Adapting Compiler Front Ends for Generalised Parsing

Submitted by

Robert Michael Walsh

for the degree of Doctor of Philosophy

provided by

Royal Holloway, University of London

2016

Declaration

I, Robert Michael Walsh, hereby declare that this thesis and the work presented in it is entirely my own. Where I have consulted the work of others, this is always clearly stated.

Signed:  (Robert Michael Walsh)

Date: 16/02/2016

Dedications

I would like to thank numerous people and organisations without whom this thesis would not have been possible.

- I would like to thank my supervisors, Adrian Johnstone and Elizabeth Scott, for the countless academic, technical and (at times) emotional support they have provided me over the years. They initially gave me the opportunity to pursue this PhD and, since then, constantly helped me to develop and refine my ideas. I am especially grateful for the assistance and feedback they provided during the final stage of the work, as well as helping me to keep up my motivation and energy.
- I would like to thank Joe Reddington and Laurence O'Toole for all the advice they provided me throughout my stay as a PhD student. I would also like to thank them for their wit and friendship, that made my stay enjoyable.
- Although he arrived during the final stage of my thesis, Thomas van Binsbergen's enthusiasm and ideas have certainly inspired me which I would like to thank him for.
- I would like to thank James Smith, Jamie Alnasir and Andrej Zukov Gregoric for their friendship and for the lunches we have had together.
- Ulrich Schaechtle and Bedour Al-Rayes started in my laboratory at roughly the same time I did, and together we have shared the PhD experience and I would like to thank them for the many discussions we have had.
- I, of course, would like to thank my mother who has raised me over the years and has provided countless support.
- I would like to thank all my friends, both in and out of Royal Holloway, who have supported me and provided me a respite at particularly difficult moments
- I would like to thank EPSRC for the financial assistance they provided as part of the studentship funding of the PLanCompS project
- Finally, I would like to thank the Swansea University half of the PLanCompS project, headed by Peter Mosses, as well as Microsoft Corporation for providing the interesting case study from which my work is built on.

Abstract

Traditional compiler front-ends are designed around near-deterministic parsing algorithms, restricting the form of the parsing grammar. The corresponding lexical analysers are required to provide only a single partitioning of an input string into tokens. This requires the lexical analyser to apply lexical disambiguation techniques.

A generalised parser can in principle generate all derivations from a grammar whose tokens are the characters of the underlying alphabet (so called character-level parsing) but this results in a very large data structure. This thesis investigates a form of lexical analysis that can offer all tokenisations of a string to a parser, as well as a corresponding multiple input parsing algorithm, MGLL, which can simultaneously construct derivations over all tokenisations. The goal is to provide equivalent generality to a character-level parser, but with significantly improved performance in terms of both the space required to represent the derivations and the time required to construct them. In addition, lexical-level disambiguation constructs are provided which may be used to model traditional lexical disambiguation strategies under the new framework.

The thesis will also discuss the translation of derivation trees into abstract syntax forms suitable for use in structural semantics, by annotating a grammar with a small set of local operations called GIFT operators. Some preliminary work on how to specify ambiguity reduction at syntactic level shall also be described.

The thesis concludes with two case studies. One demonstrating the application of this new form of lexical analysis to the C# 2.0 language specification. The other drawn from the PLanCompS [Mos+15] project, which shows how to generate derivations in an appropriate abstract syntax for the C# 1.2 language, from a parser which uses the concrete grammar from the C# 1.2 language standard specification.

Contents

Chapter 1

Introduction and Background Theory

The role of a compiler is to take a string written in some source programming language and generate a semantically equivalent machine-interpretable string. As shown in Figure 1.1, the process is conceptually considered in two parts - firstly the front-end, which takes the initial string, and analyses its structure and semantics, and secondly the back-end, which generates an equivalent machine-interpretable string. This thesis focuses on the former, developing new techniques and ideas for lexical analysis all the way through to the interface between the syntax and semantic analyser. These techniques are made possible with the advent of generalised parsing, which was popularised by Earley [Ear70], Lang [Lan74] and Tomita [Tom85], then expanded on by Farshi [Noz91], Horspool and Aycock [AH99], and Johnstone and Scott [SJ13].

1.1 The Compiler Front-end

A language can be defined in terms of words and sentences. A compiler takes the characters in an input string and initially groups them into words - the lexical analysis (or lexing) stage. It then determines whether these words form syntactically valid sentences - the syntactic analysis (or parsing) stage. Words can usually be grouped according to some word set - for example, the set of identifiers in a programming language. It can be helpful for the parsing stage if these words are grouped together and denoted by a symbolic 'token', such as `ID`. Similarly, strings representing assignment operators may be grouped together, as could strings representing binary operators. Sentences for, say, an assignment statement can then be expressed as a combination of these tokens. For example, `ID assign_op ID bin_op ID;` would be such a sentence

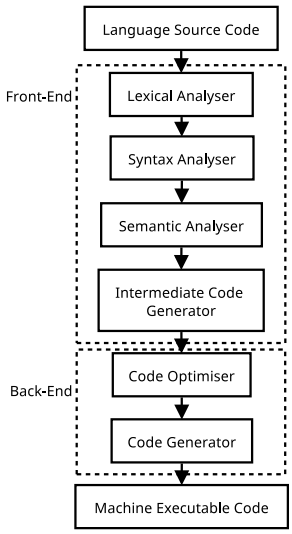


Figure 1.1: Simplified overview of the structure of a compiler

in a C-like language

Traditionally, lexical analysis produces a single grouping of words over the character string, which is then given to the parser. However, it is not always possible to know the correct word grouping, without knowing the sentential context in which it appears. Chapter 2 will introduce and explore a new technique that produces multiple groupings of words over a given character string. Chapter 3 will then present a parsing algorithm that can take multiple groupings as input, and generate all derivations over all of those groupings.

The parser produces a structure representing the way in which these sentences are built. Extracting meaning from the sentences requires breaking the sentence down into phrases. Chapter 4 will present a way to translate the structure of the sentence into a different structure for the phrases needed for semantic analysis.

The role of the language designer is to define the set of words and valid sentence structures. The former is achieved by specifying *pattern sets* (tokens) for the words. The latter is achieved by specifying a context-free grammar containing *productions* that determine the structure of the sentences. The language designer is then responsible for specifying how these sentences are restructured to obtain meaning.

In the rest of this chapter, the background theory and definitions that are necessary to understand the content of this thesis are introduced. Where appropriate, standard treatments as found, for instance, in Aho, Sethi and Ullman [ASU86] are used. The chapter will also introduce a structure that embeds all context-free derivations of a sentence, called an SPPF, which is less well known. More details can be found in

Grune [GJ08].

1.2 Strings and tokens

Definition 1.2.1. *An alphabet is a finite set of symbols, A. The set of strings whose elements lie in A is denoted by A^*. A string, $s \in A^*$, is a sequence $a_1 \dots a_n$ such that $a_1, \dots, a_n \in A$, where n is the length of the string. If n is 0 then this is the empty string, denoted by the symbol, ϵ.*

Two strings, $s, t \in A^*$ can be concatenated to form a string, $st \in A^*$. For all $\alpha \in A^*$, $\alpha\epsilon = \alpha = \epsilon\alpha$. If R and Q are sets of strings, then the concatenation RQ is the set of strings such that if $\alpha \in R$ and $\beta \in Q$ then $\alpha\beta \in RQ$.

Any string defined over some alphabet, A, can be given as input to the compiler though usually only a subset of these strings will be accepted. The role of the language designer is to develop a language specification that determines which strings should be accepted and how these strings should be subsequently processed. Usually, the first stage is to determine whether the string can be partitioned into words and produce partitions of the string as sequences of *tokens*.

Definition 1.2.2. *A token is a pair (x, Y) where x is a unique name for the token, and $Y \subseteq A^*$ is the set of strings that x denotes (the pattern set). An element of Y is known as a lexeme, and Y is the pattern of x. In this thesis, a token will often simply be referred to by its name, x. A string s is said to be matched by x if $s \in Y$.*

A language specification includes the definitions of the alphabet A and the set of tokens T. This will be referred to as the lexical specification. Some tokens have patterns which contain just a single string, for example, keywords such as **if** and **while**. These are called single-lexeme tokens and in this thesis, the bolded lexeme will be used as the name of the token, for example, (**if**, {`if`}).

A lexical analyser takes a character string, and determines if there is some sequence of tokens in T that partitions the entire string, and produces some token sequences as output if there is. If there exists more than one token sequence, i.e. there is more than one way to partition the string, then this is known as a lexical ambiguity. The next few sections describe the mechanisms for determining how to partition a character string into a sequence of tokens.

1.3 Regular Expressions

The patterns of tokens are typically defined using regular expressions. If A is the alphabet of the language, regular expressions specify subsets of A^* as follows

- ϵ is a regular expression denoting the set containing only the empty string
- a is a regular expression denoting the set containing only the character $a \in A$.

If r and q are regular expressions then

- rq is a concatenated regular expression denoting the set of all strings $\alpha\beta$ such that α is in the set denoted by r and β is in the set denoted by q.
- $r|q$ is an alternated regular expression denoting the set obtained from the union of the set denoted by r and the set denoted by q.
- (r) is a parenthesised regular expression denoting the set denoted by r.
- $r?$ is an optional regular expression denoting the union of the set denoted by r and the set containing the empty string.
- r^+ is a positive closure regular expression denoting the set of all strings obtained by concatenating one or more strings in the set denoted by r.
- r^* is a Kleene closure regular expression denoting the union of the set denoted by r^+ and the set containing the empty string.

To prevent ambiguity, the operators of a regular expression are given an order of precedence. Parentheses always have the highest precedence, followed by the Kleene closure, positive closure and optional operators. This is then followed by the concatenation, with alternation having the lowest precedence.

In traditional lexical analysis, regular expressions specify the patterns for token definitions. A string is matched by a regular expression if it is in the pattern denoted by the regular expression. The next section describes how such a match is determined for a given regular expression.

1.4 Lexical Analysis

For a lexical specification defining a set of tokens, T, $t \in T$ has a corresponding deterministic-finite state automaton [RS59], DFA_t, which matches exactly the strings in the pattern of t. The lexical analyser takes a character string, u, and partitions this

string such that $u = u_1u_2$. If u_1 is accepted by some DFA_{r_i} then r_i matches u_1. If u_1 is accepted, then the lexical analyser determines inductively whether the remaining input, u_2, is accepted. u is accepted by the lexical analyser and token sequences (called tokenisations) of form $r_ir_j \dots r_n$ are produced if u_1 is matched by r_i and u_2 is matched by $r_j \dots r_n$.

In general, there will be multiple ways to partition u. A lexical analyser could consider all ways to partition u but there are worst case exponentially many partitions. Consider a string of n characters. A partition can be inserted in $n-1$ positions between two characters. Therefore, there are $2^{(n-1)}$ ways to partition the string.

1.5 Chomsky's Grammar Hierarchy

Regular expressions are often sufficient to specify the patterns of tokens. However, to specify the set of strings accepted by a programming language, one needs a more complex description of the *language* of strings.

Formal grammars were explored by Chomsky in [Cho56]. A grammar is a formal description that generates a language, $\mathcal{L}_\Gamma$. If T is a set of tokens used by a grammar, then $\mathcal{L}_\Gamma \subseteq T^*$. A grammar is a 4-tuple, $\Gamma = (N, T, S, P)$, where N is a set of non-terminal symbols, T is a set of tokens (in this context known as the set of terminal symbols), $S \in N$ is the start symbol, and P is a set of pairs of form (x, γ) with $x \in (N \cup T)^*N(N \cup T)^*$ and $\gamma \in (N \cup T)^+ \cup \{\epsilon\}$.

P is called the set of productions. The first element of the pair is known as the left-hand side of the production, whilst the second element is known as the right-hand side of the production. T is the alphabet of the grammar, Γ.

Chomsky identified four different classes of grammars. These classes are defined by their constraints on the productions.

(Type 0) Unrestricted Grammars The set of all grammars satisfying the above definition. The problem of deciding whether a string is in the language of an unrestricted grammar is, in general, undecidable.

(Type 1) Context-sensitive Grammars For all $(x, \gamma) \in P$, then x is of the form $\alpha X \beta$ and γ is of the form $\alpha\delta\beta$ where $\alpha, \beta \in (N \cup T)^+ \cup \{\epsilon\}$, $X \in N$ and $\delta \in (N \cup T)^+$. $(S, \epsilon) \in P$ is also allowed in a context-sensitive grammar if S does not appear on the right-hand side of any production in P. In general, the problem of deciding whether a string is in the language of a context-sensitive grammar is worst-case exponential.

(Type 2) Context-free Grammars For all $(x, \gamma) \in P$, then $x \in N$ and $\gamma \in (N \cup T)^+ \cup \{\epsilon\}$. Deciding whether a string is in the language of a context-free grammar is known to be sub-cubic, but it is not known if it is linear, or even worst case quadratic.

(Type 3) Regular Grammars For all $(x, \gamma) \in P$ then $x \in N$ and either the grammar is right regular and $\gamma \in TN$ or the grammar is left regular and $\gamma \in NT$. $(S, \epsilon) \in P$ is also allowed in a regular grammar if S does not appear on the right-hand side of any production in P. A regular grammar cannot be simultaneously left regular and right regular. Language membership for regular languages is decidable in linear time.

The classes form a containment hierarchy, Type 3 $\subset$ Type 2 $\subset$ Type 1 $\subset$ Type 0.

Typically, syntactic analysis of programming languages is the process of determining whether a string is in the language of some context-free grammar. A programming language may have features that are context-sensitive but not context-free - however these features are usually handled as part of the semantic analysis, as general context-sensitive parsing algorithms offer poor performance.

1.6 Backus-Naur Form (BNF)

Backus-Naur Form (BNF) and its various extensions form a notation for describing context-free grammars by their set of productions. If $(x_1, \gamma_1), \ldots, (x_n, \gamma_n) \in P$ then

$$\begin{aligned} x_1 &::= \gamma_1 \\ &\vdots \\ x_n &::= \gamma_n \end{aligned}$$

are production rules.

A non-terminal, x_i, may be the left-hand side of multiple productions,

$$(x_i, \gamma_i), \ldots, (x_i, \gamma_p)$$

For short-hand, a vertical bar | is used to represent this choice of productions

$$x_i ::= \gamma_i | \ldots | \gamma_p$$

$\gamma_i, \ldots, \gamma_p$ are called the *alternates* for the production rule for x_i.

For any production

$$x ::= \gamma$$

$\gamma = g_1 g_2 \dots g_p$ is a string of terminal and non-terminal *instances*.

1.6.1 Extended Backus-Naur Form (EBNF)

BNF can be extended by allowing the right-hand sides of rules to be regular expressions over the symbols of the grammar. Such grammars, called EBNF [Wir96] grammars, do not increase the class of languages that can be specified, but the format can be more concise and extends the class of grammars which can be parsed using certain efficient but limited techniques. In this thesis, parsing techniques which are based on EBNF will not be discussed, and derivation trees for such grammars will not be defined. However, later an EBNF recogniser version of GLL will be given, which can be used to generate tokenisations. Extended Backus-Naur Form (EBNF) is often expressed in Wirth-style form [Wir77] but this thesis shall use regular expression notation in which productions are of the form $X ::= \gamma$, where X is a non-terminal and γ is a regular expression defined over $N \cup T \cup \{\epsilon\}$.

1.7 Derivations

For each $X \in N$, the language generated by X, $\mathcal{L}_\Gamma(X)$, comprises the strings $u \in T^*$ such that X derives u. A derivation step is of the form $\alpha X \beta \Rightarrow \alpha\gamma\beta$ where $\alpha, \beta, \gamma \in (N \cup T)^*$ and $(X ::= \gamma) \in P$. A sequence of derivation steps $S \Rightarrow \alpha_1 \Rightarrow \cdots \Rightarrow \alpha_n \Rightarrow s \in T^*$ is referred to as a derivation. The derivation step $\alpha X \beta \Rightarrow \alpha\gamma\beta$ is left-most if $\alpha \in T^*$ and right-most if $\beta \in T^*$. A derivation consisting solely of left-most derivation steps is a left-most derivation and a derivation consisting solely of right-most derivation steps is a right-most derivation.

$X \stackrel{*}{\Rightarrow} u$ is used to represent the closure of a sequence of derivation steps. If $X \stackrel{*}{\Rightarrow} u$ then $u \in \mathcal{L}_\Gamma(X)$. The role of the parser is to construct structural representations of the derivations, $S \stackrel{*}{\Rightarrow} u$, for an input u, given a grammar $\Gamma = (N, T, S, P)$. The language generated by the grammar, $\mathcal{L}_\Gamma$ is the language of S, $\mathcal{L}_\Gamma(S)$.

Consider the grammar

$$S ::= \mathbf{q}\, S \mid A\, B$$
$$A ::= \mathbf{a}$$
$$B ::= \mathbf{b}$$

A left-most derivation of the string `qqab` is

$$S \Rightarrow \mathbf{q}S \Rightarrow \mathbf{qq}S \Rightarrow \mathbf{qq}AB \Rightarrow \mathbf{qqa}B \Rightarrow \mathbf{qqab}$$

An equivalent right-most derivation is

$$S \Rightarrow \mathbf{q}S \Rightarrow \mathbf{qq}S \Rightarrow \mathbf{qq}AB \Rightarrow \mathbf{qq}A\mathbf{b} \Rightarrow \mathbf{qqab}$$

If there is a non-terminal $X \in N$ such that there exists a derivation $X \Rightarrow \alpha \overset{*}{\Rightarrow} X\gamma$, then Γ is said to be left-recursive. Similarly, if there exists a derivation $X \Rightarrow \alpha \overset{*}{\Rightarrow} \gamma X$ then Γ is said to be right-recursive.

1.7.1 Derivation Trees

A derivation tree is an ordered, rooted tree with the following properties

- Leaf nodes have labels $t \in (T \cup \epsilon)$
- Interior nodes have labels $n \in N$
- If X is the label of an interior node, then the children of this node are labelled with symbols in α for some $X \Rightarrow \alpha$. The children are ordered as they appear, left-to-right, in the sequence α.
- When read left-to-right, the leaf nodes, omitting nodes labelled ϵ, yield some string in the language generated by the grammar, $\mathcal{L}_\Gamma$.

Each derivation corresponds to exactly one derivation tree although a derivation tree can correspond to one or more derivations. The two derivations given of the string `qqab` for the grammar on page 14 correspond to a single derivation tree, seen in Figure 1.2.

1.7.2 Syntactic Ambiguity

The syntactic structure is usually the basis of semantic analysis. If for all strings in $\mathcal{L}_\Gamma$, there exists exactly one left-most derivation, then Γ is syntactically unambiguous. If there exists more than one left-most derivation for a given string, then Γ is syntactically ambiguous. The existence of more than one left-most derivation is problematic for semantics as this means there is more than one way to interpret the string. Chapter 5 will give a discussion of some elementary disambiguation techniques which are used in the C# 1.2 case study in Chapter 6. Note, if there exists more than one left-most

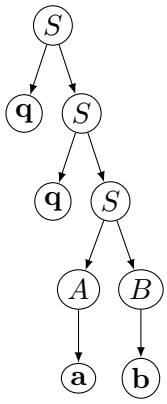


Figure 1.2: Derivation tree for the string `qqab` for the grammar on page 14

derivation of a given string, then there also exists more than one derivation tree. For example, consider the grammar

$$S ::= \mathbf{q}\, S \mid A\, B \mid \mathbf{a}$$
$$A ::= \mathbf{a}$$
$$B ::= \mathbf{b}\, S \mid \epsilon$$

The string `qaba` has two left-most derivations corresponding to the derivation trees in Figure 1.3a and Figure 1.3b respectively.

$$S \Rightarrow \mathbf{q}S \Rightarrow \mathbf{q}AB \Rightarrow \mathbf{q}aB \Rightarrow \mathbf{qab}S \Rightarrow \mathbf{qaba}$$
$$S \Rightarrow \mathbf{q}S \Rightarrow \mathbf{q}AB \Rightarrow \mathbf{qa}B \Rightarrow \mathbf{qab}S \Rightarrow \mathbf{qab}AB \Rightarrow \mathbf{qab}$$

1.7.3 Shared Packed Parse Forests (SPPFs)

In general, there can be a very large (and in some cases infinite) number of left-most derivations. Constructing each derivation tree individually would be neither efficient nor practical. Instead, these derivation trees can be combined into a single structure called a shared packed parse forest (SPPF) [Tom85].

To combine derivation trees, one must determine which nodes can be shared. Simply sharing nodes in the derivation tree that have the same label will not work as nodes with the same label can refer to different contexts. The symbols in a derived string can be denoted by their position (or index) in the string, starting at an index of 0. A portion of the string can be denoted by two indices, the left extent representing the position of the first character in the string portion, and the right extent representing the position

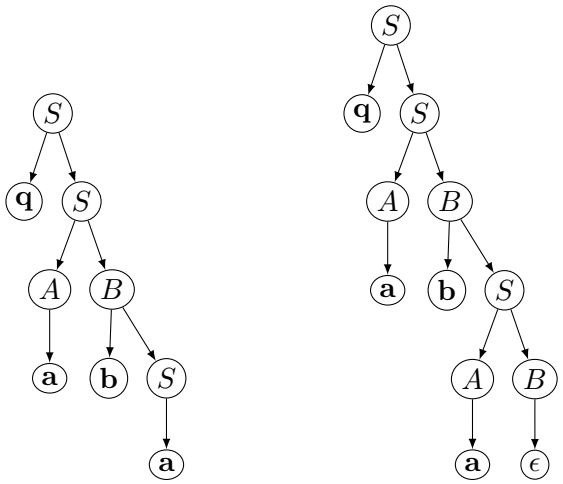


Figure 1.3: The derivations of the string `qaba` for the grammar on page 16

after the last character of this string portion. Initially, left and right extents are added to the node labels, so that every label is a triple (a, i, j) where $a \in N \cup T \cup \{\epsilon\}$ and $i \leq j$, $i, j \in \mathbb{N}$. A leaf node of the form (t, i, j) means that the terminal t is matched by the segment of the input beginning at a left extent of i and ending in a right extent of j. An interior node has a label of the form (X, k, l) where the left-most child of the node has a left-extent of k and the right-most child has a right-extent of k.

For the derivations of `qaba` for the grammar on page 16, the derivation trees with extents are as given in Figure 1.4a and Figure 1.4b respectively. By comparing the two trees, it is easy to see where it may be possible to *share* nodes. In this case, the only point where the two trees differ is in the subtree rooted at $(S, 3, 4)$. To manage these separate derivation trees, a *packed node* is introduced for each family of children. This packed node is labelled with a production of form $X ::= \gamma$ denoting the production that this family of children correspond to. Figure 1.5a and Figure 1.5b show how packed nodes are included for the running example.

A shared packed parse forest (SPPF) is the union of these packed derivation trees with extents. Non-packed nodes are shared if they have the same label. Shared nodes may have distinct packed node children, these packed nodes will be the children of this single node. For a set of derivation trees, $\mathcal{X}$, the result is a directed, bipartite graph with two disjoint node sets, V_p and V_s which have the following properties

- Each $u \in V_s$ has a label of the form (r, i, j) where $r \in (N \cup T \cup \{\epsilon\})$, $i, j \in \mathbb{N}$ and $i \leq j$.
- Each $w \in V_p$ has a label of the form $(X ::= \alpha\beta)$, where $X ::= \alpha\beta \in P$. Elements

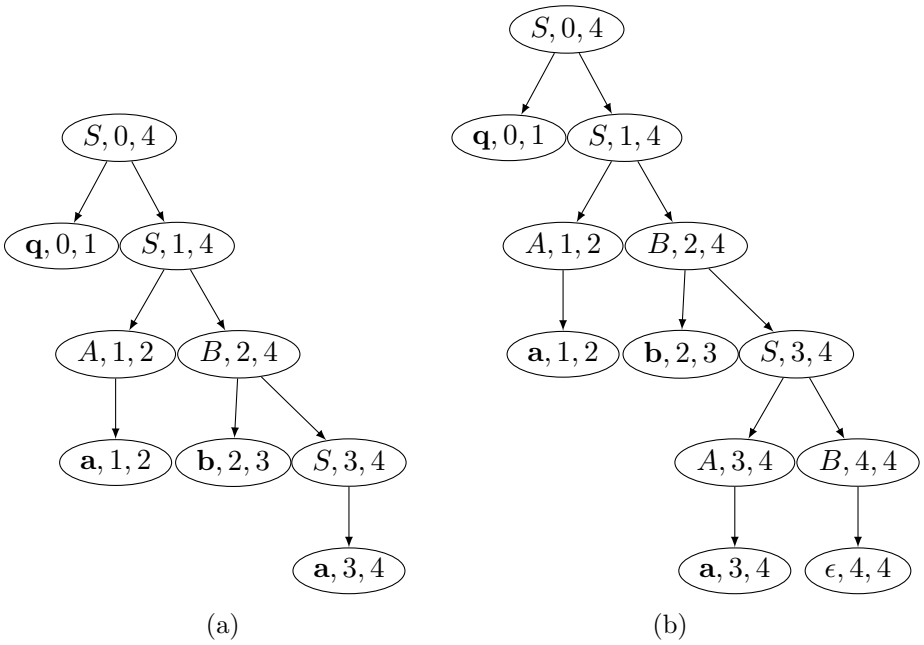


Figure 1.4: Derivation trees with extents for the string `qaba` for the grammar on page 16

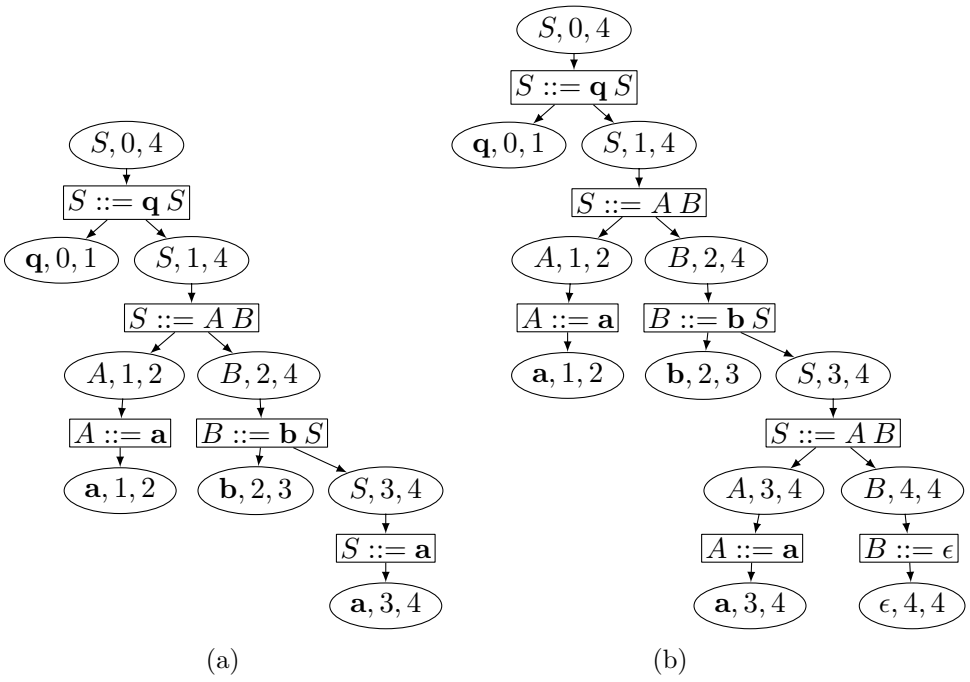


Figure 1.5: Derivation trees with extents and packed nodes for the string `qaba` for the grammar on page 16

of V_p are called packed nodes.

- $u \in V_s$ with labels of the form (r, i, j) where $r \in T$ are terminal nodes.
- $u \in V_s$ with labels of the form (ϵ, i, i) where $i \in \mathbb{N}$ are epsilon nodes.
- $u \in V_s$ with labels of the form (r, i, j) where $r \in N$ are non-terminal nodes. Each non-terminal node has one or more elements of V_p as its children.
- If u is a node in V_s with more than one child, then there is a syntactic ambiguity at u.
- A node $w \in V_p$ has one or more children. These children are elements of V_s.
- For any two trees in $\mathcal{X}$, if there is a node in the two trees with the same label and same yield, then there is exactly one corresponding node in the SPPF.

The final SPPF for the example is given in Figure 1.6. For brevity, in the rest of this thesis, where there are no ambiguities, packed nodes will be omitted.

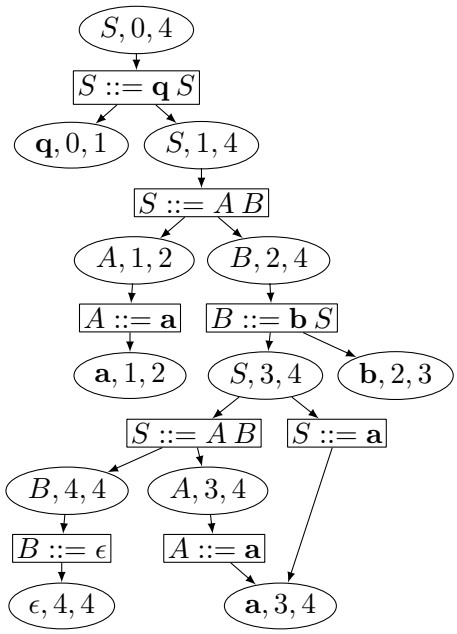


Figure 1.6: SPPF for the derivations of string `qaba` over the grammar on page 16

1.7.4 Binarised SPPF

As noted in [SJ10b], if k is the maximum number of nonterminals in a given alternate in a grammar, then the grammar can generate an SPPF of size $O(n^k)$. To constrain the worst-case size complexity to cubic, a binarised form of the SPPF is constructed.

Consider the grammar

$$S ::= \mathbf{q}\, S\, B \mid A\, B\, \mathbf{c} \mid \mathbf{a}\, \mathbf{c}$$
$$A ::= \mathbf{a}$$
$$B ::= \mathbf{b}\, S \mid \epsilon$$

The string `qabacc` would result in the non-binarised SPPF given in Figure 1.7.

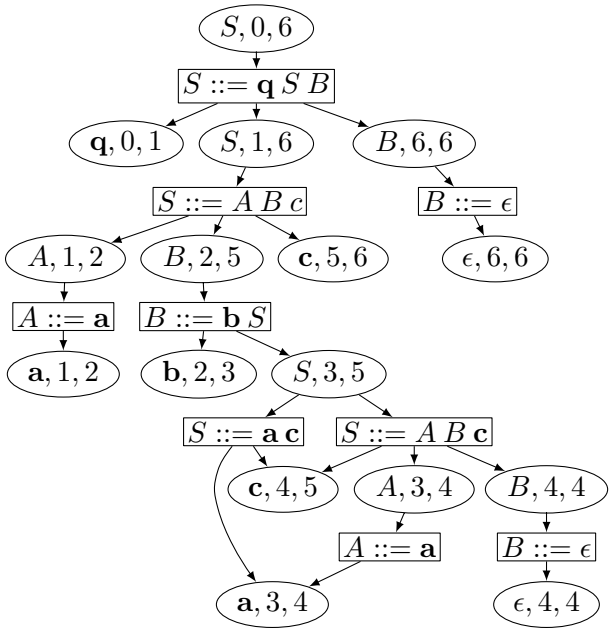


Figure 1.7: Non-binarised SPPF for the derivations of string `qabacc` over the grammar on page 20

A binarised SPPF introduces intermediate nodes representing grammar slots. Packed nodes can have at most two children - a left child, and a right child. For any given packed node, all but the right-most child of the packed node is grouped under a new intermediate node. Packed nodes additionally include a pivot value, representing the right-extent reached by its left child. The properties of the binarised SPPF are then as follows:

- Each $u \in V_s$ has a label of form (r, i, j) where $r \in (N \cup T \cup \{\epsilon\})$ or r is a grammar slot $X ::= \alpha \cdot \beta$, $X ::= \alpha\beta \in P$, $i, j \in \mathbb{N}$ and $i \leq j$.
- Each $w \in V_p$ has a label of form $(X ::= \alpha \cdot \beta, i)$, $X ::= \alpha\beta \in P$ and $i \in \mathbb{N}$. Elements of V_p are called packed nodes, and i is called the pivot value.
- $u \in V_s$ with labels of the form (r, i, j) where $r \in T$ are terminal nodes.

- $u \in V_s$ with labels of the form (ϵ, i, i) where $i \in \mathbb{N}$ are epsilon nodes.
- $u \in V_s$ with labels of the form (r, i, j) where $r \in N$ are non-terminal nodes. Each non-terminal node has one or more elements of V_p as its children.
- $u \in V_s$ such that r is a grammar slot $X ::= \alpha \cdot \beta$ are intermediate nodes. Each intermediate node has one or more children that are elements of V_p.
- If u is a node in V_s with more than one child, then there is a syntactic ambiguity at u.
- A node $w \in V_p$ has one or two children - an (optional) left child and a right child. These children are elements are of V_s. If w has label (r, i) and the right child has label (t, k, j), then $k = i$.

The binarised version of the SPPF from parsing `qabacc` over the grammar on page 20 is given in Figure 1.8.

1.8 Overview of the Thesis

Chapter 2 will present a new approach to lexical analysis which can offer all tokenisations of a string to a parser. A lexer under this approach will produce a set of token triples as output for a given lexical specification. This set will embed all the tokenisations of the strings in a way that ensures worst case quadratic size with respect to the length of the string. The chapter will then investigate how elements of this set can be removed to simulate the effects of the disambiguation techniques used in traditional lexical analysis, and at the same time provide more control over how tokenisations are eliminated. The final part of the chapter will review previous work in the context of this new approach, such as character level parsing and Schrödinger's tokens.

Chapter 3 will then present an extension of the generalised parsing algorithm GLL, called MGLL, that can accept multiple tokenisations as input. The chapter will include a discussion on how the derivation trees resulting from parsing multiple inputs are represented, by giving an extension to the SPPF representation.

Chapter 4 will present a small set of grammar annotations, known as the GIFT operators, that can be used to describe the translation of derivation trees into a form more suitable for structural operational semantics. This will also discuss how the same set of annotations can be used to describe an equivalent grammar to grammar transformation.

Chapter 5 will discuss some of the implications of the theoretical concepts in Chapters 2 and 3. The chapter contains a more in-depth discussion about how the new

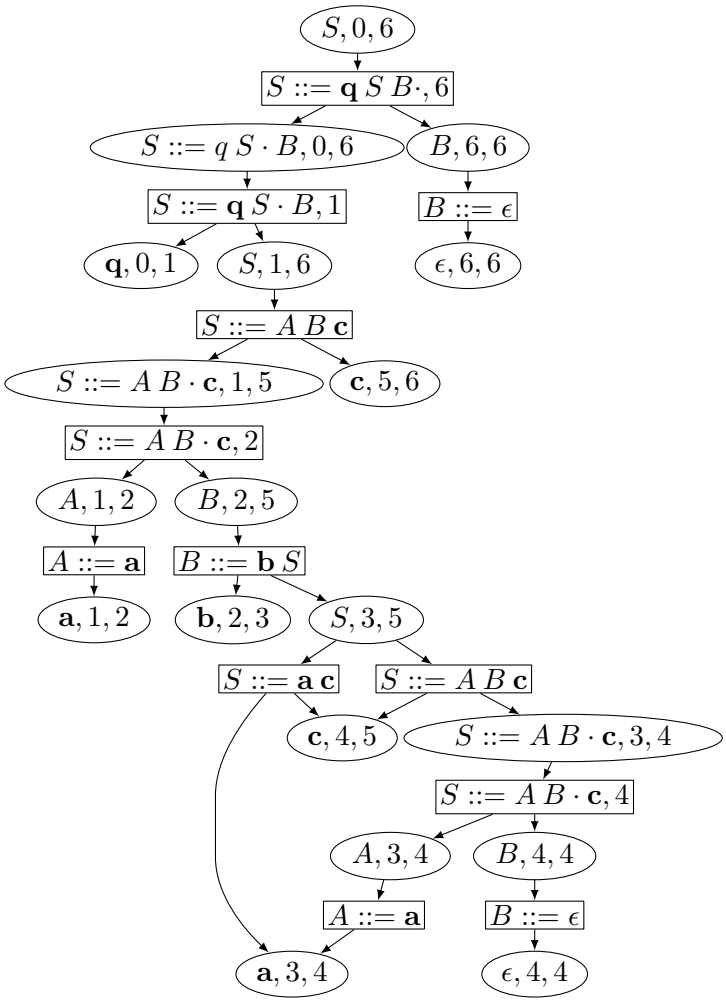


Figure 1.8: Binarised SPPF for the derivations of string `qabacc` over the grammar on page 20

lexical analysis approach compares to character-level parsing. It will then consider an approach to syntactic disambiguation that operates similarly to the lexical disambiguation rules in Chapter 2. The possibility of implementing a lexical analyser that permits tokens with context-free patterns is explored, along with a method for suppressing whitespace and comment tokens under the new lexical analysis approach. The chapter will also consider the implications that this new approach has on the well-known example of the ANSI-C type/variable name conflict.

Chapter 6 will present a set of general frameworks that implement the theoretical work discussed throughout this thesis. It will then apply this implementation to produce a lexical analyser and parser for the C# 2.0 language specification [HCC06].

This will be followed with a discussion on the use of this implementation to implement a compiler front-end for the C# 1.2 language specification [HCC02] as part of the PLanCompS [Mos+15] project.

Concluding remarks, with a discussion of future work, will then be discussed in Chapter 7.

The appendices contain the lexical and syntax specification for both C# language specifcations, as well as a hand-crafted 'abstract syntax' for C# 1.2 used for the PLanCompS project.

Chapter 2

Lexical Analysis in the Light of Generalised Parsing

In the lexical specifications for programming languages, there is nearly always more than one way to tokenise a string. A classic example can be found in the tokenisation of identifiers. A string `ab` could denote a single identifier `ab` or two identifiers, `a` and `b`, concatenated. Normally, during the lexical analysis phase, one of these tokenisations is selected to be used by the parser. As discussed in the last chapter, formal languages are typically specified in terms of sets of words (tokens) over a finite character set and, then, as sentences of tokens. A compiler takes an input character string, which is first tokenised into strings of tokens, that are then parsed. In lexical specifications designed for traditional lexical analysers, there is usually some mechanism to ensure that only one string of tokens is returned to be used by the parser. The existence of more than one tokenisation of a given character string is known as a lexical ambiguity. The process of discarding some tokenisations in favour of others shall be referred to as lexical ambiguity reduction. Typically, lexical analysers choose the tokenisation in which the tokens from the left are the longest match of the character string, with some specified priority ordering of the tokens being used when there is more than one such longest match.

Whilst the mechanisms used by traditional lexical analysis are generally good enough, there are instances where the lexer will reject tokenisations which are syntactically valid, whilst retaining one which is not valid. Languages such as Java or C# will have the following token-pattern pairs:

$$(\texttt{++}, \{\texttt{++}\}), (\texttt{+}, \{\texttt{+}\})$$

As noted in [Lip10], for the string `a+++b` there are three possible tokenisations:

1 **ID ++ + ID**

2 **ID + ++ ID**

3 **ID + + + ID**

All tokenisations are syntactically correct strings, however, most lexical specifications for these programming languages will choose the first option. In this case, any approach for selecting a single tokenisation will give a tokenisation that can be parsed.

However, consider the string `a++++b` which has five possible tokenisations:

1 **ID ++ ++ ID**

2 **ID ++ + + ID**

3 **ID + ++ + ID**

4 **ID + + ++ ID**

5 **ID + + + + ID**

A Java or C# lexer would return the first tokenisation. However, there are no grammatical constructs in which a **++** token can be followed by another **++** token. In this case, the tokenisation chosen by traditional lexical analysis will be rejected by the parser, even though other tokenisations (tokenisations two, four and five) would be accepted.

Of course, these problems can be avoided by inserting appropriate whitespace into the character string - the string `a ++ + + b` would be tokenised as the second tokenisation above. Given that these languages are essentially whitespace insensitive with respect to operators, it is uncomfortable that there is a case in which the placement of whitespace is important.

A more interesting case is nested type parameters in Java. If one considers the Java string `List<List<Integer>>`, then the tokenisation that is given by the Java parser is **ID < ID < ID >>**, which would be rejected by the standard grammar, as it expects a **>** token to close a type parameter, rather than a **>>** token. The tokenisation that would be accepted by the Java standard grammar would be the tokenisation **ID < ID < ID > >**. The Java compiler must work round this issue by using a non-standard grammar which accepts the first tokenisation as nested type parameterisation.

Additionally, there are cases where the tokenisation that should be chosen is dependent on the surrounding context. In these cases, it would be more appropriate to simply pass all these tokenisations to the parser. C# has a number of keywords (such as **method** and **get**) that are not considered reserved words by the language. This

means they can potentially be considered as identifiers or keywords depending on the context. It would be useful to be able to pass both tokenisations to the parser.

One possible approach to deal with multiple tokenisations is to output all tokenisations at the lexical analysis phase and then separately parse each tokenisation. If more than one tokenisation is accepted by the parser, then this would be treated as a syntactic ambiguity. As shown in Chapter 1, the number of ways of partitioning a string is worst case exponential in the length of the string. In practice, there will be significantly fewer tokenisations, but often the actual number can be too large to be useful for parsing. In this chapter, techniques will be developed and studied that will allow one to explore the entire spectrum from resolving all lexical ambiguities to allowing all tokenisations to be parsed. This will involve a new representation of the tokenisations of a string, as well as the formalisation and expansion of the traditional techniques for resolving lexical ambiguity. This new approach will be described as the *multilexer* approach.

2.1 Traditional approach to Lexical Analysis

As described in Chapter 1, lexical analysis concerns itself with partitioning character strings. Suppose that the patterns of the tokens $t_1, \ldots, t_p$ are specified by the regular expressions $r_1, \ldots, r_p$ respectively. If u is a character string that can be partitioned as $u = u_1 \ldots u_n$, where each u_i can be matched by some r_{u_i}, then $t_1 \ldots t_n$ is a tokenisation of u. If there exists at least one tokenisation of u for the given set of tokens, then a traditional lexical analyser will return some tokenisation. If there exists no tokenisation of u, then the lexical analyser reports an error.

Recall that there can be multiple ways in which the character string can be tokenised. A string fragment in C such as `forbid` could be tokenised (correctly) as a single identifier token, or as a concatenation of one or more identifier tokens for different partitions of the string fragment. It could also be tokenised as a **`for`** keyword concatenated with the tokenisations of the string fragment `bid`. The C lexical analyser chooses the tokenisation that is the longest match. Note that longest match is applied across all possible tokens and not just those of the same identifier - the tokenisations in which `for` is tokenised as a keyword are eliminated even though `for` is a proper prefix. This is necessary as this ensures, for example, that a string fragment `if` is tokenised as an **`if`** keyword rather than an identifier named `i` concatenated with an identifier named `f`. This string fragment could also be tokenised as an identifier whose name is `if`. Here a classical lexical analyser will use a priority mechanism. Keywords are nearly always given priority over identifiers, and so an **`if`** token is chosen. Notable exceptions

to this are the keywords in the PL/I [AC76] standard, which are never reserved and therefore can be used as identifiers, and some keywords in the C# standard (such as **`get`** and **`set`**) which are only considered as such in certain contexts.

Traditional lexical analysis is fast but does not lend itself towards constructing all tokenisations. Listing multiple tokenisations as sequences of token/lexeme pairs omits details about each tokenisation.

2.2 Tokens with Extents

Classically, the lexeme that a token matches is not expressed for the parsing phase. As will be seen in Chapter 3, aspects of the relationship between the token and the portion of the string that the token matched need to be maintained when running a parser that handles multiple inputs. In this section, lexical analysis will be expanded to store the matched character extents with each token.

Listing multiple tokenisations simply as sequences of token names does not give enough information to distinguish between each unique tokenisation. Consider the lexical specification consisting of the single token-pattern pair

$$(\mathbf{D}, \{\mathsf{a}, \mathsf{b}, \mathsf{c}, \mathsf{ab}, \mathsf{bc}, \mathsf{abc}\})$$

The string `abc` can be tokenised as **D** or **DD** or as **DDD**. For **D**, there is a single token/lexeme pair $(\mathbf{D}, \mathsf{abc})$, and for **DDD** there is a single token/lexeme sequence of pairs $(\mathbf{D}, \mathsf{a})(\mathbf{D}, \mathsf{b})(\mathbf{D}, \mathsf{c})$. But there are two possible token/lexeme sequences of pairs $(\mathbf{D}, \mathsf{ab})(\mathbf{D}, \mathsf{c})$ and $(\mathbf{D}, \mathsf{a})(\mathbf{D}, \mathsf{bc})$ for **DD**. These possibilities are presented in tabular form in Table 2.1. There is a need to distinguish between these tokenisations, as later a semantics analysis phase may need access to the underlying lexemes. This is not a problem in traditional lexical analysis, as longest match ensures only the tokenisation **D** is possible. Even if **D** was not an option, a semantics analyser can safely assume that the corresponding token/lexeme sequence chosen was $(\mathbf{D}, \mathsf{ab})(\mathbf{D}, \mathsf{c})$. For lexical analysis that allows ambiguities, one must work harder to ensure that there remains a one-to-one mapping between each tokenisation and the underlying character sequence.

To distinguish between tokenisations of a given character string, it is not necessary to record the lexemes. It is sufficient to construct sequences of token/integer pairs (a, j), where a is the token selected and j is the right character extent of the corresponding lexeme in the input string. As a sequence of pairs is read (from left to right), the left extent of the lexeme is the right extent of the previous lexeme. The tokenisations given

	a	b	c
1	**D**	**D**	**D**
2	**D**		**D**
3		**D**	**D**
4		**D**	

Table 2.1: The list of all tokenisations for the string abc

in Table 2.1 can then be represented as the sequences

$$(\mathbf{D}, 1)(\mathbf{D}, 2)(\mathbf{D}, 3)$$
$$(\mathbf{D}, 1)(\mathbf{D}, 3)$$
$$(\mathbf{D}, 2)(\mathbf{D}, 3)$$
$$(\mathbf{D}, 3)$$

Note that there are two distinct sequences, $(\mathbf{D}, 1)(\mathbf{D}, 3)$ and $(\mathbf{D}, 2)(\mathbf{D}, 3)$.

In general, when tokenising a character string, a sequence of pairs can be returned - where each pair contains the token and the right extent of the lexeme to which the token corresponds. Sequences of token/index pairs with the property that the indices form a monotonically increasing sequence of positive integers will be referred to as *indexed token strings*. In the above example, the last elements at the end of every sequence have the same right character extent. In general, the sequences could end with elements that have different right character extents. A set of indexed token strings is called an indexed token string (ITS) set if the last element of every sequence has the same right character extent.

Of course, enumerating the ITS set is still worst case exponential in the length of the string. An ITS set will normally have redundancy in the sense that the same pair may appear in the same position in more than one sequence. For example, the pairs $(\mathbf{D}, 1)$ and $(\mathbf{D}, 3)$ in the first sequence above appear in the same positions in other sequences in the ITS set. Instead, one can consider a set of triples for the above set

$$\{(\mathbf{D}, 0, 1), (\mathbf{D}, 0, 2), (\mathbf{D}, 0, 3), (\mathbf{D}, 1, 2), (\mathbf{D}, 1, 3), (\mathbf{D}, 2, 3)\}$$

These sets of triples will be referred to as the sets of tokens with extents (TWE).

A TWE set is defined to be any finite set of triples in the form (a, i, j) where a is a token and $0 \leq i < j$ are integers. Formally, a *set of tokens with extents* is a finite set

$$\Sigma \subseteq \{(t, i, j) | t \in T, i, j \in \mathbb{N}, i < j\}$$

where T is the set of tokens, and i and j are the left and right extents respectively. For a TWE set Σ, the *floor* of Σ is the smallest integer i such that there is an element $(a, i, k) \in \Sigma$, and the *height* of Σ is the largest j such that there is an element $(a, l, j) \in \Sigma$.

This set adequately encompasses the set of tokenisations given above. While there may be exponentially many tokenisations of a character string of length n, there are at most $O(n^2)$ triples, thus the corresponding TWE set representation is worst case quadratic in the length of the character string. This reduction in the complexity implies that several ITS sets correspond to the same TWE set.

Any ITS set X corresponds to a TWE set, Σ_X, as illustrated in the example above. An indexed token string, r, of form $(t_1, i_1)(t_2, i_2) \ldots (t_k, i_k)$ corresponds to a TWE set of form $\{(t_1, 0, i_1), (t_2, i_1, i_2), \ldots, (t_k, i_{k-1}, i_k)\}$. Σ_X is defined to be the union of the TWE sets of the indexed token strings of X. If the elements of X are tokenisations of a given character string, γ of length m, then Σ has floor 0 and height m. The TWE set Σ_X is uniquely defined for X, but two different ITS sets may have the same corresponding TWE set. For example, for the string `abbc`, there is the ITS set X_1 of all tokenisations

$$\begin{aligned}\{&(\mathbf{D}, 1)(\mathbf{D}, 2)(\mathbf{D}, 3)(\mathbf{D}, 4),\\ &(\mathbf{D}, 1)(\mathbf{D}, 2)(\mathbf{D}, 4),\\ &(\mathbf{D}, 2)(\mathbf{D}, 3)(\mathbf{D}, 4),\\ &(\mathbf{D}, 2)(\mathbf{D}, 4)\}\end{aligned}$$

and the subset X_2 containing just two of the tokenisations

$$\begin{aligned}\{&(\mathbf{D}, 1)(\mathbf{D}, 2)(\mathbf{D}, 3)(\mathbf{D}, 4)\\ &(\mathbf{D}, 2)(\mathbf{D}, 4)\}\end{aligned}$$

but

$$\Sigma_{X_1} = \{(\mathbf{D}, 0, 1), (\mathbf{D}, 1, 2), (\mathbf{D}, 2, 3), (\mathbf{D}, 3, 4), (\mathbf{D}, 2, 4), (\mathbf{D}, 0, 2)\} = \Sigma_{X_2}$$

Furthermore, there exist many TWE sets which do not correspond to any ITS set, for example,

$$\{(\mathbf{D}, 0, 2), (\mathbf{D}, 1, 3), (\mathbf{D}, 3, 4)\}$$

Most of the TWE sets that will be considered in practice will correspond to sets of tokenisations of a character string, and it is as TWE sets that sets of indexed token

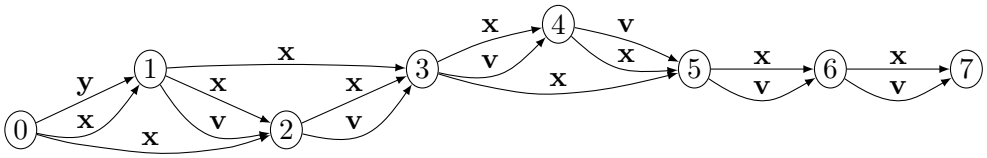
strings will eventually be given to an MGLL parser. However, the initial discussion and analysis will be on the TWE sets and their relationship to ITS sets.

2.3 TWE and ITS Sets

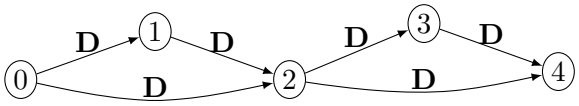
It can be helpful to visualise a TWE set as a labelled directed, acyclic graph with multiple edges between some pair of nodes. For a set $\Sigma = \{(a_1, i_1, j_1), \ldots, (a_p, i_p, j_p)\}$, the integers $i_1, \ldots, i_p, j_1, \ldots, j_p$ are the labels of nodes in the graph, and $a_1, \ldots, a_p$ are the labels of edges in the graph. An edge labelled **a** between nodes i and j exists if and only if $(\mathbf{a}, i, j) \in \Sigma$, so triples correspond precisely to edges in the graph. For example, consider the TWE set

$$\Sigma = \{(\mathbf{y}, 0, 1), (\mathbf{x}, 0, 1), (\mathbf{x}, 0, 2), (\mathbf{x}, 1, 2), (\mathbf{v}, 1, 2), (\mathbf{x}, 1, 3), (\mathbf{x}, 2, 3), (\mathbf{v}, 2, 3), (\mathbf{x}, 3, 4),$$
$$(\mathbf{v}, 3, 4), (\mathbf{x}, 3, 5), (\mathbf{x}, 4, 5), (\mathbf{v}, 4, 5), (\mathbf{x}, 5, 6), (\mathbf{v}, 5, 6), (\mathbf{x}, 6, 7), (\mathbf{v}, 6, 7)\}$$

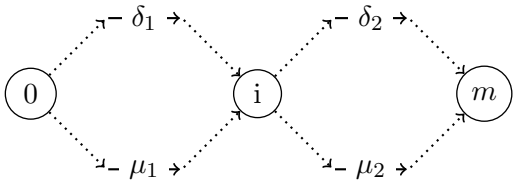
This can be visualised graphically as



For the TWE set Σ_{X_1} on page 29, the TWE graph is



Paths from 0 to the largest node, correspond to indexed token strings. It is intuitively obvious that if a TWE set Σ of height m is to correspond to an ITS set then every node in the graph must lie on a path from 0 to m. Such a path is called a *complete path*. Similarly, if two indexed token strings $\delta_1\delta_2$ and $\mu_1\mu_2$ correspond to paths which intersect at the right hand ends of δ_1 and μ_1



then there are also paths corresponding to $\delta_1\mu_2$ and $\mu_1\delta_2$. Thus if an ITS set contains $\delta_1\delta_2$ and $\mu_1\mu_2$ then its corresponding TWE set will also correspond to the ITS set

obtained by adding $\delta_1\mu_2$ and $\mu_1\delta_2$. Formally, an indexed token sequence r of form $(t_1, i_1)(t_2, i_2)\ldots(t_k, i_k)$ is embedded in a TWE set Σ if, i_k is the height of Σ, $(t_1, 0, i_1) \in \Sigma$, and for every contiguous subsequence $(t_k, i_k)(t_{k+1}, i_{k+1})$ in r one has $(t_{k+1}, i_k, i_{k+1}) \in \Sigma$. The set of all indexed token strings embedded in Σ will be denoted by $strings(\Sigma)$. Recall that Σ_X is the TWE set corresponding to X, so $\Sigma_{strings(\Sigma)} \subseteq \Sigma$ is a TWE set such that $(t, i, j) \in \Sigma_{strings(\Sigma)}$ if, and only if, a contiguous subsequence $(t_k, i)(t, j)$ exists in some string in $strings(\Sigma)$.

A TWE set Σ is said to be *tight* if every triple in the set belongs to an indexed token sequence embedded in Σ. An ITS set X is *consistent* if every string embedded in Σ_X is an element of X.

If a TWE set Σ is not tight, then there exist triples in the set whose corresponding edges in the graphical representation are not on a complete path. Triples whose corresponding edges lie on paths from the root node, but not on a complete path will generally generate some parser activity. Triples whose corresponding edges do not lie on paths from the root node will not generate MGLL parser activity but will increase the size of the data structures unnecessarily. Also, as seen later, these triples could interfere with attempts to lexically disambiguate the set. Ensuring the tightness of a TWE set is, therefore, important.

Lemma 2.3.1. *A TWE set Σ, with height m, is tight if and only if $\forall(a, i, j) \in \Sigma$*

- *$i = 0$ or there exists an element $(a', i', i) \in \Sigma$, and*
- *$j = m$ or there exists an element $(a', j, j') \in \Sigma$*

Proof. Suppose that Σ is tight, then (a, i, j) belongs to a string embedded in Σ. If $i \neq 0$ then an indexed token sequence of the form $\alpha(a', i)(a, j)\beta$ must be embedded in Σ. Therefore there must be some $(a', i', i) \in \Sigma$. Similarly, if $j \neq m$ then an indexed token sequence $\gamma(a, j)(a', j')\delta$ must be embedded in Σ. Therefore there must be some $(a', j, j') \in \Sigma$.

Now suppose both properties hold for every $(a, i, j) \in \Sigma$. Since $i < j$ in any triple, the elements of the form

$$(a_0, 0, i_0), (a_1, i_0, i_1), \ldots, (a_p, i_p, i)$$

must all be in Σ due to the first property. Also, the elements of the form

$$(b_0, j, j_0), (b_1, j_0, j_1), \ldots, (b_f, j_f, m)$$

must all be in Σ due to the second property. Hence the sequence

$$(a_0, i_0)(a_1, i_1) \ldots (a_p, i)(t, j)(b_0, j_0), (b_1, j_1) \ldots (b_f, m)$$

is embedded in Σ and Σ is tight. □

Lemma 2.3.2. *This lemma is divided into two steps:*

a) A TWE set Σ is tight if and only if $\Sigma_{strings(\Sigma)} = \Sigma$, which is if and only if $\Sigma \subseteq \Sigma_{strings(\Sigma)}$ since $\Sigma_{strings(\Delta)} \subseteq \Delta$ for all TWE sets Δ

b) For any ITS set X, Σ_X is tight.

Proof.

a) By definition, if Σ is tight then every triple $\sigma \in \Sigma$ belongs to the TWE set of some string in $strings(\Sigma)$ so $\sigma \in \Sigma_{strings(\Sigma)}$. Conversely, if $\Sigma = \Sigma_{strings(\Sigma)}$ and $\sigma \in \Sigma$, then $\sigma \in \Sigma_{strings(\Sigma)}$ and σ belongs to some string embedded in Σ.

b) By definition, if $(t, i, j) \in \Sigma_X$ then (t, i, j) belongs to some string $u \in X$. Also by definition, u is embedded in Σ_X. $(t, i, j) \in \Sigma_{strings(\Sigma)}$ and the result follows from a).

□

As stated, a TWE set will embed an ITS set. The consistency of this ITS set is another important feature.

Lemma 2.3.3. *This lemma is divided into two steps:*

a) An ITS set X is consistent if and only if $X = strings(\Sigma_X)$, the smallest enclosing consistent set of X.

b) For any TWE set Δ, $strings(\Delta)$ is consistent.

Proof.

a) By definition of Σ_X and $strings(\Sigma_X)$, $X \subseteq strings(\Sigma_X)$. By the definition of consistency, X is consistent if and only if $strings(\Sigma_X) \subseteq X$.

b) Let $X = strings(\Delta)$. The proof is to show that $strings(\Sigma_X) = X$, and thus that the result follows from a). By definition of Σ_X and $strings(\Sigma_X)$, $X \subseteq strings(\Sigma_X)$ for any X. So the remainder of the proof is to show that $strings(\Sigma_X) \subseteq X$.

For any TWE sets $\Delta \subseteq \Sigma$, if the height of Δ is equal to the height of Σ then $strings(\Delta) \subseteq strings(\Sigma)$. If $X = \emptyset$ then X is consistent. If $X \neq \emptyset$ then the height

of Σ_X is equal to the height of Δ. By definition, $\Sigma_X \subseteq \Delta$ so $strings(\Sigma_X) \subseteq X$, as required.

□

If X is the set of all tokenisations of a given character string then Σ_X will be tight. However, lexical ambiguity reduction may create TWE sets that are not tight. As discussed, tightness is important for minimising the size of the data structures - pruning is a procedure that will remove the elements that prevent the TWE set from having the tightness property. A definition of $prune(\Sigma)$ is given, which constructs the maximum size tight TWE set $\Sigma' \subseteq \Sigma$:

- Construct Σ^1 by adding all elements of Σ of form $(t, 0, j)$ for any t, j
- Form the closure of Σ^1 under the property that if $(t', i, j) \in \Sigma^1$ then $(s, j, k) \in \Sigma^1$ for all $(s, j, k) \in \Sigma$.
- Construct Σ' by adding all elements of Σ^1 of form (t, i, m) for any t, i
- Form the closure of Σ' under the property that if $(t', i, j) \in \Sigma'$ then $(s, k, i) \in \Sigma'$ for all $(s, k, i) \in \Sigma^1$.

If there exist no element of Σ whose corresponding edge in the graphical representation is on a complete path (which is precisely when $strings(\Sigma) = \emptyset$), then the maximum size tight TWE set that is given by $prune(\Sigma)$ is the empty set.

Lemma 2.3.4. *For a TWE set Σ, the set, Σ', constructed by prune(Σ), is tight. Additionally, the set of indexed token strings embedded in Σ' is exactly the set of indexed token strings embedded in Σ.*

Proof. By construction, if $\Sigma' = \emptyset$ then $strings(\Sigma) = \emptyset$ and the result holds. If an element of the form $(t, i, m) \in \Sigma'$, then there must be a sequence in $strings(\Sigma)$. By construction, the floor and height of Σ' must be the same as the floor and height of Σ, meaning there must be some $(t_0, 0, i_0) \in \Sigma'$ and some $(t_m, i_{m-1}, m) \in \Sigma'$.

If $(t, i, j) \in \Sigma'$ then $(t, i, j) \in \Sigma^1$. If $i > 0$, then

$$(t_0, 0, i_1), (t_1, i_1, i_2), \ldots, (t_i, i_i, i), (t, i, j) \in \Sigma^1$$

If $(t, i, j) \in \Sigma'$ then $(t_i, i_i, i) \in \Sigma'$. If $j < m$ then $(t_{j+1}, j, j_{j+1}) \in \Sigma'$. Therefore, Σ' is tight.

As $\Sigma' \subseteq \Sigma$, $strings(\Sigma') \subseteq strings(\Sigma)$ by construction. Suppose

$$strings(\Sigma) \setminus strings(\Sigma') \neq \emptyset$$

Then there must exist some $(t, i, j) \in \Sigma$ not in Σ' such that

$$(t_0, 0, i_0), (t_1, i_0, i_1), \ldots, (t_i, i_{i-1}, i), (t, i, j), (t_j, j, i_j), \ldots, (t_m, i_m, m) \in \Sigma$$

By construction, $(t_0, 0, i_0) \in \Sigma^1$ and therefore

$$(t_0, 0, i_0), (t_1, i_0, i_1), \ldots, (t_i, i_{i-1}, i), (t, i, j), (t_j, j, i_j), \ldots, (t_m, i_m, m) \in \Sigma^1$$

As $(t_m, i_m, m) \in \Sigma^1$, $(t_m, i_m, m) \in \Sigma'$, and so

$$(t_0, 0, i_0), (t_1, i_0, i_1), \ldots, (t_i, i_{i-1}, i), (t, i, j), (t_j, j, i_j), \ldots, (t_m, i_m, m) \in \Sigma'$$

Therefore $(t, i, j) \in \Sigma'$, a contradiction. $strings(\Sigma) \setminus strings(\Sigma') = \emptyset$ and $strings(\Sigma) = strings(\Sigma')$. □

2.4 Direct TWE Set Construction

In the previous section, the TWE set Σ_X was defined in terms of a corresponding ITS set. However, the main reason for introducing the TWE set representation is to reduce the space requirements from worst case exponential to worst case quadratic. Constructing the ITS sets in the lexical analysis phase would remove this saving. This section instead discusses the direct construction of TWE sets for any given character string. Ultimately, two approaches are considered. The one discussed in this section is based on DFAs and assumes that the patterns of tokens can be defined using regular expressions. The other approach uses a form of GLL recogniser which can be constructed from EBNF grammars, and will be discussed in Chapter 5.

2.4.1 Finite-state Machine Approach

The algorithm considered here is a simple extension of an algorithm used in typical lexical analysers, such as lex [LS75]. In a traditional lexical analyser, the set of tokens, T, would initially be sorted into a list, in order of priority. Each $t \in T$ has a corresponding deterministic finite state automaton [RS59], DFA_t, exactly matching the lexemes of t. The algorithm tries to match the character string, I, at the current character index, starting at 0, to find the first token whose DFA matches a prefix of the string starting at this index. Each DFA attempts to find the longest string from this index that allows the DFA to transition to an accepting state. Each time such a string is found, the current character index advances to the index one passed the portion of string matched, and repeats until the end of the string is reached. The ordering of the

tokens determines priority as only the first token matched is returned, and advancing the current index for the longest matched string creates a longest match for the entire string.

For an algorithm constructing TWE sets, a few aspects need to be different. The first is that a triple is added to the resulting TWE set every time an accepting state is reached. When a token match is made, all other tokens must be considered for the same start index, and the indices following every matched strings must be considered. In the algorithm below, $I[k]$ returns the kth symbol in the input, I. c_I denotes the current character index. Σ is the TWE set being constructed. Q is a set of states to be processed, whilst U keeps track of the states that have been processed.

```
Σ := ∅; Q := ∅; U := ∅
Add 0 to Q
while Q is not empty do {
    Remove i from Q
    c_I := i
    for all t ∈ T do {
        s := 0; k := c_I;                    ▷ Let s denote the current state in DFA_t
        while there exists a transition from s on I[k] to some state p do {
            s := p; k := k + 1;
            if s is an accepting state then {
                Add (t, c_I, k) to Σ
                if k ≠ m ∧ k ∉ U then add k to Q and U
            }
        }
    }
}
Return Σ
```

The first character index to be processed is 0. This is popped from Q and set as the current character index, c_I. For each $t \in T$, the string starting at the current index is given as input to DFA_t. The current character index is kept as the start index and this character index is given to a counter, k. As long as a transition can be made on $I[k]$ in DFA_t, the transition is made. If, after a transition, DFA_t is in an accepting state, a new triple (t, c_I, k) is added to Σ. If k is not the end of the string and is not already in U, then it is added to Q and U. The addition of k to Q adds it to the set of indices to process, whilst the addition to U ensures that the same k cannot be processed more than once (as the same k could be reached as a match by other tokens). Transitions for the current DFA_t continue until no further transitions can be made. This continues for the same starting index, c_I, for every token. After all tokens have been processed, a new c_I is taken from Q and the same process occurs. This continues until $Q = \emptyset$.

The result of this algorithm will be a TWE set, Σ. Because only character indices that are either 0 or the right extent of some match are considered in each iteration, if the height of Σ is m then I is accepted by the lexical analyser. If I is accepted then Σ will be tight.

2.5 Lexical Disambiguation

The main motivation for the work in this chapter is to give a language designer more control over the tokenisations which are chosen from the set of possibilities within a specified lexer scheme. A lexical ambiguity occurs when there exists more than one indexed token string for the same input character string. Many of the tokenisations will not be syntactically correct, that is they will not be derivable in the grammars used to specify the language. However, parsing each tokenisation is likely to be slower than eliminating many of the tokenisations before parsing. This section looks at developing disambiguation rules which can be used to reduce the amount of lexical ambiguity in the TWE set which is passed to the parser. The focus will be mainly on formalising and discussing variations of the frequently used longest match and priority disambiguation rules. The starting point is the full set, X_γ, of tokenisations of a character string γ and its corresponding TWE set Σ_γ. However, the discussion applies if X_γ is replaced with any consistent ITS set (as defined on page 31).

2.5.1 Lexical Ambiguity Reduction and TWE sets

Lexical ambiguity reduction removes some indexed token sequences from the ITS set of a given character string. However, formally the ambiguity reduction rules will be defined in terms of removing triples from a TWE set - which is not entirely equivalent to ITS set element removal.

Removing sequences from an ITS set may make it inconsistent. As demonstrated in Lemma 2.3.3, the corresponding TWE set will embed the smallest enclosing consistent set, and so some removed sequences will be re-instated. Recall from page 29 the ITS sets X_1 and X_2

$$\begin{aligned} X_1 = \{&(\mathbf{D},1)(\mathbf{D},2)(\mathbf{D},3)(\mathbf{D},4),\\ &(\mathbf{D},1)(\mathbf{D},2)(\mathbf{D},4),\\ &(\mathbf{D},2)(\mathbf{D},3)(\mathbf{D},4),\\ &(\mathbf{D},2)(\mathbf{D},4)\} \end{aligned}$$

$$X_2 = \{(\mathbf{D},1)(\mathbf{D},2)(\mathbf{D},3)(\mathbf{D},4)$$
$$(\mathbf{D},2)(\mathbf{D},4)\}$$

where X_2 is a proper subset of X_1 obtained by removing particular sequences. The corresponding TWE set for X_2 is the TWE set for X_1 - the tokenisations removed from X_1 are reinstated. Thus, it is not possible to model ambiguity reduction from X_1 to X_2 using TWE set element removal.

Removing a triple from a TWE set will remove all strings containing that triple embedded in the set. For instance, it is not possible to remove just the string $(\mathbf{D},2)(\mathbf{D},4)$, say, from X_1 since removing the triple $(\mathbf{D},0,2)$ also removes the string $(\mathbf{D},2)(\mathbf{D},3)(\mathbf{D},4)$.

A TWE set, Σ, embeds an ITS

$$(t_1,i_1)(t_2,i_2)\ldots(t_p,i_p)$$

if and only if there exists exactly one sequence of triples in the TWE set

$$(t_1,0,i_1)(t_2,i_1,i_2)\ldots(t_p,i_{p-1},i_p)$$

corresponding to this embedded ITS. Removing any one triple from this sequence will remove this ITS from $\mathit{strings}(\Sigma)$. In the case where the ITS set contains only one element, then the corresponding TWE set will contain only triples for this exact ITS. Likewise, if all but a single sequence of triples of the form above is removed from the TWE set, then the TWE set will only embed one ITS. So, in the case that the lexical disambiguation is complete, lexical ambiguity removal at either the level of the ITS set or the TWE set will give the same result. As the ITS set is potentially exponential in size whereas the TWE set is at worst quadratic, it is more practical to disambiguate via TWE set triple removal - and this is the approach that will be used, even though there are sometimes outcomes which cannot be obtained. Thus, sequences from $\mathit{strings}(\Sigma)$ are removed to produce a new set $\mathit{strings}(\Sigma')$ by removing triples from Σ to produce the subset Σ'. As discussed in Lemma 2.3.3, by construction $\mathit{strings}(\Sigma')$ will be consistent, and by pruning the set after disambiguation, Σ' will be tight, as demonstrated by Lemma 2.3.4.

Lexical ambiguity in a tight TWE set Σ means there exist two or more distinct triples that share the same left extent, as in $(a,i,j),(b,i,k) \in \Sigma$, or equivalently that there exist two distinct triples which share the same right extent. In terms of the graphical representations, the former focuses on nodes that have more than one out-edge, whilst the latter focuses on nodes that have more than one in-edge. Three classes of disambiguation rules for both cases are specified, which are then related to the

classical notions of longest match and priority.

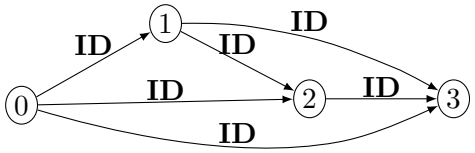
2.5.2 Priority and Longest Match

Firstly, consider the longest match lexical disambiguation strategy that is very often applied to identifiers.

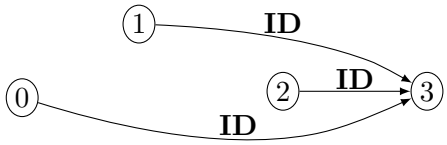
Assuming a token **ID** whose pattern is the standard C-style identifiers, the string `aaa` will have the following ITS set

$$\{(\mathbf{ID},1)(\mathbf{ID},2)(\mathbf{ID},3),\quad (\mathbf{ID},2)(\mathbf{ID},3),\quad (\mathbf{ID},1)(\mathbf{ID},3),\quad (\mathbf{ID},3)\}$$

This has the corresponding TWE graph



It should be apparent that to emulate classical longest match, it is enough to consider, for each node, all **ID** labelled out-edges removing all but the one with the greatest right extent. For the above, this results in

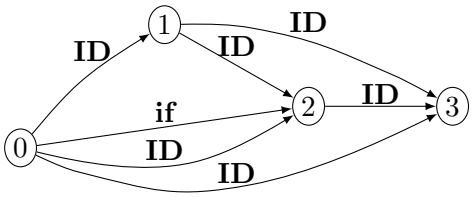


Of course, the TWE set that this graph represents is not tight. However, after pruning, the resulting set is $\Sigma' = \{(\mathbf{ID},0,3)\}$ and $strings(\Sigma') = \{(\mathbf{ID},3)\}$ - exactly the tokenisation expected from traditional longest match.

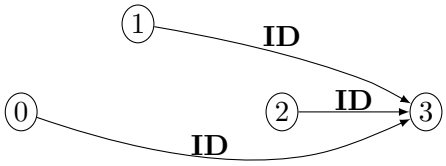
One can consider different ways in which longest match is applied. Longest match could apply across all tokens (a global longest match), or only across a restricted group of tokens. For example, in C-style languages, there exists a token $(\mathbf{if},\{\texttt{if}\})$. Consider the character string `ifa`. This will have the ITS set

$$\{(\mathbf{ID},1)(\mathbf{ID},2)(\mathbf{ID},3),\quad (\mathbf{ID},2)(\mathbf{ID},3),\quad (\mathbf{ID},3),\quad (\mathbf{if},2)(\mathbf{ID},3)\}$$

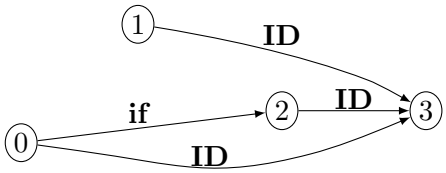
with corresponding TWE graph



If one were to use 'global' longest match the resulting TWE graph would be



After pruning, this would return just the indexed token string $(\mathbf{ID}, 3)$. Alternatively, one can restrict longest match to only apply in the case of **ID**. In this case, the resulting TWE graph would be

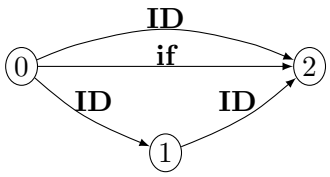


This returns two indexed token strings: $(\mathbf{ID}, 3)$ and $(\mathbf{if}, 2)(\mathbf{ID}, 3)$.

In many languages, keywords and identifiers are distinguished, with keywords taking priority over identifiers. The character string `if` will have the ITS set

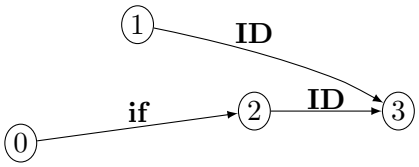
$$\{\ (\mathbf{ID}, 1)(\mathbf{ID}, 2), \quad (\mathbf{ID}, 2), \quad (\mathbf{if}, 2)\ \}$$

which has the corresponding TWE graph



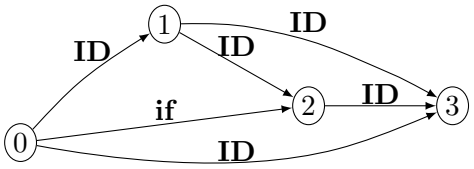
To eliminate the edge from node 0 to node 2 labelled **ID** and thus prioritise **if**, it is enough to specify that an edge labelled **ID** be deleted if there exists another out-edge from the same node, labelled **if**.

However, if one were to apply this priority rule exactly to the `ifa` example then the resulting TWE graph would be

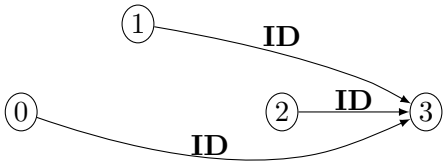


Here, the tokenisation $(\texttt{ID}, 3)$ is desired. Priority as described above would remove the edge from 0 to 3 labelled **ID** as there exists an edge from 0 to 2 labelled **if**.

The alternative version of priority is that an edge labelled **ID** should be removed in the presence of an edge labelled **if** only if they share the same start and end nodes. For the above example, applying this gives the TWE graph

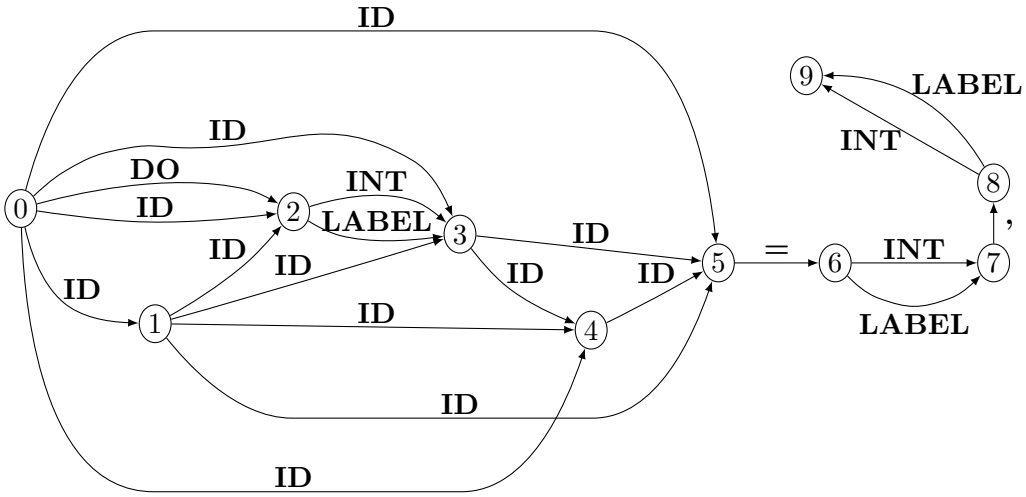


Applying longest match across all tokens then returns the TWE graph

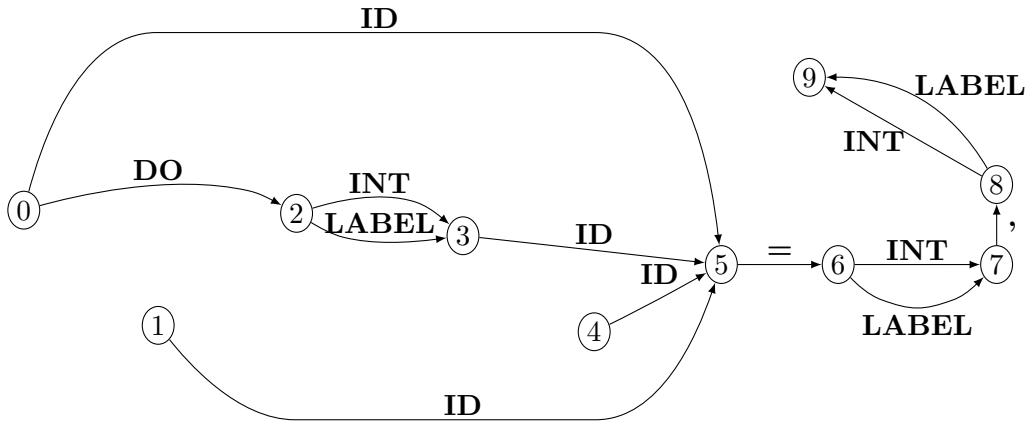


Giving the tokenisation $(\textbf{ID}, 3)$ as required, whilst still preserving $(\textbf{if}, 2)$ in the `if` character string example.

An interesting case is seen in early versions of FORTRAN [IA78] - which was designed to be completely whitespace independent. A character string `DO5AB=1,6` has the corresponding TWE graph



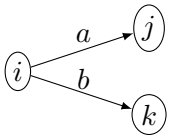
Suppose one would like an implementation that always prioritises tokenisations with `DO` as a keyword over ones with `DO` as a prefix of **ID**. Since longest match should still apply to **ID**, a first step would be to restrict longest match to only be applicable just across a particular token. Longest match just on **ID** will give the TWE graph



This will still leave tokenisations that match `DO5AB` as an **ID**. By specifying that **DO** should always have priority over **ID** regardless of the right-extent, these tokenisations will be eliminated.

2.5.3 Left-extent Pair-wise Operations

The previous section described specific ways in which a pair-wise comparison on triples with the same left-extent could be used for disambiguation. This section will describe a more general solution that can capture all these cases. Consider operations between (a, i, j) and (b, i, k). The graph visualisation for these two triples is



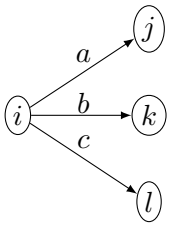
A formal expression of the comparisons in the previous examples is achieved by defining relations over the set of tokens and their extents. A binary relation, R, over the token set is defined as aRb if a can be compared to b. This relation need not be symmetric - aRb does not imply bRa. It also need not be transitive - aRb and bRc does not imply aRc. In some cases, R may be reflexive, so aRa.

Given relations R for two triples (a, i, j) and (b, i, k), classes of removal specifications can be defined by the way in which j and k are compared

- (Class 1) Remove (b, i, k) if bRa and $j = k$

- (Class 2) Remove (b, i, k) if bRa and $j > k$
- (Class 3) Remove (b, i, k) if bRa and $j < k$

As it stands, application of two or more removal decision operations can lead to non-confluence issues. Consider $(a, i, j), (b, i, k), (c, i, l) \in \Sigma$ whose graph visualisation is



Potential pair-wise removal decisions are between (a, i, j) and (b, i, k), between (a, i, j) and (c, i, l), or between (b, i, k) and (c, i, l). If a decision between (a, i, j) and (b, i, k) results in (b, i, k) being removed, then a decision between (b, i, k) and (c, i, l) no longer occurs. If a decision between (b, i, k) and (c, i, l) results in (b, i, k) being removed, then a decision between (a, i, j) and (b, i, k) no longer occurs.

A solution is to only mark a triple for removal instead of actually removing the triple, and applying the removal decisions with marked as well as unmarked triples. After all removal decisions are applied, a triple is removed from Σ if it is marked for removal. The three classes of removal specifications on (a, i, j) and (b, i, k) are then as follows:

- (Class 1) Mark (b, i, k) for removal if bRa and $j = k$
- (Class 2) Mark (b, i, k) for removal if bRa and $j > k$
- (Class 3) Mark (b, i, k) for removal if bRa and $j < k$

In the above case, an operation between (a, i, j) and (b, i, k) may mark (b, i, k) for removal. However, the operation between (b, i, k) and (c, i, l) will still occur, regardless of the state of (b, i, k). If (c, i, l) is marked for removal as a result of the operation, then this simply means that (a, i, j) is the only triple left unmarked. Of course, this has the potential to lead to all triples being marked for removal - if (b, i, k) and (c, i, l) are marked for removal, an operation between (a, i, j) and (c, i, l) could still mark (a, i, j) for removal as well.

Classes 1, 2, and 3 allow the specification of a large variety of operations. Relations will be represented as a matrix of size $|T| \times |T|$ (where T is the set of tokens). Rows in the matrix represent the left operand of R, whilst columns represent the right operand.

The disambiguation rules only mark for removal edges labelled with the left operand, so $aRb, <=$ is not the same as $bRa, >$. In the relation matrix M_R, $M_R[a,b] = 1$ if aRb and is 0 otherwise.

It is worth considering whether any of these classes of operations are redundant. Two ambiguity reduction relations are said to be equivalent if, for all possible TWE sets, applying either of the relations results in the same set. Two classes of ambiguity reduction operations are equivalent if every relation of one class has an equivalent relation in the other class. As the following lemma shows, this is not the case for these three classes.

Lemma 2.5.1. *For the given definition of what is required for two classes of ambiguity reduction operations to be equivalent:*

a) Class 1 and Class 2 are not equivalent.

b) Class 1 and Class 3 are not equivalent.

c) Class 2 and Class 3 are not equivalent.

Proof. This can be shown by example. If R is not empty then there is a relation aRb. Consider the TWE set $\{(a,0,1),(b,0,1)\}$. A class 1 relation marks $(a,0,1)$ for removal but since the right extents are the same, no class 2 or class 3 relation can result in $(a,0,1)$ being marked for removal, proving both (a) and (b).

Now suppose that R is any nonempty relation with aRb. For the TWE set $\{(a,0,1),$ $(b,0,2)\}$, a class 2 relation for R marks $(a,0,1)$ for removal but there is no class 3 relation which marks $(a,0,1)$ for removal, proving (c). □

Whilst the classes of operations are not redundant, they can still overlap. Consider the TWE set

$$\{(a,0,1),(b,0,1),(c,0,2),(d,1,3),(d,2,3)\}$$

A class 1 relation bR_1a would result in the TWE set $\{(a,0,1),(c,0,2),(d,1,3),(d,2,3)\}$ as would a class 2 relation bR_2c.

The examples in the previous sub-section are now returned to, in order to demonstrate how these mechanisms can achieve the desired results in practice. In these examples, each relation will be represented by a matrix in the form described above.

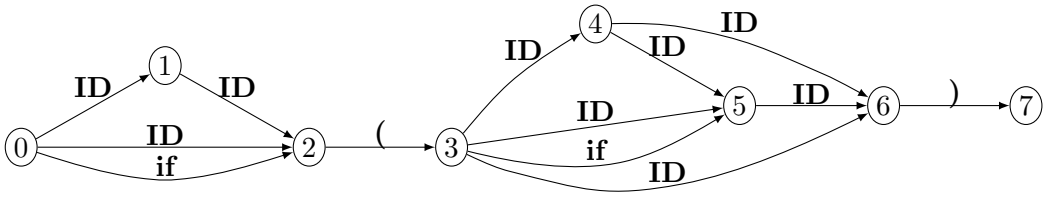
To define the longest match rules for **if** and **ID** for C-style languages one can use a class 2 rule with

$$R_2 = \begin{array}{c} \\ \textbf{if} \\ \textbf{ID} \end{array} \begin{array}{c} \begin{array}{cc} \textbf{if} & \textbf{ID} \end{array} \\ \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix} \end{array}$$

and to define the priority of **if** over **ID** one can use a class 1 rule with

$$R_1 = \begin{array}{c} \\ \textbf{if} \\ \textbf{ID} \end{array} \begin{array}{c} \begin{array}{cc} \textbf{if} & \textbf{ID} \end{array} \\ \begin{pmatrix} 0 & 0 \\ 1 & 0 \end{pmatrix} \end{array}$$

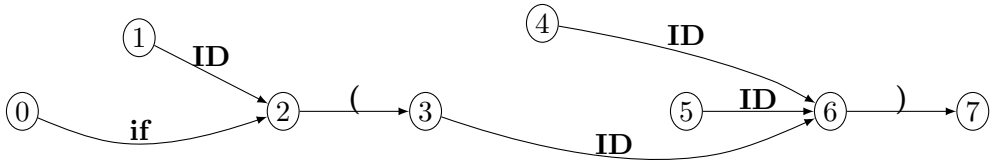
For example, for a character string `if(ifx)`, the TWE graph is



Using the class 1 relation, $(\mathbf{ID}, 0, 2)$ is marked for removal since $(\mathbf{if}, 0, 2)$ exists and $\mathbf{ID} R_1 \mathbf{if}$. Similarly $(\mathbf{ID}, 3, 5)$ is marked for removal.

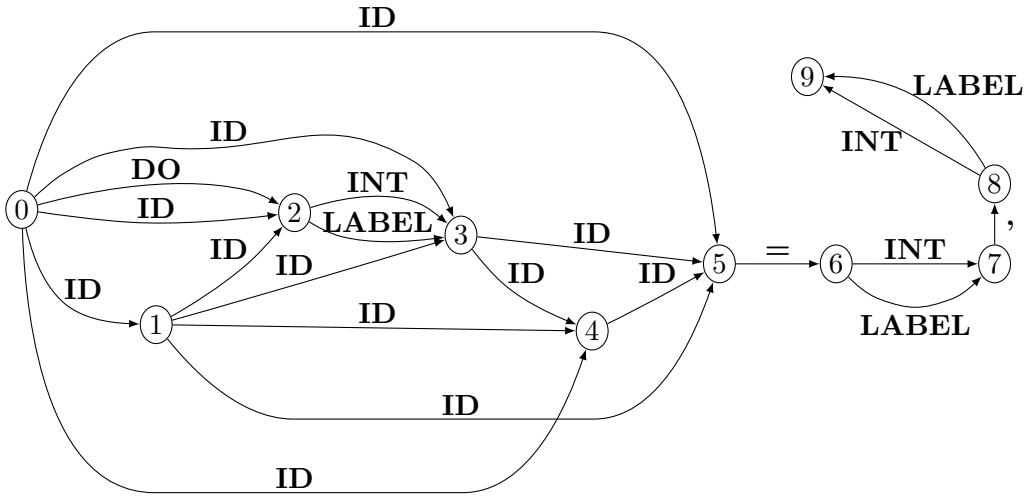
Using the class 2 relation, $(\mathbf{ID}, 0, 1)$ is marked for removal as $(\mathbf{ID}, 0, 2)$ exists and $\mathbf{ID} R_2 \mathbf{ID}$. Similarly, $(\mathbf{ID}, 3, 4)$, $(\mathbf{ID}, 3, 5)$, and $(\mathbf{ID}, 4, 5)$ are also marked for removal. $(\mathbf{if}, 3, 5)$ is marked for removal as $(\mathbf{ID}, 3, 6)$ exists and $\mathbf{if} R_2 \mathbf{ID}$.

After all marked triples are removed, the TWE graph is



Pruning this gives the final ITS $(\mathbf{if}, 2)((, 3)(\mathbf{ID}, 6)(), 7)$.

For the FORTRAN example, the TWE graph for the string `DO5AB=1,6` is



If one uses a class 1 rule with relation

$$R_1 = \begin{array}{c} \\ \textbf{DO} \\ \textbf{ID} \end{array} \begin{array}{c} \begin{array}{cc} \textbf{DO} & \textbf{ID} \end{array} \\ \begin{pmatrix} 0 & 0 \\ 1 & 0 \end{pmatrix} \end{array}$$

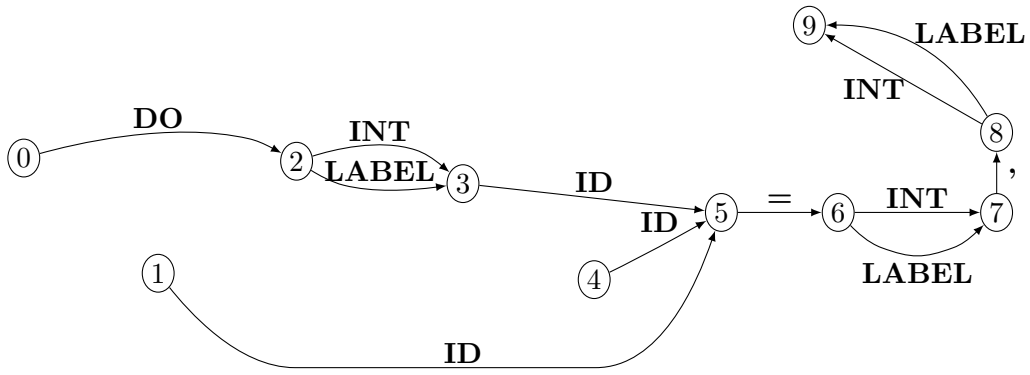
A class 2 rule with relation

$$R_2 = \begin{array}{c} \\ \textbf{DO} \\ \textbf{ID} \end{array} \begin{array}{c} \begin{array}{cc} \textbf{DO} & \textbf{ID} \end{array} \\ \begin{pmatrix} 0 & 0 \\ 1 & 1 \end{pmatrix} \end{array}$$

and a class 3 rule with relation

$$R_3 = \begin{array}{c} \\ \textbf{DO} \\ \textbf{ID} \end{array} \begin{array}{c} \begin{array}{cc} \textbf{DO} & \textbf{ID} \end{array} \\ \begin{pmatrix} 0 & 0 \\ 1 & 0 \end{pmatrix} \end{array}$$

applying the rules gives the TWE graph



which embeds the ITS set

$$\{(\textbf{DO},2)(\textbf{INT},3)(\textbf{ID},5)(=,6)(\textbf{INT},7)(\textbf{,},8)(\textbf{INT},9),$$
$$(\textbf{DO},2)(\textbf{INT},3)(\textbf{ID},5)(=,6)(\textbf{INT},7)(\textbf{,},8)(\textbf{LABEL},9),$$
$$(\textbf{DO},2)(\textbf{INT},3)(\textbf{ID},5)(=,6)(\textbf{LABEL},7)(\textbf{,},8)(\textbf{INT},9),$$
$$(\textbf{DO},2)(\textbf{INT},3)(\textbf{ID},5)(=,6)(\textbf{LABEL},7)(\textbf{,},8)(\textbf{LABEL},9),$$
$$(\textbf{DO},2)(\textbf{LABEL},3)(\textbf{ID},5)(=,6)(\textbf{INT},7)(\textbf{,},8)(\textbf{INT},9),$$
$$(\textbf{DO},2)(\textbf{LABEL},3)(\textbf{ID},5)(=,6)(\textbf{INT},7)(\textbf{,},8)(\textbf{LABEL},9),$$
$$(\textbf{DO},2)(\textbf{LABEL},3)(\textbf{ID},5)(=,6)(\textbf{LABEL},7)(\textbf{,},8)(\textbf{INT},9),$$
$$(\textbf{DO},2)(\textbf{LABEL},3)(\textbf{ID},5)(=,6)(\textbf{INT},7)(\textbf{,},8)(\textbf{LABEL},9)\}$$

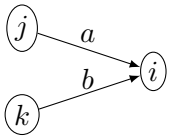
Defining **ID** R **DO** for all three classes of operations is in fact equivalent to the unrestricted version of priority necessary for this particular case.

Although the class 1-3 relations are adequate for handling all the cases seen in the traditional use of a lexer, the aim of this thesis is to expand the options available to a language designer. One may wish to consider other ambiguity reductions that are not so easily modelled with the traditional understanding of priority and longest match. Whilst the `DO5AB=1,6` FORTRAN example has been described in a scenario where the language designer would always want the keyword interpretation of `DO`, this does not match what is required by FORTRAN. The string `DO5AB=1.6` should be interpreted as the assignment of a real value to the identifier `DO5AB`. The only way to determine whether `DO` should be tokenised as a keyword or as the prefix of an identifier is to perform lookahead, choosing the former tokenisation only if there is a comma present. Whilst this will not be explored here, lexical ambiguity reduction using lookahead is one option that can be considered.

Another additional set of operations, which will be considered, is right-extent pair-wise operations.

2.5.4 Right-extent Pair-wise Operations

Rather than left-extent pair-wise operations, consider right-extent pair-wise operations. Let $r = (a, j, i)$ and $s = (b, k, i)$. The graph visualisation for these two triples is



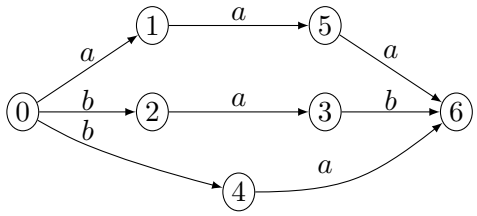
The classes of pair-wise operations are equivalent to that of common left-extent:

- (Class 1a) Mark (b, k, i) for removal if bRa and $j = k$
- (Class 2a) Mark (b, k, i) for removal if bRa and $j < k$
- (Class 3a) Mark (b, k, i) for removal if bRa and $j > k$

It should be intuitively obvious that a class 1a ambiguity rule for common right extent is semantically equivalent to a class 1 rule for common left extent. However, class 2a and class 3a are not equivalent to class 2 and class 3, and they are not redundant. Consider the TWE set

$$\{(a, 0, 1), (b, 0, 2), (b, 0, 4), (a, 1, 5), (a, 2, 3), (b, 3, 6), (a, 4, 6), (a, 5, 6)\}$$

with the TWE graph



The corresponding ITS set cannot be reduced down to just the string $(a, 0, 2)(a, 2, 3)$ $(b, 3, 6)$ using any of classes 1-3. There is no class 1 relation that can remove any triple. The only class two relation that can remove a triple is aRa which would remove $(a, 0, 2)$ (in favour of $(a, 0, 4)$), and the only class three relation that can remove a triple is aRa which would also remove $(a, 0, 2)$ (in favour of $(a, 0, 1)$). However class 2a relations on aRa and aRb will remove $(a, 5, 6)$, and $(a, 4, 6)$, leaving the single ITS $(a, 0, 2)(a, 2, 3)(b, 3, 6)$, as required.

An issue that affects both variants of pair-wise operations is the possibility that when presented a choice, all choices are marked for removal. For example if there was a relation aRb under the class 2 relation, and a relation $bR'a$ under the class 3 relation, then in all cases where the right extents are different, both (a, i, j) and (b, i, k) would be marked for removal.

2.6 Token suppression

A common aspect of traditional lexical analysis that has not yet been considered is the suppression of layout tokens (used here to refer to both whitespace and comments). For languages where layout is not semantically significant, it is usual for the lexical analyser to suppress tokens corresponding to layout before the token string is given as input to the parser. This is conceptually straightforward to apply in the multilexer approach, provided that matches to tokens that are suppressed do not overlap other token matches.

A token t is said to be *left separated* in a token set if, for every lexeme, a_k, in the pattern set of t, the only token in the token set containing a lexeme of the form μa_k (where μ is a string of characters) is t. Similarly, t is said to be *right separated* in a token set if, for every lexeme, a_k, in the pattern set of t, the only token in the token set containing a lexeme of the form $a_k\mu$ is t. If t is both left and right separated, then t is said to be *separated* in a token set.

Separated tokens can be suppressed from a constructed TWE set without a problem. From the perspective of an indexed token sequence, this is straightforward - simply

removing a pair from an indexed token sequence

$$(t_1, j_1) \ldots (t_p, i_p)(t, j)(t_q, i_q) \ldots (t_h, i_h)$$

gives a new token sequence

$$(t_1, j_1) \ldots (t_p, i_p)(t_q, i_q) \ldots (t_h, i_h)$$

whose relationship to the original is clear. In effect, the pattern match for t is 'merged' onto the front of the match made by t_q. If t occurs at the end of the sequence, then the height of the sequence will be changed. However, since t is separated in the token set, this will apply to all sequences in a set of tokenisations of a given character string, preserving consistency.

As discussed, it is often more practical to operate on TWE sets. For every indexed token sequence of form

$$(t_1, j_1) \ldots (t_p, i_p)(t, j)(t_q, i_q) \ldots (t_h, i_h)$$

in an ITS set X, the TWE set Σ_X will contain a triple of form (t, j_p, j). Given t is separated, to remove (t, j) it is enough to remove the triple (t, j_p, j) from Σ_X, and replace all triples of form (s, j, j_l) with a triple of form (s, j_p, j_l).

Where layout is separated, this approach is sufficient. However, even for C-style languages, comments are not separated in the above sense. Chapter 5 will consider combining token suppression with an initial processing stage - an approach that is sufficient for the C# case studies in Chapter 6.

Of course, layout suppression is not adequate in general. For example, in embedded languages some patterns may be whitespace in some contexts but not others. The multilexer parser approach gives the potential for dealing with non-separated layout in a formal way, and the issues and potential approaches will be discussed in Chapter 5 - although the full study is not yet complete.

2.7 Related Work

Whilst most current compilers use the traditional lexical analysis approach described in 2.1, there have been previous proposals for how one could define lexical specifications that output tokenisations other than the one obtained through longest match and priority. This section will briefly consider these other proposals in relation to this work.

2.7.1 Lex/Flex

Whilst lex (and its free software counterpart, flex, with the two together being referred to as (f)lex here) is a canonical example of a traditional lexical analyser, it includes some functionality that allows lexical ambiguities to be handled in a way that is more extensive than longest match and priority. The functionality that will be considered here is the REJECT action.

In normal usage, (f)lex will behave like a traditional lexical analyser - selecting the lexeme to give the tokenisation that is the longest match, with priority over the tokens for cases where the lexemes have the same length (specified simply according to how the tokens are ordered in the specification). However, if a rule is given the semantic action of REJECT, then any semantic action for the match will be executed, and the match is rejected, backtracking to the next match. For example, the lex specification

```
%%
forbid    REJECT;
for       printf("for");
bid       printf("bid");
```

would, for the string `forbid`, reject the tokenisation `forbid`, but accept the tokenisation `for bid`.

Although the documentation for (f)lex recommends against using the REJECT action for string partitioning, as (f)lex allows any semantic action as a side effect of a match, it is possible to construct a full TWE set with this action. By specifying that individual characters should be skipped (the symbol, `.`, is used to represent any character here), and assigning REJECT after a `printf` statement for every pattern match, one could construct the following lex specification

```
%%
int i=0;

aaa     printf("(X,%d,%d)",i,i+yyleng); REJECT;
aa      printf("(Y,%d,%d)",i,i+yyleng); REJECT;
a       printf("(X,%d,%d)",i,i+yyleng); printf("(Y,%d,%d)",i,yyleng); REJECT;
.       i=i+1;
```

The left extent is tracked by `i`, and `yyleng` returns the length of the token match. For the string `aaa`, this would produce the TWE set,

$$\{(\mathbf{X},0,3),(\mathbf{Y},0,2),(\mathbf{X},0,1),(\mathbf{Y},0,1),(\mathbf{Y},1,3),(\mathbf{X},1,2),(\mathbf{Y},1,2),(\mathbf{X},2,3),(\mathbf{Y},2,3)\}$$

Of course, this is an inelegant solution, as the TWE set is constructed through the explicit rejection of all matches. This operation requires backtracking, which is

costly. This is also much less efficient than the algorithm in 2.4, as every possible token match, at every possible character input position, must be tried. The resulting set will, in general, not be tight.

2.7.2 Schrödinger's Tokens

Aycock and Horspool [AH01] developed a concept for allowing multiple tokens to match the same lexeme. The idea is to introduce a new kind of token, which can simultaneously represent more than one token in the lexer specification, referred to as a Schrödinger's token. This presents a technique which is relatively simple in terms of concept.

This approach produces a single tokenisation. However, when a lexeme can be matched by more than one token, a special token is used that represents the union of these tokens, denoted $\bar{s}(t_1, \ldots, t_p)$, where $t_1, \ldots, t_p$ are the actual tokens matched. For example, if one considers the lexical specification

$$\mathbf{X} = \{a, bx\}$$
$$\mathbf{Y} = \{a, cy\}$$

For the string `abx`, the resulting tokenisation is

$$\bar{s}(\mathbf{X}, Y)X$$

This tokenisation simultaneously represents both tokenisations of the string, with the token given for a representing both $\mathbf{X}$ and $\mathbf{Y}$. The modifications required to allow a parser to take such a tokenisation as input are relatively straightforward, and are, in fact, conceptually, the same as those needed in the multilexer approach - which will be discussed in Chapter 3. In essence, all that is needed is to modify the parsing algorithm to repeat actions, where relevant, for all the tokens that the Schrödinger's token represents, rather than just the single token

Although the specific intention was to handle the case where two or more tokens can match the same lexeme, the paper sketches one way to handle the case where one token matches a lexeme that is the prefix of the lexeme of another token match. The suggestion was to pad the shorter tokenisation with special 'null' tokens. For example, in the Java nested parametrisation case described at the start of this chapter, the string `List<List<Integer>>` would result in, assuming longest match on **`ID`**, the tokenisation

$$\texttt{ID<ID<ID}\ \bar{s}(\texttt{>>}, \texttt{>})\ \bar{s}(null, \texttt{>})$$

In this case, whilst the parser would fail for `>>`, it would succeed on `>`. The null token would then be ignored and `>` read as the next input.

However, this approach has drawbacks. Where the longer match is required, it is necessary to modify the parsing grammar to include these 'null' tokens. Whilst the Schrödinger's token approach is effective for when the language designer only needs multiple tokenisations when a single lexeme is matched by more than one token, the multilexer approach offers a more general solution for providing all possible tokenisations.

2.7.3 Character-Level Parsing

An approach described by Visser [Vis97], used in parser generators including ASF+SDF [Bra+01], Spoofax [KV10], and Rascal [KSV11], is the use of character-level parsing. This approach removes the formal distinction between lexical and syntactic analysis. A character-level grammar is one in which the set of terminals is simply the set of characters. A is a set of tokens whose patterns contain a single element which is a single character string. A more detailed comparison between multilexer parsing and character-level parsing will be given in Chapter 5, but a brief description of the approach will be given here.

Consider a lexical specification

$$\mathbf{x} = \{b, c, bc\}$$
$$\mathbf{y} = \{d\}$$

This specification contains a lexical ambiguity as, for instance, the string `bc` can be tokenised as both $\mathbf{x}$ and $\mathbf{xx}$. Consider the following token-level parsing grammar which has $\mathbf{x}$ and $\mathbf{y}$ as terminals

$$S ::= A\,\mathbf{y}$$
$$A ::= \mathbf{x}\,A \mid \mathbf{x}$$

In a character-level parser, an equivalent grammar is

$$S ::= A\,Y$$
$$A ::= X\,A \mid X$$
$$X ::= \mathtt{b} \mid \mathtt{c} \mid \mathtt{b}\,\mathtt{c}$$
$$Y ::= \mathtt{d}$$

A parse of the string `bcd` on the character-level grammar would result in the SPPF in Figure 2.1. This parse effectively captures both tokenisations of the string, with the non-terminals X and Y being used to represent the tokens `x` and `y` that would exist in the token-level grammar. The lexical ambiguity becomes a syntactic ambiguity in the rules for A, and the ambiguity needs to be resolved using syntactic disambiguation mechanisms. At a syntactic level, the concepts of longest match and priority become more difficult to express. The approach used in tools such as ASF+SDF is to specify restrictions on the characters that may follow a non-terminal symbol. A follow restriction declares that if a particular non-terminal is matched, and the character that occurs next in the input is in the set of characters that may not follow that non-terminal, then the match is rejected. This is usually given in the form

$$X - / - G$$

where X is a non-terminal and G is the set of characters that may not follow X. In the above example, the follow restriction

$$X - / - \texttt{c}$$

would state that X may not be followed by the character `c`. [Vis97] observes that this approach can be used, in most cases, in place of an equivalent longest match technique. For the string `bcd`, the derivation in which one X matches just `b` will be rejected, as this X will be followed by `c`. This would then prune the branch of the SPPF rooted at the packed node $(A ::= X\ A\cdot, 1)$ as required, leaving the branch where X is the longest match.

The syntactic disambiguation operation that is used in place of priority is the reject production. A reject production is a grammar rule annotated with `reject`. For a production of form $X\{reject\} ::= \alpha$, if $X \Rightarrow \alpha$ then this derivation is rejected. If one were defining, for example, a character-level parser for C, the production

$$ID\{reject\} ::= \texttt{f o r}$$

would ensure that ID could never match the keyword `for`.

Although these operations can provide functionality that is similar to longest match and priority, it is much harder, conceptually, to understand the effect of a given follow restriction rule. In particular, follow restrictions may be too restrictive in the case

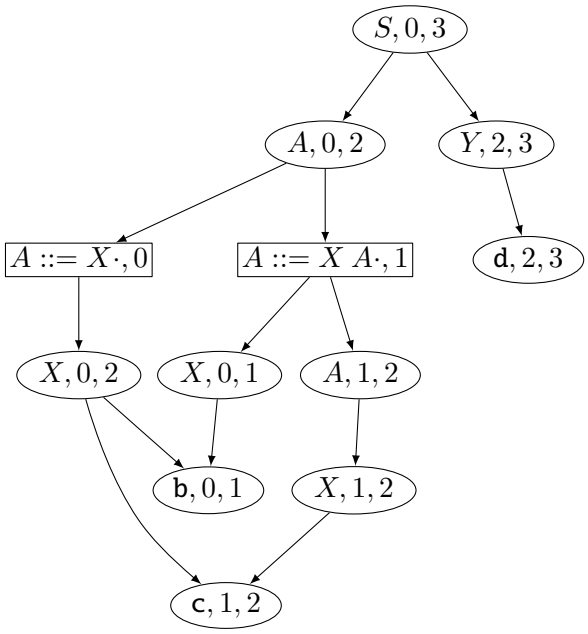


Figure 2.1: Character-level SPPF for the string `bcd`

where two token non-terminals derive similar strings. Consider the following grammar

$$S ::= A\ B \mid A\ A\ B$$
$$A ::= \mathtt{x}\ \mathtt{x} \mid \mathtt{x}$$
$$B ::= \mathtt{x}\ \mathtt{y}$$

For the string `xxxy`, the resulting SPPF is given in Figure 2.2. Ambiguity results as the string `xx` can be matched as AA or A. One may be tempted to use the following restriction to achieve the equivalent of longest match

$$A - / - \mathtt{x}$$

However, this not only eliminates the packed node ($S ::= A\ A\ B\cdot, 2$) but also the other packed node ($S ::= A\ B\cdot, 2$). This is because the third `x` in the string still follows A in both cases.

A more detailed discussion comparing character-level parsing with the multilexer approach will be given in Chapter 5. The key message, though, is that the multilexer approach gives language designers the same level of power and control as a character-level specification, whilst retaining some of the advantages of a token-level specification. The data structures constructed in the multilexer approach are generally smaller than

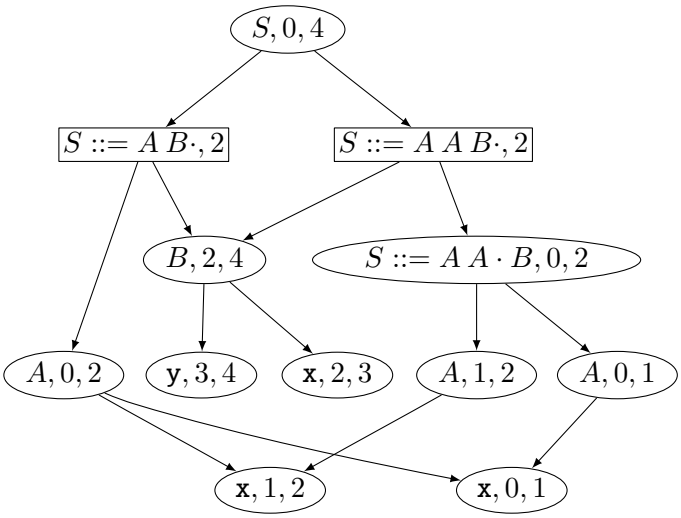


Figure 2.2: Character-level SPPF for the string `xxxy`

those constructed in character-level parsing. The lexical ambiguity reduction techniques presented for the multilexer approach are also a closer match to what one might expect from longest match and priority than follow restrictions and reject productions in character-level parsing.

2.7.4 Other Work

In addition to those already discussed, there are other approaches that should be mentioned. One approach is to use lexical feedback [Ros91], in which the parser shares context obtained during parsing to inform the tokenisation process. This requires the lexical and syntactic analysis to occur in lock-step. This is used to handle issues such as the ANSI C type/variable name conflict, which will be discussed in more detail towards the end of Chapter 5. The multilexer approach has resonances with other attempts made in the field of computational linguistics, in particular, [DFH04] and [CT96].

Chapter 3

Multiple input parsing with MGLL

Chapter 2 described a new multilexing approach that allows a language designer more flexibility and control over lexical disambiguation. However, there are some lexical choices that should be made in the syntactic context in which the token appears. Suppose there is a language which, conceptually, is the union of two languages, for example, COBOL and SQL. Then certain sequences of characters correspond to keywords when appearing in an SQL statement but as identifiers in a COBOL statement. Of course, a solution is to parse all the alternative tokenisations and reject those which are not syntactically correct. The lexical approach that was developed in the previous chapter is designed precisely to support this. However, simply parsing each token string independently is likely to be inefficient.

As already mentioned, it is possible to use a character-level grammar to specify the lexical and phrase level syntax of a language. Then input analysis can be carried out in worst case cubic time in the length of the input character string. However, by adopting the approach discussed in Chapter 2, a language designer can retain the efficient lexeme recognition techniques from traditional lexical analysis, and eliminate many token strings before parsing begins. In fact, the given approach degrades gracefully to the case of a single tokenised string when the design specification permits. Furthermore, parser level performance measures that rely on lookahead are more effective if the lookahead is specified at token rather than character level. Thus, to exploit the multilexer approach, an efficient multiple input parsing technique is required.

In this chapter, a new parsing technique, MGLL, is introduced which can efficiently parse a set of input strings if they are specified as a TWE set. The discussion will begin with parsing in general and the GLL [SJ13] technique on which MGLL is based. The

notion of an extended SPPF structure which embeds derivations of multiple strings will then be defined, and the MGLL technique for constructing such a structure will be given.

3.1 Parsing Preliminaries

Chapter 1 described how a context free grammar Γ defines a language over sentences which can be derived from the start symbol, S. At some point during execution, a compiler is presented with a string of terminals, and it needs to determine whether the string is in the language of Γ, in other words, whether the string can be derived from S. This is referred to as recognition.

In fact, the semantics of a sentence is usually determined, at least in part, by its syntactic structure, so a parser usually constructs some representation of a derivation of the sentence. A parser for a grammar Γ takes as input strings of terminals and produces a representation of one or more derivations of the string. There are many parsing techniques both in the literature and in practice. In this section, the simplest technique, recursive descent, will be described as will its generalisation, GLL, which forms the basis of MGLL.

3.1.1 Recursive Descent Parsing

A recursive descent recogniser for a context-free grammar Γ contains a parse function, parseX, for each nonterminal X in Γ and a main function which calls parseS, where S is the start symbol. The function parseX contains a block of code for each grammar rule for X, and each block is guarded by a test over a lookahead set. The block of code corresponding to a rule $X ::= x_1 \dots x_p$ contains a line for each x_i. If x_i is a terminal then this is matched to the current input symbol, and if x_i is a non-terminal then its parse function is called. The input is assumed to be held in an array I whose last element, $I[m]$, is \$. I thus has length $m + 1$.

Consider a grammar

$$S ::= \mathbf{q}\, S \mid A\, \mathbf{b} \mid B$$
$$A ::= \mathbf{a}\, A \mid \mathbf{a}$$
$$B ::= \mathbf{b}$$

The recursive descent recogniser for this grammar will then be as follows

```
procedure RECOGNISEΓ(input u) {
    Let m be the length of u
    Read each element of u into I
    I[m] := $
    i := 0
    PARSES
    if i == m then Terminate and return success
    else FAIL
}

procedure PARSES {
    if I[i] ∈ SELECT(S, qS) then {
        i := i + 1
        PARSES
        return
    }
    if I[i] ∈ SELECT(S, Ab) then {
        PARSEA
        if I[i] = b then i := i + 1
        else FAIL
        return
    }
    if I[i] ∈ SELECT(S, B) then {
        PARSEB
        return
    }
     FAIL
}

procedure PARSEA {
    if I[i] ∈ SELECT(A, aA) then {
        if I[i] = a then i := i + 1
        else FAIL
        PARSEA
        return
    }
    if I[i] ∈ SELECT(A, a) then {
        if I[i] = a then i := i + 1
        else FAIL
        return
    }
}

procedure PARSEB {
    if I[i] ∈ SELECT(B, b) then {
```

```
        if I[i] = b then i := i + 1
        else FAIL
        return
    }
}
```

(Here FAIL is a function which terminates RECOGNISEΓ and returns an error message.)

SELECT$(X, x_1 \dots x_p)$ returns the lookahead sets defined using standard first and follow sets.

$$first_T(X) = \{t \in T | \exists \gamma (X \stackrel{*}{\Rightarrow} t\gamma)\}$$
$$follow_T(X) = \{t \in T | \exists \gamma, \delta (S \stackrel{*}{\Rightarrow} \gamma X t \beta)\}$$

If X is non-nullable then $first(X) = first_T(X)$, otherwise $first(X) = first_T(X) \cup \{\epsilon\}$. If $S \stackrel{*}{\not\Rightarrow} \alpha X$ then $follow(X) = follow_T(X)$ otherwise $follow(X) = follow_T(X) \cup \{\$\}$. Then if $first(x_1 \dots x_p)$ does not contain ϵ

$$\text{SELECT}(X, x_1 \dots x_p) = first(x_1 \dots x_p)$$

and if $first(x_1 \dots x_p)$ does contain ϵ

$$\text{SELECT}(X, x_1 \dots x_p) = first(x_1 \dots x_p) \cup follow(X)$$

Recursive descent parsers do not accept all grammars. In general, the select sets for each nonterminal are required to be disjoint to ensure the parser does not fail on a valid input string. Without additional lookahead, the recursive descent parser for the grammar on page 56 would never choose the alternate $A ::= \mathbf{a}$ as $\mathbf{a}$ is also in SELECT$(A, \mathbf{a}A)$. As a result, the recursive descent parser would incorrectly reject, for example, the string `ab`. Even worse, if the grammar were left recursive (for example, with $A ::= A\mathbf{a} | \mathbf{a}$), the parser may fail to terminate at all. Provided the grammar is not left recursive, the FAIL function could instead be some kind of backtracking mechanism which initiates exploration of the other alternates. However, without care, this can become exponential in both time and space requirements.

An alternative to backtracking is to create new processes for each possible option at points of non-determinism. This requires the function call stack to be handled explicitly. The function call stack holds the return position needed at the end of each parse function call. GLL is a technique that effectively behaves in such a manner - in this sense, it behaves like a generalisation of recursive descent, in which each possible choice of action is recorded for eventual processing. Multiple call stacks are represented efficiently using a graph-structured stack (GSS).

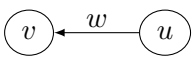
The GLL parsing technique, described next, can be applied to any context free grammar and outputs the SPPF representation of all possible derivations of an input string. The algorithm described is slightly different from that originally given in [SJ13] as GLL is presented here as a preliminary for the new MGLL algorithm.

3.1.2 GLL BNF Parsing Algorithm

Similar to a recursive descent parser, there is code in the GLL parser for Γ corresponding to every position in the grammar rules of Γ. Such a position is called a grammar slot, and a slot is denoted using item notation of form $X ::= x_1 \dots x_j \cdot x_{j_1} \dots x_p$. Lines of the parser are labelled with the corresponding grammar slot.

Conceptually, one can think of GLL as executing a recursive descent parser until a point is reached where a choice needs to be made. This can occur where there are two or more productions for a nonterminal containing the current input symbol in their select sets. At this point, a record of the current parser state for each option is created and then each option is processed in turn. A parser state comprises the current algorithm position, i.e. the line label, the current input symbol, the current call stack and, for a parser, the portion of the derivation tree constructed so far. Naïvely, tuples of the form (L, u, i, w) are needed, where L is a grammar slot, u is a stack, i is an integer, and w is a portion of a derivation tree. As remarked in the previous section, the number of such tuples is potentially exponential, and can be infinite if the grammar contains left recursion. To address this, two global data structures are maintained: a GSS combining all the call stacks, and an SPPF combining all the derivation trees. The state of the parser can then be represented by a tuple (or descriptor) (L, u, i, w), where L is a grammar slot, i is an integer, u is a GSS node corresponding to a stack top, and w is an SPPF node which is the left-most child of the next node to be constructed. To ensure only one SPPF node needs to be recorded, SPPFs are binarised as defined in Chapter 1.

GSS nodes are labelled with a pair $u = (L, i)$, where L is a grammar slot and i is an integer input index. The slot $X ::= \alpha A \cdot \beta$ is the return position after what, in a recursive descent parser, would be a call to parseA and i is the current input position. The same GSS node can be used by several different descriptors provided that the input position is the same in each case - hence the need to store the input position. When a GSS node, u, is created, an edge is added from it to the previous current GSS node, v, and this edge is labelled with the current SPPF node, w.



The GLL algorithm then proceeds to match the next portion of the input to the

non-terminal A. When this is completed, the GSS node $u = (X ::= \alpha A \cdot \beta, i)$ is 'popped'. An SPPF node w' is created whose left child is the edge label w and whose right child is the SPPF node associated with A, which has just been constructed. At some point, the algorithm will continue from position L with the current SPPF node and the current GSS node - which is now the child of u.

In general, the node u may have more than one child - this is how the GSS reduces the space required to store all the stacks. Thus, there may be more than one continuation choice. A pop action creates descriptors of the form (L, u', i, w') for each child u' of u, allowing all possible continuations to be explored.

A GLL algorithm consists of a main loop which selects a descriptor for processing. The set of unprocessed descriptors is denoted by $\mathcal{R}$. To avoid repeated processing, the set $\mathcal{U}$ of all descriptors which have been created is also maintained. A new GSS node can be added to a node which has already been popped, and a pop action will need to be subsequently applied down the new edge. Thus a set $\mathcal{P}$ of popped nodes is maintained and consulted whenever a new edge is added to an existing GSS node.

Three functions are defined to construct the GSS. The POP function performs a pop on the given GSS node. The ADD function adds descriptors to $\mathcal{U}$ and $\mathcal{R}$ where appropriate. The CREATE function generates a new GSS node and adds descriptors to ensure that the pop function is called for new nodes. POP is called at the end of a block of code corresponding to an alternate. CREATE is called at the point of what would be a call to a parse function in a recursive-descent parser.

```
function ADD(L, u, i, w) {
    if (L, u, i, w) ∉ U then add (L, u, i, w) to U and to R
}

function POP(u, i, z) {
    if u ≠ u_0 then {
        let (L, k) be the label of u
        add (u, z) to P
        for each GSS edge (u, w, v) do {
            let y be the node returned by GETNODE(L, w, z)
            ADD(L, v, i, y)
        }
    }
}

function CREATE(L, u, i, w) {
    if there is not already a GSS node labelled (L, i) then create one
    let v be the GSS node labelled (L, i)
```

```
    if there is not an edge from v to u labelled w then {
        create an edge from v to u labelled w
        for all (v, z) ∈ P do {
            let y be the node returned by getNode(L, w, z)
            let h be the right extent of z
            add(L, u, h, y)
        }
    }
    return v
}
```

Like recursive descent, GLL will use select sets to prevent the exploration of choices that are guaranteed to be invalid. Testing the current input against the select set of the current grammar slot can reduce the number of descriptors created and thus reduce the overall amount of work required by the parser.

```
function testSelect(a, Y, α) {
    if a ∈ first(α) or (ϵ ∈ first(α) ∧ a ∈ (first(Y) ∪ follow(Y))) then
        return true
    else
        return false
}
```

The SPPF is constructed with the following support functions. The function getNodeL builds SPPF nodes for terminals and ϵ. The function getNode builds SPPF nodes for productions and intermediate nodes. \$ denotes a dummy node - used as an anchor for constructing nodes. Q represents a grammar slot and is defined to be fiR (first-in-rule) if it is of the form $X ::= x \cdot \beta$ where x is a terminal or a non-nullable non-terminal, and does not represent the end of the rule (i.e. $\beta \neq \epsilon$).

```
function getNodeL(a, i, j) {
    if there is no SPPF node y labelled (a, i, j) then create one
    return y
}

function getNode(Q, w, z) {
    if Q is fiR then return z
    else{
        suppose that z has label (q, k, i)
        if Q is the end of a production then set t := left hand side of Q
        else set t := Q
        if w = $ then let j := k
```

```
        else suppose w has label (s, j, k)
        if an SPPF node y labelled (t, j, i) does not exist then create one
        if y does not have a child labelled (Q, k) then
            create one with right child z and, if w ≠ $, left child w
        return y
    }
}
```

When executing, a GLL parser is essentially traversing the grammar and the input string. It employs three variables, c_U which holds the current stack top (a GSS node), c_I which holds the current input position and c_N which holds the current SPPF node. When a descriptor is processed, in order to continue a parse these variables are set using the values in the descriptor.

For example, consider the grammar

$$S ::= A\,\mathbf{d} \mid A\,B$$
$$A ::= A\,\mathbf{a} \mid \mathbf{a}$$
$$B ::= \mathbf{b}\,\mathbf{b} \mid \mathbf{d}$$

A GLL parser for this grammar is as follows (c_R is a global variable that holds an SPPF node)

```
Create GSS node u_0 = (L_0, 0)
c_U := u_0; c_N := $; c_I := 0;
U := ∅; R := ∅; P := ∅
goto L_S
L_0 :
if R ≠ ∅ then {
    Remove (L, u, i, w) from R
    c_U := u; c_N := w; c_I := i;
    goto L
}
if there exists SPPF node labelled (S, 0, m) then
    Report success
else
    Report failure

L_S :
if testSelect(I[c_I], S, Ad) then
    add(L_{S::=Ad}, c_U, c_I, $)
```

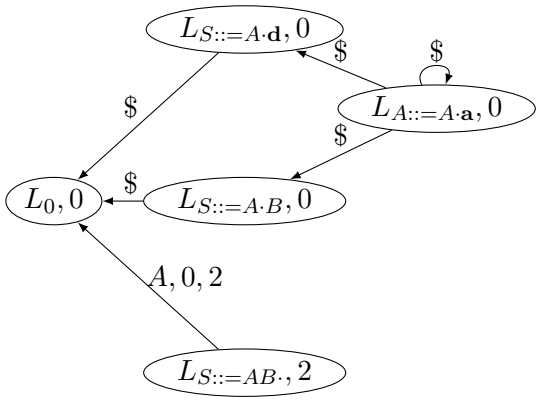
if TESTSELECT$(I[c_I], S, AB)$ **then**
  ADD$(L_{S::=AB}, c_U, c_I, \$)$
goto L_0
$L_{S::=A\mathbf{d}}$:
$c_U :=$ CREATE$(L_{S::=A\cdot\mathbf{d}}, c_U, c_I, c_N)$; **goto** L_A
$L_{S::=A\cdot\mathbf{d}}$:
if TESTSELECT$(I[c_I], S, \mathbf{d})$ is false **then goto** L_0
if $I[c_I] = \mathbf{d}$ **then** {
  $c_R :=$ GETNODEL$(\mathbf{d}, c_I, c_I + 1)$
  $c_I := c_I + 1$; $c_N :=$ GETNODE$(L_{S::=A\mathbf{d}\cdot}, c_N, c_R)$
}
if $I[c_I] \in follow(S)$ **then** POP(c_U, c_I, c_N)
goto L_0
$L_{S::=AB}$:
$c_U :=$ CREATE$(L_{S::=A\cdot B}, c_U, c_I, c_N)$; **goto** L_A
$L_{S::=A\cdot B}$:
if TESTSELECT$(I[c_I], S, B)$ is false **then goto** L_0
$c_U :=$ CREATE$(L_{S::=AB\cdot}, c_U, c_I, c_N)$; **goto** L_B
$L_{S::=AB\cdot}$:
if $I[c_I] \in follow(S)$ **then** POP(c_U, c_I, c_N)
goto L_0

L_A :
if TESTSELECT$(I[c_I], S, Aa)$ **then**
  ADD$(L_{A::=A\mathbf{a}}, c_U, c_I, \$)$
if TESTSELECT$(I[c_I], S, \mathbf{a})$ **then**
  ADD$(L_{A::=\mathbf{a}}, c_U, c_I, \$)$
goto L_0
$L_{A::=A\mathbf{a}}$:
$c_U :=$ CREATE$(L_{A::=A\cdot a}, c_U, c_I, c_N)$; **goto** L_A
$L_{A::=A\cdot\mathbf{a}}$:
if TESTSELECT$(I[c_I], A, \mathbf{a})$ is false **then goto** L_0
if $I[c_I] = \mathbf{a}$ **then** {
  $c_R :=$ GETNODEL$(\mathbf{a}, c_I, c_I + 1)$
  $c_I := c_I + 1$; $c_N :=$ GETNODE$(L_{A::=A\mathbf{a}\cdot}, c_N, c_R)$
}
if $I[c_I] \in follow(A)$ **then** POP(c_U, c_I, c_N)
goto L_0
$L_{A::=\mathbf{a}}$:
if $I[c_I] = \mathbf{a}$ **then** {
  $c_R :=$ GETNODEL$(\mathbf{a}, c_I, c_I + 1)$

$c_I := c_I + 1$; $c_N :=$ GETNODE($L_{A::=\mathbf{a}\cdot}, c_N, c_R$)
}
if $I[c_I] \in follow(A)$ **then** POP(c_U, c_I, c_N)
goto L_0

L_B :
if TESTSELECT($I[c_I], B, \mathbf{bb}$) **then**
ADD($L_{B::=\mathbf{bb}}, c_U, c_I, \$$)
if TESTSELECT($I[c_I], B, \mathbf{d}$) **then**
ADD($L_{B::=\mathbf{d}}, c_U, c_I, \$$)
goto L_0
$L_{B::=\mathbf{bb}}$:
if $I[c_I] = \mathbf{b}$ **then** {
$c_R :=$ GETNODEL($\mathbf{b}, c_I, c_I + 1$)
$c_I := c_I + 1$; $c_N :=$ GETNODE($L_{B::=\mathbf{b}\cdot\mathbf{b}}, c_N, c_R$)
}
if TESTSELECT($I[c_I], B, \mathbf{b}$) is false **then goto** L_0
if $I[c_I] = \mathbf{b}$ **then** {
$c_R :=$ GETNODEL($\mathbf{b}, c_I, c_I + 1$)
$c_I := c_I + 1$; $c_N :=$ GETNODE($L_{B::=\mathbf{bb}\cdot}, c_N, c_R$)
}
if $I[c_I] \in follow(B)$ **then** POP(c_U, c_I, c_N)
goto L_0
$L_{B::=\mathbf{d}}$:
if $I[c_I] = \mathbf{d}$ **then** {
$c_R :=$ GETNODEL($\mathbf{d}, c_I, c_I + 1$)
$c_I := c_I + 1$; $c_N :=$ GETNODE($L_{B::=\mathbf{d}\cdot}, c_N, c_R$)
}
if $I[c_I] \in follow(B)$ **then** POP(c_U, c_I, c_N)
goto L_0

The algorithm is illustrated by executing it on the string **aad**. The final GSS will be



The final U and P will be

$$
\begin{aligned}
U = \{&(LS ::= A\mathbf{d}, (L0,0), 0, \$), (LS ::= AB, (L0,0), 0, \$),\\
&(LA ::= A\mathbf{a}, (LS ::= A\cdot\mathbf{d}, 0), 0, \$), (LA ::= \mathbf{a}, (LS ::= A\cdot\mathbf{d}, 0), 0, \$),\\
&(LA ::= A\mathbf{a}, (LS ::= A\cdot B, 0), 0, \$), (LA ::= \mathbf{a}, (LS ::= A\cdot B, 0), 0, \$),\\
&(LA ::= A\mathbf{a}, (LA ::= A\cdot\mathbf{a}, 0), 0, \$), (LA ::= \mathbf{a}, (LA ::= A\cdot\mathbf{a}, 0), 0, \$),\\
&(LS ::= A\cdot\mathbf{d}, (L0,0), 1, (A,0,1)), (LS ::= A\cdot B, (L0,0), 1, (A,0,1)),\\
&(LA ::= A\cdot\mathbf{a}, (LS ::= A\cdot B, 0), 1, (A,0,1)),\\
&(LA ::= A\cdot\mathbf{a}, (LS ::= A\cdot B, 0), 2, (A,0,2)),\\
&(LS ::= A\cdot\mathbf{d}, (L0,0), 2, (A,0,2)), (LS ::= A\cdot B, (L0,0), 2, (A,0,2)),\\
&(LB ::= \mathbf{d}, (LS ::= AB\cdot, 2), 2, \$), (LS ::= AB\cdot, (L0,0), 3, (S,0,3))\}
\end{aligned}
$$

$$
\begin{aligned}
P = \{&((LS ::= A\cdot\mathbf{d}, 0), (A,0,1)), ((LS ::= A\cdot B, 0), (A,0,1)), ((LA ::= A\cdot\mathbf{a}, 0), (A,0,1)),\\
&((LA ::= A\cdot\mathbf{a}, 0), (A,0,2)), ((LS ::= A\cdot\mathbf{d}, 0), (A,0,2)), ((LS ::= A\cdot B, 0), (A,0,2)),\\
&((LS ::= AB\cdot, 2), (B,2,3))\}
\end{aligned}
$$

The parser will output the final SPPF in Figure 3.1.

The GLL parser specification is a set of 'templates', and a parsing algorithm is constructed by substituting actual grammar symbols, alternates, and sets of terminals into the templates. To perform the substitution, the slots associated with each symbol in the grammar need to be identified. Every symbol, x, on the right-hand side of every production is given a unique instance number, j, which is written as a superscript, x^j. For an instance, x^j, E_{x^j} denotes the slot immediately after that instance of x, $X ::= \alpha x\cdot\beta$. For each rule $X ::= \alpha$, $L_{\alpha'}$ denotes the slot $X ::= \cdot\alpha$, where α' is the instanced version of α, and this labels the section of the algorithm corresponding to the alternate α. L_0 denotes the line label of the start of the main program - this can be thought of as equivalent to the slot $S' ::= S\cdot$ in an LR-parser.

If r is an instanced string, $exp(r)$ denotes the underlying uninstanced string. $lhs(x^j)$ denotes the nonterminal on the left-hand side of the production rule in which the

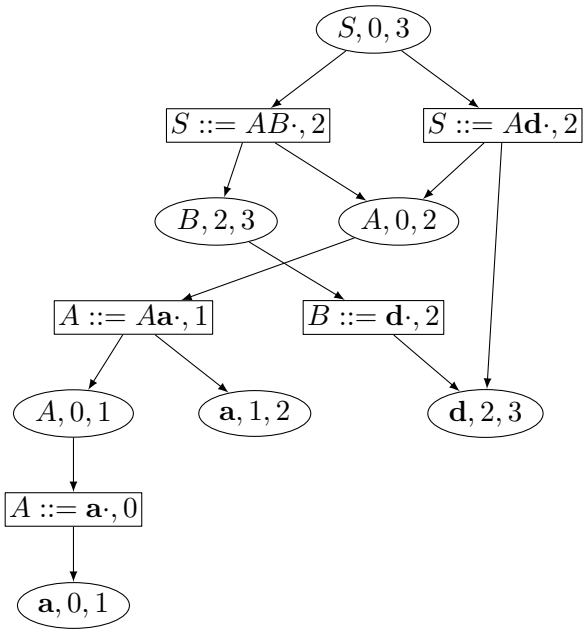


Figure 3.1: Final SPPF produced from the above GLL parse

instance of x^j occurs.

The main GLL parse function is then as follows

Create GSS node $u_0 = (L_0, 0)$
$c_U := u_0$; $c_N :=$ \$; $c_I := 0$;
$\mathcal{U} := \emptyset$; $\mathcal{R} := \emptyset$; $\mathcal{P} := \emptyset$
goto L_S
L_0 :
if $R \neq \emptyset$ **then** {
    Remove (L, u, i, w) from $\mathcal{R}$
    $c_U := u$; $c_N := w$; $c_I := i$;
    goto L
}
if there exists SPPF node labelled $(S, 0, m)$ **then**
    Report success
else
    Report failure

L_A : code(A)
...
L_Z : code(Z)

code(X) statements seen above are place-holders for where code generated by the templates for X should be inserted. If the rule for X is of the form $X ::= \alpha_1 | \ldots | \alpha_p$ then each alternate must be considered, so the following code is generated.

$code(X ::= \alpha_1 | \ldots | \alpha_p) =$
    if TESTSELECT$(I[c_I], X, \alpha_1)$ **then** ADD$(L_{\alpha'_1}, c_U, c_I, \$)$
    ...
    if TESTSELECT$(I[c_I], X, \alpha_p)$ **then** ADD$(L_{\alpha'_p}, c_U, c_I, \$)$
    goto L_0
$L_{\alpha'_1}$:
    code(r_1)
    if $I[c_I] \in follow(X)$ **then** POP(c_U, c_I, c_N)
    goto L_0
    ...
$L_{\alpha'_p}$:
    code(r_p)
    if $I[c_I] \in follow(X)$ **then** POP(c_U, c_I, c_N)
    goto L_0

The template for an instanced string, $g_1 g_2 \ldots g_d$, of terminals and non-terminals is

$code(g_1 g_2 \ldots g_d) =$
    code(g_1)
    code(g_2)
    ...
    code(g_d)

For an ϵ instance ϵ^j whose left-hand side is X

$code(\epsilon^j) =$
    $c_R :=$GETNODEL(ϵ, c_I, c_I); $c_N :=$GETNODE$(E_{\epsilon^j}, c_N, c_R)$

For a terminal $a \in T$, the code generated for an instance a^j, whose left-hand side is X, is

$code($code$(a^j) =$
    if $I[c_I] = a$ **then** {
        $c_R :=$GETNODEL$(a, c_I, c_I + 1)$

$c_I := c_I + 1; c_N :=$GETNODE(E_{a^j}, c_N, c_R)
}

For a non-terminal $Y \in N$, the code generated for an instance Y^j, whose left-hand side is X, is

$code(Y^j) =$
  $c_U :=$CREATE(E_{Y^j}, c_U, c_I, c_N); **goto** L_Y
E_{Y^j} :

3.2 Extended SPPFs

Before extending GLL to accept multiple input strings, it is necessary to consider how to efficiently represent the derivations of several different sentences.

An SPPF represents the set of all derivations for a single input string. If a TWE set is used as input to the parser, then an extended form of SPPF is required. The output of the parser is modified to handle the case where the input TWE set represents multiple tokenisations. This *Extended SPPF* (ESPPF) is a modification of the standard definition of an SPPF so that the leaf nodes now represent elements in the TWE set, Σ. For example, given the grammar

$$S ::= A\,\mathbf{d} \mid A\,B$$
$$A ::= A\,\mathbf{a} \mid \mathbf{a}$$
$$B ::= \mathbf{b}\,\mathbf{b} \mid \mathbf{d}$$

For the input token string `aabb`, a GLL parser would produce the SPPF given in Figure 3.2. However, suppose that the underlying character string has more than one tokenisation and that the TWE set is

$$\Sigma = \{(\mathbf{a},0,1),(\mathbf{a},0,2),(\mathbf{a},1,4),(\mathbf{a},1,5),(\mathbf{a},2,4),$$
$$(\mathbf{b},2,6),(\mathbf{b},4,5),(\mathbf{b},5,6),(\mathbf{d},5,6)\}$$

then the corresponding ITS set is

$$\begin{aligned} strings(\Sigma) = \{&(\mathbf{a},1)(\mathbf{a},4)(\mathbf{b},5)(\mathbf{b},6), \quad (\mathbf{a},1)(\mathbf{a},4)(\mathbf{b},5)(\mathbf{d},6), \quad (\mathbf{a},2)(\mathbf{b},6),\\ &(\mathbf{a},2)(\mathbf{a},4)(\mathbf{b},5)(\mathbf{b},6), \quad (\mathbf{a},1)(\mathbf{a},5)(\mathbf{b},6),\\ &(\mathbf{a},2)(\mathbf{a},4)(\mathbf{b},5)(\mathbf{d},6), \quad (\mathbf{a},1)(\mathbf{a},5)(\mathbf{d},6) \qquad \}\end{aligned}$$

A parser for this grammar would reject the tokenisations $(\mathbf{a},1)(\mathbf{a},4)(\mathbf{b},5)(\mathbf{d},6)$, $(\mathbf{a},1)(\mathbf{a},5)(\mathbf{b},6)$, $(\mathbf{a},1)(\mathbf{a},5)(\mathbf{d},6)$, $(\mathbf{a},2)(\mathbf{a},4)(\mathbf{b},5)(\mathbf{d},6)$, and $(\mathbf{a},2)(\mathbf{b},6)$. But, the tokenisations $(\mathbf{a},1)(\mathbf{a},4)(\mathbf{b},5)(\mathbf{b},6)$, and $(\mathbf{a},2)(\mathbf{a},4)(\mathbf{b},5)(\mathbf{b},6)$ would be accepted. Because both of these correspond to the token string `aabb`, the standard SPPF construction would lose the information about the extents. As a result, both are treated as the string `aabb`, resulting in the SPPF in Figure 3.2.

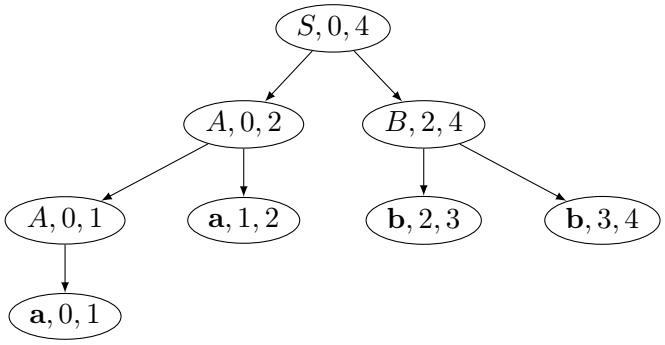


Figure 3.2: SPPF from parsing the tokenisation `aabb`

An ESPPF instead has leaf nodes whose extents match the extents in the TWE set. Suppose that a TWE set, Σ, has a corresponding ITS set

$$\begin{aligned}\{&(a_{11},i_{11})(a_{12},i_{12})\dots(a_{1j_1},m),\\ &(a_{21},i_{21})(a_{22},i_{22})\dots(a_{2j_2},n_n),\\ &\dots\\ &(a_{d1},i_{d1})(a_{d2},i_{d2})\dots(a_{dj_t},n_n)\}\end{aligned}$$

For each individual ITS

$$(a_{h1},i_{h1})(a_{h2},i_{h2})\dots(a_{hj_t},m)$$

the corresponding ESPPF will be a modified SPPF for this tokenisation in which every terminal node is relabelled to match the corresponding triple in the TWE set. The changes to the extents are then appropriately propagated throughout the graph to create the ESPPF. The final ESPPF is the union of the ESPPFs for each indexed token string.

For the example above, when the extents are adjusted to match the definition of an ESPPF, the result is three distinct ESPPFs - one for each valid tokenisation. The

ESPPF for $(\mathbf{a},1)(\mathbf{a},4)(\mathbf{b},5)(\mathbf{b},6)$ is given in Figure 3.3 and the ESPPF for $(\mathbf{a},2)(\mathbf{a},4)$ $(\mathbf{b},5)(\mathbf{b},6)$ is given in Figure 3.4. The ESPPF for the tokenisation $(\mathbf{a},1)(\mathbf{a},5)(\mathbf{d},6)$ is given in Figure 3.5. The ESPPF for the entire TWE set is then obtained by taking

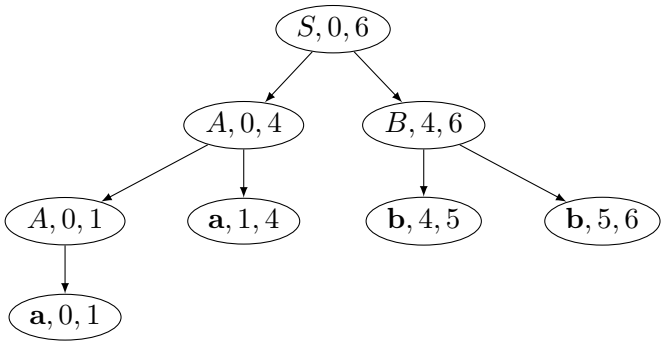


Figure 3.3: ESPPF from parsing the tokenisation $(\mathbf{a},1)(\mathbf{a},4)(\mathbf{b},5)(\mathbf{b},6)$

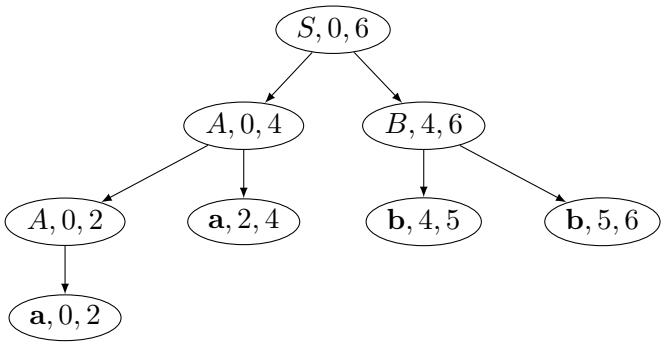


Figure 3.4: ESPPF from parsing the tokenisation $(\mathbf{a},2)(\mathbf{a},4)(\mathbf{b},5)(\mathbf{b},6)$

the union of the ESPPFs for each indexed token string. For the above ESPPFs, this is the ESPPF in Figure 3.6.

An ESPPF is a directed, bipartite graph with two disjoint node sets, V_p and V_s with the following properties

- Each $u \in V_s$ has a label of form (r,i,j) where $r \in (N \cup T \cup \epsilon)$ or r is a grammar slot $X ::= \alpha \cdot \beta$, $X ::= \alpha\beta \in P$, $i,j \in \mathbb{N}$ and $i \leq j$.
- Each $w \in V_p$ has a label of form $(X ::= \alpha \cdot \beta, i)$, $X ::= \alpha\beta \in P$ and $i \in \mathbb{N}$. Elements of V_p are called packed nodes, and i is called the pivot value.
- $u \in V_s$ with labels of the form (r,i,j), such that $(r,i,j) \in \Sigma$ are terminal nodes.
- $u \in V_s$ with labels of the form (ϵ,i,i) where $i \in \mathbb{N}$ are epsilon nodes.

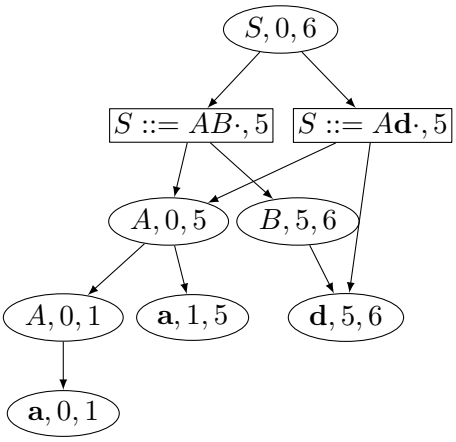


Figure 3.5: ESPPF from parsing the tokenisation $(\mathbf{a}, 1)(\mathbf{a}, 5)(\mathbf{d}, 6)$

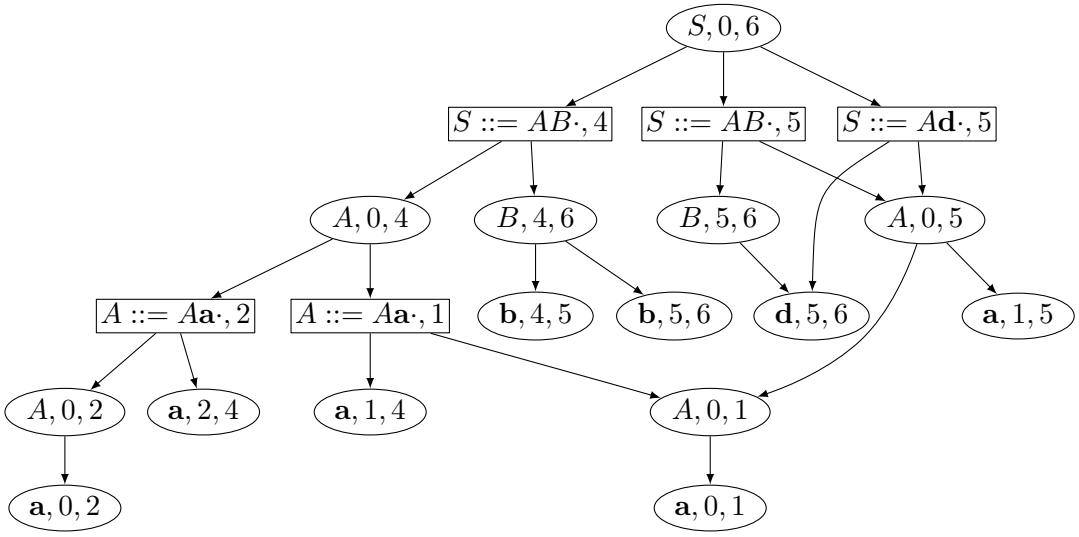


Figure 3.6: ESPPF combining the ESPPFs of tokenisations

- $u \in V_s$ with labels of the form (r, i, j), such that $r \in N$ are non-terminal nodes. Each non-terminal node has one or more children that are elements of V_p.
- $u \in V_s$ with labels of the form (r, i, j), such that r is a grammar slot $X ::= \alpha \cdot \beta$ are intermediate nodes. Each intermediate node has one or more children that are elements of V_p.
- If u is a node in V_s with more than one child, then there is a syntactic ambiguity at u.
- A node $w \in V_p$ has one or two children - an (optional) left child and a right child.

These children are elements of V_s. If w has label (r, i) and the right child has label (t, k, j), then $k = i$.

- If the original character string from which the TWE set was constructed is $\alpha = a_1 a_2 \dots a_m$, then the subgraph rooted at a node $u \in V_s$ labelled (δ, i, j) represents the substring $a_i \dots a_j$, $1 \leq i < j \leq m$.
- In all cases $0 \leq i < j \leq m$, where m is the height of Σ.
- For any two trees embedded in the ESPPF, if there is a node in the two trees with the same label, then there is exactly one corresponding node for those nodes in the ESPPF.

3.2.1 Retrieving Strings from the ESPPF

For single-input parsing, the tokenisation that is accepted can be retrieved from the SPPF by concatenating pairs consisting of the token name and right extent from each of the terminal nodes reachable from the root, in monotonically increasing order of extent. When parsing multiple inputs, hopefully, the parse would eliminate all but one input. In this case, the accepted tokenisation can be retrieved in the same way. However, this is not the case when multiple inputs are accepted by the parser.

Consider the ESPPF from Figure 3.6. Simply reading all terminal nodes in monotonically increasing order of extent will not work in this case as there are three tokenisations embedded in the ESPPF. Naïvely, one might consider constructing a TWE set by considering the labels of terminals reachable from the root as the elements of a TWE set. The strings that were parsed would then be the indexed token strings embedded in this TWE set. In this example, the TWE set

$$\{(\mathbf{a}, 0, 2), (\mathbf{a}, 2, 4), (\mathbf{a}, 1, 4), (\mathbf{b}, 4, 5), (\mathbf{b}, 5, 6), (\mathbf{a}, 0, 1), (\mathbf{d}, 5, 6), (\mathbf{a}, 1, 5)\}$$

would be constructed. This does embed the three tokenisations that were accepted - $(\mathbf{a}, 1)(\mathbf{a}, 4)(\mathbf{b}, 5)(\mathbf{b}, 6)$, $(\mathbf{a}, 2)(\mathbf{a}, 4)(\mathbf{b}, 5)(\mathbf{b}, 6)$ and $(\mathbf{a}, 1)(\mathbf{a}, 5)(\mathbf{d}, 6)$ - but it also embeds three other tokenisations - $(\mathbf{a}, 1)(\mathbf{a}, 4)(\mathbf{b}, 5)(\mathbf{d}, 6)$, $(\mathbf{a}, 2)(\mathbf{a}, 4)(\mathbf{b}, 5)(\mathbf{d}, 6)$ and $(\mathbf{a}, 1)(\mathbf{a}, 5)$ $(\mathbf{b}, 6)$ - that were rejected by the parser.

Instead, consider how one might obtain tokenisations in a manner similar to the way in which one obtains tokenisations in the single-input case. It is clear that two tokenisations cannot be part of the same derivation. Therefore, first one must identify which nodes are in a single derivation. Nodes that are children of the same packed node must be members of the same derivation. In the graph in Figure 3.7, this is illustrated by drawing a red/dashed line between these nodes. For a terminal node, (a, i, j), the

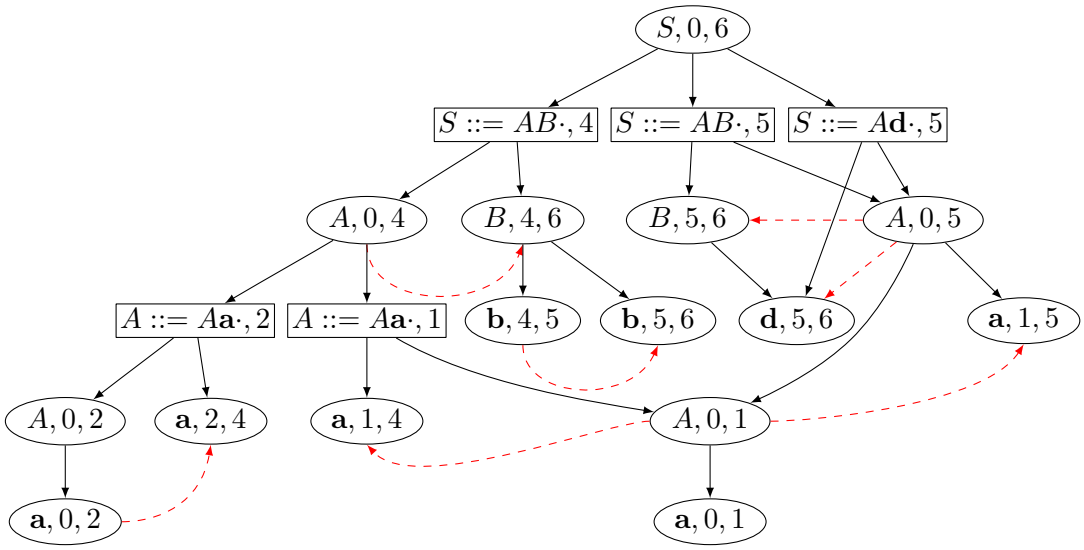


Figure 3.7: ESPPF from Figure 3.6 with red/dashed edges showing nodes that are children of the same packed node

yield set, that is the ITS set that the node represents, is the set containing one element (a, j). The yield set of a packed node is then the set obtained by concatenating elements of the yield set for its left child with elements of the yield set for its right child. If an interior node does not have multiple packed node children, then the yield set of that node is the yield set of its single (omitted in this visualisation) packed node child. So the node $(A, 0, 2)$ will have a yield set of $\{(\mathbf{a}, 2)\}$ and the packed node $(A ::= A\mathbf{a}\cdot, 2)$ will have a yield set of $\{(\mathbf{a}, 2)(\mathbf{a}, 4)\}$. Likewise, $(A, 0, 1)$ will have a yield set of $\{(\mathbf{a}, 1)\}$ and the packed node $(A ::= A\mathbf{a}\cdot, 1)$ will have a yield set of $\{(\mathbf{a}, 1)(\mathbf{a}, 4)\}$. $(A, 0, 5)$ will have a yield set of $\{(\mathbf{a}, 1)(\mathbf{a}, 5)\}$ and $(B, 5, 6)$ will have a yield set of $\{(\mathbf{d}, 6)\}$, so the packed nodes $(S ::= AB\cdot, 5)$ and $(S ::= A\mathbf{d}\cdot, 5)$ will both have a yield set of $\{(\mathbf{a}, 1)$ $(\mathbf{a}, 5)(\mathbf{d}, 6)\}$.

Where an interior node has multiple packed node children, this indicates the presence of multiple derivations under that node. The yield set of an interior node is, therefore, the union of the yield sets of its packed node children. As the yield set for $(A ::= A\mathbf{a}\cdot, 2)$ is $\{(\mathbf{a}, 2)(\mathbf{a}, 4)\}$ and the yield set for $(A ::= A\mathbf{a}\cdot, 1)$ is $\{(\mathbf{a}, 1)(\mathbf{a}, 4)\}$, the yield set for $(A, 0, 4)$ is $\{(\mathbf{a}, 2)(\mathbf{a}, 4), (\mathbf{a}, 1)(\mathbf{a}, 4)\}$. The yield set for $(B, 4, 6)$ is $\{(\mathbf{b}, 5)$ $(\mathbf{b}, 6)\}$, so the yield set for the packed node $(S ::= AB\cdot, 4)$ is $\{(\mathbf{a}, 2)(\mathbf{a}, 4)(\mathbf{b}, 5)(\mathbf{b}, 6)$, $(\mathbf{a}, 1)(\mathbf{a}, 4)(\mathbf{b}, 5)(\mathbf{b}, 6)\}$. The yield set for $(S, 0, 6)$ is then $\{(\mathbf{a}, 2)(\mathbf{a}, 4)(\mathbf{b}, 5)(\mathbf{b}, 6), (\mathbf{a}, 1)$ $(\mathbf{a}, 4)(\mathbf{b}, 5)(\mathbf{b}, 6), (\mathbf{a}, 1)(\mathbf{a}, 5)(\mathbf{d}, 6)\}$. As this is the root node, this corresponds to the set of indexed token strings that were accepted by the parser - the three tokenisations that were expected.

Formally, an algorithm that constructs the yield set, X_Γ, is as follows

- For leaf nodes u labelled (a, i, j), if $i < j$ then let $X_u = \{(a, j)\}$. If $i = j$, let $X_u = \emptyset$.
- For packed nodes u with two children $y = (t, i, k)$ and $z = (s, k, j)$, where $i \leq k \leq j$, let $X_u = X_y \cdot X_z$, the set of all strings which are concatenations of some element of X_y with an element of X_z.
- For packed nodes u with one child $y = (t, i, j)$, let $X_u = X_y$.
- For internal nodes u with packed node children $v_1, v_2, \ldots, v_p$, let $X_u = X_{v_1} \cup X_{v_2} \cup \cdots \cup X_{v_p}$.
- If u is the root node, then $X_\Gamma = X_u$.

3.3 Parsing a TWE Set

The aim is to parse the strings embedded in the TWE set more efficiently than simply parsing each string independently. The GLL approach can be modified relatively easily to allow the existence of several different terminals at a given input position. This modification, termed *MGLL*, is given in this section.

3.3.1 GLL BNF Parsing Algorithm for a TWE Set (MGLL)

The main modification that is necessary to extend GLL to multiple inputs involves a modification of the templates to consider all token matches at a given input position. This involves modifying the template for terminals so that instead of simply constructing the SPPF nodes for the current token and then advancing the input position, the nodes for the current token are constructed and then descriptors are added for all token triples that follow (in terms of the embedded ITS) the current token triple. Of course, the templates for SPPF node construction must also be modified to assign appropriate character extents.

Each descriptor stores (L, u, c, w) - where L is a grammar slot, u is a GSS Node, c is a terminal/integer pair (b, i) and w is an SPPF node. Elements of $\mathcal{P}$ are now triples (u, a, z), where u, z are GSS nodes and $a \in T$. For each terminal/integer pair $c = (b, i)$, $c.sym$ and $c.extent$ represent b and i respectively. The ADD, POP and CREATE methods are modified to take this into account.

function ADD(L, u, c, w) {

```
    if (L, u, c, w) ∉ U then add (L, u, c, w) to U and to R
}

function POP(u, c, z) {
    if u ≠ u_0 then {
        let (L, k) be the label of u
        add (u, c.sym, z) to P
        for each GSS edge (u, w, v) do {
            let y be the node returned by getNode(L, w, z)
            add(L, v, c, y)
        }
    }
}

function create(L, u, i, w) {
    if there is not already a GSS node labelled (L, i) then create one
    let v be the GSS node labelled (L, i)
    if there is not an edge from v to u labelled w then {
        create an edge from v to u labelled w
        for all a, z such that (v, a, z) ∈ P do {
            let y be the node returned by getNode(L, w, z)
            let h be the right extent of z and let f = (a, h)
            add(L, u, f, y)
        }
    }
    return v
}
```

The testSelect and SPPF node creation functions remain unchanged from the previous version.

A challenge in parsing a set of triples is that the elements of the set are unordered. The parser will need to know the triples that will follow a token that has been parsed. To lower the search cost, the follow sets of each token found at a left extent are pre-computed. For each a starting at left-extent i, a set $\Sigma_{a,i} \subseteq \Sigma$ is defined such that $(b, j) \in \Sigma_{a,i}$ if $(a, i, j), (b, j, k) \in \Sigma$ for some $k > j$.

The input is a TWE set Σ with parser lookahead sets $\Sigma_{a,j}$ as previously defined. Σ is augmented with a final triple $(\$, m, m + 1)$. c_C represents a pair (a, i). Recall from Chapter 2 that m is the height of the TWE set. c_I is an integer value $0 \leq c_I \leq m$. c_a is a variable which is either some $t \in T$ or \$. sentence(w) is the result of running the sentence finding algorithm on the ESPPF rooted at w. The main GLL parse function is then as follows

Construct sets $\Sigma_{a,i}$ from Σ
Create GSS node $u_0 = (L_0, 0)$
$\mathcal{U} := \emptyset$; $\mathcal{R} := \emptyset$; $\mathcal{P} := \emptyset$
for all $(a, 0, i) \in \Sigma$ **do**
    ADD$(L_S, u_0, (a, 0), \$)$)
L_0 :
if $R \neq \emptyset$ **then** {
    Remove (L, u, c, w) from $\mathcal{R}$
    $c_U := u$; $c_N := w$; $c_C := c$; $c_I := c.extent$; $c_a := c.sym$;
    goto L
}
if there exists SPPF node labelled $(S, 0, m)$ **then**
    output SENTENCE$((S, 0, m))$
else
    Report failure

L_A : code(A)
$\vdots$
L_Z : code(Z)

If the rule for X is of the form $X ::= \alpha_1 | \ldots | \alpha_p$ then each alternate must be considered, so the following code is generated.

code$(X ::= \alpha_1 | \ldots | \alpha_p)$ =
    if TESTSELECT(c_a, X, α_1) **then** ADD$(L_{\alpha_1}, c_U, c_C, \$)$
    $\vdots$
    if TESTSELECT(c_a, X, α_p) **then** ADD$(L_{\alpha_p}, c_U, c_C, \$)$
    goto L_0
L_{α_1} :
    code(r_1)
    POP(c_U, c_C, c_N)
    goto L_0
    $\vdots$
L_{α_p} :
    code(r_p)
    POP(c_U, c_C, c_N)
    goto L_0

If r is an instanced string, let $exp(r)$ be a function that returns the corresponding uninstanced string. The template for an instanced string, $g_1 g_2 \ldots g_d$, of terminals and

non-terminals is

$code(g_1 g_2 \ldots g_d) =$
    code(g_1)
    code(g_2)
    $\vdots$
    code(g_d)

For an ϵ instance ϵ^j whose left-hand side is X

$code(\epsilon^j) =$
    $c_R :=$GETNODEL(ϵ, c_I, c_I); $c_N :=$GETNODE$(E_{\epsilon^j}, c_N, c_R)$
    if $c_a \notin follow(X)$ **then goto** L_0

For a terminal $a \in T$ the code generated for an instance a^j, whose left-hand side is X and whose follow is β, is

$code(a^j) =$
    if $|\Sigma_{a,c_I}| > 1$ **then** {
        for all $(b,k) \in \Sigma_{a,c_I}$ **do** {
            if TESTSELECT(b, X, β) is true **then** {
                $c_R :=$GETNODEL(a, c_I, k)
                Let $c_x :=$GETNODE(E_ρ, c_N, c_R)
                ADD$(E_\rho, c_U, (b,k), c_x)$
            }
        }
        goto L_0
    }
    Let (b,k) be the singular element in Σ_{a,c_I}
    if TESTSELECT(b, X, β) is false **then goto** L_0
    $c_R :=$GETNODEL(a, c_I, k)
    $c_N :=$GETNODE(E_ρ, c_N, c_R)
    $c_I := k$; $c_C := (b,k)$; $c_a := b$
E_ρ :

For a non-terminal $Y \in N$ the code generated for an instance Y^j, whose left-hand side is X and whose follow is β, is

$code(Y^j) =$
    $c_U :=$CREATE(E_{Y^j}, c_U, c_I, c_N); **goto** L_Y
E_{Y^j} :

if TESTSELECT(c_a, x, β) is false **then goto** L_0

This gives a complete set of templates for generating an MGLL parser. As will be demonstrated in Chapter 5, the shape of the ESPPF is comparable to an SPPF produced by an equivalent character-level parser, therefore the size of an ESPPF is no worse than cubic in the length of the input string. As character indices remain the basis for descriptor creation, the algorithm should have an upper-bound time complexity no worse than quartic in the number of character indices to process, and the upper-bound may even be cubic. However, a full complexity analysis was not the focus of this study, as the main advantage of MGLL is in the extent to which it is 'recursively decent' - in the sense that it is relatively simple to understand and implement. The performance of MGLL is certainly comparable to GLL in the average use case, as will be demonstrated in Chapter 6.

Example

Recall the grammar

$$S ::= A\ \mathbf{d} \mid A\ B$$
$$A ::= A\ \mathbf{a} \mid \mathbf{a}$$
$$B ::= \mathbf{b}\ \mathbf{b} \mid \mathbf{d}$$

A MGLL parser for this grammar is as follows

Construct sets $\Sigma_{a,i}$ from Σ
Create GSS node $u_0 = (L_0, 0)$
$\mathcal{U} := \emptyset$; $\mathcal{R} := \emptyset$; $\mathcal{P} := \emptyset$
for all $(x, 0, i) \in \Sigma$ **do**
    ADD$(L_S, u_0, (a, 0), \$)$)
L_0 :
if $R \neq \emptyset$ **then** {
    Remove (L, u, c, w) from $\mathcal{R}$
    $c_U := u$; $c_N := w$; $c_C := c$; $c_I := c.extent$; $c_a := c.sym$;
    goto L
}
if there exists SPPF node labelled $(S, 0, m)$ **then**
    output SENTENCE$((S, 0, m))$
else
    Report failure

L_S :
if TESTSELECT$(c_a, S, A\mathbf{d})$ **then**
  ADD$(L_{S::=A\mathbf{d}}, c_U, c_C, \$)$
if TESTSELECT(c_a, S, AB) **then**
  ADD$(L_{S::=AB}, c_U, c_C, \$)$
goto L_0
$L_{S::=A\mathbf{d}}$:
$c_U :=$ CREATE$(E_{S::=A\cdot\mathbf{d}}, c_U, c_I, c_N)$; **goto** L_A
$E_{S::=A\cdot\mathbf{d}}$:
if TESTSELECT$(c_a, S, \mathbf{d})$ is false **then goto** L_0
if $|\Sigma_{\mathbf{d},c_I}| > 1$ **then** {
  for all $(x,k) \in \Sigma_{\mathbf{d},c_I}$ **do** {
    if $x \in follow(S)$ **then** {
      $c_R :=$GETNODEL$(\mathbf{d}, c_I, k)$
      Let $c_x :=$GETNODE$(E_{S::=A\mathbf{d}\cdot}, c_N, c_R)$
      ADD$(E_{S::=A\mathbf{d}\cdot,c_U}, (x,k), c_x)$
    }
  }
  goto L_0
}
Let (x,k) be the singular element in $\Sigma_{\mathbf{d},c_I}$
if $x \notin follow(S)$ **then goto** L_0
$c_R :=$GETNODEL$(\mathbf{d}, c_I, k)$
$c_N :=$GETNODE$(E_{S::=A\mathbf{d}\cdot}, c_N, c_R)$
$c_I := k$; $c_C := (x,k)$; $c_a := x$
$E_{S::=A\mathbf{d}\cdot}$:
POP(c_U, c_C, c_N)
goto L_0
$L_{S::=AB}$:
$c_U :=$ CREATE$(E_{S::=A\cdot B}, c_U, c_I, c_N)$; **goto** L_A
$E_{S::=A\cdot B}$:
if TESTSELECT(c_a, S, B) is false **then goto** L_0
$c_U :=$ CREATE$(E_{S::=AB\cdot}, c_U, c_I, c_N)$; **goto** L_B
$E_{S::=AB\cdot}$:
if $c_a \notin follow(S)$ **then goto** L_0
POP(c_U, c_C, c_N)
goto L_0

L_A :
if TESTSELECT$(c_a, A, A\mathbf{a})$ **then**
  ADD$(L_{A::=A\mathbf{a}}, c_U, c_C, \$)$
if TESTSELECT$(c_a, A, \mathbf{a})$ **then**

ADD($L_{A::=\mathbf{a}}, c_U, c_C, \$$)

goto L_0

$L_{A::=A\mathbf{a}}$:

c_U := CREATE($E_{A::=A\cdot\mathbf{a}}, c_U, c_I, c_N$); **goto** L_A

$E_{A::=A\cdot\mathbf{a}}$:

if TESTSELECT($c_a, A, \mathbf{a}$) is false **then goto** L_0

if $|\Sigma_{\mathbf{a},c_I}| > 1$ **then** {
  for all $(x,k) \in \Sigma_{\mathbf{a},c_I}$ **do** {
    if $x \in follow(A)$ **then** {
      c_R :=GETNODEL($\mathbf{a}, c_I, k$)
      Let c_x :=GETNODE($E_{A::=A\mathbf{a}\cdot}, c_N, c_R$)
      ADD($E_{A::=A\mathbf{a}\cdot}, c_U, (x,k), c_x$)
    }
  }
  goto L_0
}

Let (x,k) be the singular element in $\Sigma_{\mathbf{a},c_I}$

if $x \notin follow(A)$ **then goto** L_0

c_R :=GETNODEL($\mathbf{a}, c_I, k$)

c_N :=GETNODE($E_{A::=A\mathbf{a}\cdot}, c_N, c_R$)

$c_I := k$; $c_C := (x,k)$; $c_a := x$

$E_{A::=A\mathbf{a}\cdot}$:

POP(c_U, c_C, c_N)

goto L_0

$L_{A::=\mathbf{a}}$:

if $|\Sigma_{a,c_I}| > 1$ **then** {
  for all $(x,k) \in \Sigma_{\mathbf{a},c_I}$ **do** {
    if $x \in follow(A)$ **then** {
      c_R :=GETNODEL($\mathbf{a}, c_I, k$)
      Let c_x :=GETNODE($E_{A::=\mathbf{a}\cdot}, c_N, c_R$)
      ADD($E_{A::=\mathbf{a}\cdot}, c_U, (x,k), c_x$)
    }
  }
  goto L_0
}

Let (x,k) be the singular element in $\Sigma_{\mathbf{a},c_I}$

if $x \notin follow(A)$ **then goto** L_0

c_R :=GETNODEL($\mathbf{a}, c_I, k$)

c_N :=GETNODE($E_{A::=\mathbf{a}\cdot}, c_N, c_R$)

$c_I := k$; $c_C := (x,k)$; $c_a := x$

$E_{A::=\mathbf{a}\cdot}$:

POP(c_U, c_C, c_N)

goto L_0

L_B :
if TESTSELECT$(c_a, B, \mathbf{bb})$ **then**
  ADD$(L_{B::=\mathbf{bb}}, c_U, c_C, \$)$
if TESTSELECT$(c_a, B, \mathbf{d})$ **then**
  ADD$(L_{B::=\mathbf{d}}, c_U, c_C, \$)$
goto L_0
$L_{B::=\mathbf{bb}}$:
if $|\Sigma_{\mathbf{b},c_I}| > 1$ **then** {
  for all $(x,k) \in \Sigma_{\mathbf{b},c_I}$ **do** {
    if TESTSELECT$(x, B, \mathbf{b})$ is true **then** {
      c_R :=GETNODEL$(\mathbf{b}, c_I, k)$
      Let c_x :=GETNODE$(E_{B::=\mathbf{b}\cdot\mathbf{b}}, c_N, c_R)$
      ADD$(E_{B::=\mathbf{b}\cdot\mathbf{b}}, c_U, (x,k), c_x)$
    }
  }
  goto L_0
}
Let (x,k) be the singular element in $\Sigma_{\mathbf{b},c_I}$
if TESTSELECT$(x, B, \mathbf{b})$ is false **then goto** L_0
c_R :=GETNODEL$(\mathbf{b}, c_I, k)$
c_N :=GETNODE$(E_{B::=\mathbf{b}\cdot\mathbf{b}}, c_N, c_R)$
$c_I := k$; $c_C := (x,k)$; $c_a := x$
$E_{B::=\mathbf{b}\cdot\mathbf{b}}$:
if $|\Sigma_{\mathbf{b},c_I}| > 1$ **then** {
  for all $(x,k) \in \Sigma_{\mathbf{b},c_I}$ **do** {
    if $x \in follow(B)$ **then** {
      c_R :=GETNODEL$(\mathbf{b}, c_I, k)$
      Let c_x :=GETNODE$(E_{B::=\mathbf{bb}\cdot}, c_N, c_R)$
      ADD$(E_{B::=\mathbf{bb}\cdot}, c_U, (x,k), c_x)$
    }
  }
  goto L_0
}
Let (x,k) be the singular element in $\Sigma_{\mathbf{b},c_I}$
if $x \notin follow(B)$ **then goto** L_0
c_R :=GETNODEL$(\mathbf{b}, c_I, k)$
c_N :=GETNODE$(E_{B::=\mathbf{bb}\cdot}, c_N, c_R)$
$c_I := k$; $c_C := (x,k)$; $c_a := x$
$E_{B::=\mathbf{bb}\cdot}$:
POP(c_U, c_C, c_N)
goto L_0
$L_{B::=\mathbf{d}}$:

if $|\Sigma_{\mathbf{d},c_I}| > 1$ **then** {
  for all $(x,k) \in \Sigma_{\mathbf{d},c_I}$ **do** {
    if $x \in follow(B)$ **then** {
      $c_R :=$GETNODEL$(\mathbf{d}, c_I, k)$
      Let $c_x :=$GETNODE$(E_{B::=\mathbf{d}\cdot}, c_N, c_R)$
      ADD$(E_{B::=\mathbf{d}\cdot}, c_U, (x,k), c_x)$
    }
  }
  goto L_0
}
Let (x,k) be the singular element in $\Sigma_{\mathbf{d},c_I}$
if $x \notin follow(B)$ **then goto** L_0
$c_R :=$GETNODEL$(\mathbf{d}, c_I, k)$
$c_N :=$GETNODE$(E_{B::=\mathbf{d}\cdot}, c_N, c_R)$
$c_I := k$; $c_C := (x,k)$; $c_a := x$
$E_{B::=\mathbf{d}\cdot}$:
POP(c_U, c_C, c_N)
goto L_0

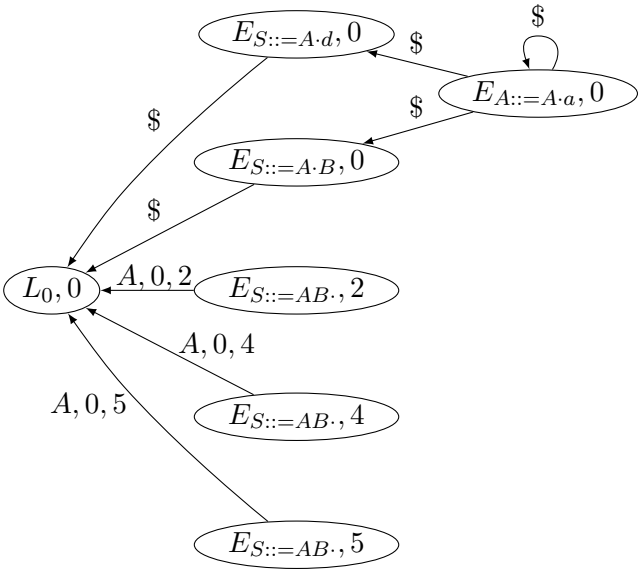
Consider the TWE set

$$\Sigma = \{(\mathbf{a},0,1),(\mathbf{a},0,2),(\mathbf{a},1,4),(\mathbf{a},1,5),(\mathbf{a},2,4),$$
$$(\mathbf{b},2,6),(\mathbf{b},4,5),(\mathbf{b},5,6),(\mathbf{d},5,6)\}$$

The first step of the algorithm creates triple follow sets for this TWE set as follows

$$\Sigma_{\mathbf{a},0} = \{(\mathbf{a},1),(\mathbf{a},2),(\mathbf{b},2)\}$$
$$\Sigma_{\mathbf{a},1} = \{(\mathbf{b},4),(\mathbf{b},5),(\mathbf{d},5)\}$$
$$\Sigma_{\mathbf{a},2} = \{(\mathbf{b},4)\}$$
$$\Sigma_{\mathbf{b},2} = \{(\$,6)\}$$
$$\Sigma_{\mathbf{b},4} = \{(\mathbf{b},5),(\mathbf{d},5)\}$$
$$\Sigma_{\mathbf{b},5} = \{(\$,6)\}$$
$$\Sigma_{\mathbf{d},5} = \{(\$,6)\}$$

The final GSS will be



The final values of U and P will be

$$
\begin{aligned}
U = \{&(LS,(L0,0),(\mathbf{a},0),\$),(LS ::= A\mathbf{d},(L0,0),(\mathbf{a},0),\$),\\
&(LS ::= AB,(L0,0),(\mathbf{a},0),\$),(LA ::= A\mathbf{a},(ES ::= A\cdot\mathbf{d},0),(\mathbf{a},0),\$),\\
&(LA ::= \mathbf{a},(ES ::= A\cdot\mathbf{d},0),(\mathbf{a},0),\$),(LA ::= A\mathbf{a},(ES ::= A\cdot B,0),(\mathbf{a},0),\$),\\
&(LA ::= \mathbf{a},(ES ::= A\cdot B,0),(\mathbf{a},0),\$),(LA ::= A\mathbf{a},(EA ::= A\cdot\mathbf{a},0),(\mathbf{a},0),\$),\\
&(LA ::= \mathbf{a},(EA ::= A\cdot\mathbf{a},0),(\mathbf{a},0),\$),(EA ::= \mathbf{a}\cdot,(ES ::= A\cdot\mathbf{d},0),(\mathbf{a},1),(A,0,1)),\\
&(EA ::= \mathbf{a}\cdot,(ES ::= A\cdot\mathbf{d},0),(\mathbf{a},2),(A,0,2)),(EA ::= \mathbf{a}\cdot,(ES ::= A\cdot\mathbf{d},0),(\mathbf{b},2),(A,0,2)),\\
&(EA ::= \mathbf{a}\cdot,(ES ::= A\cdot B,0),(\mathbf{a},1),(A,0,1)),(EA ::= \mathbf{a}\cdot,(ES ::= A\cdot B,0),(\mathbf{a},2),(A,0,2)),\\
&(EA ::= \mathbf{a}\cdot,(ES ::= A\cdot B,0),(\mathbf{b},2),(A,0,2)),(EA ::= \mathbf{a}\cdot,(EA ::= A\cdot\mathbf{a},0),(\mathbf{a},1),(A,0,1)),\\
&(EA ::= \mathbf{a}\cdot,(EA ::= A\cdot\mathbf{a},0),(\mathbf{a},2),(A,0,2)),(EA ::= \mathbf{a}\cdot,(EA ::= A\cdot\mathbf{a},0),(\mathbf{b},2),(A,0,2)),\\
&(ES ::= A\cdot\mathbf{d},(L0,0),(\mathbf{a},1),(A,0,1)),(ES ::= A\cdot\mathbf{d},(L0,0),(\mathbf{a},2),(A,0,2)),\\
&(ES ::= A\cdot\mathbf{d},(L0,0),(\mathbf{b},2),(A,0,2)),(ES ::= A\cdot B,(L0,0),(\mathbf{a},1),(A,0,1)),\\
&(ES ::= A\cdot B,(L0,0),(\mathbf{a},2),(A,0,2)),(ES ::= A\cdot B,(L0,0),(\mathbf{b},2),(A,0,2)),\\
&(EA ::= A\cdot\mathbf{a},(EA ::= A\cdot\mathbf{a},0),(\mathbf{a},1),(A,0,1)),(EA ::= A\cdot\mathbf{a},(ES ::= A\cdot\mathbf{d},0),(\mathbf{a},1),(A,0,1)),\\
&(EA ::= A\cdot\mathbf{a},(ES ::= A\cdot B,0),(\mathbf{a},1),(A,0,1)),(EA ::= A\cdot\mathbf{a},(EA ::= A\cdot\mathbf{a},0),(\mathbf{a},2),(A,0,2)),\\
&(EA ::= A\cdot\mathbf{a},(ES ::= A\cdot\mathbf{d},0),(\mathbf{a},2),(A,0,2)),(EA ::= A\cdot\mathbf{a},(ES ::= A\cdot B,0),(\mathbf{a},2),(A,0,2)),\\
&(EA ::= A\cdot\mathbf{a},(EA ::= A\cdot\mathbf{a},0),(\mathbf{b},2),(A,0,2)),(EA ::= A\cdot\mathbf{a},(ES ::= A\cdot\mathbf{d},0),(\mathbf{b},2),(A,0,2)),\\
&(EA ::= A\cdot\mathbf{a},(ES ::= A\cdot B,0),(\mathbf{b},2),(A,0,2)),(LB ::= \mathbf{bb},(ES ::= AB\cdot,2),(\mathbf{b},2),\$),\\
&(EA ::= A\mathbf{a}\cdot,(EA ::= A\cdot\mathbf{a},0),(\mathbf{b},4),(A,0,4)),(EA ::= A\mathbf{a}\cdot,(EA ::= A\cdot\mathbf{a},0),(\mathbf{b},5),(A,0,5)),\\
&(EA ::= A\mathbf{a}\cdot,(EA ::= A\cdot\mathbf{a},0),(\mathbf{d},5),(A,0,5)),(EA ::= A\mathbf{a}\cdot,(ES ::= A\cdot\mathbf{d},0),(\mathbf{b},4),(A,0,4)),\\
&(EA ::= A\mathbf{a}\cdot,(ES ::= A\cdot\mathbf{d},0),(\mathbf{b},5),(A,0,5)),(EA ::= A\mathbf{a}\cdot,(ES ::= A\cdot\mathbf{d},0),(\mathbf{d},5),(A,0,5)),\\
&(EA ::= A\mathbf{a}\cdot,(ES ::= A\cdot B,0),(\mathbf{b},4),(A,0,4)),(EA ::= A\mathbf{a}\cdot,(ES ::= A\cdot B,0),(\mathbf{b},5),(A,0,5)),\\
&(EA ::= A\mathbf{a}\cdot,(ES ::= A\cdot B,0),(\mathbf{d},5),(A,0,5)),(EA ::= A\cdot\mathbf{a},(EA ::= A\cdot\mathbf{a},0),(\mathbf{b},4),(A,0,4)),\\
&(EA ::= A\cdot\mathbf{a},(ES ::= A\cdot\mathbf{d},0),(\mathbf{b},4),(A,0,4)),(EA ::= A\cdot\mathbf{a},(ES ::= A\cdot B,0),(\mathbf{b},4),(A,0,4)),
\end{aligned}
$$

$(ES ::= A \cdot \mathbf{d}, (L0, 0), (\mathbf{b}, 4), (A, 0, 4)), (ES ::= A \cdot B, (L0, 0), (\mathbf{b}, 4), (A, 0, 4)),$
$(EA ::= A \cdot \mathbf{a}, (EA ::= A \cdot \mathbf{a}, 0), (\mathbf{b}, 5), (A, 0, 5)), (EA ::= A \cdot \mathbf{a}, (ES ::= A \cdot \mathbf{d}, 0), (\mathbf{b}, 5), (A, 0, 5)),$
$(EA ::= A \cdot \mathbf{a}, (ES ::= A \cdot B, 0), (\mathbf{b}, 5), (A, 0, 5)), (EA ::= A \cdot \mathbf{a}, (EA ::= A \cdot \mathbf{a}, 0), (\mathbf{d}, 5), (A, 0, 5)),$
$(EA ::= A \cdot \mathbf{a}, (ES ::= A \cdot \mathbf{d}, 0), (\mathbf{d}, 5), (A, 0, 5)), (EA ::= A \cdot \mathbf{a}, (ES ::= A \cdot B, 0), (\mathbf{d}, 5), (A, 0, 5)),$
$(ES ::= A \cdot \mathbf{d}, (L0, 0), (\mathbf{b}, 5), (A, 0, 5)), (ES ::= A \cdot \mathbf{d}, (L0, 0), (\mathbf{d}, 5), (A, 0, 5)),$
$(ES ::= A \cdot B, (L0, 0), (\mathbf{b}, 5), (A, 0, 5)), (ES ::= A \cdot B, (L0, 0), (d, 5), (A, 0, 5)),$
$(LB ::= \mathbf{bb}, (ES ::= AB\cdot, 4), (\mathbf{b}, 4), \$), (LB ::= \mathbf{bb}, (ES ::= AB\cdot, 5), (\mathbf{b}, 5), \$),$
$(LB ::= \mathbf{d}, (ES ::= AB\cdot, 5), (\mathbf{d}, 5), \$), (EB ::= \mathbf{b} \cdot \mathbf{b}, (ES ::= AB\cdot, 4), (\mathbf{b}, 5), (\mathbf{b}, 4, 5)),$
$(ES ::= AB\cdot, (L0, 0), (\$, 6), (S, 0, 6))\}$

$P = \{((ES ::= A \cdot \mathbf{d}, 0), \mathbf{a}, (A, 0, 1)), ((ES ::= A \cdot \mathbf{d}, 0), \mathbf{a}, (A, 0, 2)), ((ES ::= A \cdot \mathbf{d}, 0), \mathbf{b}, (A, 0, 2)),$
$((ES ::= A \cdot B, 0), \mathbf{a}, (A, 0, 1)), ((ES ::= A \cdot B, 0), \mathbf{a}, (A, 0, 2)), ((ES ::= A \cdot B, 0), \mathbf{b}, (A, 0, 2)),$
$((EA ::= A \cdot \mathbf{a}, 0), \mathbf{a}, (A, 0, 1)), ((EA ::= A \cdot \mathbf{a}, 0), \mathbf{a}, (A, 0, 2)), ((EA ::= A \cdot \mathbf{a}, 0), \mathbf{b}, (A, 0, 2)),$
$((EA ::= A \cdot \mathbf{a}, 0), \mathbf{b}, (A, 0, 4)), ((ES ::= A \cdot \mathbf{d}, 0), \mathbf{b}, (A, 0, 4)), ((ES ::= A \cdot B, 0), \mathbf{b}, (A, 0, 4)),$
$((EA ::= A \cdot \mathbf{a}, 0), \mathbf{b}, (A, 0, 5)), ((EA ::= A \cdot \mathbf{a}, 0), \mathbf{d}, (A, 0, 5)), ((ES ::= A \cdot \mathbf{d}, 0), \mathbf{b}, (A, 0, 5)),$
$((ES ::= A \cdot \mathbf{d}, 0), \mathbf{d}, (A, 0, 5)), ((ES ::= A \cdot B, 0), \mathbf{b}, (A, 0, 5)), ((ES ::= A \cdot B, 0), \mathbf{d}, (A, 0, 5)),$
$((ES ::= AB\cdot, 5), \$, (B, 5, 6)), ((ES ::= AB\cdot, 4), \$, (B, 4, 6))\}$

The final output ESPPF is then as given in Figure 3.8.

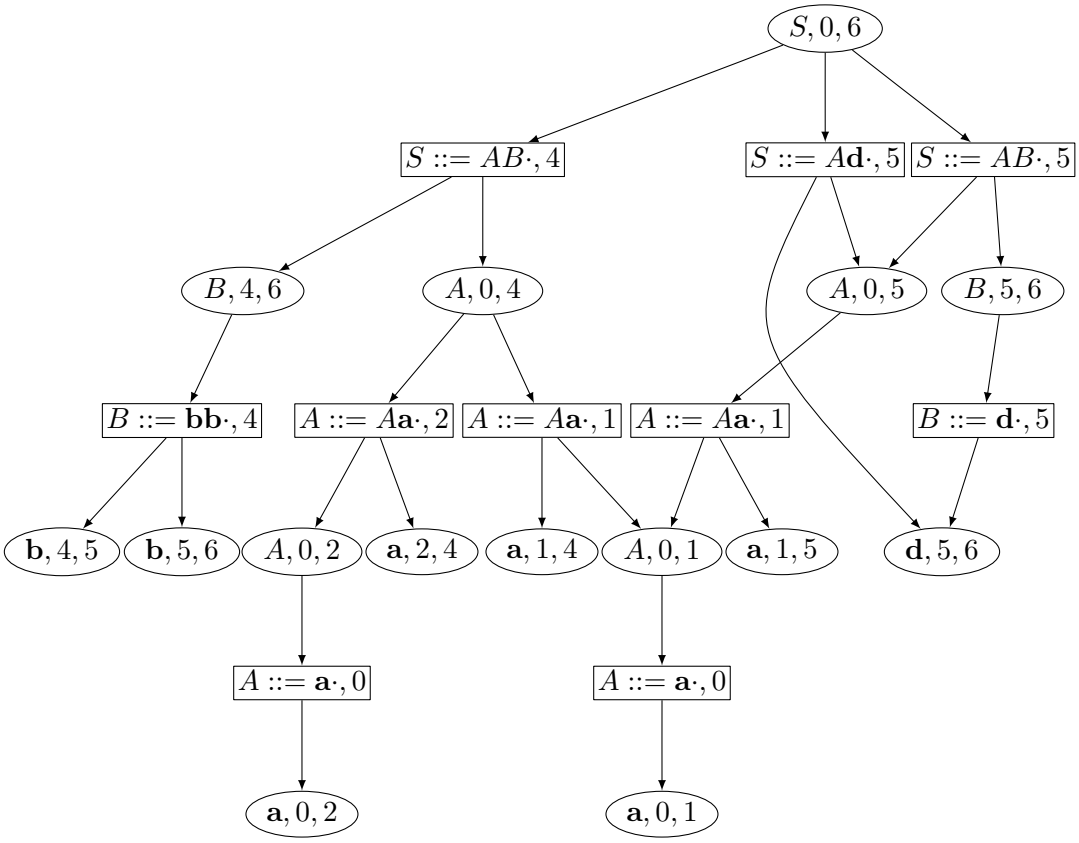


Figure 3.8: Final ESPPF produced for this example

Chapter 4

Abstract Syntax Conversion with GIFT Operators

Syntactic structure is usually the basis of semantic interpretation. In early programming language design, it was often assumed that semantics would be interpreted as the string was parsed, due to the memory constraints of early computers. In contemporary language design, where resources are more readily available, it is generally assumed that the result of parsing a string is a structural representation of the string, which is then transformed into a more suitable semantics form. This form could be some tree-like structure, or it could be a collection of data structures for modelling the semantics. This internal form is an *abstraction* of the language in which language concepts are presented in a concise manner.

This internal abstraction of the language is called an *abstract syntax* - which may or may not be grammar-based. The grammar that is used for parsing is then called a *concrete syntax*. This chapter considers a style of abstract syntax called a *structural abstract syntax* and examines a set of operators and its extension (the TIF and GIFT operators respectively) for specifying a concrete to abstract syntax conversion. Chapter 6 will later describe how the GIFT operators are used to perform a translation of derivation trees in the C# concrete syntax into trees in a structural abstract syntax for C# designed as part of the PLanCompS [Mos+15] project.

4.1 Models of Abstract Syntax

Abstract syntax forms range from those closely based on derivation trees to those which impose no particular structure.

Attribute grammars [Knu68] specify semantics in terms of attributes associated

with grammar elements. The semantics results from computation over trees, and the initial tree is a derivation tree in the concrete syntax. As such, there is no separate abstract syntax. An attribute grammar can define the complete semantics of a language. Alternatively, an attribute grammar can be used as an intermediate step, describing the translation from the derivation tree to some internal form. Later, an attribute grammar style specification is used as the basis of TIF [JS10] and GIFT transformations.

The idea of abstraction, in general, is to reduce complex structures into a (conceptually) simpler form. A derivation tree itself is already an abstraction from derivations - for instance, a derivation tree does not distinguish between a left-most and right-most derivation.

An example of an unstructured abstract syntax is seen in the Vienna Development Method (VDM) [BJ78]. VDM includes a formal language that supports high-level abstractions of mathematical concepts such as sets and maps. It can be used to model and validate the semantics of software systems, before production implementation. The role of the concrete structure is de-emphasised, focussing only on the abstractions.

Although the compiler architecture flowchart given at the start of Chapter 1 shows a simple progression from syntactic to semantic analysis, and then to intermediate code generation, this in itself is an abstraction. In practice, the transition between the phases involves a chain of intermediate abstract syntax conversions. For example, in the GNU C Compiler [SG15], after parsing, a tree-based representation is constructed before being transformed into a sequence of three-address instructions. This sequence is then itself transformed into the low level architectural independent representation. This representation is then used to construct the architectural specific representation. This provides a simpler interface for the compiler to perform optimisations before producing the final semantics.

4.2 Structural Abstract Syntax for Structural Operational Semantics

Plotkin [Plo04] introduced the idea of specifying operational semantics in terms of logical inference rules applied over the programming language syntax. Inference rules must be defined for every syntactic form, so it is beneficial to reduce the number of syntactic forms. The starting point for structural operational semantics specifications is usually an abstract syntax tree that is the derivation tree of some structurally simpler, abstract syntax grammar.

The idea is to take the non-terminals which represent syntactic categories in the concrete syntax and map these into a smaller set of matching semantics categories

- for example, grouping together semantically similar constructs under a single non-terminal. Overbey and Johnson [OJ09] identified common transformations one would wish to make when constructing an abstract syntax. Some of these include

- Omission of keywords that represent syntactic sugar
- Relabelling symbols to distinguish elements of the same production
- Merging symbols in different productions with similar semantics
- Renaming symbols to better document the semantic intent
- Converting recursive structures into list structures
- Removal of non-terminals that are necessary for unambiguous parsing, but not for semantics

The specification of a concrete to abstract syntax transformation may be done informally, or through more principled techniques such as term rewriting [KBV01; BN98].

In the rest of this thesis, abstract syntax will be taken to mean a structural abstract syntax grammar. The transformation operators that are defined in this chapter have a derivation tree to abstract syntax tree interpretation, but may also be thought of as grammar to grammar translations.

4.2.1 The relationship between concrete and structural abstract syntax

The structural abstract syntax grammar may be thought of as a structurally more concise version of the concrete syntax. Such an abstract syntax is unlikely to be suitable for near-deterministic parsing techniques as they are often ambiguous. For example, a common concrete to abstract syntax transformation is to take the subgrammar for expressions which are usually of form

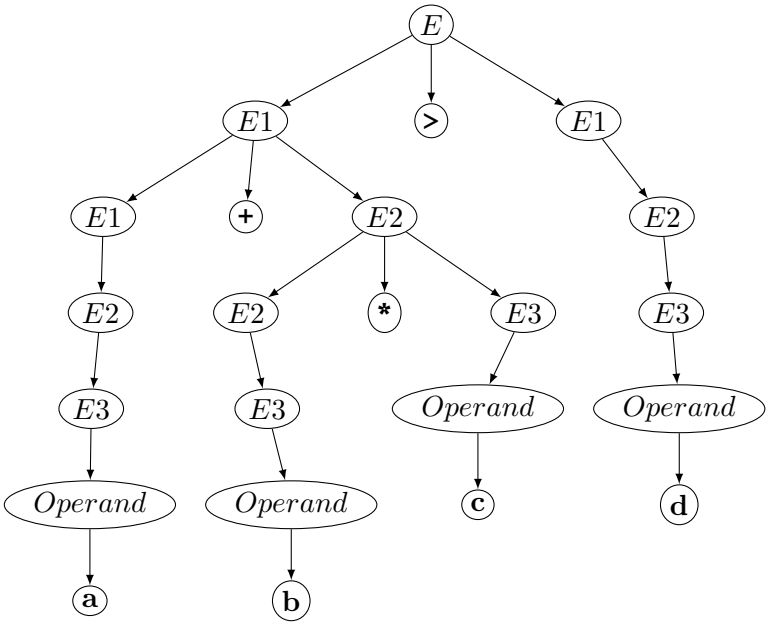
$$
\begin{aligned}
E &::= E1 \mid E1 \texttt{<} E1 \mid E1 \texttt{>} E1 \\
E1 &::= E2 \mid E1 \texttt{+} E2 \mid E1 \texttt{-} E2 \\
E2 &::= E3 \mid E2 \texttt{*} E3 \mid E2 \texttt{/} E3 \\
E3 &::= \mathit{Operand} \mid \texttt{(}\ E\ \texttt{)} \\
\mathit{Operand} &::= \mathbf{a} \mid \mathbf{b} \mid \ldots
\end{aligned}
$$

and translate it into an abstract syntax pattern of form

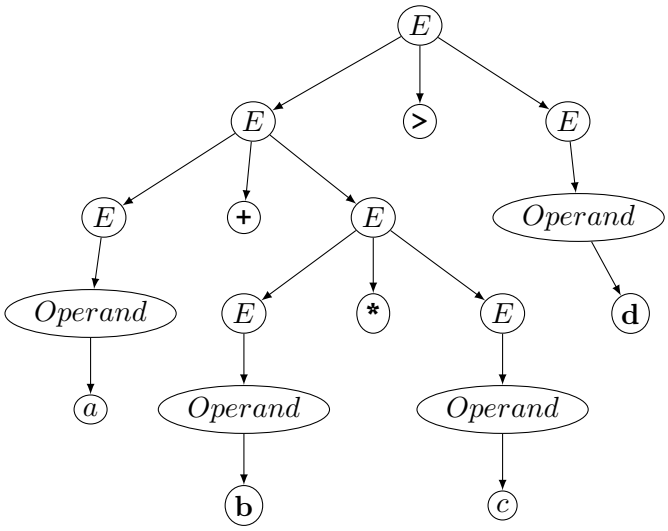
$$E ::= E\ Op\ E \mid Operand$$

$$Op ::= \texttt{+} \mid \texttt{-} \mid \texttt{*} \mid \texttt{/} \mid \texttt{>} \mid \texttt{<}$$
$$Operand ::= \mathbf{a} \mid \mathbf{b} \mid \ldots$$

The latter (abstract) syntax rule pattern supports compact semantics specification but is not very useful as a parsing grammar without some well-defined disambiguation strategy. Whilst the concrete syntax rule pattern for E makes the order of evaluation explicit through the structure, the abstract syntax leaves this open. For the string `a+b*c>d`, the concrete syntax generates the following derivation tree



A parser for the abstract syntax generates many derivation trees. There is one derivation tree in the abstract syntax that is equivalent in structure to the derivation tree for the concrete syntax:



This tree has a similar shape to the tree for the concrete syntax except nodes corresponding to non-terminals which are used to determine the evaluation order are removed. Since this tree makes the correct evaluation order explicit whilst being structurally concise, it is the tree required as the starting point for structurally defined semantics.

In structurally defined semantics, the expectation is that the concrete syntax is the grammar that will parse the string and from which a derivation tree will be obtained. This derivation tree is then transformed, using the operators which will be described in this chapter, into a derivation tree in the abstract syntax. One could view the concrete syntax parser as being the disambiguation strategy for the many abstract syntax parses.

4.3 Derivation Tree Manipulation

To obtain the preferred abstract syntax tree (AST) from a given concrete syntax derivation tree, transformations are performed on the nodes of the derivation tree. It is generally required to transform the structure of the tree, for instance by adding or removing nodes or subtrees.

4.3.1 Node and subtree removal

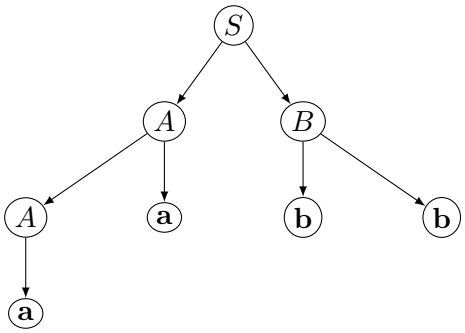
If a single node is deleted, then its children could be reattached to some other part of the tree. Possible removal transformations are considered in this section.

Consider the grammar

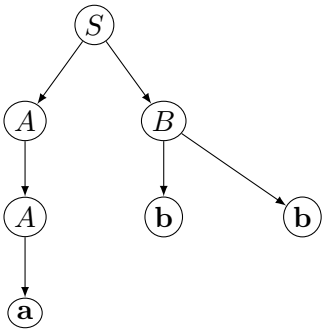
$$S ::= A\,\mathbf{d} \mid A\,B$$

$$A ::= A\,\mathbf{a} \mid \mathbf{a}$$
$$B ::= \mathbf{b}\,\mathbf{b} \mid \mathbf{d}$$

A derivation tree for the string `aabb` is

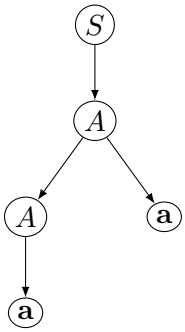


If a node, x, has no children, then removing that node involves simply deleting it. For example, if one were to remove the second **a** in the above tree then the resulting tree would be

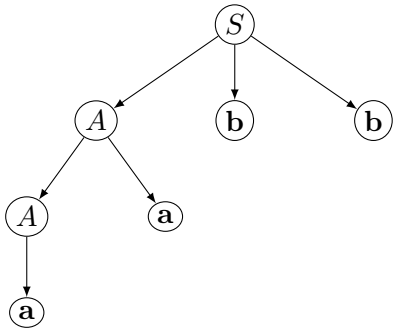


Now suppose x has children. There are several different ways to handle the children when x is deleted.

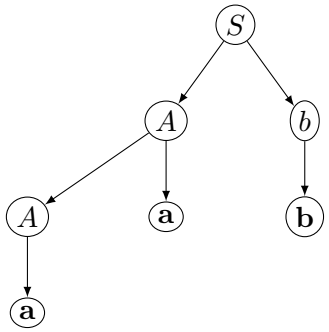
- The entire tree rooted at x could be removed (x is *torn* from the tree). For the above example, if the node B was torn, then the result would be



- x could be removed with the children of x being attached to the parent of x (x is *folded under* its parent). For the same example, if the node B was folded under S, then the result would be



- One child, y, of x is distinguished. When x is removed from the tree, it is relabelled y and inherits the children of y (y is folded over x). For the same example, if the left-most b node is distinguished and is folded over B then the result would be

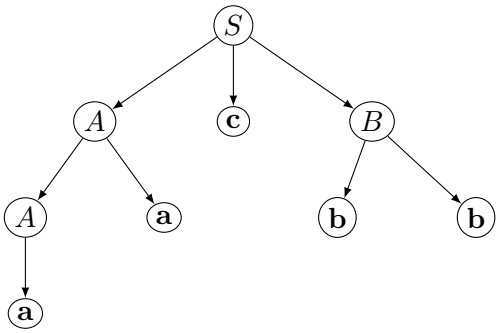


  The B node has been replaced by the left-most **b** node. Since only one child could possibly be the successor, only a single distinguished child is allowed.

The root node may not be sensibly removed with tear or fold-under. Removing the entire tree rooted at S would, of course, result in the empty tree, and since the root node by definition has no parent, its children could not be attached to a parent.

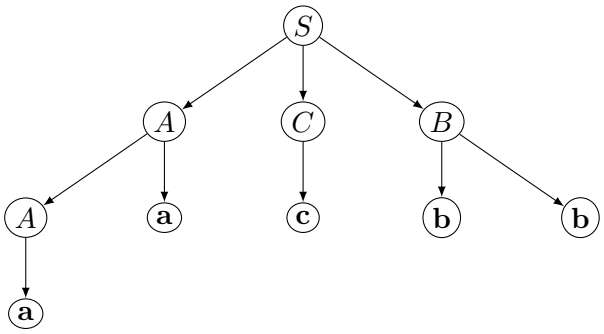
4.3.2 Subtree insertion

For a subtree that comprises of a single node, insertion is straightforward. One simply adds a new node amongst the list of existing children. For example, one could insert a **c** node between the A and B children of S in the above example to obtain the tree

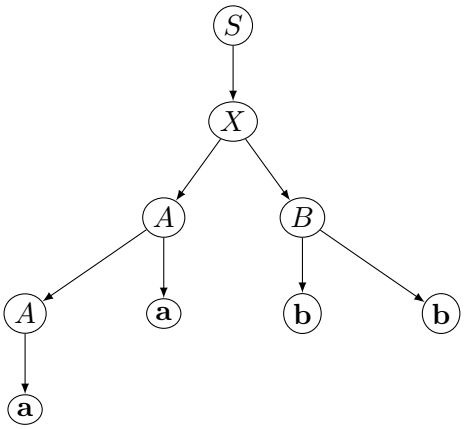


Some less trivial node addition operations include:

- The *insertion* of an entire tree rooted at some node y amongst the list of children for a node X. For the previous example, one could add a new node C, which has **c** as a child:



- Take one or more children of X and add them as children of a new node Y (Y *gathers* some children of X). This Y then takes the place of these children under X. In the running example, consider creating a new node, X, under S, that gathers the nodes A and B



One could easily envisage transformations that go beyond these, but only local tree transformation will be considered for this thesis. First, the Tear-Insert-Fold (TIF) operators proposed in [JS10] shall be reviewed, and potential problems with the interaction between operations will be noted. The Gather-Insert-Fold-Tear (GIFT) formalism, which modifies and extends TIF, will then be described. In Chapter 6, it will be shown how the GIFT operators may be used to transform the C# 1.2 parsing grammar from the language specification into a structural abstract syntax, developed by the PLanCompS [Mos+15] project, used in the formal semantics of C# 1.2.

4.4 TIF Operators on Derivation Trees

The Tear-Insert-Fold (TIF) operators were originally implemented in RDP [JS98] but were formalised in [JS10]. This formalisation of the TIF operators will be discussed here as the background for the GIFT operators discussed later. TIF provides a set of annotations applied to symbols on the right-hand side of productions. For a production of form $X ::= x_1 x_2 \dots x_n$, an element x_i can be annotated as follows

- x_i (no annotation) - The node corresponding to this symbol is constructed 'as is' in the derivation tree
- $x_i^{\wedge}$ (fold-under) - The node constructed for x_i is removed and the children of x_i are attached to the node constructed for X.
- $x_i^{\wedge\wedge}$ (fold-over) - As for fold-under, but the node X is relabelled x_i. Only one node may relabel X, so the right-most fold-over is chosen to relabel X, with the rest converted to fold-under.
- $x_i^{\wedge\wedge\wedge}$ (tear) - The tree rooted at the node constructed by x_i is removed. Note, the original RDP formulation prescribed very different semantics to the $\wedge\wedge\wedge$, this former semantic interpretation was discarded in the TIF formalism.
- $[x_i]$ (insert) - The tree rooted at x_i is inserted at this position.

There is a natural correspondence between these operators and the transformations described in the previous section. Fold-under, fold-over and tear are operations that remove nodes from the tree. Fold-under removes the node corresponding to the annotated symbol, fold-over removes the parent of the node corresponding to the annotated symbol, whilst tear removes the tree rooted at the node corresponding to the annotated symbol.

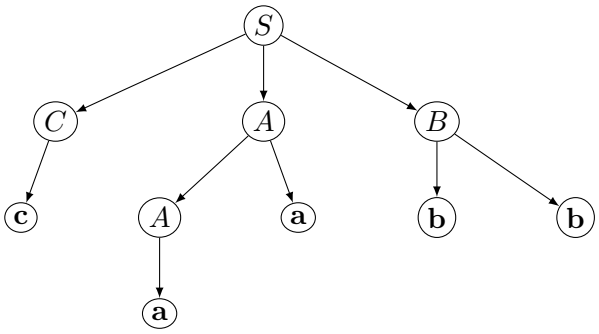
In TIF, insert is an operation that can insert a subtree which has been torn from elsewhere in the current derivation. If a symbol is annotated with a name using the

form $x_i^{\wedge\wedge\wedge} : id$, then the torn tree is stored and an occurrence of $[id]$ means insert the tree named id here.

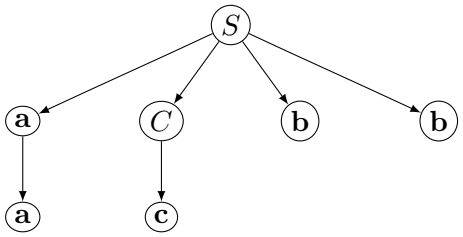
For example, consider the annotated grammar

$$
\begin{aligned}
S &::= A\ \mathbf{d} \mid C^{\wedge\wedge\wedge} : t\ A\ [t]\ B^{\wedge} \\
A &::= A\ \mathbf{a}^{\wedge\wedge} \mid \mathbf{a}^{\wedge\wedge} \\
B &::= \mathbf{b}\ \mathbf{b} \mid \mathbf{d} \\
C &::= \mathbf{c}
\end{aligned}
$$

The derivation tree for the string `caabb` is



The result of applying the node operations gives the AST



The tree rooted at the node C under S is torn with the identifier t, and then is reinserted after node A. The node for A under S is replaced by its second child, and likewise, its first child is also replaced by that node's child. The node for B under S is removed and its children attached under S.

A syntax-directed definition of the TIF operators can be made using an attribute grammar. Given a production rule $X ::= x_1 \ldots x_n$, each x_i has three attributes, a tree *node*, a string *label*, and a sequence of tree node *children*. A function $newNode()$ constructs and returns a new tree node. If x_i denotes an unannotated terminal then the following are defined

$$x_i.node = newNode() \quad x_i.label = x_i \quad x_i.children = \emptyset$$

For each grammar rule $X ::= x_1 x_2 \ldots x_n$, the following are defined

$$X.node = newNode() \quad X.label = Z$$
$$X.children = (u_1, \ldots, u_n)$$

where $u_i = x_i.node$ if x_i is unannotated, $u_i = x_i.children$ if x_i is annotated with $^\wedge$ or $^{\wedge\wedge}$, and u_i is the empty sequence if x_i is annotated with $^{\wedge\wedge\wedge}$. Additionally, for each $x_i^{\wedge\wedge\wedge} : id$, an additional attribute labelled id is stored

$$id = x^i.node$$

$[id]$ moves the node stored in id to a new position in the list of children. In the list of children for X, u_T such that $1 \leq T \leq n$ represents the position of the insertion. Then $u_T = id$. If there is a x_i annotated with $^{\wedge\wedge}$ then $Z = x_i.label$, otherwise $Z = X$.

For the example annotated grammar on page 95, the corresponding syntax-directed definition is

$S ::= A\ \mathbf{d}$
$\{S.node = getNode() \quad S.label = S \quad S.children = (A.node, \mathbf{d}.node)$
$\mathbf{d}.node = getNode() \quad \mathbf{d}.label = \mathbf{d} \quad \mathbf{d}.children = \emptyset\}$
$S ::= C^{\wedge\wedge\wedge} : tA\ [t]\ B^\wedge$
$\{S.node = getNode() \quad S.label = S$
$t = C.node \quad S.children = (A.node, t, B.children)\}$

$A ::= A^1\ \mathbf{a}^{\wedge\wedge}$
$\{A.node = \mathbf{a}.node \quad A.label = \mathbf{a}.label \quad A.children = (A^1.node)\}$
$A ::= \mathbf{a}^{\wedge\wedge}$
$\{A.node = \mathbf{a}.node \quad A.label = \mathbf{a}.label \quad A.children = \emptyset\}$

$B ::= \mathbf{b}^1\ \mathbf{b}^2$
$\{B.node = getNode() \quad B.label = B \quad B.children = (\mathbf{b}^1.node, \mathbf{b}^2.node)$
$\mathbf{b}^1.node = getNode() \quad \mathbf{b}^1.label = \mathbf{b} \quad \mathbf{b}^1.children = \emptyset$
$\mathbf{b}^2.node = getNode() \quad \mathbf{b}^2.label = \mathbf{b} \quad \mathbf{b}^2.children = \emptyset\}$
$B ::= \mathbf{d}$
$\{B.node = getNode() \quad B.label = B \quad B.children = (\mathbf{d}.node)$

$$\mathbf{d}.node = getNode() \quad \mathbf{d}.label = \mathbf{d} \quad \mathbf{d}.children = \emptyset\}$$

$$C ::= c$$

$$\{C.node = getNode() \quad C.label = C \quad C.children = (\mathbf{c}.node)$$
$$\mathbf{c}.node = getNode() \quad \mathbf{c}.label = \mathbf{c} \quad \mathbf{c}.children = \emptyset\}$$

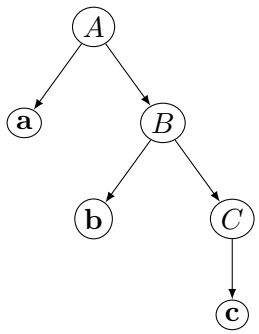
In this thesis, only annotations on BNF grammars are considered, since for EBNF the appropriate semantics for operators, such as folds under closure operations, are unclear. This is partly because there is no formally agreed upon standard as to how closures are represented in derivation trees (for instance, RDP [JS98] produces derivation trees where the items in a closure are represented at the top-level whilst the SDF formalism [Hee+89] considers closures as anonymous non-terminals).

4.4.1 Evaluation order and interaction of operators

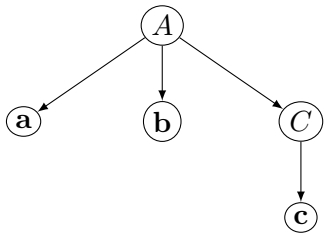
The TIF operators were designed to provide a small set of transformations that would be easy for language engineers to reason about - as they perform only local changes to the derivation tree - local here meaning that a node only interacts with its parent. Unfortunately, the evaluation algorithm in the RDP interpretation for folds given in [JS98] causes the impact of some fold operations to be dependent on fold operations in other grammar rules, and thus is not locally determined. Consider

$$A ::= \mathbf{a}\, B^{\wedge}$$
$$B ::= \mathbf{b}\, C^{\wedge\wedge}$$
$$C ::= \mathbf{c}$$

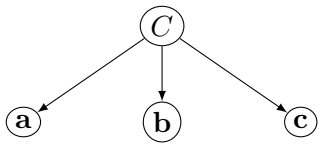
The derivation tree for the string `abc` is



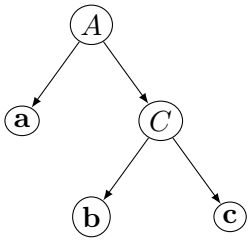
If the fold-under operator on B was applied first, then an intermediate tree would be



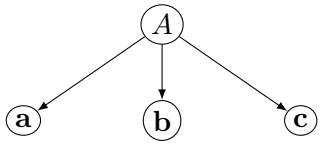
and applying the fold-over operator on C would result in the tree



Alternatively, if the fold-over operator on C was applied first, then the intermediate tree would be



and then the final resulting tree would be



In the first case, the combination causes C to replace its former parent's parent which breaches the locality principle of the TIF operators. The second interpretation maintains this locality.

The syntax-directed definition described in the introduction to this section works to the second interpretation, and thus maintains locality.

$$A ::= \mathbf{a}\, B^{\wedge}$$
$$\{A.node = getNode() \quad A.label = A \quad A.children = (\mathbf{a}.node, B.children)$$
$$\mathbf{a}.node = getNode() \quad \mathbf{a}.label = \mathbf{a} \quad \mathbf{a}.children = \emptyset\}$$
$$B ::= \mathbf{b}\, C^{\wedge\wedge}$$
$$\{B.node = C.node \quad B.label = C.label \quad B.children = (\mathbf{b}.node, C.children)$$
$$\mathbf{b}.node = getNode() \quad \mathbf{b}.label = \mathbf{b} \quad \mathbf{b}.children = \emptyset\}$$

$$C ::= \mathbf{c}$$
$$\{C.node = getNode() \quad C.label = C \quad C.children = (\mathbf{c}.node)$$
$$\mathbf{c}.node = getNode() \quad \mathbf{c}.label = \mathbf{c} \quad \mathbf{c}.children = \emptyset\}$$

The interpretation of the TIF operators given in [JS10] uses the interpretation in the syntax directed definition above, and thus can only ever match the second interpretation.

4.5 The GIFT Operators

When discussing structural abstract syntax grammars, the usefulness of merging syntactic categories under a single semantic category was noted. An example was given with the expression grammar on page 88. More generally, for the purposes of formal semantic definition, it may be useful to bring together non-terminals with similar sublanguages in the grammar. For instance, it is not unusual for languages to have different non-terminals for different types of declarations. For example, the C# grammars in appendices A and B have separate non-terminals for class member declarations and struct member declarations, where the only distinction is that structs cannot have a destructor. The formal semantics, on the other hand, may wish to treat these declarations uniformly, relying on the concrete syntax grammar to filter structs with destructors. Therefore, when transforming concrete derivations to abstract syntax interpretations, being able to merge subtrees under a new non-terminal is often desired.

In this section, the TIF notation is extended to include a new operator, *gather*. When processed, the nodes for $x_i \ldots x_{i_p}$ are replaced by an instance of a node labelled Y and reattached as children of that instance of Y. Y may be a new or a pre-existing non-terminal.

Formally, the gather annotation defines a new symbol instance Y to be defined with one or more children

$$Y.node = newNode() \quad Y.label = Y \quad Y.children = (u_i, \ldots, u_{i_p})$$

For a rule of form $X ::= x_1 x_2 \ldots x_n$ this is denoted by $(x_i \ldots x_{i_p})!Y$. Where the children of X would normally be

$$X.children = (u_1, \ldots, u_i, \ldots, u_{i_p}, \ldots u_n)$$

this is redefined by the gather operator as simply

$$X.children = (u_1, \ldots, Y.node, \ldots, u_n)$$

$u_i, \ldots, u_{i_p}$ become the children of Y, and Y takes the places of these nodes in the children of X.

This allows for symbols that are operands of gather to themselves have annotations. $u_i, \ldots, u_{i_p}$ are defined according to the annotation on their corresponding symbols. In addition, a subsequence of this sequence of symbols may also be annotated with gather, and this behaves as if Y is the parent. For every $(x_i \ldots (x_h \ldots x_k)!W \ldots x_{i_p})!Y$, an extra symbol is defined on top of the definitions for Y

$$W.node = newNode() \quad W.label = W$$

and then the children of each node are defined as such

$$\begin{aligned} X.children &= (u_1, \ldots, Y.node, \ldots, u_n) \\ Y.children &= (u_i, \ldots, W.node, \ldots, u_{i_p}) \\ W.children &= (u_h, \ldots, u_k) \end{aligned}$$

As fold-over modifies not just the list of children but the label of the parent, a fold-over annotation within a gather is not permitted in the formulation described in this thesis, as it is unclear whether X should be relabelled, or Y.

Given the example GIFT-annotated grammar

$$\begin{aligned} S &::= A\ \mathbf{d} \mid (C^{\wedge\wedge\wedge} : t\ A\ [t]\ B^{\wedge})!A \\ A &::= A\ \mathbf{a}^{\wedge\wedge} \mid \mathbf{a}^{\wedge\wedge} \\ B &::= (\mathbf{b}!W\ \mathbf{b})!D \mid \mathbf{d} \\ C &::= \mathbf{c} \end{aligned}$$

The corresponding syntax-directed definition is as follows

$$\begin{aligned} &S ::= A\ \mathbf{d} \\ &\quad \{S.node = getNode() \quad S.label = S \quad S.children = (A.node, \mathbf{d}.node) \\ &\quad \mathbf{d}.node = getNode() \quad \mathbf{d}.label = \mathbf{d} \quad \mathbf{d}.children = \emptyset\} \\ &S ::= (C^{\wedge\wedge\wedge} : tA^1\ [t]\ B^{\wedge})!A^2 \\ &\quad \{S.node = getNode() \quad S.label = S \end{aligned}$$

$$t = C.nodeS.children = (A^1.node)$$
$$A^1.node = getNode() \quad A^1.label = A$$
$$A^1.children = (A^2.node, t, B.children)\}$$

$$A ::= A^1\ \mathbf{a}^{\wedge\wedge}$$
$$\{A.node = \mathbf{a}.node \quad A.label = \mathbf{a}.label \quad A.children = (A^1.node)\}$$

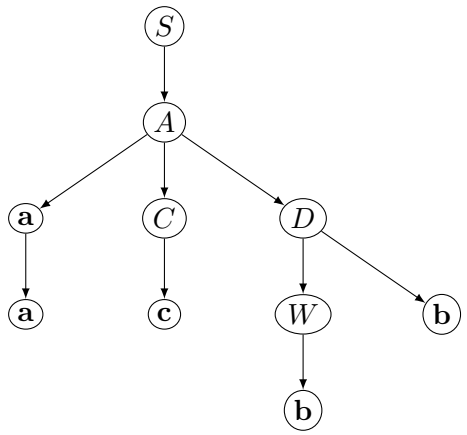
$$A ::= \mathbf{a}^{\wedge\wedge}$$
$$\{A.node = \mathbf{a}.node \quad A.label = \mathbf{a}.label \quad A.children = \emptyset\}$$

$$B ::= (\mathbf{b}^1!W\ \mathbf{b}^2)!D$$
$$\{B.node = getNode() \quad B.label = B \quad B.children = (D.node)$$
$$\mathbf{b}^1.node = getNode() \quad \mathbf{b}^1.label = \mathbf{b} \quad \mathbf{b}^1.children = \emptyset$$
$$\mathbf{b}^2.node = getNode() \quad \mathbf{b}^2.label = \mathbf{b} \quad \mathbf{b}^2.children = \emptyset$$
$$W.node = getNode() \quad W.label = W \quad W.children = (\mathbf{b}^1.node)$$
$$D.node = getNode() \quad D.label = D \quad D.children = (W.node, \mathbf{b}^2.node)\}$$

$$B ::= \mathbf{d}$$
$$\{B.node = getNode() \quad B.label = B \quad B.children = (\mathbf{d}.node)$$
$$\mathbf{d}.node = getNode() \quad \mathbf{d}.label = \mathbf{d} \quad \mathbf{d}.children = \emptyset\}$$

$$C ::= \mathbf{c}$$
$$\{C.node = getNode() \quad C.label = C \quad C.children = (\mathbf{c}.node)$$
$$\mathbf{c}.node = getNode() \quad c.label = \mathbf{c} \quad \mathbf{c}.children = \emptyset\}$$

The result of applying the node operations on the derivation tree for the string `caabb` (seen on page 95) gives the AST

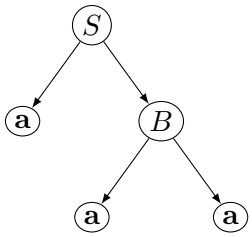


By combining the fold-under and gather operators, it is possible to perform an action equivalent to renaming a symbol. This is used in a few areas of the C# 1.2 case

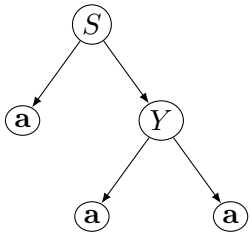
study, which will be discussed in Chapter 6. Consider the annotated grammar

$$S ::= \mathbf{b}\ B^{\wedge}$$
$$B ::= (\mathbf{aa})!Y$$

The derivation tree for the string `baa` is



The resulting tree after applying all transformations will be the following



This combination of operations is effectively equivalent to relabelling the B node into Y.

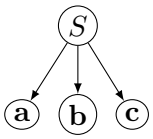
This section has examined a set of simple operators that allow a language designer to define the structure of abstract syntax trees over a concrete syntax in terms of tree-to-tree transformations. A semantics designer may also need a grammar that can act as the basis for inference rules. In the next section, the GIFT operators are considered as grammar-to-grammar transformations.

4.6 Grammar-to-Grammar Transformation

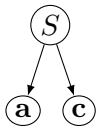
It is relatively easy to see that, for some GIFT derivation tree to AST operation, there is a corresponding operation that takes the concrete syntax and transforms it into an abstract syntax such that the AST is a derivation tree in the abstract syntax. For instance, the application of fold-under to a terminal instance removes the corresponding terminal nodes from derivation trees. This corresponds to removing that terminal instance from the grammar. If one considers the following GIFT-annotated grammar, Γ

$$S ::= \mathbf{a}\ \mathbf{b}^{\wedge}\ \mathbf{c}$$

the derivation tree of the string abc for this Γ is



Applying the fold-under operator transforms this into the tree



The transformation of Γ into a new grammar Γ' is then

$$S ::= a\ c$$

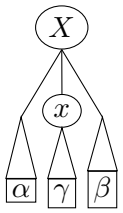
and the second tree above is clearly a derivation tree in Γ'.

Of course, for a finite language, one way to generate this Γ' is to enumerate all the transformed trees that are generated by Γ and construct a grammar which is the union of the most trivial grammars that generate these trees. As many languages are non-finite, this would not be possible, in general.

The following section shall consider an operational approach for transforming Γ into Γ', through inductive steps defined over each GIFT operator.

4.6.1 The Fold-Under Operator

Derivations using a production of the form $X ::= \alpha x \hat{} \beta$, where $x \in N \cup T \cup \{\epsilon\}$ and $\alpha, \beta \in (N \cup T \cup \{\epsilon\})^*$ will have derivations trees with subtrees of the form



where γ represents one of the right-hand sides of x.

After applying the fold-under operator, this will be transformed into

The corresponding abstract syntax will replace the rule $X ::= \alpha x \beta$ with rules such that for each right-hand side, $\gamma_1, \gamma_2, \ldots, \gamma_n$ of x

$$\begin{aligned} X &::= \alpha\gamma_1\beta \\ X &::= \alpha\gamma_2\beta \\ &\vdots \\ X &::= \alpha\gamma_n\beta \end{aligned}$$

Embedded fold-under operators

The symbols of some γ that are the right-hand side of x may themselves include fold-under operators, in other words, there may be productions of the form $x ::= \eta y^{\wedge} \zeta$. In this case, the working grammar will simply have a production of the form

$$X ::= \alpha\eta y^{\wedge}\zeta\beta$$

which is processed subsequently.

A production of the form

$$X ::= \alpha X^{\wedge} \beta$$

constructs a new abstract syntax rule in the working grammar

$$X ::= \alpha\alpha X^{\wedge}\beta\beta$$

For this case, it is difficult to find a BNF context-free grammar whose derivations trees exactly match the generated ASTs. This issue will be returned to in 4.6.6.

Example

Given

$$\begin{aligned} S &::= A\ \mathbf{d} \mid C\ A\ B^{\wedge} \\ A &::= A\ \mathbf{a} \mid \mathbf{a}^{\wedge} \\ B &::= \mathbf{b}\ C^{\wedge}\ \mathbf{b} \mid \mathbf{d} \end{aligned}$$

$$C ::= \mathbf{c}$$

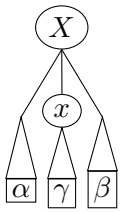
For the rule $A ::= \mathbf{a}^{\wedge}$ the resulting rule in the abstract syntax will be $A ::= \epsilon$. In the case of the rule $S ::= C\ A\ B^{\wedge}$, B corresponds to two rules - $B ::= \mathbf{b}\ C^{\wedge}\ \mathbf{b}$ and $B ::= \mathbf{d}$. As a result, two additional rules are generated for the abstract syntax. $C^{\wedge}$ will then transform one of these rules as well as the existing rule into the final form. The resulting abstract syntax will be

$$\begin{aligned}
S &::= A\ \mathbf{d} \mid C\ A\ \mathbf{b}\ \mathbf{c}\ \mathbf{b} \mid C\ A\ \mathbf{d} \\
A &::= A\ \mathbf{a} \mid \epsilon \\
B &::= \mathbf{b}\ \mathbf{c}\ \mathbf{b} \mid \mathbf{d} \\
C &::= \mathbf{c}
\end{aligned}$$

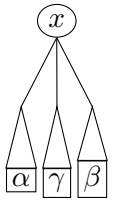
Note that the rule for B can now be deleted as B is no longer reachable.

4.6.2 The Fold-over Operator

The fold-over operator is similar to the fold-under operator. Derivations using a production of the form $X ::= \alpha x^{\wedge\wedge}\beta$, where $x \in N \cup T \cup \{\epsilon\}$ and $\alpha, \beta \in (N \cup T \cup \{\epsilon\})^*$ will have derivation trees with subtrees of the form



After application of the fold-over operator, this will be transformed to



The abstract syntax will contain rules corresponding to each right hand side $\gamma_1, \gamma_2, \ldots, \gamma_n$. As the fold-over operator changes the parent of the node, grammar rules need to be generated for each annotated x

$$\begin{aligned}
x &::= \alpha\gamma_1\beta \\
x &::= \alpha\gamma_2\beta
\end{aligned}$$

$$\vdots$$
$$x ::= \alpha\gamma_n\beta$$

In the tree transformation, the node labelled X may be the child of another node, W. After transformation x becomes the new child of W. In the grammar transformation, this needs to be reflected. For example, in the annotated grammar

$$S ::= \mathbf{a}\, X\, \mathbf{b}$$
$$X ::= \mathbf{c}\, Y^{\wedge\wedge} \mid \mathbf{f}$$
$$Y ::= \mathbf{d}$$

One rule for X contains a symbol annotated with fold-over, whilst the other rule does not. The resulting abstract grammar will need to be

$$S ::= \mathbf{a}\, Y\, \mathbf{b} \mid \mathbf{a}\, X\, \mathbf{b}$$
$$Y ::= \mathbf{c}\, \mathbf{d}$$
$$X ::= \mathbf{f}$$

For every x^i on the right-hand side of X annotated with fold-over, a set of $exp(x^i)$, Δ_X, is maintained. X itself is also in this set if there exists a right-hand side of X that does not contain a symbol with a fold-over annotation.

After all other transformations are applied, for all instances of X on right-hand sides of production in P, new rules are created for each $x_i \in \Delta_X$. For productions of form $Y ::= \eta X \zeta$, these new rules will be

$$Y ::= \eta x_1 \zeta$$
$$\vdots$$
$$Y ::= \eta x_p \zeta$$

If Δ_S contains more than one symbol, a new start symbol, S', is created, and for each $\sigma \in \Delta_S$, a rule $S' ::= \sigma$ is created.

Embedded Fold-over operators

As with the fold-under operator, the symbols of some γ that is the right-hand side of x may themselves include fold-over operators. Similar to the fold-under over operator,

the working grammar will simply have a production of the form

$$x ::= \alpha\eta y^{\wedge\wedge}\zeta\beta$$

which is processed subsequently.

The fold-over operator also has the same problem with recursion. This will be returned to in 4.6.6.

Example

Consider the annotated grammar

$$\begin{aligned} S &::= A\,\mathbf{d} \mid C\,A\,B^{\wedge\wedge} \\ A &::= A\,\mathbf{a}^{\wedge\wedge} \mid \mathbf{a}^{\wedge\wedge} \\ B &::= \mathbf{b}\,\mathbf{b} \mid \mathbf{d} \\ C &::= \mathbf{c} \end{aligned}$$

The rule $A ::= A\mathbf{a}^{\wedge\wedge}$ will resolve to a rule $a ::= A$. a is added to Δ_A. The rule $A ::= \mathbf{a}^{\wedge\wedge}$ will resolve to a rule $a ::= \epsilon$, as a is already in Δ_A it is not added again.

In the rules of S, the rule $S ::= C\,A\,B^{\wedge\wedge}$ will resolve to rules $B ::= C\,A\,\mathbf{b}\,\mathbf{b}$ and $B ::= C\,A\,\mathbf{d}$. B is added to Δ_S. As $S ::= A\,\mathbf{d}$ is not annotated with any fold-over operator, S is also added to Δ_S. After resolving all operators, one gets

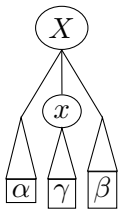
$$\begin{aligned} S &::= A\,\mathbf{d} \\ a &::= A \mid \epsilon \\ B &::= C\,A\,\mathbf{d}\,\mathbf{d} \mid C\,A\,\mathbf{d} \mid \mathbf{b}\,\mathbf{b} \mid \mathbf{d} \\ C &::= \mathbf{c} \end{aligned}$$

This is then transformed into

$$\begin{aligned} S' &::= S \mid B \\ S &::= a\,\mathbf{d} \\ a &::= a \mid \epsilon \\ B &::= C\,a\,\mathbf{d}\,\mathbf{d} \mid C\,a\,\mathbf{d} \mid \mathbf{b}\,\mathbf{b} \mid \mathbf{d} \\ C &::= \mathbf{c} \end{aligned}$$

4.6.3 The Tear Operator

Derivations using a production of the form $X ::= \alpha x^{\wedge\wedge\wedge} \beta$, where $x \in N \cup T \cup \{\epsilon\}$ and $\alpha, \beta \in (N \cup T \cup \{\epsilon\})^*$ will have derivation trees with subtrees of the form



After application of the tear operator, this will be transformed to

The abstract syntax will simply replace the former rule with a rule of form

$$X ::= \alpha\beta$$

Example

In the grammar

$$S ::= A\,\mathbf{d} \mid C^{\wedge\wedge\wedge} : t\,A\,B$$
$$A ::= A\,\mathbf{a} \mid \mathbf{a}$$
$$B ::= \mathbf{b}\,\mathbf{b} \mid \mathbf{d}$$
$$C ::= \mathbf{c}$$

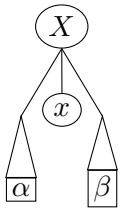
The rule $S ::= C^{\wedge\wedge\wedge} : t\,A\,B$ is resolved to the rule $S ::= A\,B$ and $t = C$. The resulting grammar is

$$S ::= A\,\mathbf{d} \mid A\,B$$
$$A ::= A\,\mathbf{a} \mid \mathbf{a}$$
$$B ::= \mathbf{b}\,\mathbf{b} \mid \mathbf{d}$$
$$C ::= \mathbf{c}$$

4.6.4 The Insert Operator

Derivations using productions of the form $X ::= \alpha[x]\beta$, where $\alpha, \beta \in (N \cup T \cup \{\epsilon\})^*$ will have derivation trees of the form

After application of the insert operators, this will be transformed to



The abstract syntax will contain a rule $X ::= \alpha x \beta$. In the case that the insert operator is $[id]$ where id is the name of some stored symbol, y, then the abstract syntax will contain the rule

$$X ::= \alpha y \beta$$

Example

Consider

$$S ::= A\ \mathbf{d} \mid C^{\wedge\wedge\wedge} : t\ A\ [t]\ B$$
$$A ::= A\ \mathbf{a} \mid \mathbf{a}$$
$$B ::= \mathbf{b}\ \mathbf{b} \mid \mathbf{d}$$
$$C ::= \mathbf{c}\ [\mathbf{d}]$$

Resolving $[d]$ will generate the rule

$$C ::= \mathbf{c}\ \mathbf{d}$$

in the abstract syntax.

In the rule $S ::= C^{\wedge\wedge\wedge} : t\ A\ [t]\ B$, C is torn and stored as t. The abstract syntax will therefore contain a rule $S ::= A\ C\ B$.

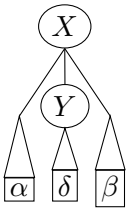
The resulting abstract syntax grammar is

$$S ::= A\,\mathbf{d} \mid A\,C\,B$$
$$A ::= A\,\mathbf{a} \mid \mathbf{a}$$
$$B ::= \mathbf{b}\,\mathbf{b} \mid \mathbf{d}$$
$$C ::= \mathbf{c}\,\mathbf{d}$$

4.6.5 The Gather Operator

With the gather operator, the right-hand operator can either be an existing non-terminal or a new non-terminal symbol. Derivations using a production of the form $X ::= \alpha(\delta)!Y\beta$, where $\alpha, \delta, \beta \in (N \cup T \cup \{\epsilon\})^*$ will have derivation trees of the form

After applying the gather operator, this will be transformed to



The abstract syntax will contain a rule of the form

$$X ::= \alpha Y \beta$$

as well as a rule of the form.

$$Y ::= \delta$$

Example

Given the grammar

$$S ::= A\,\mathbf{d} \mid C\,A\,B$$
$$A ::= A\,\mathbf{a} \mid \mathbf{a}$$
$$B ::= (\mathbf{b}\,\mathbf{b})!D \mid \mathbf{d}$$

$$C ::= \mathbf{c}$$

In the rule $B ::= (\mathbf{b}\ \mathbf{b})!D$, the gather operator extracts $\mathbf{b}\ \mathbf{b}$ into a new production $D ::= \mathbf{b}\ \mathbf{b}$, and then $B ::= D$. The resulting grammar is then

$$\begin{aligned} S &::= A\ d \mid C\ A\ B \\ A &::= A\ a \mid a \\ B &::= D \mid d \\ C &::= c \\ D &::= b\ b \end{aligned}$$

4.6.6 Resolution order of operators

As was the case in the tree-to-tree transformation, the resolution of the GIFT operators from the grammar-to-grammar perspective is mostly declarative. Even in the case where an operator is embedded in the left operand of a GIFT annotation, it is possible to resolve both annotations in either order without affecting the final result. For example, consider the GIFT grammar

$$\begin{aligned} S &::= (\mathbf{a}\ A^{\wedge})!X \\ A &::= \mathbf{a}\ A \mid \mathbf{a} \end{aligned}$$

If the gather operator is resolved first, then the working grammar will be

$$\begin{aligned} S &::= X \\ A &::= \mathbf{a}\ A \mid \mathbf{a} \\ X &::= \mathbf{a}\ A^{\wedge} \end{aligned}$$

If instead the fold-under operator is resolved first, then the working grammar will be

$$\begin{aligned} S &::= (\mathbf{a}\ \mathbf{a}\ A)!X \mid (\mathbf{a}\ \mathbf{a})!X \\ A &::= \mathbf{a}\ A \mid \mathbf{a} \end{aligned}$$

In both cases, resolving all other operators will result in the same final abstract syntax

$$\begin{aligned} S &::= X \\ A &::= \mathbf{a}\ A \mid \mathbf{a} \\ X &::= \mathbf{a}\ \mathbf{a}\ A \mid \mathbf{a}\ \mathbf{a} \end{aligned}$$

Of course, performance is an important consideration. In the above example, resolving the gather operator first and then the fold-under operator requires only two resolution steps. Meanwhile, when fold-under is resolved first, the need to create two productions for each rule of A means there are two gather resolution steps, generating three resolution steps overall. The gather operator only ever moves symbol instances, it does not create them. The tear operator only ever removes symbol instances, and the insert operator only reinstates the tear operator instance. However, the fold-over and fold-under operators copy symbol instances, which, in general, will increase the number of resolution steps needed if the symbol instances copied have annotations.

The conflict discussed in 4.4.1 becomes more pertinent in the grammar-to-grammar context. Consider the GIFT-annotated grammar

$$A ::= \mathbf{a}\ B^{\wedge}$$
$$B ::= \mathbf{b}\ C^{\wedge\wedge}$$
$$C ::= \mathbf{c}$$

Applying the annotation on B first will result in the intermediate grammar

$$A ::= \mathbf{a}\ \mathbf{b}\ C^{\wedge\wedge}$$
$$C ::= \mathbf{c}$$

Resolving $C^{\wedge\wedge}$ would then lead to the abstract syntax

$$S' ::= C$$
$$C ::= \mathbf{a}\ \mathbf{c}$$

The application of the fold-over operator breaks the locality of the GIFT operators, leading to an interpretation that is different to the interpretation that results from the tree-to-tree transformation. Resolving the operators in the other way results in an abstract syntax grammar that matches the tree-to-tree transformation

$$S ::= \mathbf{a}\ \mathbf{b}\ \mathbf{c}$$

For both this issue, and the performance issue, the solution is to resolve fold operators only when all symbol instances that are to be copied have their own annotations resolved.

There is also the outstanding issue of what happens when the fold-under and fold-

over annotation recursively copies instances of itself such as in the rule

$$X ::= \mathbf{a}\, X^{\wedge}\, \mathbf{b} \mid \mathbf{c}$$

In this case, there is no easy solution. The problem becomes even more difficult when the recursion is not obvious in the initial GIFT-annotated grammar, such as in the grammar

$$\begin{aligned} X &::= \mathbf{a}\, Y^{\wedge}\, \mathbf{b} \\ Y &::= \mathbf{c}\, Z^{\wedge} \\ Z &::= \mathbf{d} \mid X^{\wedge} \end{aligned}$$

For this thesis, the proposal is that these *fold-cycles* will be identified and will remain annotated without resolution in the final abstract syntax.

Definition 4.6.1. *X is in a fold-cycle if there exists a sequence $X_1, X_2, \cdots, X_k$, such that for all $i, 1 \leq i \leq k$ there exists a production of the form $X_i ::= \alpha X^{\wedge}_{(i+1)} \beta$ or $X_i ::= \alpha X^{\wedge\wedge}_{(i+1)} \beta$ and $X_1 = X = X_k$.*

For the best time performance, and to handle the issues illustrated above, the construction of the abstract syntax will occur in a series of steps. Initially, a definition of terms is provided. A production, $p \in P$ is *annotation-free* if none of the symbols on its right hand side are annotated. For a given $X \in N$, if every $p_i \in P$ such that X is on the left hand side of p_i is annotation-free, then X is annotation-free.

Definition 4.6.2. *If a grammar symbol $x \in N \cup T \cup \{\epsilon\}$ is a terminal, ϵ or an annotation-free non-terminal, then it is annotation-resolved in Γ.*

The steps taken are then as follows

1. Resolve all instances of the gather, insert and tear operators
2. For each annotation instance of form $x^{\wedge}$ or $x^{\wedge\wedge}$, apply this annotation instance if:
   - x is annotation-resolved.
   - x is not annotation-resolved, but all annotations on the right-hand sides of productions of x that are not annotation-free are of the form $x ::= \alpha Y^{\wedge} \beta$ or $x ::= \alpha Y^{\wedge\wedge} \beta$ such that Y is in a fold-cycle.
3. Repeat the previous step until all symbols are either annotation-resolved or in a fold-cycle.

Of course, in the case where a fold-cycle is present the grammar that is produced will not be the grammar whose derivation trees are ASTs in the original grammar. However, the grammar that is produced will be a GIFT-annotated grammar, and the ASTs that are produced by this grammar will be the ASTs of the original grammar.

The primary purpose of the grammar-to-grammar transformation is to document the abstract syntax that the abstract syntax trees correspond to - which would be particularly useful for debugging the GIFT translation of the trees. For this purpose, a grammar-to-grammar transformation with minimal unresolved annotations may be good enough.

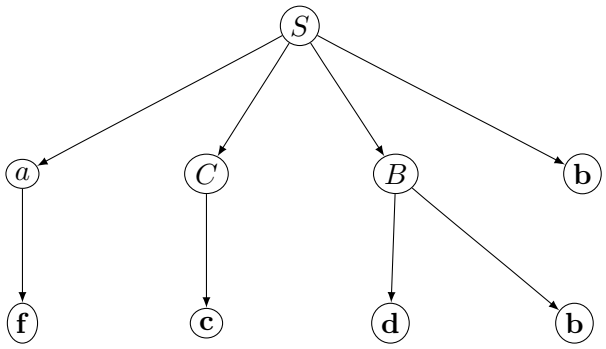
For the grammar

$$
\begin{aligned}
S &::= A\ \mathbf{d} \mid C^{\wedge\wedge\wedge} : t\ A\ [t]\ B^{\wedge} \\
A &::= A\ \mathbf{a}^{\wedge\wedge}\ \mathbf{f} \mid \mathbf{a}^{\wedge\wedge} \\
B &::= D^{\wedge}\ \mathbf{b} \mid \mathbf{d} \\
C &::= \mathbf{c} \\
D &::= B^{\wedge\wedge}\ \mathbf{b}
\end{aligned}
$$

the result of the described procedure would be the grammar

$$
\begin{aligned}
S &::= a\ \mathbf{d} \mid a\ C\ D^{\wedge}\ \mathbf{b} \mid a\ C\ \mathbf{d}\ \mathbf{b} \\
a &::= a\ \mathbf{f} \mid \epsilon \\
B &::= D^{\wedge}\ \mathbf{b} \mid \mathbf{d} \\
C &::= \mathbf{c} \\
D &::= B^{\wedge\wedge}\ \mathbf{b}
\end{aligned}
$$

For the string `cafdbb`, the resulting AST will be



Chapter 5

Topics in Lexical Analysis

This chapter considers some issues arising from the theoretical approaches described in Chapters 2 and 3. In the first instance, the ESPPF constructed using a multilexer parser approach is compared to an equivalent SPPF constructed using a character-level parser. Then an approach to syntactic ambiguity reduction operating similarly to the lexical ambiguity reduction rules in Chapter 2 is considered. An alternative TWE set construction algorithm is given, that uses an EBNF GLL Recogniser instead of a finite-state automaton. A more detailed discussion of how the token suppression technique in Chapter 2 can be used to suppress layout tokens then follows. The final part of this chapter will consider the ANSI-C type/variable name ambiguity in the context of a generalised parser.

5.1 The relationship between character-level SPPF and token-level ESPPF parsing

In the multilexer parser approach, a user specifies a set of tokens, T, whose patterns are non-empty strings over some alphabet A, along with a context-free grammar Γ, whose set of terminals is T. The goal is to find, given T and Γ, all derivations of all tokenisations of an input character string.

An alternative approach is to use character-level parsing with no formal distinction between lexical and syntactic analysis. A character-level grammar is one in which the set of terminals is simply the set A, where each $a \in A$ denotes a single string, `a`. In more traditional terms, A is a set of tokens whose patterns contain a single element which is a string containing a single character.

This section gives a brief discussion on the relationship between the character-level and multilexer parser approaches. The C# 2.0 case study in Chapter 6 will then go

further by giving experimental results that include a practical comparison of the two approaches.

Suppose that $\Gamma = (N, T, S, P)$ is a token-level grammar under the multilexer parser approach, and that T is a set of tokens whose patterns are context-free strings over A. One can define a corresponding character-level grammar for Γ as follows. Let the set of terminals of the grammar be simply the alphabet, A. Define a set of non-terminals $T' = \{t' | t \in T\}$ and a set of productions Q such that for each $t' \in T'$, the sub-language generated is the pattern for the corresponding t. Defining the productions that generate the sub-language for t' may require additional non-terminals (that must not be in N), denoted by the set M. Each $q \in Q$ is therefore a pair (t', γ') where $t' \in T'$ and $\gamma \in (M \cup A \cup T')^+$. For each $p \in P$, symbols in T are replaced with the corresponding $t' \in T'$, with the resulting production being added to a set R. Let $N' = N \cup T' \cup M$ and $P' = R \cup Q$. If Γ is the grammar defined under the multilexer parser approach, then $\Gamma' = (N', A, S, P')$ is a corresponding character-level grammar.

For example, consider the lexical specification

$$\mathbf{x} = \{\mathtt{b}, \mathtt{c}, \mathtt{bc}, \mathtt{cc}\}$$
$$\mathbf{y} = \{\mathtt{d}\}$$

$A = \{\mathtt{b}, \mathtt{c}, \mathtt{d}\}$ and $T = \{\mathbf{x}, \mathbf{y}\}$. A token-level grammar, Γ over T is given by

$$S ::= A\,\mathbf{y}$$
$$A ::= \mathbf{x}\,C$$
$$C ::= \mathbf{x}$$

A corresponding character-level grammar, Γ', would then be

$$S ::= A\,Y$$
$$A ::= X\,C$$
$$C ::= X$$
$$X ::= \mathtt{b} \mid \mathtt{c} \mid \mathtt{b\ c} \mid \mathtt{c\ c}$$
$$Y ::= \mathtt{d}$$

Of course, this is not the only corresponding character-level grammar that could be used. Another possible Γ' is

$$S ::= A\,Y$$

$$
\begin{aligned}
A &::= X\ C \\
C &::= X \\
X &::= B \mid C \mid B\ C \mid C\ C \\
Y &::= \mathtt{d} \\
B &::= \mathtt{b} \\
C &::= \mathtt{c}
\end{aligned}
$$

The first definition of Γ' will be the one that is used here.

There is a natural correspondence between the SPPFs that are produced by Γ' and the ESPPFs that are produced by Γ. Given an input character string, δ, the parser for the character-level grammar Γ' will produce an SPPF for δ. If all nodes that are descendants of a symbol node labelled (t', i, j) are removed from this SPPF and the symbol node labelled (t', i, j) is relabelled (t, i, j) (where t is the corresponding token in the token-level grammar), then the result is the ESPPF obtained by tokenising δ and giving the resulting TWE set to the parser for Γ.

Consider the character string `bccd`, and the example grammars above. Tokenising the string yields the TWE set

$$\Sigma = \{(\mathbf{x}, 0, 1), (\mathbf{x}, 0, 2), (\mathbf{x}, 1, 2), (\mathbf{x}, 1, 3), (\mathbf{x}, 2, 3), (\mathbf{y}, 3, 4)\}$$

The MGLL parser for Γ would then produce the ESPPF in Figure 5.1. The parser for Γ', when given the character string, would produce the SPPF in Figure 5.2. If, for this SPPF, all the nodes that descend from $(X, 0, 1)$, $(X, 0, 2)$, $(X, 1, 3)$, $(X, 2, 3)$, and $(Y, 3, 4)$ are removed, all instances of X are replaced with $\mathbf{x}$, and all instances of Y are replaced with $\mathbf{y}$, the result is Figure 5.1.

A token-level grammar specification can always be transformed into an equivalent character-level grammar specification. A character-level grammar specification is a single unified specification in which all information is available throughout the input string analysis. This has advantages over the traditional, lexical analysis followed by syntactic analysis approach as this allows situations in which the token choice depends on the syntactic context to be handled cleanly.

As will be demonstrated in Chapter 6, the multilexer approach provides several advantages:

- When the patterns of the tokens are regular languages, producing a TWE set for a string, and then parsing that set is likely to be more efficient than parsing the character string in a corresponding character-level parser. Whereas generalised

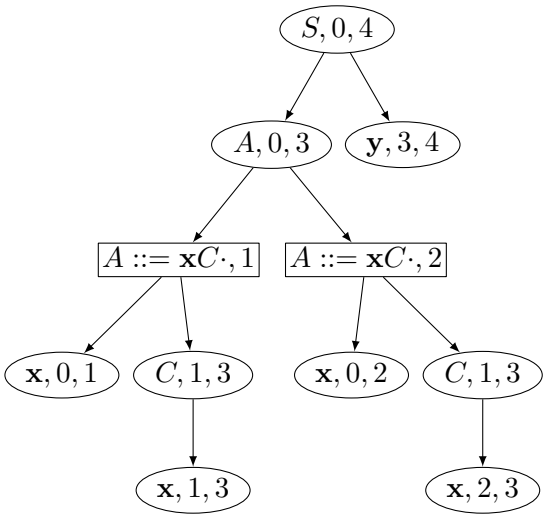


Figure 5.1: ESPPF produced by Γ for the string `bccd`

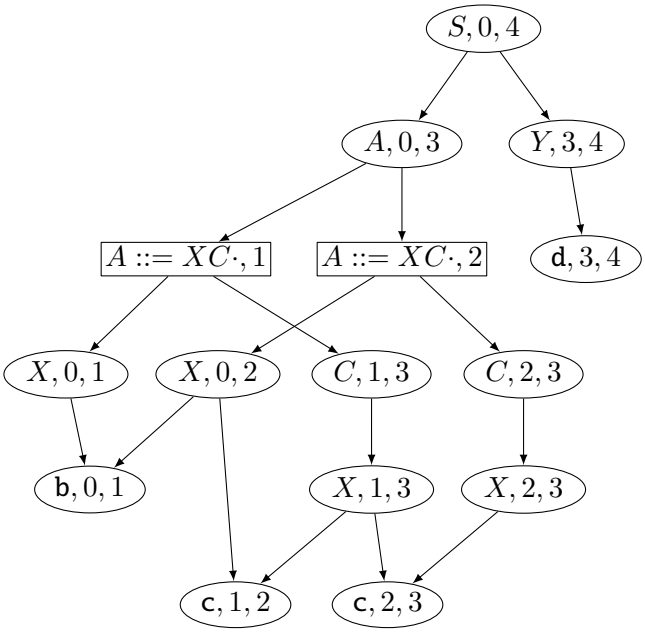


Figure 5.2: SPPF produced by Γ' for the string `bccd`

parsing is worst-case time complexity cubic in the size of the string, the DFA approach used by the multilexer is worst-case quadratic in the size of the string.

- The structure of a lexeme is usually not important for semantic evaluation and the design of the multilexer parser approach means the construction of these structures in the derivation tree can be omitted.

- Character-level parsers are restricted to character-level lookahead. If, for example, two tokens had lexemes that share the same first character, character-level lookahead would not be able to eliminate the exploration of derivation steps involving a token non-terminal that could not result in a valid parse. Meanwhile, token-level lookahead could allow these derivation steps to be eliminated before they are explored.

- A user will obtain more useful feedback from error reporting at token-level than character-level. For instance, it is more useful to report that an "identifier" cannot appear in a particular context, than to simply say that a certain sequence of characters cannot appear in that context.

The multilexer parser approach is aimed at giving language designers the same level of power and control as a character-level specification, whilst retaining the advantages of a token-level specification. A multilexer specification also provides techniques for reducing the amount of lexical ambiguity in the TWE set provided to an MGLL parser, whereas in a character-level specification the parser must treat all lexical ambiguities as syntactic ambiguities. As will be discussed in the next section, lexical level ambiguity reduction mechanisms can be cleaner than attempting to disambiguate at syntactic level. Where lexical choice is dependent on contextual information, all alternative possibilities can be passed down to the MGLL parser for resolution.

It is up to the language designer to decide where the boundary between the lexer and parser is set, and the language designer decides the disambiguation rules to use. The next section compares the effects of these disambiguation rules at both lexical and syntactic level.

5.2 Lexical Ambiguity Reduction and Syntactic Ambiguity Reduction

Although syntactic ambiguity reduction is not a central topic for this thesis, ambiguity reduction mechanisms are needed as part of the C# 1.2 case study. Of course, for a character-level grammar, the lexical ambiguities that occur in a token-level approach are moved to become syntactic ambiguities. So for a character-level grammar, to capture the behaviour of the lexical ambiguity mechanisms given in Chapter 2, one must decide on syntactic equivalents.

Recall from Chapter 1 that for any context-free grammar, Γ, and token string, δ in the language of Γ, there is an SPPF, $\mathcal{S}_\delta$, which embeds all the derivation trees for

δ. Let $V_s(\mathcal{S}_\delta)$ denote the set of symbols and intermediate nodes in $\mathcal{S}_\delta$, and let $V_p(\mathcal{S}_\delta)$ denote the set of packed nodes in $\mathcal{S}_u$.

Whilst the problem of finding syntactic ambiguity in a context-free grammar is undecidable, it is trivial to determine whether an ambiguity exists in a given SPPF for a string. An SPPF, $\mathcal{S}_\delta$ is said to contain a syntactic ambiguity if and only if an element of $V_s(\mathcal{S}_\delta)$ has two or more children - representing the point where two derivations diverge. For an ESPPF, an element of $V_s(\mathcal{S}_\delta)$ having two or more children still represents where two derivations diverge, but this does not necessarily mean the ESPPF contains a syntactic ambiguity. The divergence of derivations in an ESPPF means one of two things: either that there is a syntactic ambiguity, or that there exists a derivation for more than one ITS. As was the case with lexical ambiguity, it may not be possible, or even desirable, to reduce the number of derivation trees to one. Therefore, the ideas presented in this section aim only to *reduce* the number of derivation trees.

This section gives a description of some syntactic ambiguity reduction techniques that are needed for the C# 1.2 case study explored in Chapter 6. How the behaviour of these techniques compares to lexical ambiguity reduction techniques is then considered, with a discussion on the difference between choosing to resolve at lexical or parsing level.

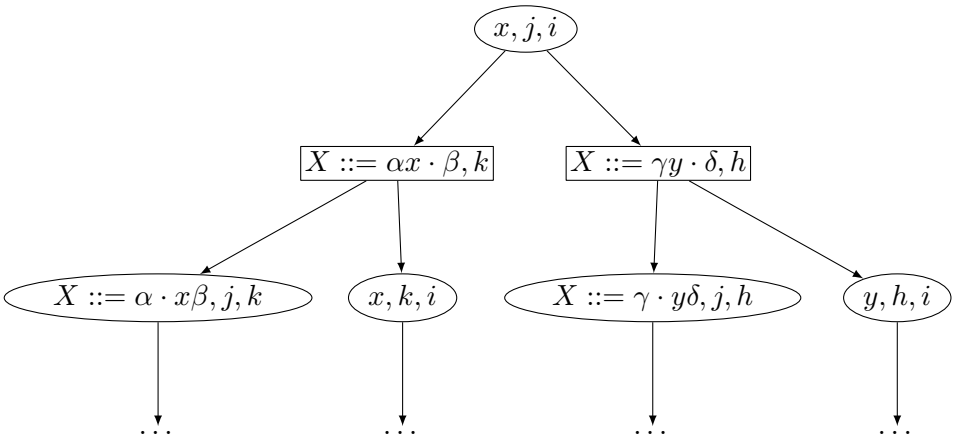
5.2.1 Syntactic level ambiguity reduction in an SPPF

As the presence of more than one derivation tree embedded in an SPPF is indicated by the presence of a symbol or intermediate node having more than one packed node, it is natural to treat ambiguity reduction as the removal of packed nodes. In the given scheme, although a node may have more than two packed node children, ambiguity reduction will be pairwise. In addition, each ambiguity reduction operation will simply mark packed nodes for suppression, they do not remove packed nodes from the tree. That way, marked nodes continue to participate in other pairwise operations. It is an error for an ambiguity reduction scheme to suppress all derivation trees as this would change the language being parsed. In such cases, a warning is issued and one arbitrarily chosen packed node is unmarked. After all marking operations are complete, all trees rooted at marked nodes are removed from the SPPF.

The syntactic ambiguity reduction rules are in the format

$$rule(r, t)$$

rule is a label denoting the class of the disambiguation rule, r and t are grammar slots of form $X ::= \alpha \cdot \beta$ where $X ::= \alpha\beta \in P$. Consider a pair of packed nodes $X ::= \alpha x \cdot \beta, k$ and $X ::= \gamma y \cdot \delta, h$ with a common parent x, j, i as depicted by the graph



Either $\alpha = \gamma, x = y$, and $\beta = \delta$ and x is some $X ::= \alpha x \cdot \beta$, or $\beta = \epsilon = \delta$ and $x = X$.

The three classes of rules are as follows. The reader will see that these rules are thematically similar to the lexical ambiguity reduction rules given in Chapter 2.

- *suppress*$(X ::= \alpha x \cdot, X ::= \gamma y \cdot)$ - Mark the node labelled $(X ::= \alpha x \cdot, h)$ for suppression if there exists a node labelled $(X ::= \gamma y \cdot, k)$, $\gamma y \neq \alpha x$ sharing the same parent node such that $k = h$

- *longest*$(X ::= \alpha x \cdot \beta, X ::= \gamma y \cdot \delta)$ - Mark the node labelled $(X ::= \alpha x \cdot \beta, h)$ for suppression if there exists a node labelled $(X ::= \gamma y \cdot \delta, k)$ sharing the same parent node such that $k > h$

- *shortest*$(X ::= \alpha x \cdot \beta, X ::= \gamma y \cdot \delta)$ - Mark the node labelled $(X ::= \alpha x \cdot \beta, h)$ for suppression if there exists a node labelled $(X ::= \gamma y \cdot \delta, k)$ sharing the same parent node such that $k < h$

Informally, *suppress*(r, t) can be thought of as the negation of a priority mechanism - marking all nodes labelled with slots that are lower priority than the node labelled with t. *longest*(r, t) and *shortest*(r, t) can be viewed as longest and shortest-match mechanisms. The pivot value of a packed node refers to the right-extent of the left child of the packed node. As such, the pivot value effectively refers to the portion of the string matched up until the symbol just before the slot position. A higher pivot value means a longer match.

As was the case for the lexical ambiguity reduction rules, for two grammar slots r and t, it is possible to specify that r should be suppressed if t exists regardless of the pivot values, by specifying *suppress*(r, t), *longest*(r, t), and *shortest*(r, t) together in the ambiguity reduction scheme.

In general, these rules will not be sufficient to reduce the number of derivations for all grammars. However, these rules are sufficient for the purposes of this thesis.

5.2.2 Syntactic Ambiguity Reduction in an ESPPF

For an ESPPF, the presence of a symbol/intermediate node with more than one packed node child could either mean the underlying grammar is syntactically ambiguous or that the ESPPF embeds derivations for more than one ITS. The same class of rules defined above can still be applied in these circumstances. This section compares, with examples, the use of these rules with corresponding lexical ambiguity reduction rules.

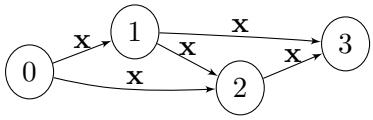
Consider a token set, T_1, containing a single token

$$(\mathbf{x}, \{\mathtt{aa}, \mathtt{a}\})$$

as well as a grammar Γ_1 with a single rule

$$S ::= \mathbf{x}\ \mathbf{x}$$

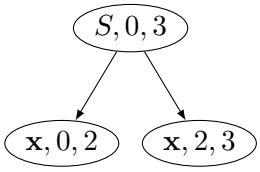
Given the character string `aaa`, the lexical analysis phase initially generates the TWE set represented by the graph



If the relation $\mathbf{x}R\mathbf{x}$ under the matrix for class 2 operations is specified and applied, the result after pruning is the TWE set

$$\{(\mathbf{x}, 0, 2), (\mathbf{x}, 2, 3)\}$$

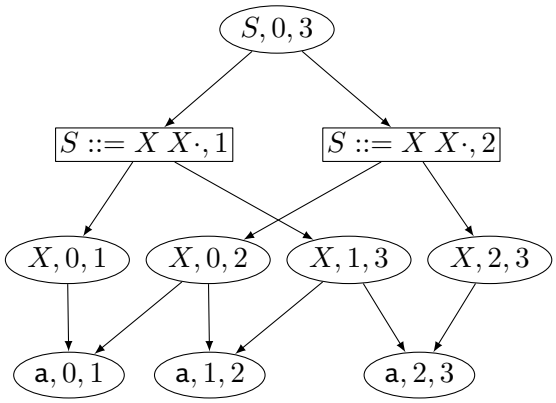
This is equivalent to lexical longest match. The MGLL parser for Γ_1 would then construct the ESPPF



Consider the corresponding character-level grammar

$$S ::= X\ X$$
$$X ::= \mathtt{a}\ \mathtt{a} \mid \mathtt{a}$$

The SPPF that results from parsing the string `aaa` is the following



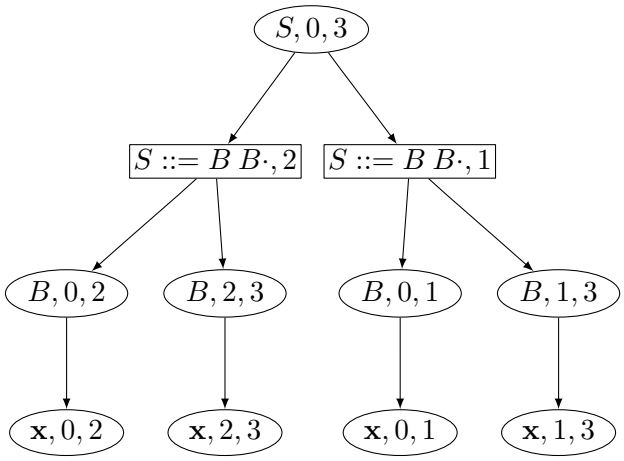
To perform the equivalent derivation reduction, it is necessary to use the rule

$$longest(S ::= X\ X\cdot, S ::= X\ X\cdot)$$

This is conceptually more difficult for the user, as although the ambiguity is the result of what X can derive, the choice appears under S. This can be illustrated more clearly with Γ_2

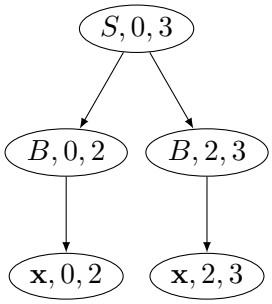
$$S ::= B\ B$$
$$B ::= \mathbf{x}$$

If the full TWE set for the string $\mathtt{aaa}$ is given to the MGLL parser for Γ_2, then the result is the ESPPF



The presence of two derivation trees embedded in the ESPPF is the result of the lexical ambiguity in $\mathbf{x}$ - the grammar itself is syntactically unambiguous. As before, defining the relation $\mathbf{x}R\mathbf{x}$ under the matrix for class 2 operations, and applying this to the

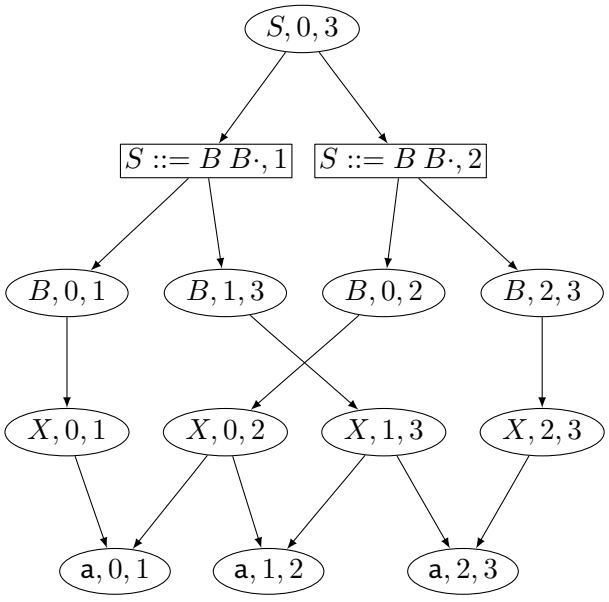
TWE set, reduces the TWE set such that the result of parsing is a single derivation tree. resulting in the ESPPF



For the corresponding character-level grammar

$$S ::= B\ B$$
$$B ::= X$$
$$X ::= \mathsf{a}\ \mathsf{a} \mid \mathsf{a}$$

the SPPF for the string aaa is



To achieve the same effect as longest match on the TWE set above, it is necessary to specify $\mathit{longest}(S ::= B\ B\cdot, S ::= B\ B\cdot)$. The syntactic ambiguity that the definition of X creates only appears at the nodes under S. The sublanguages generated by non-terminals representing tokens are unambiguous, but strings in these sublanguages can cause ambiguities elsewhere in the grammar.

The syntactic versions of longest match and priority are not sufficient for implementing the lexical equivalents (class two and class one operations respectively). Consider the token set

$$\{(\mathbf{x}, \{\mathsf{aa}, \mathsf{a}\}), (\mathbf{y}, \{\mathsf{bc}, \mathsf{ab}\}), (\mathbf{z}, \{\mathsf{c}\})\}$$

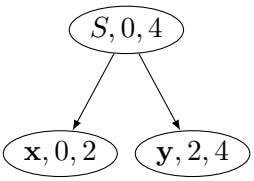
and the grammar, Γ_3

$$S ::= \mathbf{x}\ \mathbf{y}\ \mathbf{z} \mid \mathbf{x}\ \mathbf{y}$$

Let $\mathbf{x}R\mathbf{x}, \mathbf{x}R\mathbf{y}$, and $\mathbf{z}R\mathbf{y}$ under the matrix for class 2 operations. After applying lexical ambiguity reduction and pruning, the result of lexical analysis for the character string aabc is the TWE set

$$\{(\mathbf{x}, 0, 2), (\mathbf{y}, 2, 4)\}$$

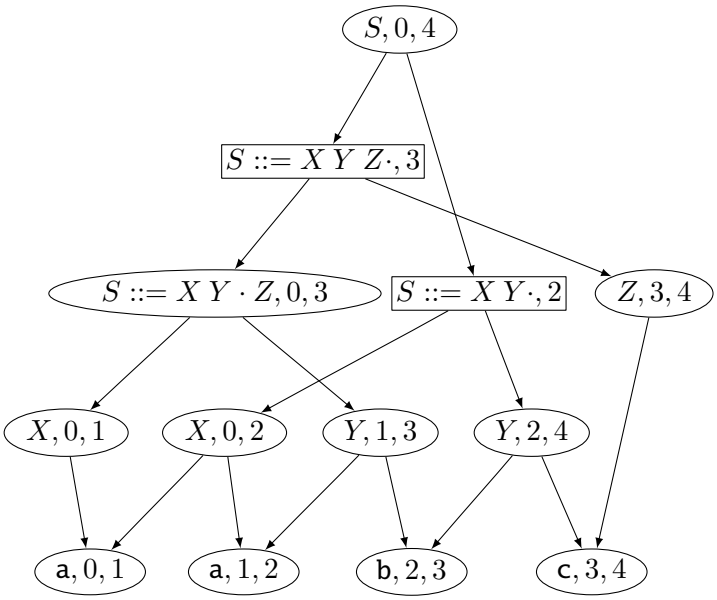
The MGLL parser for Γ_3 will then generate the ESPPF



Now consider the corresponding character-level grammar

$$\begin{aligned} S &::= X\ Y\ Z \mid X\ Y \\ X &::= \mathsf{a}\ \mathsf{a} \mid \mathsf{a} \\ Y &::= \mathsf{b}\ \mathsf{c} \mid \mathsf{a}\ \mathsf{b} \\ Z &::= \mathsf{c} \end{aligned}$$

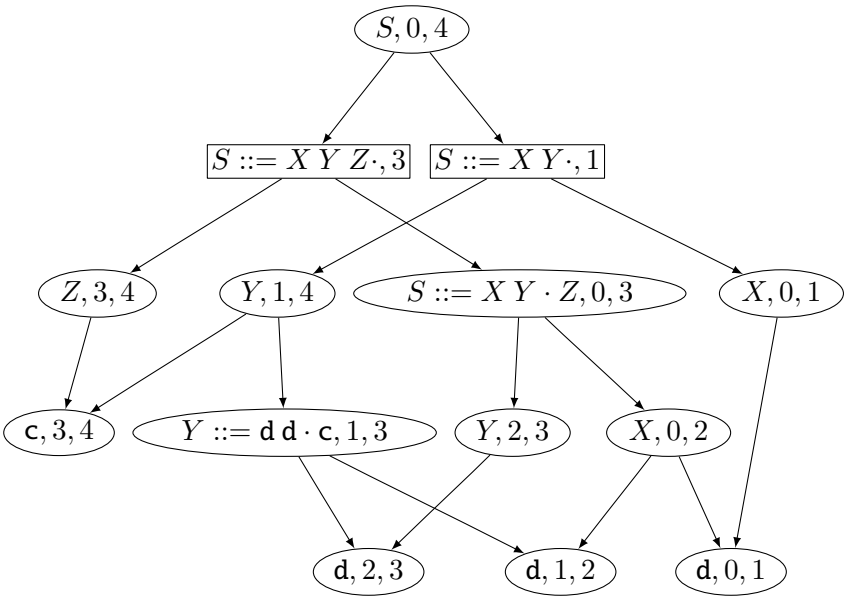
A character-level parser on the same character string will produce the following SPPF



Derivation reduction rules can only be applied to nodes under the root, as this is the only node with more than one packed node child. $longest(S ::= X\ Y\cdot, S ::= X\ Y\ Z\cdot)$ will suppress $(S ::= XY\cdot, 2)$, making the opposite decision that was made in the MGLL case. In this case $shortest(S ::= X\ Y\ Z\cdot, S ::= X\ Y\cdot)$ is what is needed to achieve the same result. However, this fails for a slightly more complicated example. Consider

$$
\begin{aligned}
S &::= X\ Y\ Z \mid X\ Y \\
X &::= \mathsf{a\ a} \mid \mathsf{a} \mid \mathsf{d\ d} \mid \mathsf{d} \\
Y &::= \mathsf{b\ c} \mid \mathsf{a\ b} \mid \mathsf{d\ d\ c} \mid \mathsf{d} \\
Z &::= \mathsf{c}
\end{aligned}
$$

$shortest(S ::= X\ Y\ Z\cdot, S ::= X\ Y\cdot)$ is still needed for the equivalent of lexical longest match on X for input aabc. However for the character string dddc, the character-level grammar generates the SPPF



and the rule $shortest(S ::= X\ Y\ Z\cdot, S ::= X\ Y\cdot)$ selects the derivation in which X matches `d` rather than the one in which it matches `dd`.

This demonstrates that the lexical ambiguity reduction mechanisms of the multilexer parsing approach described in Chapter 2 cannot be simply mimicked in character-level grammars with equivalent syntactic ambiguity reduction mechanisms. A more complicated syntactic ambiguity reduction strategy that requires more than local information about two sibling packed nodes could be written that may achieve similar goals. However, it is clear from the discussion that the techniques from lexical level disambiguation do not easily carry over in a natural manner to grammar level techniques - a point that was previously highlighted by Visser [Vis97]. This gives the multilexer parser approach a strong advantage over a character-level grammar approach.

5.3 Constructing TWE sets with a GLL Recogniser

Section 2.4 described how to construct TWE sets using deterministic finite state automata to represent token patterns. However, this only works with tokens whose patterns represent a regular language. In some cases, it may be desirable for the tokens to have patterns which are context-free. For example, OCaml allows the user to specify nested comments such as the string `(* This (* is (* a comment *) *) *)`. Meanwhile the string `(* This (* comment is malformed *)` is considered invalid. The language for specifying nested comments needs to be context-free to ensure that a match is only made on matching brackets.

This section gives a GLL-style recogniser for grammars whose start symbol S, where

S has a single, EBNF rule of the form

$$S ::= (t_1 | \ldots | t_f)^*$$

where $t_1, \ldots, t_f$ are unique non-terminals, referred to as the token non-terminals, that do not appear on the right-hand side of any other production rule. The set of terminals for this grammar is a set of characters, A. The principle is that $t_1, \ldots, t_f$ will represent the set of tokens, T, for the lexer specification. The sublanguage generated by the productions of each t_i is the pattern of t_i. This grammar is effectively a character-level grammar for the language defined by the set T^*.

In principle, one could, after conversion to BNF, simply use this grammar to generate a classical generalised parser. When given a character string, this parser would produce an SPPF. One could then extract the TWE set by creating triples for every SPPF node with labels of the form (t_i, i, j). However, the structural context information about how each token was matched is not needed, and this would offer no performance advantage over simply using a character-level grammar for parsing.

A recogniser is much more efficient for the task of constructing TWE sets. Furthermore, an EBNF recogniser will, provided the patterns for each token can be defined using regular expressions over A, perform with efficiency comparable to that of finite state automaton recognition.

A tokeniser based on an EBNF GLL recogniser takes as input a character string over A and returns a corresponding TWE set. As this is a recogniser only, there are differences to the GLL parsing algorithm described in Chapter 3. The first major difference is the introduction of code templates for EBNF constructs (using the notation given in Chapter 1). Secondly, in a GLL recogniser, the edges of the GSS are unlabelled - since there is no SPPF construction. Also, a descriptor does not contain a reference to an SPPF node. As in Chapter 3, m denotes the length of the input, c_I denotes the current input position, c_U denotes the current GSS node, $\mathcal{P}$ is a set of GSS node, integer pairs, and $\mathcal{U}$ and $\mathcal{R}$ are sets of descriptors. The character string input is read character-by-character into an array I and $I[m]$ is set to \$ denoting the end of string. Additionally Σ is used to denote the TWE set.

The recogniser uses a modified version of the support functions defined in Chapter 3. Additionally, a fourth support function is defined named testRepeat, whose purpose is to detect potentially repeated actions as a result of closure and alternated regular expressions.

function ADD(L, u, i) {
  if $(L, u, i) \notin \mathcal{U}$ **then** add (L, u, i) to $\mathcal{U}$ and to $\mathcal{R}$

```
}

function POP(u, i) {
    let (L, k) be the label of u
    add (u, z) to P
    for each child v of u do {
        ADD(L, v, i)
    }
}

function CREATE(L, u, i) {
    if there is not already a GSS node labelled (L, i) then create one
    let v be the GSS node labelled (L, i)
    if there is not an edge from v to u then {
        create an edge from v to u
        for all (v, z) ∈ P do {
            ADD(L, u, k)
        }
    }
    return v
}

function TESTSELECT(a, Y, r) {
    if a ∈ first(r) or (ϵ ∈ first(r) ∧ a ∈ follow(Y)) then
        return true
    else
        return false
}

function TESTREPEAT(T, u, i) {
    if (T, u, i) ∈ TR then {
        return true }
    else{
        add (T, u, i) to TR
        return false
    }
}
```

The main GLL recognition function is then as follows

Create GSS nodes $u_0 = (\$, 0)$ and $u_1 = (L_0, 0)$ and create an edge from u_1 to u_0.
$c_U := u_1$; $c_I := 0$;
$\mathcal{U} := \emptyset$; $\mathcal{R} := \emptyset$; $\mathcal{P} := \emptyset$; $\Sigma = \emptyset$
goto L_S

L_0 :
if $R \neq \emptyset$ **then** {
    Remove (L, u, i) from $\mathcal{R}$
    $c_U := u$; $c_I := i$;
    goto L }
else if $(L_0, u_0, m) \in \mathcal{U}$ **then**
    Report success
else
    Report failure

L_A : code(A)
⋮
L_Z : code(Z)

In this scheme, there are two templates for grammar rules, one for non-terminals representing tokens and one for any other non-terminal. For a grammar rule $X ::= \alpha_1 | \ldots | \alpha_p$, let ρ_i denote the instanced version of α_i, $1 \leq i \leq p$.

If X is non-terminal representing a token, then the following template is used. If g is a GSS node with a label of the form (L, i), then $level(g)$ returns i.

code($X ::= \alpha_1 | \ldots | \alpha_p$) =
    if TESTSELECT($I[c_I], X, \alpha_1$) **then** ADD(L_{α_1}, c_U, c_I)
    ⋮
    if TESTSELECT($I[c_I], X, \alpha_p$) **then** ADD(L_{α_p}, c_U, c_I)
    goto L_0
L_{α_1} :
    code(ρ_1, ϵ)
    if $I[c_I] \in FOLLOW(X)$ **then** {
        POP(c_U, c_I)
        add $(Y, level(c_U), c_I)$ to Σ
    }
    goto L_0
    ⋮
L_{α_p} :
    code(ρ_p, ϵ)
    if $I[c_I] \in FOLLOW(X)$ **then** {
        POP(c_U, c_I)
        add $(Y, level(c_U), c_I)$ to Σ
    }
    goto L_0

If X is not a non-terminal representing a token, then the following template is used instead

$code(X ::= \alpha_1 | \ldots | \alpha_p) =$
  if TESTSELECT$(I[c_I], X, \alpha_1)$ **then** ADD(L_{α_1}, c_U, c_I)
  $\vdots$
  if TESTSELECT$(I[c_I], X, \alpha_p)$ **then** ADD(L_{α_p}, c_U, c_I)
  goto L_0
L_{α_1} :
  code(ρ_1, ϵ)
  if $I[c_I] \in FOLLOW(X)$ **then** {
    POP(c_U, c_I)
  }
  goto L_0
  $\vdots$
L_{α_p} :
  code(ρ_p, ϵ)
  if $I[c_I] \in FOLLOW(X)$ **then** {
    POP(c_U, c_I)
  }
  goto L_0

For an ϵ instance ϵ^j whose left-hand side is X, no code is generated. For a terminal $a \in T$ the code generated for an instance a^j whose left-hand side is X is

$code$(code$(a^j, r) =$
  if $I[c_I] = a$ **then** {
    $c_I := c_I + 1$;
  }

For a non-terminal $Y \in N$ the code generated for an instance Y^j whose left-hand side is X is

$code(Y^j, r) =$
  $c_U :=$CREATE(E_{Y^j}, c_U, c_I); **goto** L_Y
E_{Y^j} :

For a rule instance $\rho = (\mu)$, where μ is the instance of the body of some ρ

$code((\mu), r) =$
    code(μ, r)

For a rule instance $\rho = (\mu)?$ where μ is the instance of the body of some ρ, there are two templates depending on whether or not $\epsilon \in first(\mu)$. It would be possible to use the same template for both cases, however, $\epsilon \in first(\mu)$ implies μ is nullable, making the optional statement redundant. If $\epsilon \in first(\mu)$ then one can use a more efficient template:

$code((\mu)?, r)) =$
    code(μ, r)

Otherwise, if the left-hand side of μ is X then the template is as follows

$code((\mu)?, r)) =$
    if TESTSELECT$(I[c_I], X, r)$ **then** {
        ADD(E_ρ, c_U, c_I)
    }
    if TESTSELECT$(I[c_I], X, exp(\mu))$ is false **then goto** L_0
    code(μ, r)
E_ρ :

To define the template for closure, an additional label C_ρ needs to be defined. A flag, T_ρ, is used to prevent potential repeated actions. If ρ is an instanced string, let $exp(\rho)$ be a function that returns the corresponding underlying uninstanced string.

If $\rho = (\mu)^+$ whose left-hand side is X then the code generated is

$code((\mu)^+, r) =$
C_ρ :
    if TESTREPEAT(T_ρ, c_U, c_I) **then goto** L_0
    if TESTSELECT$(I[c_I], X, exp(\mu)r)$ is false **then goto** L_0
    code$(\mu, (exp(\mu))?r)$
    if TESTSELECT$(I[c_I], X, r)$ **then** {
        ADD(E_ρ, c_U, c_I)
    }
E_ρ :

For $\rho = (\mu)^*$ whose left hand side is X, one must also consider the nullable case so the code generated is

$code((\mu)^*, r) =$
    if TESTSELECT$(I[c_I], X, r)$ **then** {
        ADD(E_ρ, c_U, c_I)
    }
C_ρ :
    if TESTREPEAT(T_ρ, c_U, c_I) **then goto** L_0
    if TESTSELECT$(I[c_I], X, exp(\mu)r)$ is false **then goto** L_0
    code$(\mu, (exp(\mu))?r)$
    if TESTSELECT$(I[c_I], X, r)$ **then** {
        ADD(E_ρ, c_U, c_I)
    }
E_ρ :

If $\rho = \mu_1 \dots \mu_d$ where the left-hand side of ρ is X, then the following template is used

$code(\rho, r) =$
    code(μ_1)
    if TESTSELECT$(I[c_I], X, exp(\mu_2 \dots \mu_d)r)$ is false **then goto** L_0
    code$(\mu_2, exp(\mu_3 \dots \mu_d)r)$
    $\vdots$
    if TESTSELECT$(I[c_I], X, exp(\mu_d)r)$ is false **then goto** L_0
    code(μ_d, r)

If $\rho = \mu_1 | \dots | \mu_d$ where the left-hand side of ρ is X, then the following template is used

$code(\rho, r) =$
    if TESTSELECT$(I[c_I], X, exp(\mu_1)r)$ **then** {
        ADD(L_{μ_1}, c_U, c_I)
    }
    $\vdots$
    if TESTSELECT$(I[c_I], X, exp(\mu_d)r)$ **then** {
        ADD(L_{μ_d}, c_U, c_I)
    }
    goto L_0
L_{μ_1} :
    code(μ_1, r)
    goto C_ρ

$\vdots$

$L_{\mu_{d-1}}$:

code(μ_{d-1}, r)

goto C_ρ

L_{μ_d} :

code(μ_d, r)

C_ρ :

if TESTREPEAT(T_ρ, c_U, c_I) **then goto** L_0

The templates presented above are not those that can produce the most efficient recognisers. Calls to TESTREPEAT are only necessary when the body of closures can derive ϵ. For a more efficient implementation, one could provide two templates: one for nullable bodies and one for non-nullable bodies.

5.4 General Approaches to Layout Handling

As mentioned in Section 2.6, layout (whitespace and comment) strings are traditionally treated differently from other strings in many languages, with them effectively being suppressed from the character string. Section 2.6 defined a property that determined whether tokens are considered separated and gave a method for suppressing triples containing these separated tokens. In many languages, layout can be expressed with separated tokens.

For some languages, simple layout suppression is inadequate. For example, in Python, whitespace is insignificant in most places, the string

```
print "foo"; print "bar"
```

has the same semantics as the string

```
print "foo";print   "bar"
```

However, whitespace will have a significant impact on the semantics of the surrounding code when it occurs at the beginning of a new line. The level of indentation determines the nesting of the current line, with sequences of whitespace characters treated as one or more 'indent' tokens. For example, the string

```
if (x==1):
    print(x)
    x = x + 1
```

will print `x` and increment only when `x` is equal to 1. Meanwhile, the string

```
if (x==1):
    print(x)
```

```
x = x + 1
```

will print `x` when `x` is equal to 1 but will increment `x` regardless as it has no preceding indent.

An extreme approach is to define tokens for layout, let the lexer construct a corresponding TWE set for the character string and pass all triples (after lexical ambiguity reduction and pruning) to the parser without suppressing layout. The parser is then responsible for handling layout tokens. The lexer is able to handle any lexical level ambiguity relating to the definition of the layout tokens themselves - therefore giving performance enhancements over, say, a character-level approach, as discussed previously, whilst giving the parser control over whether layout should be suppressed. Of course, to do this, the parsing grammar needs references to layout tokens throughout. Generally, this would require placing an instance of a layout token before (or after) every token instance on every production right-hand side. It is possible for this to be done automatically [JSB11]. Regardless of whether this is done manually or automatically, this approach dramatically increases the size of the parser and the size of the data structures it produces. It also does not provide a clean separation in the lexer/parser interface.

Where possible, it is much cleaner and efficient to use token suppression to remove layout tokens. One naïve approach to consider would be to remove layout token triples and then appropriately making new triples for each of the triples that follow after the lexical ambiguity reduction step. For example, for the TWE set

$$\{(\mathbf{ws}, 0, 1), (\mathbf{ws}, 1, 3), (\mathbf{b}, 0, 3), (\mathbf{a}, 3, 5), (\mathbf{ws}, 5, 6), (\mathbf{a}, 6, 7)\}$$

If **ws** is a layout token, then the approach would first remove $(\mathbf{ws}, 0, 1)$ and add a new triple $(\mathbf{ws}, 0, 3)$. Then both $(\mathbf{ws}, 1, 3)$ and $(\mathbf{ws}, 0, 3)$ are removed with $(\mathbf{a}, 0, 5)$ added. The triple $(\mathbf{ws}, 5, 6)$ would then be removed with $(\mathbf{a}, 5, 7)$ being added. The resulting TWE set would be

$$\{(\mathbf{b}, 0, 3), (\mathbf{a}, 3, 5), (\mathbf{a}, 0, 5), (\mathbf{a}, 5, 7), (\mathbf{a}, 6, 7)\}$$

This would remove the layout tokens, whilst preserving the tokenisation involving the **b** token. However, it can be easily seen that, in general, this would have a significant performance impact as one must process all triples generated by this approach as well as the triples already in the set - since a layout token could be followed by another layout token.

Instead, even where layout tokens are not separated, it may be possible to process the character string before tokenisation to simplify how layout is used. An approach

that combines layout token suppression with some grammar modification technique may be the better long-term solution - allowing layout tokens that can only be disambiguated with contextual or semantic information to be given to the syntactic analysis phase. However, a full solution to the issue of layout handling is beyond the scope of this thesis. This section will conclude by describing how initial processing can be used to simplify layout token suppression. This is powerful enough for layout tokens in both C# language specifications (as well as most standalone C-like languages).

Given a set of tokens, T, whose patterns are non-empty strings over an alphabet A, suppose that one has a processor P, which takes a string over A and identifies which sequences of characters are matched by layout tokens. For example, one could define an initial processor to remove the whitespace and comments in a C# string. To do so, the processor would need to be able to recognise where whitespace and comments occur. As whitespace characters can occur inside of a string literal, the processor would also need to recognise string literals. The processor, P, produces a new string from the input string, in which strings of characters recognised as whitespace or comments are replaced with a single special character w, and all string literals and other character strings are carried across. The set of tokens for the lexer is then defined to be $T' = T \cup \{(\mathbf{ws}, w)\}$. By construction, the token **ws** is separated in T'

The output of the processor, P, is a new string δ' which can be tokenised by the main lexer, using token suppression on **ws**. For example, suppose the token set contains $(\mathbf{t_1}, \{\mathtt{a}, \mathtt{b}, \mathtt{aa}\}), (\mathbf{t_2}, \{\mathtt{1}, \mathtt{2}, \mathtt{3}\}), (\mathbf{t_3}, \{\mathtt{=}, \mathtt{+=}\})$, and the C-style whitespace and comments. Then the character string

```
a=3 /* base case */b =aa a+=b
```

could be processed to give the string

$$\mathtt{a=3}w\mathtt{b}w\mathtt{=aa}w\mathtt{a+=b}$$

which is tokenised to form the ITS

$$(\mathbf{t_1}, 1)(\mathbf{t_3}, 2)(\mathbf{t_2}, 3)(\mathbf{ws}, 4)(\mathbf{t_1}, 5)(\mathbf{ws}, 6)$$
$$(\mathbf{t_3}, 7)(\mathbf{t_1}, 9)(\mathbf{ws}, 10)(\mathbf{t_1}, 11)(\mathbf{t_3}, 13)(\mathbf{t_1}, 14)$$

After token suppression, the result is the ITS

$$(\mathbf{t_1}, 1)(\mathbf{t_3}, 2)(\mathbf{t_2}, 3)(\mathbf{t_1}, 5)(\mathbf{t_3}, 7)(\mathbf{t_1}, 9)(\mathbf{t_1}, 11)(\mathbf{t_3}, 13)(\mathbf{t_1}, 14)$$

There is one further modification that is used for the case studies. To avoid potentially large ITS sets, the TWE sets are constructed directly. Section 2.6 showed how to modify the TWE triples for token suppression, however, this would require iterating over the TWE set to search for the 'adjacent' triples to adjust their extents, a costly operation. It would be more efficient to ensure the triples are constructed in a manner such that the extents are already correctly adjusted. This can be achieved by treating the w character as a delimiter, that splits the string into a list of string fragments - a function achieved by the initial processor. The processor thus creates a list of substrings $P(\delta) = (\delta_1, \delta_2, \ldots, \delta_d)$ such that $\delta = \delta_1 w \delta_2 w \ldots w \delta_d$.

In this approach, the lexer processes each element of $P(\delta)$ separately, building the final TWE set from the union of the TWE sets for each substring. For each element being processed, an offset value is also given representing the distance of the current element from the beginning of the full input, including the preceding w character. This ensures that the extents of the constructed TWE set match character indexes in the full string. For example, the string

```
a=3 /* base case */b=aa a+=b
```

would be processed to give a sequence of three strings

```
a=3
b=aa
a+=b
```

The lexer runs on the first string to construct $\{\mathbf{t_1}, 0, 1), (\mathbf{t_3}, 1, 2), (\mathbf{t_2}, 2, 3)\}$. For the second string, the TWE set $\{(\mathbf{t_1}, 3, 5), (\mathbf{t_3}, 5, 6), (\mathbf{t_1}, 6, 8)\}$ is constructed. For the third string, the TWE set $\{(\mathbf{t_1}, 8, 10), (\mathbf{t_3}, 10, 12), (\mathbf{t_1}, 12, 13)\}$ is constructed. The final TWE set produced is then

$$\{(\mathbf{t_1}, 0, 1), (\mathbf{t_3}, 1, 2), (\mathbf{t_2}, 2, 3), (\mathbf{t_1}, 3, 5), (\mathbf{t_3}, 5, 6),$$
$$(\mathbf{t_1}, 6, 8), (\mathbf{t_1}, 8, 10), (\mathbf{t_3}, 10, 12), (\mathbf{t_1}, 12, 13)\}$$

5.5 Handling of the Standard C type/variable name ambiguity

This chapter concludes with a brief discussion on the standard C type/variable name ambiguity. In the standard C [ISO11] specification, a statement such as `sizeof(a)`,

where the argument is a single identifier, can have two different meanings depending on the context. The symbol `a` could either be a type name defined in an earlier `typedef` definition, or it could be a variable `a`. The correct interpretation can normally only be determined by looking at how `a` is defined in the symbol table - part of the semantic analysis phase. This full C example demonstrates how this can cause problems.

```
typedef int a;

void main() {
    int sizeTA = sizeof(a);
    char a;
    int sizeVA = sizeof(a);
}
```

In the initialisation of `sizeTA`, `a` is interpreted as a type, due to the earlier typedef definition of `a` - in this case, returning the size of `int`. However, then a local variable `a` is defined. For `sizeVA`, `a` is interpreted as a unary expression, and since `a` is of type `char`, returns the size of `char`.

To understand how this happens, one must look into the language specification. In the C standard grammar, there is a grammar rule

$$\textit{typedef-name} ::= \textbf{identifier}$$

where **identifier** is a token. This is designed to ensure that types and variables are maintained in separate namespaces. However, this results in syntactic ambiguities. A parser for this grammar maintains a symbol table for **identifier** and stores the information on whether **identifier** represents a type or a variable when a declaration is found. This information is then used to remove the syntactic ambiguity. Of course, non-generalised parsing techniques are not designed to handle parses that contain syntactic ambiguities, although there are exceptions, for example where longest match and priority can be applied (which is used to handle the dangling-else problem [ISO11, p. 149 in N1570]). For longest match and priority, an LR parser can, for example, perform parse table conflict removal. This is not possible for ambiguities where semantic context is needed.

The alternative approach used in most C parsers (with notable exception to Clang, which uses a modified C parsing grammar that does not distinguish between types and variable names), is to treat **typedef-name** as a terminal. This moves the problem away from syntactic analysis as `a` in the above example will be tokenised as either **typedef-name** or **identifier** with only one derivation for each during syntactic analysis. However, as the two tokens have identical patterns, this simply moves the problem

from a syntactic ambiguity to a lexical ambiguity.

Many bespoke compiler front ends have the parser act as a 'controller' of the entire front-end - the lexer is called only when it needs the next token, and it is able to execute semantic actions on sentence recognition. In this case, the lexer has access to the symbol tables that the parser is constructing and is able to use them to determine which token to return.

This is not always desirable and is not naturally compatible with general lexing and parsing techniques - as with ambiguity there can be more than one semantic interpretation and, therefore, more than one version of the symbol tables. However, a multilexer can simply return a TWE set containing triples for both token matches. The MGLL parser then can parse both interpretations. **`typedef-name`** is syntactically invalid in some places in the grammar where **`identifier`** is valid, and vice versa. Where both interpretations are syntactically valid, such as in the example at the beginning of this section, both derivations can be kept, with one being chosen later through the semantic interpretation.

With generalised parsing, however, one does simply have the option of tokenising both interpretations as **`identifier`**. A generalised parser can handle syntactic ambiguities, so the need to eliminate this ambiguity at lexical level is no longer pertinent. In principle, the ANSI-C standard grammar, as it appears in the language specification, could be used as is, with the ambiguity handled after parsing. What MGLL does offer, though, is a choice over how this ambiguity is handled. The language designer can choose to tokenise a string as both **`typedef-name`** and **`identifier`** - which would eliminate the need for an equivalent *`typedef-name`* non-terminal in the parsing grammar. Or they could tokenise the string as simply **`identifier`** and use the grammar as it appears in the specification. Having such a choice would be particularly useful where one might be adapting a legacy compiler to general multilexing techniques.

Chapter 6

Case Studies

The previous chapters set out a theoretical basis for new approaches which can be used in the automatic generation of a compiler front-end. This chapter will describe how an implementation of these techniques is used to provide solutions to problems in realistic scenarios.

The first section will describe the multilexer parser design for the C# 2.0 language specification [HCC06] (given in Appendix A) up to the construction of the ESPPF via the MGLL algorithm. This section will demonstrate how the multilexer provides a solution to the problems of contextual keywords and nested type parameterisation. Additionally, a comparison will be made to an equivalent character-level GLL parser.

The later sections will then discuss the translation of a string in the C# 1.2 language specification [HCC02] (given in Appendix B) into an AST suitable for applying funcon [Chu+15] semantics - used as the main case study of the PLanCompS [Mos+15] project. After tokenising and parsing the string (using a specification not dissimilar to that used for the C# 2.0 case study), the result is an ESPPF embedding multiple derivations. Section 6.2.2 will discuss the syntactic ambiguity reduction rules which are needed to reduce the ESPPF to a single derivation tree. Section 6.2.3 will then demonstrate how GIFT annotations are used to transform this derivation tree into an abstract syntax tree, as specified by the hand-written abstract syntax for C# 1.2 (given in Appendix C) designed for the PLanCompS project.

The theoretical concepts described in this thesis were implemented as a set of frameworks for a Java 8 JRE [Gos+15]. These frameworks were then used to create compiler front-ends for the case studies described in this chapter. All parsers used in these case studies have been generated by the ART [JS11] parser generator, and the implementation has dependencies on API functions provided by ART. A framework that implements the multilexer techniques, along with the C# 2.0 case study, can be found as a

GitHub project [Mic15b], as can the frameworks for providing the syntactic ambiguity reduction rules and GIFT transformations, applied to the C# 1.2 case study [Mic15a].

6.1 A lexer/parser interface for the C# 2.0 language specification

The C# 2.0 language specification [HCC06] describes a lexical grammar for the C# language. This lexical grammar is given in Appendix A.

The aim is to develop a lexer that, given a lexically valid C# string, produces a lexical ambiguity reduced TWE set embedding C# token strings. Of course, to produce a TWE set for C# strings, an instance of a multilexer needs to be implemented. In this section, the multilexer framework is used to specify the C# lexemes.

As the language specification was designed with traditional lexical analysis in mind, one would expect most lexical ambiguities to be resolvable by longest match and priority. In some cases, there are ambiguities that cannot be simply resolved through these disambiguation mechanism, and require additional context provided by the parser. C# contains tokens that are keywords but are not reserved. This means that a match on one of these keywords could also legitimately be a match on **`identifier`**. This situation cannot be handled by traditional lexical analysis, but can be handled by Schrödinger's tokens [AH01] and the multilexer approach. C# 2.0 also introduced the concept of parameterised types to C#. This introduces a special case of lexical ambiguity as nested parameterised types (such as `List<List<Integer>>`) require strings of > characters to be tokenised as individual **`>`** tokens rather than as **`>>`** tokens. This cannot be handled by traditional lexical analysis, and cannot be sensibly handled by Schrödinger's tokens. This section will demonstrate that this situation can be rather elegantly handled with the multilexer approach.

6.1.1 Layout-token initial processing

Strings are initially processed to replace all strings corresponding to layout (whitespace) tokens with a single `\n` character. If a string is matched by a layout token, then a `\n` character is appended to the output, otherwise the string is appended to the output. **`whitespace`**, **`new-line`**, and **`comment`** are layout tokens in this implementation, corresponding to the layout tokens of C#. **`string-literal`** and **`character-literal`** are included as they are tokens whose patterns contain characters that also appear in patterns of layout tokens. Adding these token definitions ensures characters are not treated as layout if they are part of a character or string literal.

The initial processor takes a string and starts at a character index of 0. An attempt to match against each of the tokens is made. For each token, if a transition on the DFA can be made from the current character index, the transition is made and the character index is incremented. If, after transition, the DFA for the token is in an accepting state, then the current character index is stored, and a flag is set if this token is layout. This continues until there are no more transitions that can be applied or the end of the string is reached. This results in a longest match on the token. This is repeated for the same start index for each token. After all tokens are tried, the longest match over all tokens is used to select the portion of the string matched. If the layout flag was set, then this string is replaced by a `\n` character in the output, otherwise the string is copied to the output and the current character index is advanced to the index after this string. If none of the tokens in the scheme were matched, then a single character is copied and the current character index is incremented.

This is not a general approach, however, this is good enough to initially process C# strings as there is no valid overlap between layout and other tokens in the lexer specification. For the input string given in Figure 6.1, the initial processor will produce the string in Figure 6.2.

```
// This is a comment
class Program {

  /* This is
     a block
     comment */
  public static void Main() {

        System.Console.WriteLine("Hello␣World");

  }

}
```

Figure 6.1: An example of a C# string

The `lexSegmented()` function in the multilexer frameworks splits the string into segments, using `\n` as a delimiter. Each segment is then tokenised as described in 5.4.

6.1.2 Lexer Specification Implementation

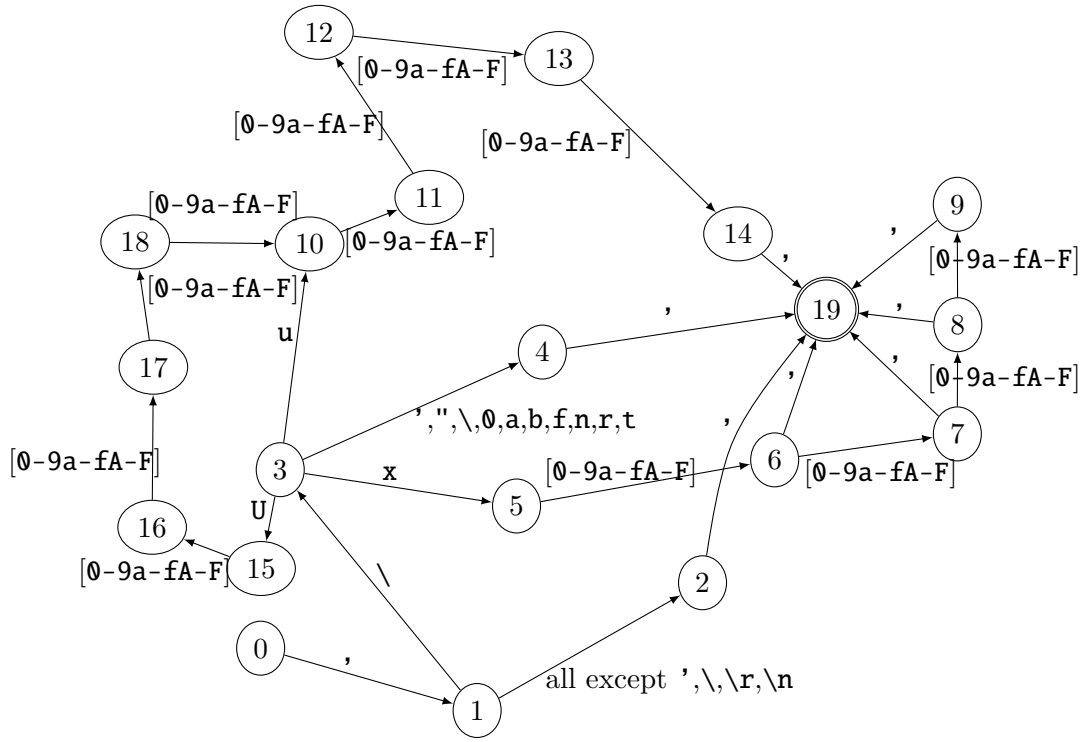
The multilexer framework [Mic15b] is used to generate TWE sets under the C# 2.0 lexer specification. The `DFAMap` interface is implemented to create a class `CS2DFAMap`. Tokens with a single pattern are matched using string comparison; the other tokens are matched using DFAs. These DFAs will be manually constructed although a lexer

```
class
Program
{
public
static
void
Main()
{
System.Console.WriteLine("Hello World");
}
}
```

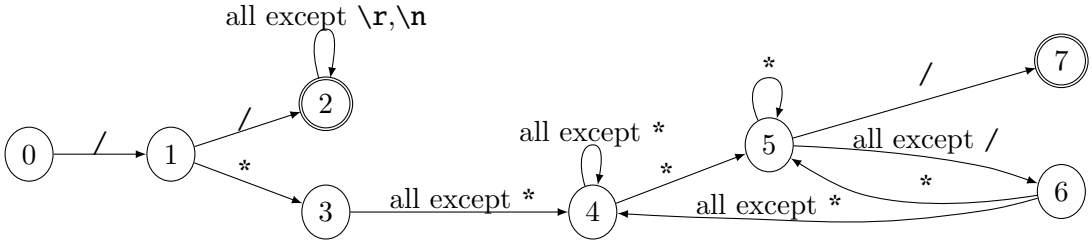
Figure 6.2: Figure 6.1 after processing

generator could automatically generate an implementation of `DFAMap`. When initialised, DFAs are constructed for every token that has more than a single pattern. These DFAs are constructed by hand, although there is no reason why DFAs cannot be constructed mechanically.

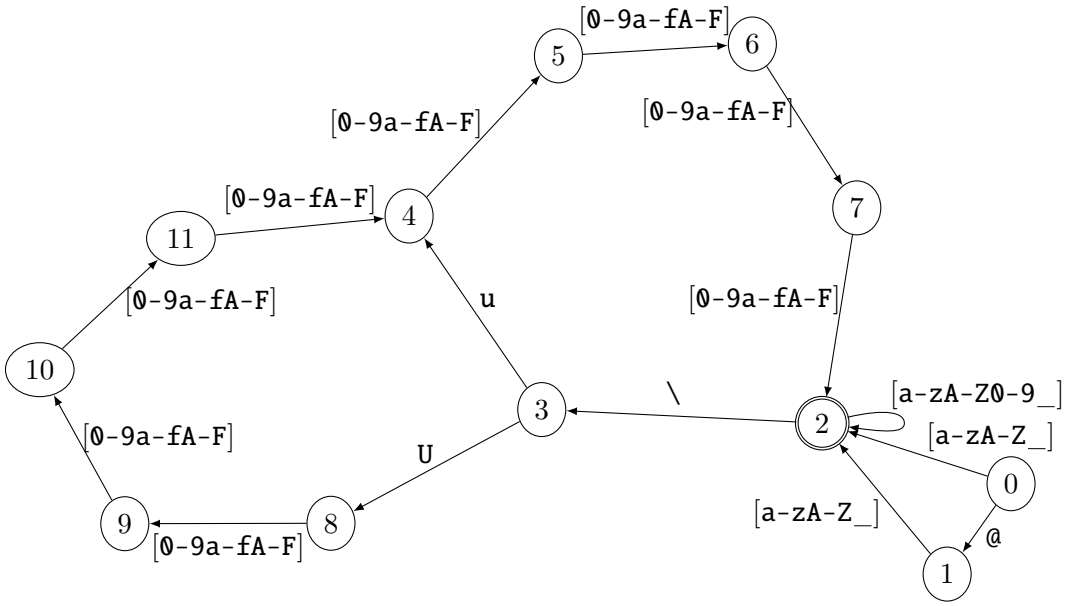
character-literal



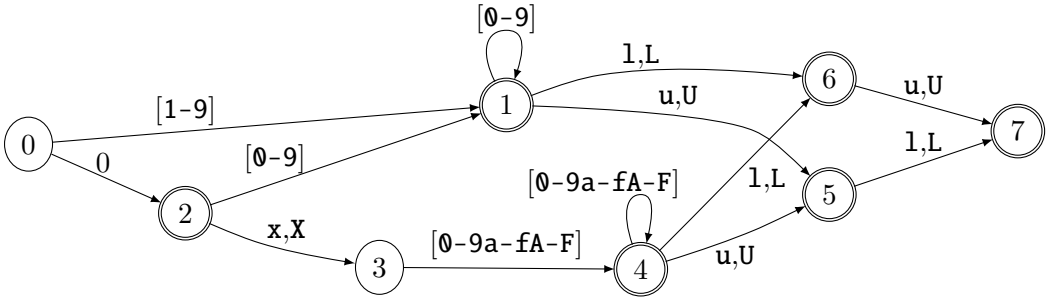
comment



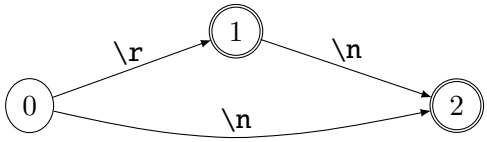
identifier



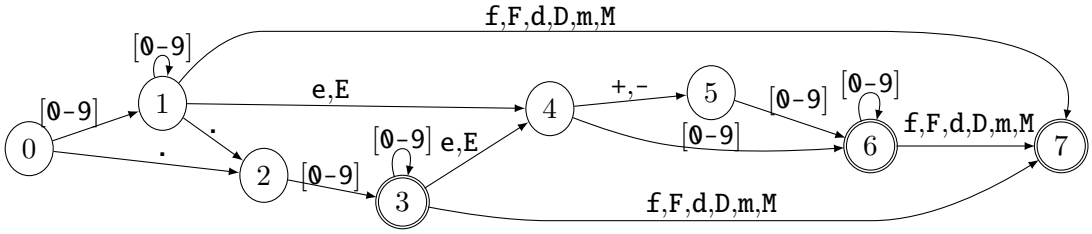
integer-literal



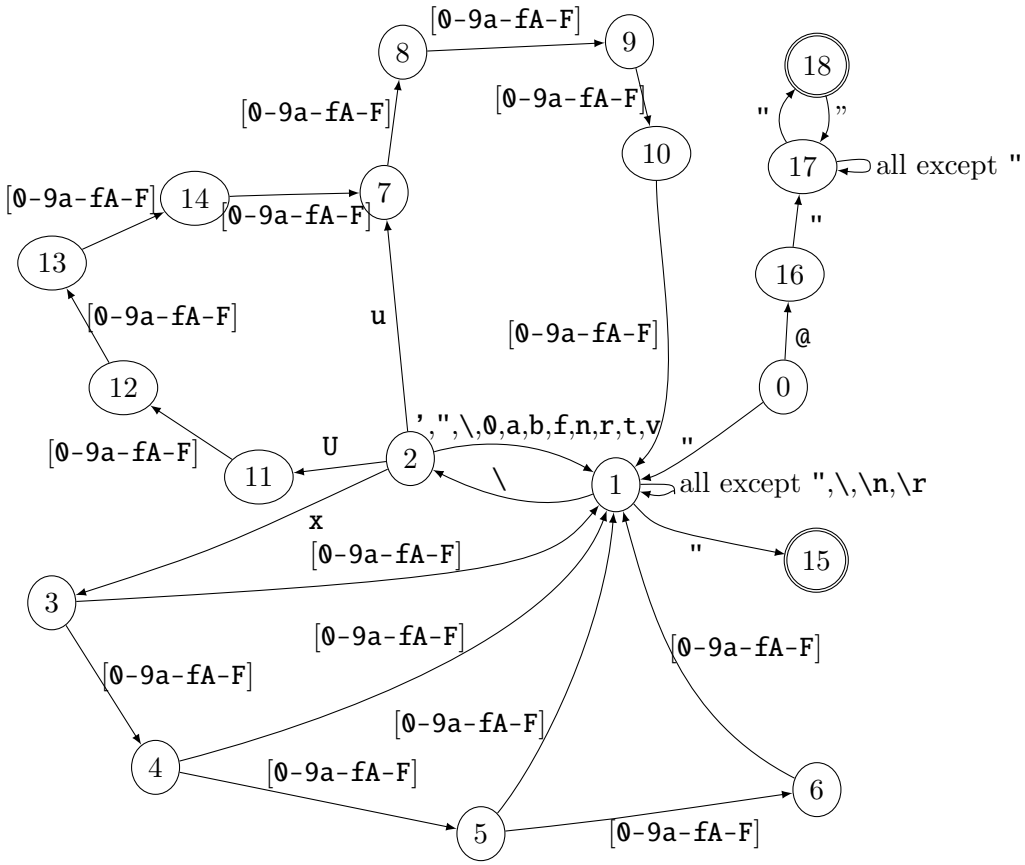
new-line



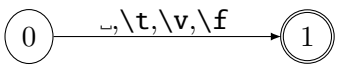
real-literal



string-literal



whitespace



All of these DFAs are stored as entries in `DFAMap`, indexed by the token name. This DFA Map implementation for C# is passed in when initialising an instance of `RegularLexer` to create a lexer for C#.

With the lexer defined, TWE sets for a given processed input string are then constructed using the Multilexer framework.

6.1.3 Lexical Disambiguation

There are lexical ambiguities in the C# lexical specification. In traditional lexical analysis, these lexical ambiguities are handled by longest match and priority. As discussed at the beginning of this section, there are two cases that traditional lexical disambiguation mechanisms cannot handle. In one case, the requirements of the lexer cannot be sensibly provided by even Schrödinger's tokens. As the multilexer approach is designed to give the user more control, it can be demonstrated that a multilexer specification can comfortably handle both of these cases. The lexical ambiguity reduction techniques described in 2.5 are used to reduce the set of indexed token strings to only those strings that are expected according to the C# specification.

Ambiguities Handled only by the Multilexer Approach

Some operators are prefixes of other operators - such as **+** and **++**. In this case the requirement is that **++** is chosen over **+**. This behaviour can be modelled by specifying the relation **+** R **++** under the relation matrix for class two operations. Similarly, for the operators **<** and **<<**, **<<** should be chosen over **<** so the class two relation **<** R **<<** is specified. These situations can also be comfortably handled by traditional lexical disambiguation.

Logically, one would expect a similar relation for the operators **>** and **>>**. However, the C# 2.0 parsing grammar contains the following sequence of grammar rules for specifying parameterised types

```
type ::= reference-type | ...
reference-type ::= class-type | ...
class-type ::= type-name | ...
type-name ::= namespace-or-type-name
namespace-or-type-name ::= identifier type-argument-list | ...
type-argument-list ::= < type-arguments >
type-arguments ::= type-argument | type-arguments , type-argument
type-argument ::= type
```

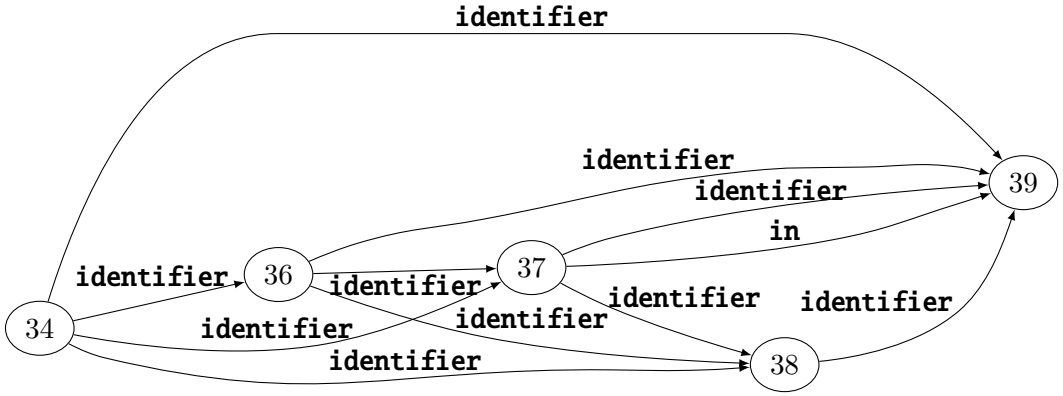
The string `A<B<C>>` should be derivable from ***type***. If the class two relation **>** R **>>** was specified, then the closing brackets of the type parameters would be tokenised as a single **>>** token. This would cause the string to be rejected by the parser as it is expecting two **>** tokens. As the **>>** tokenisation is still desired in the case of a right-shift expression, which tokenisation to use is dependent on the surrounding context. The multilexer approach is able to provide both tokenisations to the parser, with the parser rejecting one tokenisation when the context becomes clear. The multilexer approach can resolve this simply by not specifying a relation for **>** and **>>**.

Ambiguities Handled by Other Techniques

Of course, it is also important that the multilexer approach can handle cases of lexical ambiguity that are handled by traditional lexical analysis and Schrödinger's tokens.

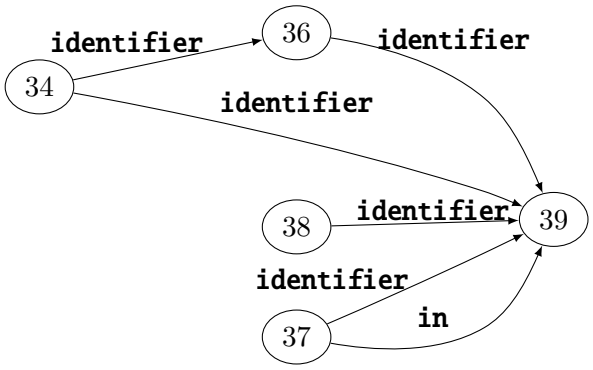
A situation handled by Schrödinger's tokens but not by traditional lexical analysis are the *contextual* keywords. A contextual keyword (such as **`yield`**) is only interpreted as such under certain contexts, being interpreted as **`identifier`** otherwise. This would traditionally require the intervention of the parser. These contextual keywords would traditionally be matched as **`identifier`** before being rewritten during syntactic analysis. This is not necessary with Schrödinger's tokens, as these would be simply tokenised as a Schrödinger's token. In the multilexer, it is equally as straightforward. Both keyword and **`identifier`** tokens are kept in the resulting TWE set. An MGLL parser will then eliminate one tokenisation as a result of parsing.

A lexical ambiguity typically handled by traditional lexical analysis is demonstrated with **`identifier`**. For example, the string `Main` in Figure 6.2 on page 143 corresponds to the following segment of the TWE graph for the entire string



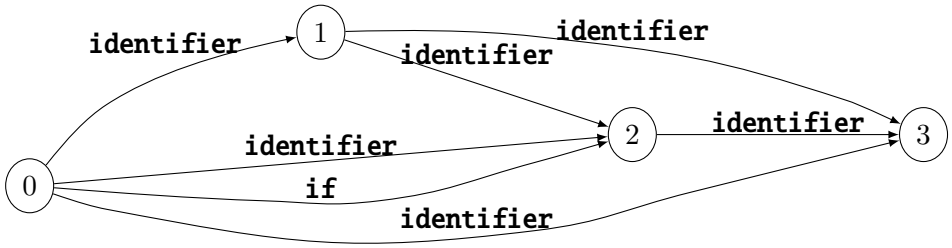
For any string matched by **`identifier`** whose length is greater than 1, every substring of this string is also matched by **`identifier`**. This is an instance of the worst case in the multilexer approach as, for an identifier of length n, there are n^2-1 ways to tokenise this identifier. In practice, only the longest match of **`identifier`** is required, which is specified with the relation **`identifier`** R **`identifier`** under the relation matrix for class 2 operations[1] - this avoids all these **`identifier`** matches being passed to the parser. By applying the rule in this example, one gets the TWE graph

[1]The reader may also notice the presence of (**`in`**, 37, 39) in this TWE graph. The relation between keywords and **`identifier`** will be discussed shortly. This particular triple will be removed through pruning without direct intervention after the **`identifier`** ambiguity is handled.

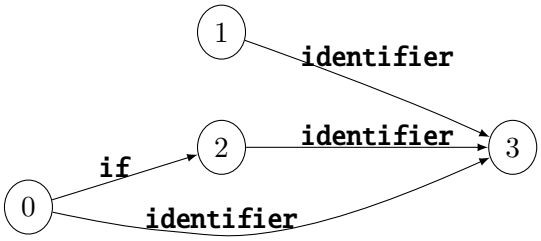


After pruning of the TWE set, only (**identifier**, 34, 39) will remain.

Many of the keywords in the lexical specification have a pattern that overlaps with patterns for **identifier**; these are all handled using the same technique: for example consider the string `ifa` which will have the TWE graph



The use of **identifier** R **identifier** under the relation matrix for class 2 operations will reduce this graph to



This still leaves two indexed token strings, (**identifier**, 3) and (**if**, 2)(**identifier**, 3). The keyword token match is a proper prefix of the longer **identifier** token match. In this particular case, the C# specification requires the ITS, (**identifier**, 3). This is specified with the relation **if** R **identifier** under the relation matrix for class 2 operations. Equivalent rules are specified for all keyword tokens. Longest match is symmetric in traditional lexical analysis. Whilst the relation above is asymmetric, one may optionally ensure symmetry by additionally specifying that **identifier** R **if** under the relation matrix for class 2 operations. This would eliminate **identifier** in the case where the **identifier** match is a proper prefix of the keyword match. This is not

necessary, however, since C# keywords are always prefixes of a string in the pattern of **identifier**. The class 2 relation **identifier** R **identifier** eliminates **identifier** token matches that are proper prefixes of a keyword, as the longest **identifier** match is either the same length as the keyword match or longer.

There is also an issue for keywords whose pattern is a prefix of the pattern of another keyword. An example of this is **for** and **foreach**. The resolution of this is straightforward - the longer keyword should always be chosen, by specifying the class 2 relation **for** R **foreach**. A more complex case occurs for the keyword **in**, whose pattern is a prefix of the pattern for **int**, which is also a prefix of the patterns for **interface**, and **internal**. In this case, the following class 2 relations are used

in R **int**

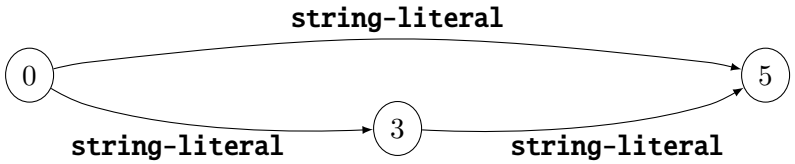
in R **internal**

in R **interface**

int R **internal**

int R **interface**

There are two types of **string-literal** in C# - regular and verbatim. A verbatim string, which is preceded with an `@` symbol, will accept any sequence of characters between two double quotes, with double quotes themselves being escaped if they are preceded by a double quote. This can lead to a lexical ambiguity: consider the TWE graph for the string `@""""`



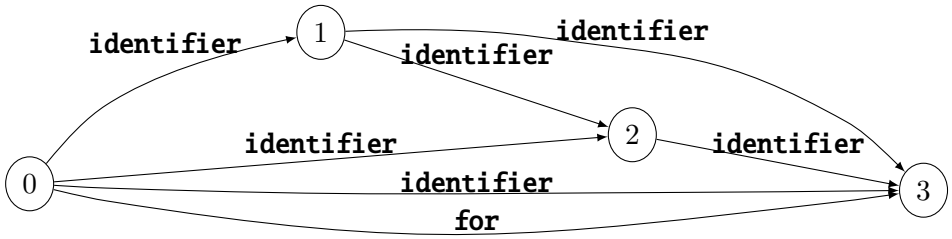
There are two tokenisations embedded in this TWE graph

$$\{(\textbf{string-literal}, 3)(\textbf{string-literal}, 5),$$
$$(\textbf{string-literal}, 5)\}$$

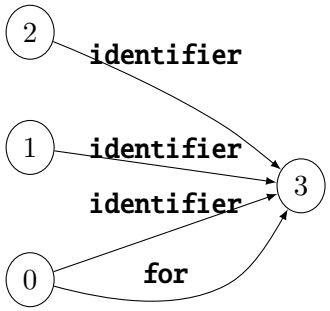
It is clear that the latter tokenisation (which would interpret this string as a verbatim string containing an escaped double quote) is the correct interpretation. The class 2 relation **string-literal** R **string-literal** is used to ensure this tokenisation is selected.

The rules described so far handle the case where a keyword pattern is a proper prefix

of an **identifier** pattern, as well as **identifier** patterns that are proper prefixes of other **identifier** patterns. The case where an **identifier** pattern is a proper prefix of a keyword is also handled implicitly. However, there is still the situation where the **identifier** pattern is the same length as the keyword pattern. This will occur for every keyword in C# since every keyword pattern is also a pattern in **identifier**. Consider the TWE graph for the string `for`

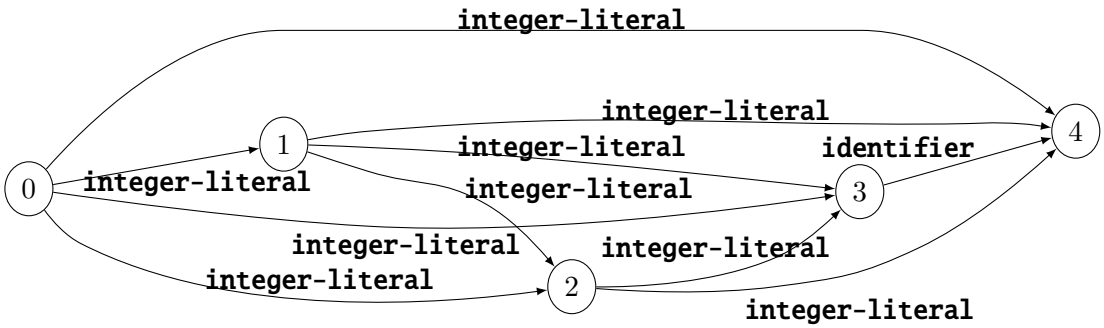


After applying all previously defined disambiguation rules, the resulting TWE graph is

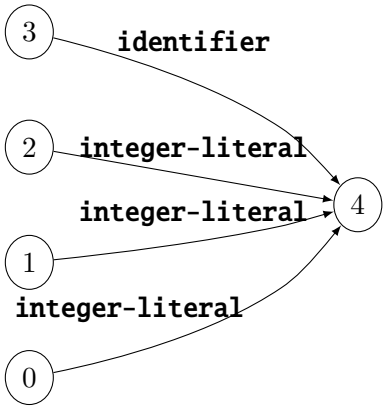


This leaves two indexed token strings: (**identifier**, 3) and (**for**, 3). The match chosen depends on whether the keyword is *reserved* or *contextual*. As described, the multilexer approach handles *contextual* keywords by maintaining both tokenisations. By contrast, a reserved keyword is one that can never be used as an identifier - **for** is an example of such a keyword. As the keyword is reserved, **identifier** cannot be a match for this string. This relationship is specified by **identifier** R **for** under the relation matrix for class 1 operations, with similar relationships being specified for every reserved word. This removes the triple (**identifier**, 0, 3), leaving the ITS (**for**, 3) as the only match.

The one remaining lexical ambiguity to discuss surrounds **integer-literal** and **real-literal**. Consider the string `123u`, whose TWE graph is

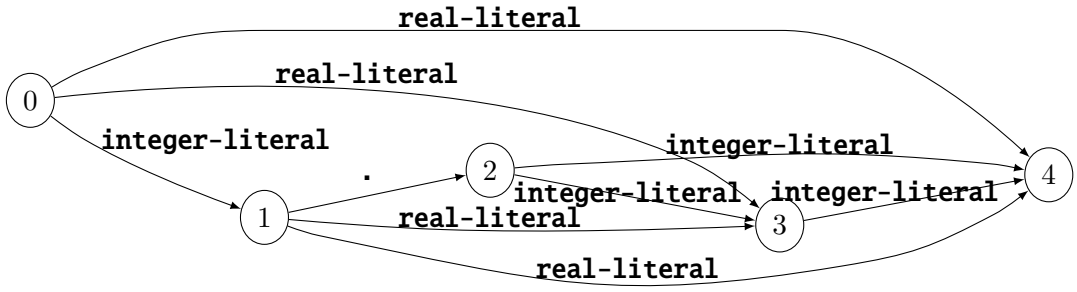


There are patterns of **integer-literal** that are proper prefixes of other patterns of **integer-literal**. In practice, only the longest match of **integer-literal** is wanted - achieved by specifying that **integer-literal** R **integer-literal** under the relation matrix for class 2 operations. This results in the TWE graph



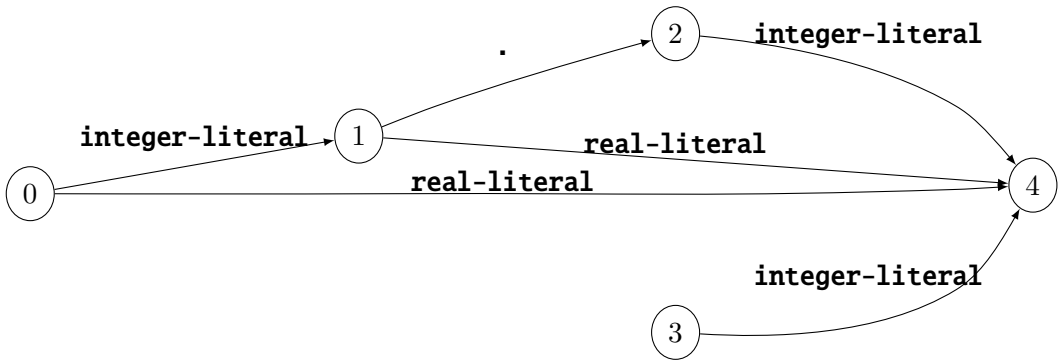
The remaining ITS is (**integer-literal**, 0, 4). Although **identifier** matches **u**, this does not need to be handled explicitly, as any rule that ensures the longest match of **integer-literal** on itself will eliminate this match after pruning.

There is also a lexical ambiguity between **integer-literal** and **real-literal**. Consider the TWE graph for the string `1.55`



The rule for **integer-literal** above will eliminate the triple (**integer-literal**, 2, 3), however this still leaves other ambiguities. One such ambiguity is caused as a result of

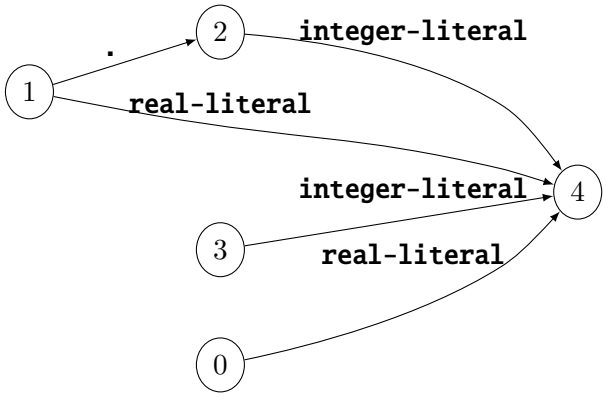
patterns for **real-literal** being proper prefixes of other patterns in **real-literal**. As with **integer-literal**, specifying that **real-literal** R **real-literal** under the relation matrix for class 2 operations is enough to eliminate this particular ambiguity, giving the TWE graph



This leaves three embedded indexed token strings

$$\{(\textbf{real-literal}, 4),$$
$$(\textbf{integer-literal}, 1)(\textbf{real-literal}, 4),$$
$$(\textbf{integer-literal}, 1)(\textbf{.}, 2)(\textbf{integer-literal}, 4)\}$$

Only the first ITS is considered valid by the C# language specification. To capture this, it is enough to specify that **integer-literal** R **real-literal** under the relation matrix for class 2 operations. This leaves the TWE graph



This reduces to just a single match. Had the string been simply `.55`, then there would still be two matches $(\textbf{.}, 1)(\textbf{integer-literal}, 3)$, and $(\textbf{real-literal}, 3)$. This is resolved by adding an additional relation $\textbf{.}R$ **real-literal** under the relation matrix for class 2 operations.

For the example C# string in Figure 6.2, the result given by the multilexer, including lexical ambiguity reduction and pruning, is the TWE set

$$
\begin{aligned}
\{ \\
&(\texttt{class}, 0, 5), (\texttt{identifier}, 5, 13), (\texttt{\{}, 13, 15), \\
&(\texttt{public}, 15, 22), (\texttt{static}, 22, 29), (\texttt{void}, 29, 34), \\
&(\texttt{identifier}, 34, 39), (\texttt{(}, 39, 40), (\texttt{)}, 40, 41), \\
&(\texttt{\{}, 41, 43), (\texttt{identifier}, 43, 50), (\texttt{.}, 50, 51), \\
&(\texttt{identifier}, 51, 58), (\texttt{.}, 58, 59), (\texttt{identifier}, 59, 68), \\
&(\texttt{(}, 68, 69), (\texttt{string-literal}, 69, 82), (\texttt{)}, 82, 83), \\
&(\texttt{;}, 83, 84), (\texttt{\}}, 84, 86), (\texttt{\}}, 86, 88) \\
\}
\end{aligned}
$$

Notably, in this case it also only embeds a single ITS. This TWE set is given to an MGLL parser for the grammar in Appendix A.

6.1.4 Comparison with a Character-level Implementation

As discussed in Chapter 5, one alternative to this approach is to remove the lexical analysis phase altogether and instead parse at character-level. The discussion suggested that the multilexer approach offered the same level of power and control as a character-level approach, whilst retaining some of the advantages of a token-based approach. This included a claim of greater efficiency when the patterns of tokens are regular languages, as well as efficiency from token-level lookahead. Lexical level disambiguation also allows tokenisations to be filtered in a clean manner before being given to the parser. This section will explore this by testing the character-level and multilexer approach on a lexer/parser specification for the C# 2.0 specification.

An MGLL parser for the grammar in Appendix A was constructed. Additionally, a character-level version of this grammar was constructed, by replacing all token terminals with an equivalent non-terminal, whose productions are the productions given in Appendix A (keywords, of course, have productions that derive the sequence of characters in the pattern) with every token non-terminal also optionally deriving a newline character as its first symbol. As the test string set, the set of test strings for the Mono implementations of the C# 1.2 and 2.0 specifications [Pro06, `/mono/tests`; Pro11, `/mono/tests`] were used along with the file Lexer.cs from the LitJSON library [B14, `/src/LitJson/Lexer.cs`] as an example of a large C# string. These strings were processed to remove all C# preprocessing directives and then given to the initial processor.

The final set of test strings used is the output of this initial processor.

The character-level parser does not apply any disambiguation - the SPPF constructed is the full SPPF for the character string. For the multilexer parser approach, there are two variations considered - one in which no lexical ambiguity reduction rules are applied, and the other in which all the lexical ambiguity reduction rules given above are applied.

The tests measure three different criteria. The first is the total amount of time (in seconds) it takes to complete the construction of the (E)SPPF for the string. For the character-level parser, this is simply the time it takes to parse the string. For the multilexer parser approach, this is the time that was taken to tokenise the character string and produce the TWE set that is given as output, and then the time taken to parse this set with the MGLL parser.

The second criterion is the size of the resulting (E)SPPF - this is measured in the number of nodes. There are two measures of the (E)SPPF size, the first is the measure of the number of nodes reachable from the root node, and the second is the measure of the number of nodes constructed by the parser in total - including those constructed by 'abandoned' parses . For the character-level parser, any nodes that are descendants of token non-terminal nodes are not included in the count. The expectation is that the number of nodes that are reachable from the root for a given character string will be the same for both character-level parsing and multilexing parsing without lexical disambiguation. One would expect the number of nodes to be less for multilexing parsing with lexical disambiguation.

The third criterion is the number of descriptors constructed in each approach. This will give an insight as to how much parser activity occurs under the different approaches.

To ensure consistency in the timing results, the Java Runtime is executed with the `CompileThreshold` flag set to 10. For each string, every test is run 21 times with only the last 10 runs recorded. The median of the last 10 runs is then given as the timing result. The experiment was carried out on a 64-bit Linux machine, with an Intel Core i7-4710MQ CPU @ 2.50GHz, and 7.7GiB of memory. Time measurements were made using Java's `System.nanotime()` method.

6.1.5 Results

For each criterion, a table of results was produced, providing data for each string in the test set. The length of the string was also recorded in each case. These strings are varied, and two strings of the same length may have vastly different results depending on what these strings contain. Therefore, the results for each criterion are plotted on scatter diagrams to determine the general trend. As there are relatively

few very large examples, a base-2 logarithmic scale is used in the representation. The full table of results is given in the GitHub repository [Mic15b, `/testSuite/2_0/_performanceResults.csv`].

The results for measuring the total time taken to lex and parse a string are found in Figure 6.3. When the multilexer parser approach is used with lexical disambiguation, the time that was taken to lex and parse is faster, on average, than both character-level parsing and multilexer parsing without lexical disambiguation. That multilexer parsing with lexical disambiguation performs better than without is not surprising - when the number of tokenisations is reduced, there is certainly less work required by the parser. The multilexer performs worst case quadratic in the length of the character string, whilst MGLL may be upper-bounded at quartic, so reducing the workload of the parser will lead to greater gains in performance. Even without lexical disambiguation, the multilexer parser generally performs better than the character-level parser, with the difference most noticeable in the largest case.

In some cases, the character-level parser does appear to perform better than both variations of the multilexer parser approach. In these cases, the string contains a significant number of usages of **`identifier`** which invokes worst case behaviour in the lexer, whilst being relatively simple for the parser. As lexical disambiguation only happens after the initial TWE set is constructed, even the multilexer with lexical disambiguation will invoke this worst case behaviour. On the other hand, the character-level parser has the advantage of being able to eliminate tokenisations through context. As there is no point in the parsing grammar where two **`identifier`** tokens are concatenated, the character-level parser can eliminate some tokenisations during the parse. This demonstrates that, depending on the grammar, the character-level may have the advantage of being able to eliminate tokenisations through context. However, the multilexer approach performs better in most other cases.

For the count of (E)SPPF nodes reachable from the root, the number of nodes, without counting nodes under token non-terminals, constructed by the character-level parser match the number of nodes constructed by multilexing without disambiguation, supporting the hypothesis. When comparing the number of (E)SPPF nodes constructed in total, the results are seen in Figure 6.4. It is notable that the character-level parser constructs more SPPF nodes than the multilexer parser approach. This demonstrates the savings that result from a token-based lookahead compared to a character-based lookahead, leading to fewer failed parse attempts in the former.

For the descriptor count, the need to also parse token non-terminals leads to a much greater descriptor count for character-level parsing, as seen in Figure 6.5.

Overall, the results are promising for the multilexer parser approach. The results

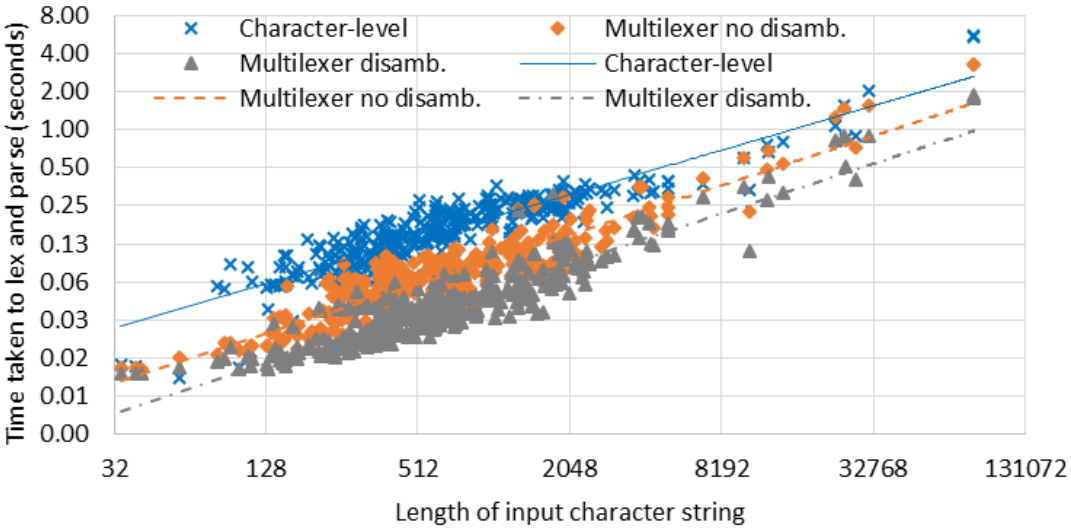


Figure 6.3: Plot of C# string lengths and time taken to lex and parse them

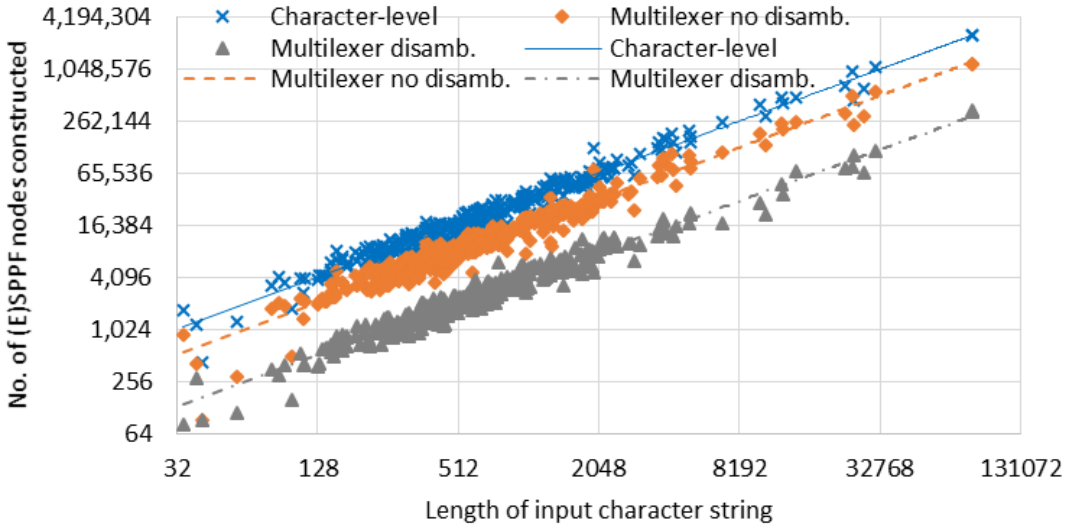


Figure 6.4: Plot of C# string lengths against (E)SPPF node counts

generally show an improvement over character-level parsing. Space savings are particularly marked. As the multilexer parser approach constructs ESPPFs that are analogous to the SPPFs constructed by character-level parsing, this demonstrates that the multilexer parser approach can offer the same power and control as character-level parsing but with greater efficiency.

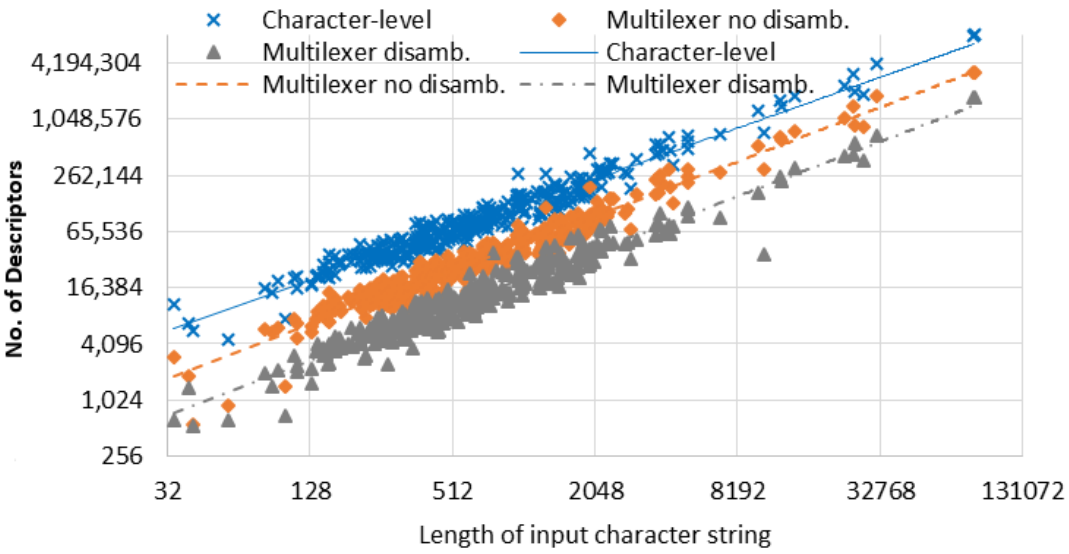


Figure 6.5: Plot of C# string lengths against descriptor counts

6.2 Generating a Structural Abstract Syntax Tree for the C# 1.2 language specification

The PLanCompS [Mos+15] project includes a case study that considers an implementation for the C# 1.2 language specification [HCC02]. The goal was to give a component-based specification of the syntax and semantics of the language. The syntax can be described using the parsing grammar in conjunction with the lexical specification (given in Appendix B). The semantics can then be described using a composition of fundamental programming constructs, or *funcons* [Chu+15]. However, moving from the syntax to semantics requires, first, the disambiguation of the resulting derivation structure and, second, the translation of the derivation structure to an AST suitable for the semantics. In this case, a description of the ASTs required for the semantics is specified by a hand-written abstract syntax (given in Appendix C) for the C# 1.2 language.

Of course, to describe the transition from the ESPPF to AST, it is first necessary to construct the ESPPF. An MGLL parser for the C# 1.2 concrete syntax is used along with a multilexer specification that is similar to that described in the previous section. The C# 1.2 multilexer specification differs only in that a different set of contextual keywords is used, and that, as C# 1.2 does not include generics, the class two relation `>` R `>>` is specified. The implementation of the C# 1.2 lexer specification can be seen in the class `CSDFAMap` in the GitHub repository [Mic15a].

This section will first describe the syntactic ambiguity reduction rules needed to resolve the ambiguities in the ESPPF for the parsed string. It will then follow with a description of the GIFT transformations necessary to translate the resulting derivation trees into an equivalent form described by the abstract syntax.

6.2.1 Funcons

Whilst the formal semantics is not considered for this thesis, a brief and informal introduction of the semantics formalism used for this case study will be given. Works that give a more detailed description of the formal semantics can be found in the bibliography [Chu+15; Mos08].

The aim of the PLanCompS project was to develop a component-based approach to semantics specifications. Many programming constructs are common across many languages (such as while loops and if statements). In many cases, how programming constructs are evaluated is independent of the result of other constructs - for example, the contents of an if statement do not affect how an if statement evaluates its contents.

These observations are taken advantage of in component-based semantics through the specification of *fundamental constructs* (funcons). These funcons are effectively simplified language constructs. For example, the funcon `if-then-else`(E_1, E_2, E_3) corresponds to the concept of an if statement. Funcons are composed to specify the semantics of a sentence. For example, the C# string `a=b+c` could translate to the funcon string `bind("a",integer-add(bound("b"),bound("c")))`. The semantics of each funcon is specified by some modular semantics framework (such as MSOS [D04]).

Source language constructs are mapped to funcons through inductively defined rewrite functions, for instance

$$expr[\![E_1 == E_2]\!] = \texttt{supply}(expr[\![E_1]\!], \texttt{equal}(\texttt{given}, expr[\![E_2]\!]))$$

A rewrite function will need to be written for every syntactic form. Therefore, as mentioned in Chapter 4, it is beneficial to reduce the number of syntactic forms. The abstract syntax used for the semantics is a structurally concise version of the concrete syntax - representing only the structure essential to the semantics. As the AST would have been constructed from derivations in the concrete syntax, ambiguity in the abstract syntax is not an issue. As the aim is to produce human-readable descriptions of the semantics, terminal symbols are maintained in this abstract syntax. This is to make it easier to document how the semantics relates to source language constructs without requiring an explicit mapping back to the concrete syntax [Mos08]. The language of the abstract syntax is therefore a superset of the language of the concrete syntax.

6.2.2 Syntactic Ambiguity Reduction of C# 1.2

After the lexical analysis step, the resulting TWE set is given to the MGLL parser for C# 1.2, that is based on the grammar given in Appendix B. Although this grammar is less ambiguous than the abstract syntax for C# given in Appendix C, it is still ambiguous in a number of areas. As such, the resulting ESPPF will, in most cases, embed more than one derivation tree. This section uses the syntactic ambiguity reduction techniques described in Chapter 5 to reduce the level of ambiguity in the resulting ESPPF. Creating a syntactic ambiguity reduction specification requires decisions to be made, using a combination of the informal descriptions given in the C# specification and knowledge of the abstract syntax trees produced by the C# abstract grammar in Appendix C. The result after applying all ambiguity reduction rules is, in this case, a single derivation tree.

In a few cases, ambiguities result from the concrete syntax providing more structural information than is required by the semantics. The derivation trees rooted at these ambiguities will map to the same abstract syntax tree. Therefore, it does not matter which derivation tree is selected. An example of this is found in the grammar segment

```
reference-type ::= class-type | interface-type | array-type | delegate-type
class-type ::= type-name | object | string
interface-type ::= type-name
delegate-type ::= type-name
type-name ::= namespace-or-type-name
namespace-or-type-name ::= identifier | namespace-or-type-name . identifier
```

it is possible to derive *type-name* from *reference-type* in three ways as shown by the ESPPF fragment in Figure 6.6. Similarly, in the grammar segment

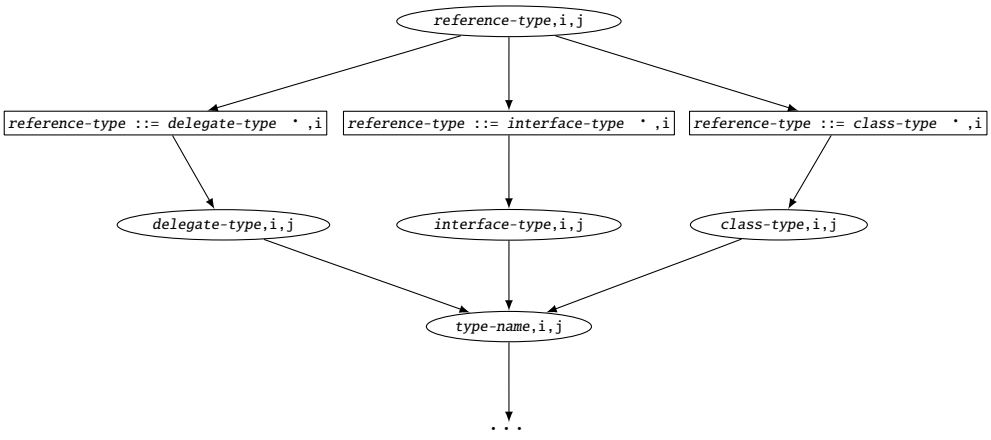


Figure 6.6: Demonstration of ambiguity in *reference-type*

```
value-type ::= struct-type | enum-type
struct-type ::= type-name | simple-type
enum-type ::= type-name
```

it is possible to derive *type-name* from *value-type* in two ways as seen in Figure 6.7. In addition, the grammar rule for *type* is

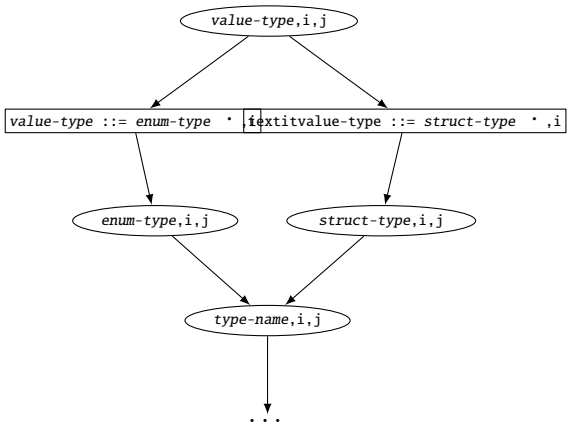


Figure 6.7: Demonstration of ambiguity in *value-type*

```
type ::= value-type | reference-type
```

which means the ambiguities in *value-type* and *reference-type* also cause an additional ambiguity under *type* as seen in Figure 6.8.

To resolve this ambiguity, it is not enough simply to make a decision between the packed nodes that are under *type*. One must also make a decision for the packed nodes under *reference-type* and *value-type*.

In the abstract syntax, the rule for *type* is

```
type ::= predefined-type | qualified-identifier | array-type
qualified-identifier ::= identifier ( . identifier )*
```

The non-terminal *qualified-identifier* replaces *type-name*, and is now the immediate descendant of *type*. For strings in the sublanguage generated by *type-name* in the concrete syntax, there is only a single derivation from *type* in the abstract syntax. All derivation trees from *type* deriving *type-name* in the concrete syntax will be transformed into the same AST. As a result, the decisions that need to be made to reduce the number of ambiguities do not matter and the following rules select packed nodes arbitrarily

$suppress(\texttt{type} ::= \texttt{value-type}\cdot, \texttt{type} ::= \texttt{reference-type}\cdot)$

$suppress(\texttt{reference-type} ::= \texttt{delegate-type}\cdot, \texttt{reference-type} ::= \texttt{class-type}\cdot)$

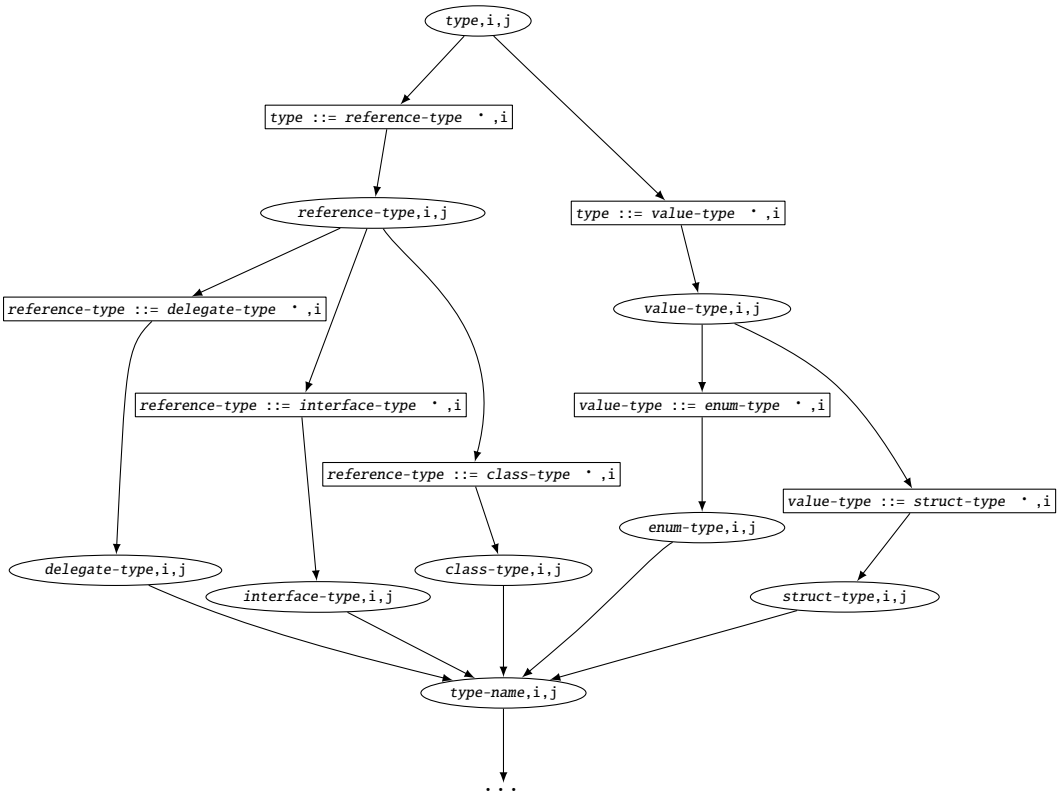


Figure 6.8: Demonstration of ambiguity under *type*

suppress(*reference-type* ::= *delegate-type*·,
reference-type ::= *interface-type*·)
suppress(*reference-type* ::= *interface-type*·,
reference-type ::= *class-type*·)
suppress(*value-type* ::= *struct-type*·, *value-type* ::= *enum-type*·)

Another case of an ambiguity where multiple derivation trees map to a single abstract syntax tree is in the following concrete syntax grammar segment

```
class-base ::= : class-type | : interface-type-list | : class-type ,
    interface-type-list
interface-type-list ::= interface-type | interface-type-list , interface-type
class-type ::= type-name | object | string
interface-type ::= type-name
```

It is possible to derive `:type-name` in two ways. Additionally for a comma-delimited list of *type-name*, such as in `:type-name,type-name`, there are two ways to derive from *class-base*. In the C# language, a class can inherit from one superclass, and

can implement one of more interfaces. The syntax, as shown here, is the same for both cases. The ambiguity arises as `type-name` could be either a class or an interface. For the funcon semantics, it is not necessary to make the distinction. In the abstract syntax, the rule for *class-base* is

```
class-base ::= : ( object | string | qualified-identifier ) ( ,
    qualified-identifier )*
```

Again, the decision on which packed nodes to suppress can be made arbitrarily. The rules that were chosen are as follows

$$
\begin{aligned}
&\mathit{suppress}(\texttt{class-base} ::= \texttt{:}\ \texttt{class-type}\cdot,\\
&\quad \texttt{class-base} ::= \texttt{:}\ \texttt{interface-type-list}\cdot)\\
&\mathit{longest}(\texttt{class-base} ::= \texttt{:}\ \texttt{class-type}\ \texttt{,}\ \texttt{interface-type-list}\cdot,\\
&\quad \texttt{class-base} ::= \texttt{:}\ \texttt{interface-type-list}\cdot)\\
&\mathit{shortest}(\texttt{class-base} ::= \texttt{:}\ \texttt{class-type}\ \texttt{,}\ \texttt{interface-type-list}\cdot,\\
&\quad \texttt{class-base} ::= \texttt{:}\ \texttt{interface-type-list}\cdot)\\
&\mathit{suppress}(\texttt{class-base} ::= \texttt{:}\ \texttt{class-type}\ \texttt{,}\ \texttt{interface-type-list}\cdot,\\
&\quad \texttt{class-base} ::= \texttt{:}\ \texttt{interface-type-list}\cdot)
\end{aligned}
$$

When the ambiguity involves comma-delimited lists, the pivot values are different, *class-base* ::= `:` *class-type* `,` *interface-type-list*· is suppressed regardless of the pivot value. As discussed in 5.2, this is achieved by defining a relation for this grammar slot for all three types of syntactic ambiguity rule.

There is one last case of ambiguity in the concrete syntax in which the abstract syntax removes the distinction between the ambiguous derivations

```
primary-no-array-creation-expression ::= delegate-creation-expression |
    object-creation-expression | ...
delegate-creation-expression ::= new delegate-type ( expression )
object-creation-expression ::= new type ( argument-list ) new type ( )
delegate-type ::= type-name
```

As established earlier, *type* is able to derive *type-name*. Also *argument-list* can derive *expression*. As a result, there is an ambiguity rooted at *primary-no-array-creation-expression* for a string of form `new` *type* `(` *expression* `)`. In the abstract syntax, *primary-no-array-creation-expression* is folded to become part of *expression*. *delegate-type* is replaced by *type* and the two productions are merged into a single production:

```
expression ::= new type ( argument-list? ) | ...
```

This eliminates the ambiguity in the abstract syntax. The decision on which rule to use is arbitrary

$$
\begin{aligned}
&\mathit{suppress}(\texttt{primary-no-array-creation-expression} ::= \\
&\quad \texttt{delegate-creation-expression}\cdot, \\
&\texttt{primary-no-array-creation-expression} ::= \\
&\quad \texttt{object-creation-expression}\cdot)
\end{aligned}
$$

Of course, not every ambiguity is resolved by the abstract syntax. Others must be resolved through an understanding of the language specification. Like most C-like languages, C# experiences the dangling-else ambiguity. The subgrammar for dealing with if-then-else statements is as follows

```
if-statement ::= if ( boolean-expression ) embedded-statement
  | if ( boolean-expression ) embedded-statement else embedded-statement
embedded-statement ::= selection-statement | ...
selection-statement ::= if-statement | ...
```

A sentence of form

```
if(boolean-expression)if(boolean-expression) embedded-statement
  else embedded-statement
```

can be derived in two ways from *if-statement* as seen in Figure 6.9. From the derivation tree rooted at the packed node labelled with pivot **h**, the **else** clause is a member of the second if statement. From the derivation tree rooted at the packed node labelled with pivot **k**, the **else** clause is a member of the first if statement. The C# specification states that an **else** clause binds to the lexically closest unmatched **if** statement. The derivation tree rooted at the packed node labelled with pivot **h** is therefore chosen. As the pivots are different, the rules needed to capture this desired result are as follows

$$
\begin{aligned}
&\mathit{longest}(\texttt{if-statement} ::= \textbf{if (}\ \texttt{boolean-expression}\ \textbf{)} \\
&\quad \texttt{embedded-statement}\ \textbf{else}\ \texttt{embedded-statement}\cdot, \\
&\texttt{if-statement} ::= \textbf{if (}\ \texttt{boolean-expression}\ \textbf{)}\ \texttt{embedded-statement}) \\
&\mathit{shortest}(\texttt{if-statement} ::= \textbf{if (}\ \texttt{boolean-expression}\ \textbf{)} \\
&\quad \texttt{embedded-statement}\ \textbf{else}\ \texttt{embedded-statement}\cdot, \\
&\texttt{if-statement} ::= \textbf{if (}\ \texttt{boolean-expression}\ \textbf{)}\ \texttt{embedded-statement})
\end{aligned}
$$

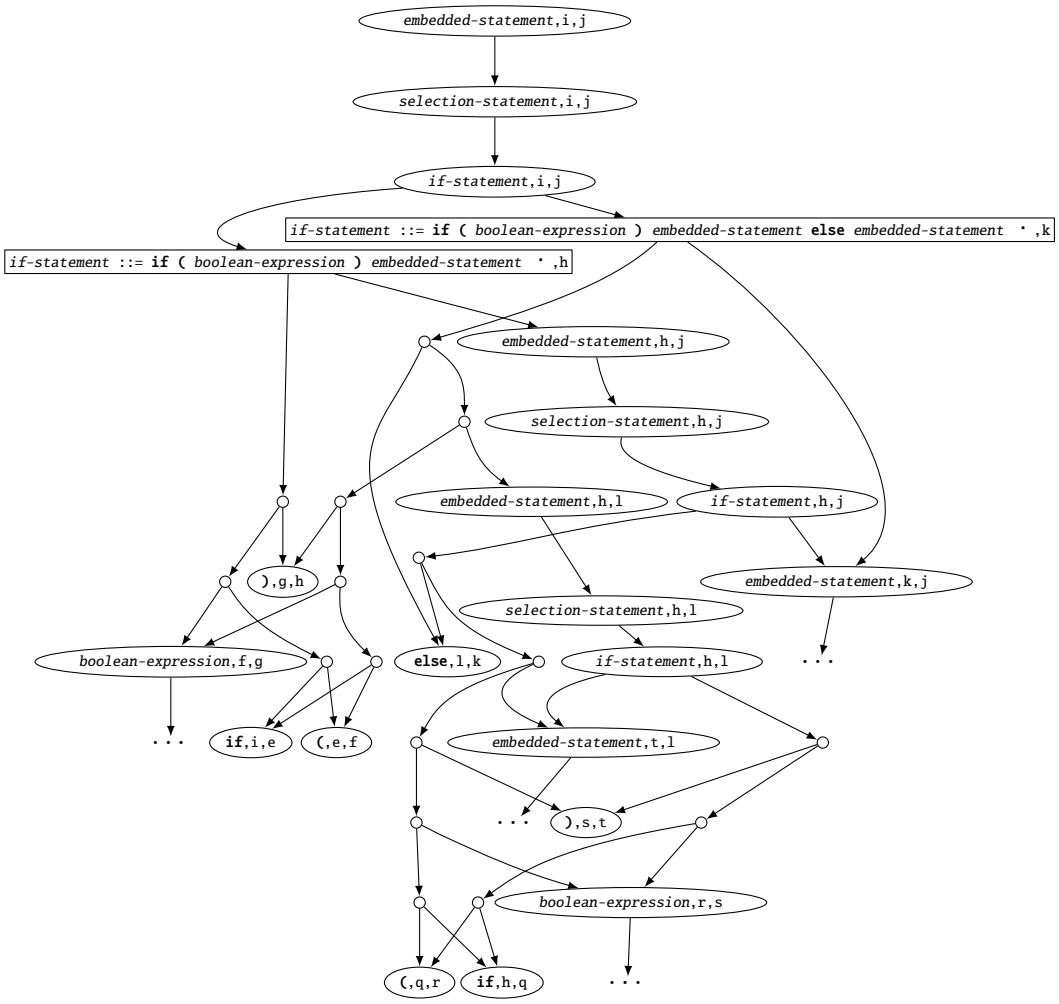


Figure 6.9: Demonstration of ambiguity under *if-statement*

$$\begin{aligned}&\mathit{suppress}(\textit{if-statement} ::= \textbf{if (}\ \textit{boolean-expression}\ \textbf{)}\\ &\qquad \textit{embedded-statement}\ \textbf{else}\ \textit{embedded-statement}\cdot,\\ &\quad \textit{if-statement} ::= \textbf{if (}\ \textit{boolean-expression}\ \textbf{)}\ \textit{embedded-statement})\end{aligned}$$

There is an ambiguity between the syntax for cast and invocation expressions. Consider the following grammar segment

```
unary-expression ::= primary-expression | + unary-expression | - unary-expression |
    ... | cast-expression
primary-expression ::= primary-no-array-creation-expression  | ...
primary-no-array-creation-expression ::= simple-name | parenthesized-expression |
    invocation-expression | member-access | ...
simple-name ::= identifier
```

```
invocation-expression ::= primary-expression ( argument-list ) | primary-expression
    ( )
argument-list ::= argument | argument-list , argument
argument ::= expression | ...
parenthesized-expression ::= ( expression )
cast-expression ::= ( type ) unary-expression
```

A string of form `(x)(y)` results in an ESPPF of the form seen in Figure 6.10. In one

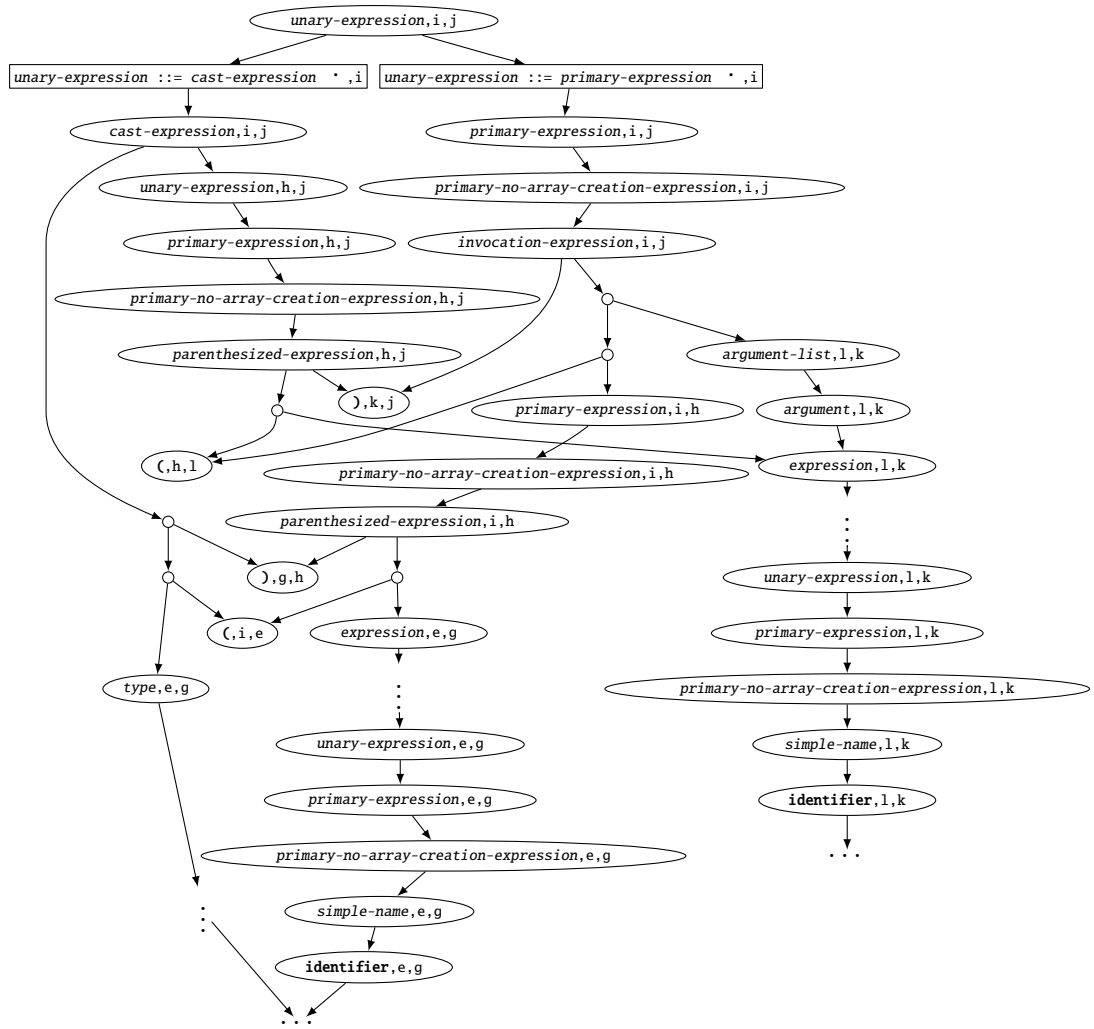


Figure 6.10: Demonstration of ambiguity under *unary-expression*

interpretation, `(x)(y)` is a cast expression which takes the result of the parenthesised expression `(y)` and casts it to a type `x`. In the other interpretation, the parenthesised expression `(x)` is an invocation with the parameter `y`. The latter is rejected in C# - an invocation expression is only valid for a method group identifier or a delegate

type: it cannot be an expression surrounded by parentheses. Therefore, the invocation expression interpretation is suppressed in favour of the cast expression interpretation.

$$suppress(\texttt{unary-expression} ::= \texttt{primary-expression}\cdot,$$
$$\texttt{unary-expression} ::= \texttt{cast-expression}\cdot)$$

additive-expression is another case where there is an ambiguity involving cast expressions

```
additive-expression ::= multiplicative-expression | additive-expression +
    multiplicative-expression | additive-expression - multiplicative-expression |
    ...
multiplicative-expression ::= unary-expression | ...
```

A string of form `(x)-y` will generate an ambiguity rooted at *additive-expression* as seen in Figure 6.11.

In this case, `(x)-y` could be a cast expression which casts a unary expression `-y` to a type `x`, or it could be a binary subtraction between a parenthesised expression `(x)` and `y`. To understand which interpretation is correct, the C# language specification [HCC02] describes the exact conditions under which an expression can be considered a cast expression:

> "A sequence of one or more tokens enclosed in parentheses is considered the start of a *cast-expression* only if at least one of the following are true:
>
> - The sequence of tokens is correct grammar for a *type*, but not for an *expression*.
> - The sequence of tokens is correct grammar for a *type*, and the token immediately following the closing parentheses is the token `~`, the token `!`, the token `(`, an **identifier**, a *literal*, or any keyword except **as** and **is**."

From this, in the `(x)(y)` case, `(x)` represents a cast since it is immediately followed by a `(`. In the `(x)-y` case, `(x)` represents a parenthesized expression since `x` can be an expression and `-` is not in the list of follow symbols for a cast. Of course, the latter is also true for the string `(x)+y`. The following rules in the disambiguation scheme capture the desired result

$$longest(\texttt{additive-expression} ::= \texttt{multiplicative-expression}\cdot,$$

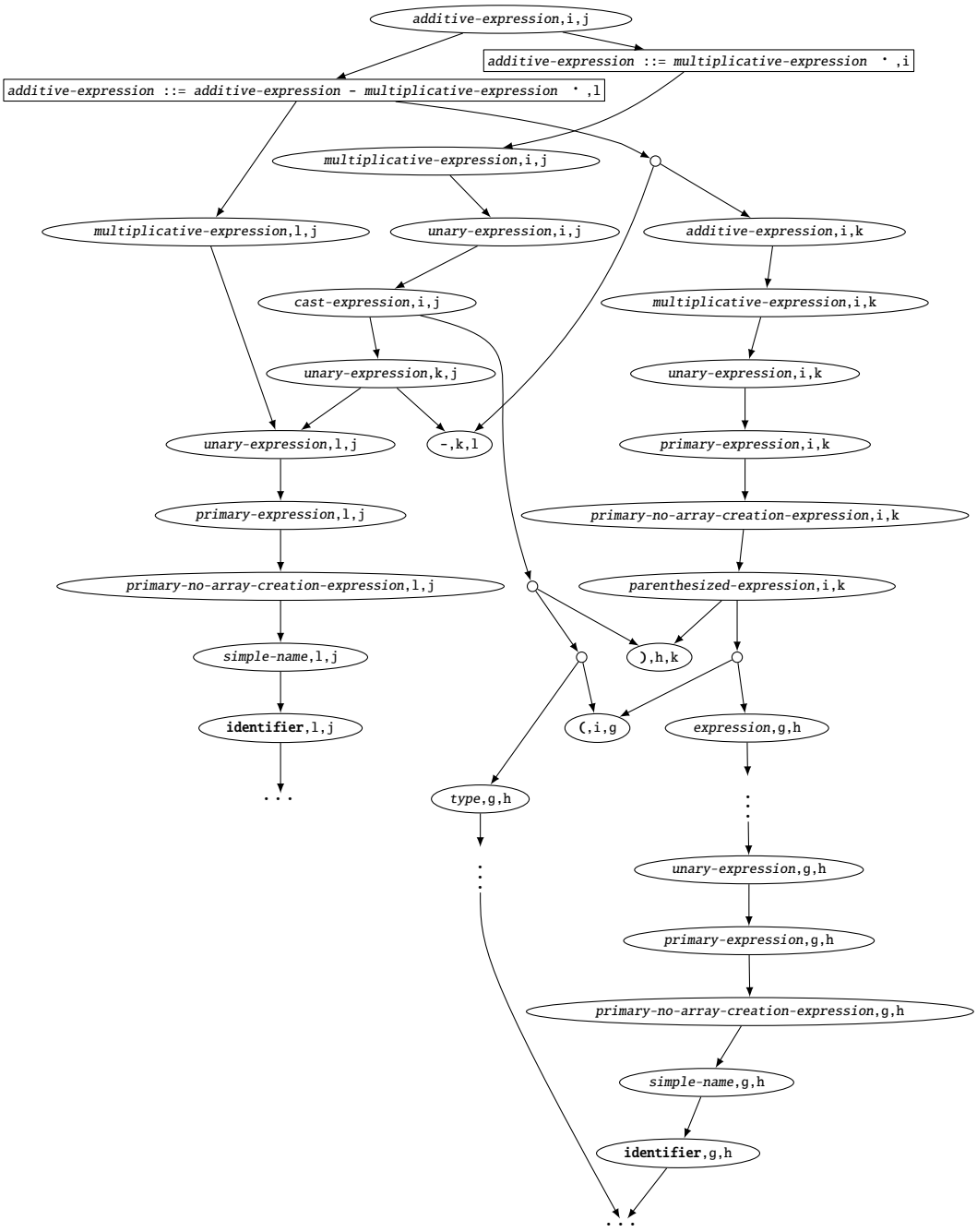


Figure 6.11: Demonstration of ambiguity under *additive-expression*

additive-expression ::= *additive-expression* + *multiplicative-expression*•)

shortest(*additive-expression* ::= *multiplicative-expression*•,

additive-expression ::= *additive-expression* **+**
multiplicative-expression·)
suppress(*additive-expression* ::= *multiplicative-expression*·,
additive-expression ::= *additive-expression* **+**
multiplicative-expression·)
longest(*additive-expression* ::= *multiplicative-expression*·,
additive-expression ::= *additive-expression* **-**
multiplicative-expression·)
shortest(*additive-expression* ::= *multiplicative-expression*·,
additive-expression ::= *additive-expression* **-**
multiplicative-expression·)
suppress(*additive-expression* ::= *multiplicative-expression*·,
additive-expression ::= *additive-expression* **-**
multiplicative-expression·)

The syntax definition of jagged arrays in C# leads to a syntactic ambiguity. Consider the production rules

```
type ::= reference-type | ...
reference-type ::= array-type | ...
array-type ::= non-array-type rank-specifiers
non-array-type ::= type
rank-specifiers ::= rank-specifier | rank-specifiers rank-specifier
rank-specifier ::= [ dim-separators ] | [ ]
dim-separators ::= , | dim-separators ,
```

The ambiguity occurs for any string with more than one *rank-specifier*. For example, the ESPPF fragment for a string of form `a[][]` will be as given in Figure 6.12.

As *non-array-type* can derive *array-type*, `a[]` can be derived from *non-array-type*. There are two derivations, one where `a[]` is derived from *non-array-type*, and one where `[][]` is derived from `rank-specifiers`. The number of derivation trees increases exponentially with the number of rank specifiers. For instance, for a string of form `a[][][]`, at the top level there is a branch where `a[][]` is derived from `non-array-type`, whose subtree looks like the one in Figure 6.12, as well as the branch where `[][][]` is derived from *rank-specifiers*.

How this is resolved is made clear in the specification - a *non-array-type* cannot derive a *array-type*. The correct interpretation can be obtained by suppressing the

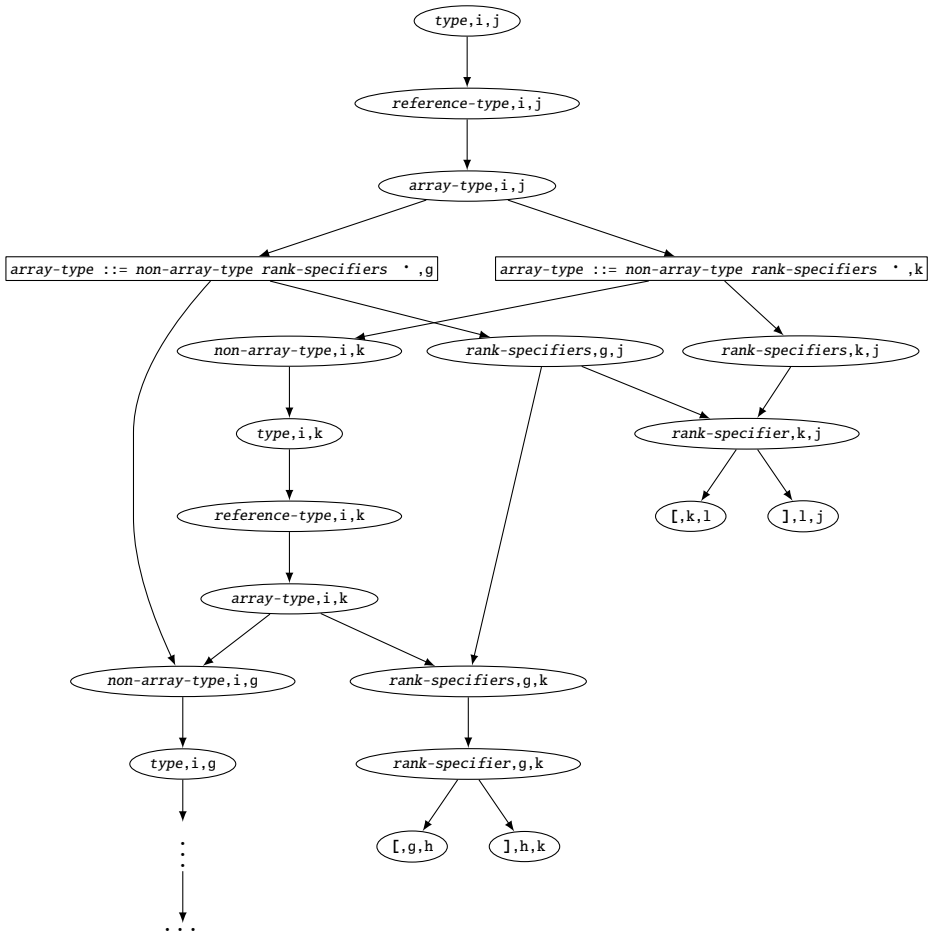


Figure 6.12: Demonstration of ambiguity under *array-type*

packed node with the highest pivot value

$$shortest(\textit{array-type} ::= \textit{non-array-type}\ \textit{rank-specifiers}\cdot,$$
$$\textit{array-type} ::= \textit{non-array-type}\ \textit{rank-specifiers}\cdot)$$

If *non-array-type* does not derive *array-type* then it must not derive strings that are derivable from *rank-specifiers*. As seen in Figure 6.12, the lowest pivot value is the one in which the top level *rank-specifiers* matches the most, and therefore the derivation tree in which *array-type* does not derive *non-array-type*.

The remaining syntactic ambiguity to look at is found in the specification of attribute arguments. Consider the grammar segment

```
attribute-arguments ::= ( positional-argument-list , named-argument-list ) | (
    named-argument-list ) | ( ) | ( positional-argument-list )
```

```
positional-argument-list ::= positional-argument | positional-argument-list ,
    positional-argument
positional-argument ::= attribute-argument-expression
named-argument-list ::= named-argument | named-argument-list , named-argument
named-argument ::= identifier = attribute-argument-expression
attribute-argument-expression ::= expression
expression ::= assignment | ...
assignment ::= unary-expression assignment-operator expression
assignment-operator ::= = | ...
```

Given that *unary-expression* $\stackrel{*}{\Rightarrow}$ *identifier*, an ambiguity occurs for strings of the form **(identifier=***expression***)**, which results in an ESPPF of the form given in Figure 6.13. In this case **identifier=***expression* can be derived either as an assignment expression or as a named argument. This ambiguity becomes more complex if there is more than one attribute argument of this form such as in strings of the form

(identifier=*expression***,identifier=***expression***,identifier=***expression***)**

which will have an ESPPF of the form in Figure 6.14.

The C# language specification limits the circumstances in which an expression can be interpreted as an *attribute-argument-expression* as follows:

> "An expression E is an *attribute-argument-expression* if all of the following statements are true:
>
> - The type of E is an attribute parameter type.
> - At compile-time, the value of E can be resolved to one of the following:
>   - A constant value.
>   - A System.Type object.
>   - A one-dimensional array of attribute-argument-expressions."

It is not valid for an *attribute-argument-expression* to be an assignment statement. Therefore, strings of the form **identifier=***expression* must be derived from *named-argument* in this context. The ambiguity reduction rules to ensure this are as follows

suppress(*attribute-arguments* ::= **(** *positional-argument-list* **)**·,
  attribute-arguments ::= **(** *positional-argument-list*
    , *named-argument-list* **)**·)
suppress(*attribute-arguments* ::= **(** *positional-argument-list* **)**·,
  attribute-arguments ::= **(** *named-argument-list* **)**·)

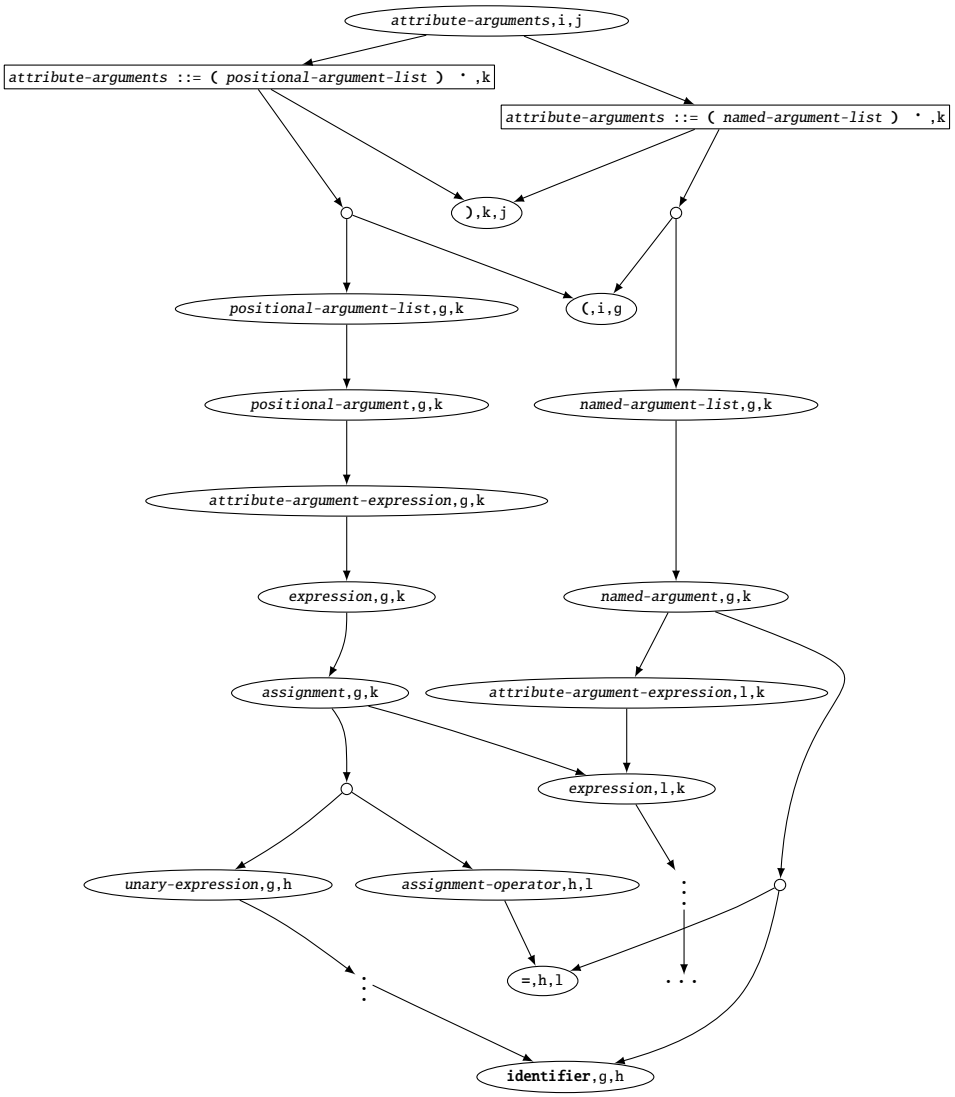


Figure 6.13: Demonstration of ambiguity under *attribute-arguments*

suppress(attribute-arguments ::= (*positional-argument-list*
, *named-argument-list*)·,
attribute-arguments ::= (*named-argument-list*)·)
shortest(attribute-arguments ::= (*positional-argument-list*
, *named-argument-list* ·),
attribute-arguments ::= (*positional-argument-list*
, *named-argument-list* ·))

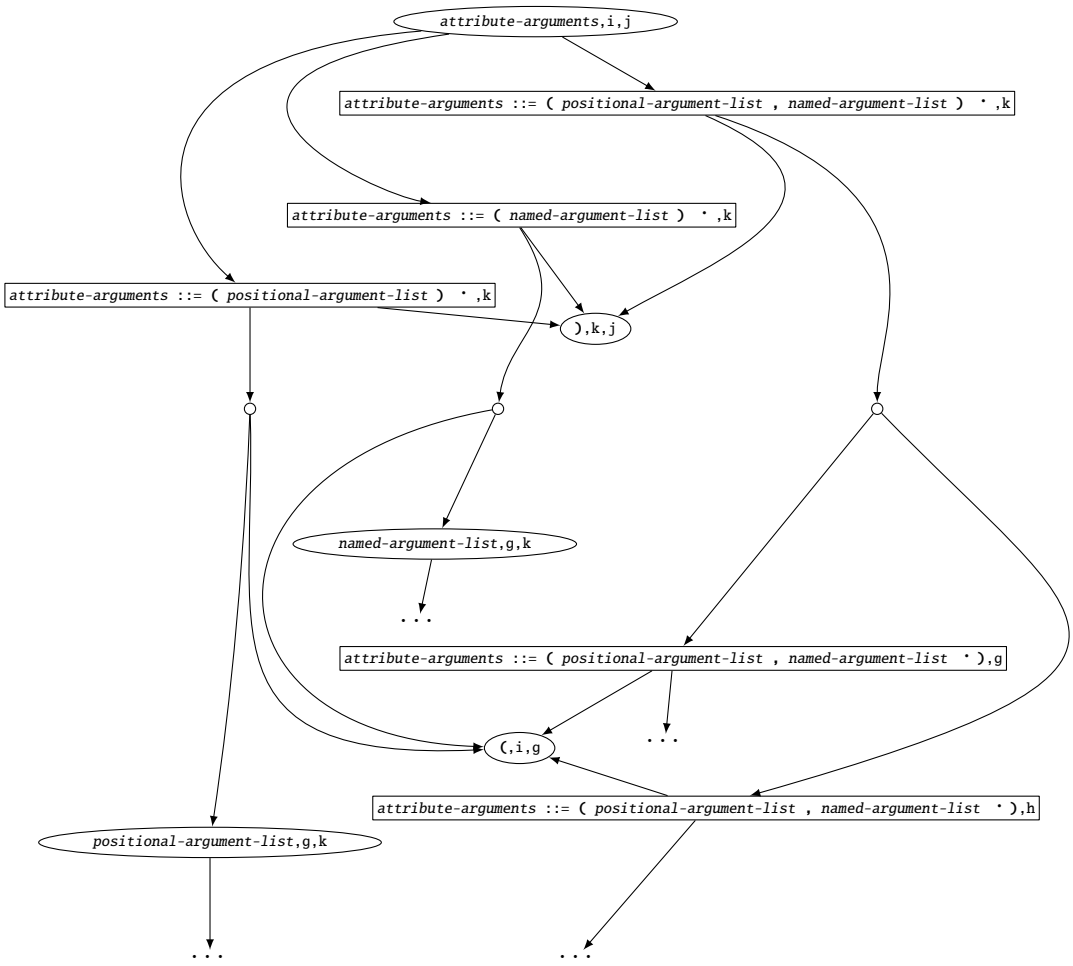


Figure 6.14: Demonstration of another ambiguity under *attribute-arguments*

These rules ensure, in all cases, that when an argument can be either a named argument or a positional argument, only the named argument interpretation is used.

The set of rules given in this section will reduce the set of derivations for any valid C# 1.2 input string to just a single derivation tree.

6.2.3 Transformation of a C# 1.2 derivation tree to an abstract syntax tree

The derivation tree resulting from the specification given in the previous section needs to be transformed into a form suitable for the funcon [Chu+15] semantics used in the PLanCompS project. As funcon is a form of structural operational semantics, this abstract syntax tree is of the form described in Chapter 4. The AST corresponds to a

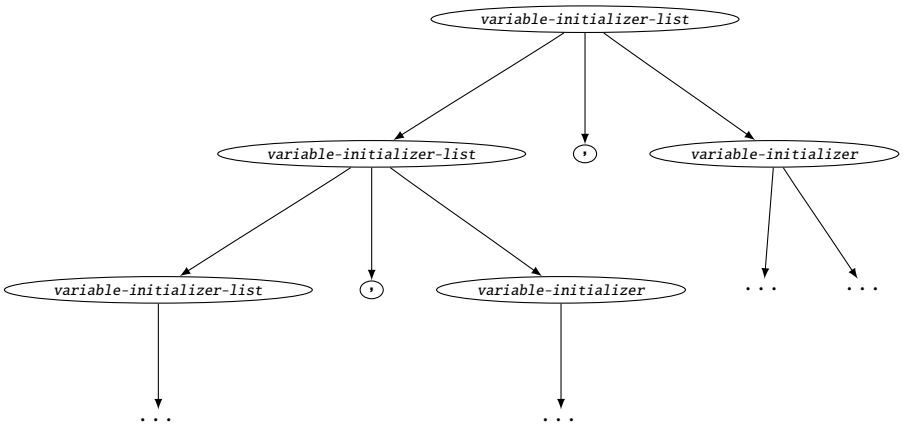
derivation tree generated by the abstract syntax grammar in Appendix C. This abstract syntax grammar was not constructed using the GIFT operators in Chapter 4, rather it was constructed by hand. The aim in this section is to produce a GIFT-annotated version of the concrete syntax for C# that constructs an AST that, if not exactly matching, is similar to the ASTs that the abstract syntax describes.

The annotated grammar will not be given in full here, instead this section shall look at the patterns and interesting cases where the GIFT annotations are necessary. The full annotated concrete syntax can be found in Appendix B.

The GIFT annotations are defined over BNF grammars yet the abstract syntax is defined in terms of EBNF. However, the impact that this has in terms of the AST construction is minimal, even in the case of closure operations. For example, consider the concrete syntax rules for *variable-initializer-list*

```
variable-initializer-list ::= variable-initializer | variable-initializer-list ,
    variable-initializer
```

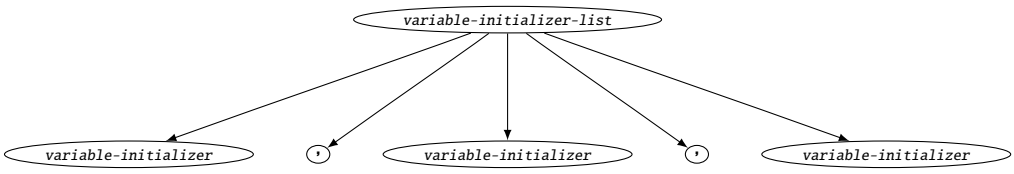
A typical derivation tree involving this production is



In the abstract syntax, the equivalent rule is

```
variable-initializer-list ::= variable-initializer (, variable-initializer)*
```

So the corresponding AST would be



Although it is not possible to capture the EBNF abstract syntax rule using the GIFT operators, it is possible to capture the resulting abstract syntax tree with the following annotated concrete syntax rule

```
variable-initializer-list ::= variable-initializer | variable-initializer-list^ ,
    variable-initializer
```

This is a rule featuring a fold-cycle, as described in 4.6.6. The grammar-to-grammar translation of the fold-under operator will result in the same rule being returned. However, the AST produced from the derivation tree will match the expected tree constructed by the abstract syntax - a sequence of *variable-initializer* nodes separated by **,** nodes.

A significant portion of the changes in the abstract syntax involve reducing a number of rules that are redundant from a semantics context. For example, consider the following fragment of the concrete syntax

```
type ::= value-type | reference-type
reference-type ::= class-type | interface-type | array-type | delegate-type
class-type ::= type-name | object | string
interface-type ::= type-name
delegate-type ::= type-name
value-type ::= struct-type | enum-type
struct-type ::= type-name | simple-type
enum-type ::= type-name
simple-type ::= number-type | bool
numeric-type ::= integral-type | floating-point-type | decimal
floating-point-type ::= float | double
type-name ::= namespace-or-type-name
namespace-or-type-name ::= identifier | namespace-or-type-name . identifier
predefined-type ::= bool | byte | char  | decimal | double | float | int | long |
    object | sbyte | short | string | uint | ulong | ushort
qualified-identifier ::= identifier | qualified-identifier . identifier
```

This would require a large number of inference rules for the semantics. The corresponding abstract syntax rules are much simpler

```
type ::= predefined-type | qualified-identifier | array-type
predefined-type ::= integral-type | bool | decimal | double | float | object |
    string
integral-type ::= sbyte | byte | short | ushort | int | uint | long | ulong | char
```

To capture this transformation in the grammar, one can begin by appropriately propagating fold-under operators across the sets of rules in the concrete syntax

```
type ::= value-type^ | reference-type^
reference-type ::= class-type^ | interface-type^ | array-type | delegate-type^
class-type ::= type-name^ | object | string
interface-type ::= type-name^
delegate-type ::= type-name^
value-type ::= struct-type^ | enum-type^
struct-type ::= type-name^ | simple-type^
```

```
enum-type ::= type-name^
simple-type ::= number-type^ | bool
numeric-type ::= integral-type | floating-point-type^ | decimal
floating-point-type ::= float | double
type-name ::= namespace-or-type-name^
namespace-or-type-name ::= identifier | namespace-or-type-name^ . identifier
predefined-type ::= bool | byte | char  | decimal | double | float | int | long |
    object | sbyte | short | string | uint | ulong | ushort
qualified-identifier ::= identifier | qualified-identifier^ . identifier
```

After all fold-under operators are evaluated, the production rule for *type* is as follows. Note that rules that would be duplicated become a single rule.

```
type ::= array-type | identifier | namespace-or-type-name^  .  identifier | object |
     string | bool | integral-type | float | double | decimal
```

This does not quite match the abstract syntax rule for *type*. *array-type* is now a direct alternate of *type*. **identifier** | *namespace-or-type-name*^**.** **identifier** derives the same strings as *qualified-identifier*, and the rest of the alternates all derive strings in *predefined-type*. To better match the abstract syntax, the gather operator is also used

```
type ::= value-type^ | reference-type^
reference-type ::= class-type^ | (interface-type^)!qualified-identifier |
    array-type | delegate-type^
class-type ::= (type-name^)!qualified-identifier | object!predefined-type | string!
    predefined-type
interface-type ::= type-name^
delegate-type ::= (type-name^)!qualified-identifier
value-type ::= struct-type^ | enum-type^
struct-type ::= (type-name^)!qualified-identifier | simple-type^
enum-type ::= (type-name^)!qualified-identifier
simple-type ::= number-type^ | bool!predefined-type
numeric-type ::= integral-type!predefined-type | floating-point-type^ | decimal!
    predefined-type
floating-point-type ::= float!predefined-type | double!predefined-type
type-name ::= namespace-or-type-name^
namespace-or-type-name ::= identifier | (namespace-or-type-name^)!
    qualified-identifier . identifier
predefined-type ::= bool | byte!integral-type | char!integral-type | decimal |
    double | float | int!integral-type | long!integral-type | object | sbyte!
    integral-type | short!integral-type | string | uint!integral-type | ulong!
    integral-type | ushort!integral-type
qualified-identifier ::= identifier | qualified-identifier^  . identifier
```

The rules in *predefined-type* that are also in *integral-type* are gathered into *integral-type*. Similarly all terminals derivable by *predefined-type* are gathered

into *predefined-type*. *type-name* is gathered to *qualified-identifier*. The instance of *namespace-or-type-name* that is under its own rule is also gathered under *qualified-identifier*. Fold-under is then applied in both cases giving the rules that already existed for *qualified-identifier*. The result is that the resulting AST will match the derivation trees for the abstract syntax for this set of rules.

In the above set of rules, *class-type* is folded under *reference-type* which is then folded under *type*. However, *class-type* is also referred to by other rules in the concrete syntax. One example is the rule

```
specific-catch-clause ::= catch ( class-type ) block | catch ( class-type
    identifier ) block
```

In the abstract syntax, the corresponding rule is

```
specific-catch-clause ::= catch ( type identifier? ) block
```

class-type is replaced with *type*. This can be reflected with the following GIFT annotated rule

```
specific-catch-clause ::= catch ( (class-type^)!type ) block | catch ( (
    class-type^)!type identifier ) block
```

As the rule for *type* now includes the transformed rules for *class-type*, this can be done without further modification.

A problem occurs in the case of class-base

```
class-base ::= : class-type | : interface-type-list | : class-type ,
    interface-type-list
interface-type-list ::= interface-type | interface-type-list , interface-type
```

The equivalent abstract syntax rule is

```
class-base ::= : (object | string | qualified-identifier) (, qualified-identifier)*
```

As seen earlier, the instance of **object** and **string** in *class-type* is gathered into *predefined-type*, so it is unclear how one could produce an equivalent GIFT annotated grammar that could simultaneously model that behaviour whilst modelling the behaviour seen in the rule for *class-base*. This separation is more convenient in the case of *class-base* as **object** is always a valid base class for inheritance in C#, whilst **string** is always invalid as a base class for inheritance (as **string** is a 'sealed' class in C#). In the other cases, the separation is not necessary as both can be treated as predefined types. As the GIFT operators are local to the current production rule, it is simply not possible to model these two different desired behaviours. In the case study, the closest GIFT annotated grammar possible is

```
class-base ::= : class-type^ | : interface-type-list^ | : class-type^ ,
    interface-type-list^
```

```
interface-type-list ::= (interface-type^)!qualified-identifier | interface-type-list^
     , (interface-type^)!qualified-identifier
```

which gives the resulting grammar

```
class-base ::= : predefined-type | : qualified-identifier | : interface-type-list^
    , qualified-identifier | : predefined-type , qualified-identifier | :
    predefined-type , interface-type-list^ , qualified-identifier | :
    qualified-identifier , qualified-identifier | : qualified-identifier ,
    interface-type-list^ , qualified-identifier
```

In some cases in the grammar, there are non-terminals whose languages are either identical or whose languages are such that one non-terminal's language is a subset of the other. In the abstract syntax, these are often 'merged' into a single non-terminal. One example of this is in the following rules in the concrete syntax

```
accessor-body ::= block | ;
constructor-body ::= block | ;
destructor-body ::= block | ;
method-body ::= block | ;
```

with an example of one of these non-terminals being referenced in the following set of rules

```
accessor-declarations ::= get-accessor-declaration | set-accessor-declaration |
    get-accessor-declaration set-accessor-declaration | set-accessor-declaration
    get-accessor-declaration
get-accessor-declaration ::= get accessor-body | attributes get accessor-body
```

The abstract syntax reduces the number of rules necessary by merging the non-terminals with the same language into a single non-terminal ***body***, as seen here

```
body ::= block | ;
accessor-declarations ::= attribute-section^* get body (attribute-section^* set
    body)? | attribute-section^* set body (attribute-section^* set body)?
```

There are two ways that one could capture this in the GIFT-annotated concrete syntax. The first is to gather each of the alternates of ***body*** non-terminals into ***body***, and then fold-under all instances of these non-terminals as follows

```
accessor-body ::= block!body | ;!body
constructor-body ::= block!body | ;!body
destructor-body ::= block!body | ;!body
method-body ::= block!body | ;!body
accessor-declarations ::= get-accessor-declaration^ | set-accessor-declaration^ |
    get-accessor-declaration^ set-accessor-declaration^ | set-accessor-declaration^
    get-accessor-declaration^
get-accessor-declaration ::= get accessor-body^ | attributes^ get accessor-body^
```

The alternative is to fold-under the **-body* non-terminals and gather into *body* as follows

```
accessor-body ::= block | ;
constructor-body ::= block | ;
destructor-body ::= block | ;
method-body ::= block | ;
accessor-declarations ::= get-accessor-declaration^ | set-accessor-declaration^ |
    get-accessor-declaration^ set-accessor-declaration^ | set-accessor-declaration^
    get-accessor-declaration^
get-accessor-declaration ::= get (accessor-body^)!body | attributes^ get (
    accessor-body^)!body
```

In the case study, the former solution is chosen. In general, the former solution is better when the change needs to be applied across the entire grammar (as is in this case), whilst the latter is better for when the change only occurs locally.

Another example of this type of modification in the abstract syntax is in the case of *class-member-declaration* and *struct-member-declaration*

```
class-member-declaration ::= constant-declaration | field-declaration |
    method-declaration | property-declaration | event-declaration |
    indexer-declaration | operator-declaration | constructor-declarator |
    destructor-declaration | static-constructor-declaration | type-declaration
struct-member-declaration ::= constant-declaration | field-declaration |
    method-declaration | property-declaration | event-declaration |
    indexer-declaration | operator-declaration | econstructor-declaration |
    static-constructor-declaration | type-declaration
```

The language of *struct-member-declaration* is a subset of *class-member-declaration*, with the only difference being that *struct-member-declaration* cannot derive *destructor-declaration*. The abstract syntax effectively merges these two non-terminals into *member-declaration*, which reduces the number of grammar rules at the (small) cost of widening the language of struct declarations.

```
member-declaration ::= constant-declaration | field-declaration |
    method-declaration | property-declaration | event-declaration |
    indexer-declaration | operator-declaration | constructor-declarator |
    destructor-declaration | static-constructor-declaration | type-declaration
```

In the GIFT-annotated concrete syntax, this transformation is captured by applying gather to each of the right-hand sides of *class-member-declaration* and *struct-member-declaration* and then applying fold-under to all right-hand instances of the two non-terminals

```
class-member-declaration ::= constant-declaration!member-declaration |
    field-declaration!member-declaration | method-declaration!member-declaration |
```

```
    property-declaration!member-declaration | event-declaration!member-declaration |
    indexer-declaration!member-declaration | operator-declaration!member-declaration
    | constructor-declarator!member-declaration | destructor-declaration!
    member-declaration | static-constructor-declaration!member-declaration |
    type-declaration!member-declaration
struct-member-declaration ::= constant-declaration!member-declaration |
    field-declaration!member-declaration | method-declaration!member-declaration |
    property-declaration!member-declaration | event-declaration!member-declaration |
    indexer-declaration!member-declaration | operator-declaration!member-declaration
    | constructor-declaration!member-declaration | static-constructor-declaration!
    member-declaration | type-declaration!member-declaration
```

The most significant difference between the concrete and abstract syntax is found in the rules for expressions. In the concrete syntax, the evaluation order is made explicit by the structure as seen in this fragment of the rules for `expression`

```
expression ::= conditional-expression | assignment
assignment ::= unary-expression assignment-operator expression
conditional-expression ::= conditional-or-expression | conditional-or-expression ?
    expression : expression
conditional-or-expression ::= conditional-and-expression | conditional-or-expression
    || conditional-and-expression
conditional-and-expression ::= inclusive-or-expression | conditional-and-expression
    && inclusive-or-expression
inclusive-or-expression ::= exclusive-or-expression | inclusive-or-expression |
    exclusive-or-expression
exclusive-or-expression ::= and-expression | exclusive-or-expression ^ and-expression
```

In the abstract syntax, this is reduced to a more concise set of rules

```
expression ::= expression assignment-operator expression | expression ? expression
    : expression | expression binary-operator expression | ...
binary-operator ::= overloadable-binary-operator | || | &&
overloadable-binary-operator ::= | | ^ | ...
```

All binary expressions are reduced to a single rule, by extracting binary operators into a new non-terminal `binary-operator` (with overloadable binary operators also in turn extracted into the existing `overload-binary-operator` non-terminal). The entire structure for expression is flattened. It is clear that applying the fold-under operator can collapse rules such as *`expression ::= conditional-expression`*, however, it is also necessary to ensure all the right-hand side instances of the non-terminals that are folded under *`expression`* are changed to refer to *`expression`*. This is achieved by applying fold-under to the non-terminal instance and gathering the result to *`expression`*. The GIFT-annotated grammar for this is

```
expression ::= conditional-expression^ | assignment^
```

```
assignment ::= (unary-expression^)!expression assignment-operator*@) (*@expression
conditional-expression ::= conditional-or-expression^ | (conditional-or-expression^)
    !expression ? expression : expression
conditional-or-expression ::= conditional-and-expression^ | (
    conditional-or-expression^)!expression ||!binary-operator (
    conditional-and-expression^)!expression
conditional-and-expression ::= inclusive-or-expression^ | (
    conditional-and-expression^)!expression &&!binary-operator (
    inclusive-or-expression^)!expression
inclusive-or-expression ::= exclusive-or-expression^ | (inclusive-or-expression^)!
    expression (|!overloadable-binary-operator)!binary-operator (
    exclusive-or-expression^)!expression
exclusive-or-expression ::= and-expression^ | (exclusive-or-expression^)!expression
    (^!overloadable-binary-operator)!binary-operator (and-expression^)!expression
```

Evaluation of the GIFT Operators

This case study has demonstrated the application of the GIFT operators in a practical scenario. In this case study, the fold-over, insert and tear operators are not applied. A fold-over operator would typically be used to promote terminals to become non-terminals - which removes symbols from strings in the language. The tear and insert operators would typically be used to re-order symbols in a string - which removes and replaces strings in the language. As the language of the abstract syntax used in this case study is a superset of the language of the concrete syntax, these operators were not applicable. Only fold-under and gather were needed for this case study.

Although the abstract syntax was handwritten, and designed using ad-hoc principles, these operators were enough to transform derivation trees generated by the C# 1.2 concrete syntax into a form matching the derivation trees generated by the abstract syntax for all but one production. Whilst the GIFT operators have only been designed to work with BNF grammars, the abstract syntax provided uses EBNF. This proved not to be an issue, as annotations could still be applied as tree-to-tree transformations to produce the ASTs that match one expectation of an EBNF tree structure. Interestingly, this is often achieved by creating fold-cycle behaviour. This may suggest that the solutions to the fold-cycle problem, discussed in 4.6.6, may be resolvable through translation to some EBNF construct. Nonetheless, this shows that GIFT operators have potential as a scheme for abstract syntax translations that is straightforward to use whilst being powerful enough, in most cases, to model a useful abstract syntax.

However, this case study did highlight one case where the GIFT operators may not be robust enough. Recall the example of `class-base` from page 176. To be able to match the abstract syntax rule for `class-base`, it is necessary for the translation of

class-type to be different depending on which non-terminal derived it - only gathering **object** and **string** under *predefined-type* if *class-type* has not been derived by *class-base*. For the tree-to-tree transformation, the solution to this would be allowing the language specifier to state that a GIFT annotation should only apply if certain conditions are met. In this particular case, there would be a condition on the application of the gather operator to **object** and **string** stating that the operator only applies if the parent node *class-type* has not got a node labelled *class-base* as its parent. The workaround used in the implementation applies exactly this condition. More generally, inherited attributes could be used in the syntax directed definition to facilitate this. However, it is not clear how this would work in the grammar-to-grammar translation. This example does highlight the limitations of applying translations within the scope of a single production only. A more robust solution would need to consider translations that can use information outside this scope.

The GIFT operators could be expressed as term rewrite rules [KBV01]. The original TIF operators were intended to be a convenient way of expressing localised tree rewrite rules. The situation encountered with *class-base* would be equivalent to a situation where conditional rewrite rules are required.

Chapter 7

Conclusions

This thesis has introduced and analysed novel approaches to dealing with the problems faced in the design of compiler front-ends. This chapter will summarise what has been presented and propose future directions for possible research.

7.1 Multilexing and Parsing

After introductory material in Chapter 1, a new approach to lexical analysis, known as multilexing, was introduced in Chapter 2. This chapter considered the drawbacks of a traditional single-tokenisation output style lexical analyser. It then considered the theoretical and practical issues of a naïve approach that simply produced all the tokenisations of a character string. This provided the context for a new approach, that is able to embed all tokenisations of a string by producing a set of triples (a TWE set) as output, and described a mechanism for ensuring that this set only contained the minimal set of triples required to embed all the tokenisations necessary.

The chapter then presented an algorithm that takes a character string as input, and produces a TWE set as output. This algorithm is a relatively simple modification of a traditional finite-state automata-based tokenising algorithm. Chapter 5 presented a different solution, using a GLL recogniser, that would permit tokens whose patterns represent languages that are context-free not just regular languages - such as that required by nested comments.

Mechanisms for reducing the number of tokenisations in a given TWE set were presented. These mechanisms were shown to provide functionality analogous to mechanisms commonly used to perform lexical disambiguation under traditional models, whilst also providing more flexibility for more complex cases. Chapter 5 showed that these mechanisms provide a means of reducing lexical ambiguity with respect only

to the token matches, a benefit not offered by equivalent syntactic-level strategies in character-level based parsing approaches. A mechanism was given that suppresses tokens whose patterns have no overlap with other tokens. This mechanism was expanded on in Chapter 5 to describe a way in which whitespace and comment tokens can be removed from a TWE set - through the use of an initial processor.

Chapter 2 also discussed how this multilexing approach compared with other approaches for allowing multiple tokenisations, such as backtracking in lex, Schrödinger's tokens, and character-level parsing. Chapter 5 carried out further comparisons of multilexing to character-level parsing, by demonstrating how the data structures produced by the parsers for each compare. This showed that the multilexing approach offers the same level of power and control as character-level parsing, whilst offering the simplicity and efficiency of a token-level parser.

Chapter 3 started by reviewing a parsing algorithm, GLL, that can produce multiple derivations as output for a single input string. The chapter then considered how to extend the SPPF data structure that is produced as a result of a GLL parse so that it is be able to embed multiple derivations from multiple input strings - the result being an Extended SPPF (ESPPF). The GLL algorithm was then extended, to MGLL, through some conceptually straightforward modifications so that it can take a TWE set as input and produce an ESPPF representing all derivations of all strings embedded in that TWE set.

7.2 The GIFT Operators

Chapter 4 began by discussing the various models of abstract syntax. It then considered one particular model of abstract syntax - the abstract syntax used in the specification of structural operational semantics. This form of abstract syntax is essentially a concise form of the grammar used for parsing - the concrete syntax.

The chapter then explored the ways a derivation tree in the concrete syntax could be transformed into an equivalent derivation tree in the abstract syntax. This considered local transformations on the tree, looking at the effects that adding or removing nodes could have on surrounding nodes. This led to a review of the TIF operators, a set of annotations on a grammar for describing the local transformations that transform a derivation tree into an equivalent abstract syntax tree. The semantics of the individual operators along with the corresponding syntax-directed definition of the tree construction semantics was given. Potential issues with the evaluation order, and how the syntax-directed definition resolves these issues were discussed. The TIF operators were then extended to the GIFT operators by adding a new *gather* annotation, that

inserts a new node in between a parent node and a group of children.

After exploring the GIFT operators as a set of operators for transforming derivation trees into abstract syntax trees, the chapter discussed how one could derive the abstract syntax that generates the AST. It demonstrated how the same operators could be used to perform local transformations to the concrete syntax to generate a BNF grammar whose derivation trees are close, if not entirely equivalent, to the generated ASTs. However, it also demonstrated some problems, such as when fold operators are in a recursive cycle and the performance impact of evaluating the operators in certain orders. The end of the chapter discussed a proposal for the order in which the operators are to be evaluated, to maximise performance and proposed a means of detecting and preventing the evaluation of fold operators in a recursive cycle.

7.3 Syntactic Ambiguity Reduction

Although discussed mainly as syntactic-level equivalents of the lexical ambiguity reduction rules, Chapter 5 proposed some primitive operations for reducing the number of derivation trees embedded in an (E)SPPF. These operations work by considering relations between grammar slots, which are the labels of packed nodes. An equivalent of longest and shortest match, based on highest and lowest pivot were considered, alongside a simple priority of grammar slots. Chapter 5 showed that these syntactic ambiguity reduction mechanisms cannot simply mimic the behaviour of the equivalent lexical ambiguity reduction rules.

The ambiguity reductions are simplistic and do not capture the full scope of the problems faced in syntactic ambiguity reduction. Discussions on issues that cannot be captured by the syntactic ambiguity reduction mechanisms presented in this thesis can be found in more dedicated studies on the subject [San+14; Afr+13; BV12]. However, the ambiguity reduction mechanisms seen in this thesis were good enough for the C# 1.2 case study presented in Chapter 6.

7.4 Implementation and the C# Case Study

Chapter 6 considered an implementation of the theoretical concepts described in this thesis as a set of general frameworks. It first described an implementation of a multilexer for the C# 2.0 language specification. Whilst providing a lexical ambiguity reduction scheme that matches the intentions of the C# lexer, it was shown how the multilexer allowed a more robust lexical specification - by allowing non-reserved keywords to also be treated as identifiers. This allows the parser to consider both in-

terpretations, choosing one based on context. It was also shown how the multilexer handles the lexical ambiguities surrounding **>** and **>>** - demonstrating that it is possible to have nested type parameterisation and right-shift expressions without requiring difficult workarounds in the parsing grammar. The end of the first part compared this implementation to an equivalent character-level implementation. The results generally demonstrated the conclusions made in Chapter 5 regarding the efficiency of the multilexer, although they also demonstrated that the character-level approach may have advantages in the situation where the input is the worst-case for the lexer but average-case for the parser.

The second part of the chapter considered the application of the syntactic ambiguity reduction mechanisms to resolve ambiguities in the C# 1.2 language specification parser - with the aim of performing a translation to abstract syntax trees described by a structural abstract syntax created for the PLanCompS [Mos+15] project. It demonstrated that, for the needs of the PLanCompS project, these mechanisms are enough to reduce the number of derivation trees to a single tree.

The final part of the chapter described how the GIFT annotations are applied to transform the resulting derivation tree into an equivalent tree in the abstract syntax created for the PLanCompS project. It was shown that the GIFT operators are able to transform the derivation tree into a form that is identical to the desired abstract syntax tree in all but one case.

7.5 Directions for Future Research

The work in this thesis also provides a platform on which further work can be developed. This will be discussed here.

7.5.1 'On-the-fly' Lexical Ambiguity Reduction

The lexical ambiguity reduction rules are designed to be applied after all tokenisations are found. As demonstrated by the results in Chapter 6, this creates an overhead, as even if the lexical ambiguity reduction scheme removes all but one tokenisation, the TWE set embedding all tokenisations needs to be constructed - and this could exhibit worst-case time and size complexity.

It would be more efficient if these lexical ambiguity reductions could be applied as the initial TWE set is constructed. Instead of constructing the full set and then removing triples, such a scheme would only initially add triples to the set if they are not suppressed by the ambiguity reduction scheme. For most cases, this would then lead to performance that is comparable to classical lexical analysis.

7.5.2 More Sophisticated Layout Token Removal

The proposed mechanism for layout token removal involves an initial processing step, with token suppression. This is good enough for the cases where tokens are separated but is not good enough in general. Further research could look at ways that layout tokens can be removed without requiring the tokens to be separated, and without requiring an initial processing step. Such an approach could involve removing layout tokens after the TWE set is constructed and lexical ambiguity reduction rules applied.

7.5.3 Identification of Flawed Ambiguity Reduction Schemes

A problem highlighted in Chapter 2 is that a lexical ambiguity reduction scheme may remove all embedded tokenisations from the set. Similarly, a syntactic ambiguity reduction scheme defined using the mechanisms described in Chapter 5 may remove all derivations from the (E)SPPF.

It would be useful to find techniques for determining whether a particular scheme can create such conflicts and find ways to prevent or resolve these conflicts.

7.5.4 Multiple-Input Parsing for Other Parsing Algorithms

Chapter 3 gave an extension to the BNF version of the GLL parsing algorithm for multiple inputs. It may be useful to consider extensions to allow multiple inputs for other parsing algorithms. It would seem that the design of GLL makes its naturally easier to extend to multiple inputs than, for example, GLR. It may be interesting to see to what extent this suggestion is true.

7.5.5 Complexity analysis of MGLL

A full complexity analysis of MGLL was not the focus of this study, however it is certainly of interest. As stated at the end of Chapter 3, the hypothesis is that MGLL should have upper-bound time complexity no worse than quartic in the number of character indices to process and may even be worst-case cubic. The worst-case scenario for MGLL would involve combining the worst-case lexical specification for the multilexer with the worst-case grammar for a generalised parser. Given a parser whose lexical specification consists of the single token-pattern pair

$$(\mathbf{b}, \{\mathsf{a}\}^{+})$$

and with the parsing grammar [SJ10a]

$$S ::= \mathbf{b} \mid S\ S \mid S\ S\ S$$

it should be possible to test this hypothesis by parsing increasing strings of `a`.

7.5.6 GIFT Translation for EBNF

The GIFT operator scheme in Chapter 4 considered only BNF grammars as both the input and the output. This is mainly due to problems in determining the semantics of folds and gather under closure operations. Future work can consider ways that the GIFT operators can be defined for EBNF grammars.

In addition, one may wish to consider whether the grammar-to-grammar transformation should create EBNF constructs where necessary. A closure operation may, for instance, resolve the issue around fold cycles. For the C# 1.2 case study, all that was considered was which operators were necessary to perform the necessary tree-to-tree transformation. The annotated grammar in Appendix B does not transform into the grammar in Appendix C, as the latter is written in EBNF. One might consider the additional semantics and operators needed to provide a complete transformation.

7.5.7 Implementation of Grammar-to-Grammar GIFT translations

Unfortunately, the implementation of the GIFT transformations does not operate through direct analysis of the annotated grammar, rather it traverses the derivation tree and applies one of the operator transformations where it is appropriate. Whilst the concrete syntax grammar to abstract syntax grammar conversion was used to illustrate the transformation, the implementation focussed entirely on transforming the derivation tree to an abstract syntax tree. The implementation does not produce an abstract syntax grammar, although earlier experimental work showed promising results for a textual based conversion of the grammar.

7.5.8 Grammar transformations outside the scope of a single production

There was one case in the C# case study where it was not possible to transform the derivation tree into a tree matching that generated by the abstract syntax. This is because the operation required information beyond the local production context. One could consider other transformations that go beyond the local production context, perhaps using more sophisticated attribute grammar schemes.

7.5.9 Mapping an abstract syntax to a parsing grammar

Although a single abstract syntax could correspond to many possible parsing grammars, there is scope for considering whether there is an inverse to the GIFT operators for generating a parsing grammar from an abstract syntax. For example, the grammars provided by the specifications of programming languages such as OCaml [Ler+12] are much closer to an abstract syntax than a parsing grammar and are highly ambiguous as a result. Therefore, it may be useful to consider whether one can take an abstract syntax, and through some grammar annotations, transform it into a less ambiguous parsing grammar.

7.5.10 Expansion of Syntactic Ambiguity Reduction Schemes

Whilst the syntactic ambiguity reduction scheme described in Chapter 5 was good enough for the concrete syntax for C# 1.2, it is not sufficient in general. For instance, the scheme is unable to handle the operator associativity and precedence ambiguities in the expression subgrammar for the C# abstract syntax in Appendix C.

The syntactic ambiguity reduction scheme proposed in this thesis allows for an easy to use specification, and it will be worth determining how other works that address this problem could be integrated into this scheme.

Appendix A

C# 2.0 Language Specification

This appendix describes the C# 2.0 language specification used in 6.1. The first part of this appendix is the lexical specification. The second part is the grammar used for parsing.

A.1 Lexical specification

This lexical specification is based on the lexical syntax given in the appendix of the language specification [HCC06]. Tokens are described using EBNF rules. A string is in the pattern of the token if there is a derivation of the string in the grammar, whose start symbol is the non-terminal labelled as the token. For readability, non-token non-terminals will be listed in italics and token non-terminals will be in bold. Characters will be underlined. Additionally, $[\alpha]$ will be interpreted to mean one symbol in the sequence depicted by α.

identifier ::= (*letter-character* | _) *identifier-part-character** |
    @ (*letter-character* | _) *identifier-part-character**
integer-literal ::= [1-9]+ *integer-type-suffix*? |
    0 ((x|X) [0-9a-fA-F]+)? *integer-type-suffix*?
real-literal ::= [0-9]* . [0-9]+ *exponent-part*? *real-type-suffix*? |
    [0-9]+ (*exponent-part real-type-suffix*? | *real-type-suffix*)
character-literal ::= ' *character* '
string-literal ::= " *regular-string-literal-character* " |
    @ " (*single-verbatim-string-literal* | " ")* "
letter-character := [a-zA-Z_]
identifier-part-character ::= [a-zA-Z0-9_] | *unicode-character-escape-sequence*
unicode-character-escape-sequence ::= \ (u [0-9a-fA-F] [0-9a-fA-F] [0-9a-fA-F] [0-
    9a-fA-F] | U [0-9a-fA-F] [0-9a-fA-F] [0-9a-fA-F] [0-9a-fA-F] [0-9a-fA-F] [0-
    9a-fA-F] [0-9a-fA-F] [0-9a-fA-F])
integer-type-suffix ::= (u|U) (l|L)? | (l|L) (u|U)?

```
exponent-part ::= (e|E) (+|-)? [0-9]+
real-type-suffix ::= f|F|d|D|m|M
character ::= single-character | simple-escape-sequence |
    hexadecimal-escape-sequence | unicode-character-escape-sequence
single-character ::= any unicode character except \ ' \n \r
simple-escape-sequence ::= \ ('|"|\|0|a|b|f|n|r|t|t|v
hexadecimal-escape-sequence ::= \ x [0-9a-fA-F] [0-9a-fA-F]? [0-9a-fA-F]? [0-9a-
    fA-F]?
regular-string-literal-character ::= single-regular-string-literal-character |
    simple-escape-sequence | hexadecimal-escape-sequence |
    unicode-character-escape-sequence
single-regular-string-literal-character ::= any unicode character except " \ \n
    \r
single-verbatim-string-literal-character ::= any unicode character except "
```

Additionally there are three layout tokens in the lexical specification

```
new-line ::= \r \n? | \n
whitespace ::= ␣ | \t | \v | \f
comment ::= // input-character* | /* (not-asterisk | *+ not-slash)* *+ /
input-character ::= any unicode character except \n \r
not-slash ::= any unicode character except /
not-asterisk ::= any unicode character except *
```

As with many languages, C# has a number of tokens whose patterns are the same as their label - known as the keywords. The following list of such tokens are *reserved*, that is these patterns should be recognised as matching only these tokens.

{ } [ ] ( ) . , : :: ; + - * / % & | ^ ! ~ = -> < > ? ?? ++ -- && || << >> == != <= >= += -= *= /= %= &= |= ^= <<= >>= -> abstract as base bool break byte case catch char checked class const continue decimal default delegate double do else enum event explicit extern false finally fixed float foreach for goto if implicit interface internal int in is lock long namespace new null object operator out override params private protected public readonly ref return sbyte sealed short sizeof stackalloc static string struct switch this throw true try typeof uint ulong unchecked unsafe ushort using virtual void volatile while

There are also keywords that are not reserved - known as the *contextual* keywords.

The patterns for these tokens can also be matched by other tokens

add alias get partial remove set where yield

A.2 Parsing Grammar

The parsing grammar is based on the grammar given in the appendix of [HCC06]. This grammar is given as a set of BNF grammar rules. The terminals of the grammar are in bold. The start symbol of this grammar is the non-terminal *compilation-unit*.

literal ::= **true** | **false** | **integer-literal** | **real-literal** | **characte-literal** | **string-literal** | **null**

compilation-unit ::= ϵ |
namespace-member-declarations |
global-attributes |
global-attributes namespace-member-declarations |
using-directives |
using-directives namespace-member-declarations |
using-directives global-attributes |
using-directives global-attributes namespace-member-declarations |
extern-alias-directives |
extern-alias-directives namespace-member-declarations |
extern-alias-directives global-attributes |
extern-alias-directives global-attributes namespace-member-declarations |
extern-alias-directives using-directives |
extern-alias-directives using-directives namespace-member-declarations |
extern-alias-directives using-directives global-attributes |
extern-alias-directives using-directives global-attributes namespace-member-declarations

namespace-name ::= *namespace-or-type-name*

type-name ::= *namespace-or-type-name*

namespace-or-type-name ::= *qualified-alias-member* |
identifier |
identifier *type-argument-list* |
namespace-or-type-name **.** **identifier** |
namespace-or-type-name **.** **identifier** *type-argument-list*

type ::= *value-type* | *reference-type* | *type-parameter*

value-type ::= *struct-type* | *enum-type*

struct-type ::= *type-name* | *simple-type* | *nullable-type*

simple-type ::= *numeric-type* | **bool**

numeric-type ::= *integral-type* | *floating-point-type* | **decimal**

integral-type ::= **sbyte** | **byte** | **short** | **ushort** | **int** | **uint** | **long** | **ulong** | **char**

floating-point-type ::= **float** | **double**

enum-type ::= *type-name*

nullable-type ::= *non-nullable-value-type* **?**

non-nullable-value-type ::= *enum-type* | *type-name* | *simple-type*

reference-type ::= *class-type* | *interface-type* | *array-type* | *delegate-type*

class-type ::= *type-name* | **object** | **string**

interface-type ::= *type-name*

array-type ::= *non-array-type rank-specifiers*

non-array-type ::= *value-type* | *class-type* | *interface-type* | *delegate-type* |
    type-parameter

rank-specifiers ::= *rank-specifier* | *rank-specifiers rank-specifier*

rank-specifier ::= **[ ]** | **[** *dim-separators* **]**

dim-separators ::= **,** | *dim-separators* **,**

delegate-type ::= *type-name*

variable-reference ::= *expression*

argument-list ::= *argument* | *argument-list* **,** *argument*

argument ::= *expression* | **ref** *variable-reference* | **out** *variable-reference*

primary-expression ::= *array-creation-expression* |
    primary-no-array-creation-expression

primary-no-array-creation-expression ::= *literal* | *simple-name* |
    parenthesized-expression | *member-access* | *invocation-expression* |

element-access | *this-access* | *base-access* | *post-increment-expression* |
post-decrement-expression | *object-creation-expression* |
delegate-creation-expression | *typeof-expression* | *checked-expression* |
unchecked-expression | *default-value-expression* | *anonymous-method-expression*

simple-name ::= **identifier** | **identifier** *type-argument-list*

parenthesized-expression ::= **(** *expression* **)**

member-access ::= *primary-expression* **.** **identifier** |
primary-expression **.** **identifier** *type-argument-list* |
predefined-type **.** **identifier** |
predefined-type **.** **identifier** *type-argument-list* |
qualified-alias-member **.** **identifier** |
qualified-alias-member **.** **identifier** *type-argument-list*

predefined-type ::= **bool** | **byte** | **char** | **decimal** | **double** | **float** | **int** | **long** |
object | **sbyte** | **short** | **string** | **uint** | **ulong** | **ushort**

invocation-expression ::= *primary-expression* **(** **)** |
primary-expression **(** *argument-list* **)**

element-access ::= *primary-no-array-creation-expression* **[** *expression-list* **]**

expression-list ::= *expression* | *expression-list* **,** *expression*

this-access ::= **this**

base-access ::= **base** **[** *expression-list* **]** |
base **.** **identifier** |
base **.** **identifier** *type-argument-list*

post-increment-expression ::= *primary-expression* **++**

post-decrement-expression ::= *primary-expression* **--**

object-creation-expression ::= **new** *type* **(** **)** |
new *type* **(** *argument-list* **)**

array-creation-expression ::= **new** *array-type array-initializer* |
new *non-array-type* **[** *expression-list* **]** |
new *non-array-type* **[** *expression-list* **]** *array-initializer* |
new *non-array-type* **[** *expression-list* **]** *rank-specifiers* |
new *non-array-type* **[** *expression-list* **]** *rank-specifiers array-initializer*

delegate-creation-expression ::= **new** *delegate-type* **(** *expression* **)**

typeof-expression ::= **typeof** **(** *type* **)** | **typeof** **(** *unbound-type-name* **)** |
    typeof **(** **void** **)**

unbound-type-name ::= **identifier** |
    identifier *generic-dimension-specifier* |
    identifier **::** **identifier** |
    identifier **::** **identifier** *generic-dimension-specifier* |
    unbound-type-name **.** **identifier** |
    unbound-type-name **.** **identifier** *generic-dimension-specifier*

generic-dimension-specifier ::= < > | < *commas* >

commas ::= , | *commas* ,

checked-expression ::= **checked** **(** *expression* **)**

unchecked-expression ::= **unchecked** **(** *expression* **)**

default-value-expression ::= **default** **(** *type* **)**

anonymous-method-expression ::= **delegate** *block* |
    delegate *anonymous-method-signature block*

anonymous-method-signature ::= **(** **)** | **(** *anonymous-method-parameter-list* **)**

anonymous-method-parameter-list ::= *anonymous-method-parameter* |
    anonymous-method-parameter-list , *anonymous-method-parameter*

anonymous-method-parameter ::= *type* **identifier** | *parameter-modifier type* **identifier**

unary-expression ::= *primary-expression* | **+** *unary-expression* | **-** *unary-expression* |
    ! *unary-expression* | **~** *unary-expression* | *pre-increment-expression* |
    pre-decrement-expression | *cast-expression*

pre-increment-expression ::= **++** *unary-expression*

pre-decrement-expression ::= **--** *unary-expression*

cast-expression ::= **(** *type* **)** *unary-expression*

multiplicative-expression ::= *unary-expression* |
    multiplicative-expression ***** *unary-expression* |
    multiplicative-expression **/** *unary-expression* |
    multiplicative-expression **%** *unary-expression*

additive-expression ::= *multiplicative-expression* |
    additive-expression **+** *multiplicative-expression* |
    additive-expression **-** *multiplicative-expression*

shift-expression ::= *additive-expression* | *shift-expression* **<<** *additive-expression* |
     shift-expression **>>** *additive-expression*

relational-expression ::= *shift-expression* | *relational-expression* **<**
    shift-expression | *relational-expression* **>** *shift-expression* |
    relational-expression **<=** *shift-expression* | *relational-expression* **>=**
    shift-expression | *relational-expression* **is** *type* | *relational-expression* **as** *type*

equality-expression ::= *relational-expression* | *equality-expression* **==**
    relational-expression | *equality-expression* **!=** *relational-expression*

and-expression ::= *equality-expression* | *and-expression* **&** *equality-expression*

exclusive-or-expression ::= *and-expression* | *exclusive-or-expression* **^** *and-expression*

inclusive-or-expression ::= *exclusive-or-expression* |
    inclusive-or-expression **|** *exclusive-or-expression*

conditional-and-expression ::= *inclusive-or-expression* |
    conditional-and-expression **&&** *inclusive-or-expression*

conditional-or-expression ::= *conditional-and-expression* |
    conditional-or-expression **||** *conditional-and-expression*

null-coalescing-expression ::= *conditional-or-expression* |
    conditional-or-expression **??** *null-coalescing-expression*

conditional-expression ::= *null-coalescing-expression* |
    null-coalescing-expression **?** *expression* **:** *expression*

assignment ::= *unary-expression assignment-operator expression*

assignment-operator ::= **=** | **+=** | **-=** | ***=** | **/=** | **%=** | **&=** | **|=** | **^=** | **<<=** | **>>=**

expression ::= *conditional-expression* | *assignment*

constant-expression ::= *expression*

boolean-expression ::= *expression*

statement ::= *labeled-statement* | *declaration-statement* | *embedded-statement*

embedded-statement ::= *block* | *empty-statement* | *expression-statement* |
    selection-statement | *iteration-statement* | *jump-statement* | *try-statement* |
    checked-statement | *unchecked-statement* | *lock-statement* | *using-statement* |
    yield-statement

block ::= **{ }** | **{** *statement-list* **}**

statement-list ::= *statement* | *statement-list statement*

empty-statement ::= **;**

labeled-statement ::= **identifier :** *statement*

declaration-statement ::= *local-variable-declaration* **;** | *local-constant-declaration* **;**

local-variable-declaration ::= *type local-variable-declarators*

local-variable-declarators ::= *local-variable-declarator* |
    local-variable-declarators **,** *local-variable-declarator*

local-variable-declarator ::= **identifier** | **identifier =** *local-variable-initializer*

local-variable-initializer ::= *expression* | *array-initializer*

local-constant-declaration ::= **const** *type constant-declarators*

constant-declarators ::= *constant-declarator* | *constant-declarators* **,**
    constant-declarator;

constant-declarator ::= **identifier =** *constant-expression*

expression-statement ::= *statement-expression* **;**

statement-expression ::= *invocation-expression* | *object-creation-expression* |
    assignment | *post-increment-expression* | *post-decrement-expression* |
    pre-increment-expression | *pre-decrement-expression*

selection-statement ::= *if-statement* | *switch-statement*

if-statement ::= **if (** *boolean-expression* **)** *embedded-statement* |
    if (*boolean-expression* **)** *embedded-statement* **else** *embedded-statement*

switch-statement ::= **switch (** *expression* **)** *switch-block*

switch-block ::= **{ }** | **{** *switch-sections* **}**

switch-sections ::= *switch-section* | *switch-sections switch-section*

switch-section ::= *switch-labels statement-list*

switch-labels ::= *switch-label* | *switch-labels switch-label*

switch-label ::= **case** *constant-expression* **:** | **default :**

iteration-statement ::= *while-statement* | *do-statement* | *for-statement* |
    foreach-statement

while-statement ::= **while (** *boolean-expression* **)** *embedded-statement*

do-statement ::= **do** *embedded-statement* **while (** *boolean-expression* **) ;**

for-statement ::= **for (; ;)** *embedded-statement* |
    for (; ; *for-iterator* **)** *embedded-statement* |
    for (; *for-condition* **;)** *embedded-statement* |
    for (; *for-condition* **;** *for-iterator* **)** *embedded-statement* |
    for (*for-initializer* **; ;)** *embedded-statement* |
    for (*for-initializer* **; ;** *for-iterator* **)** *embedded-statement* |
    for (*for-initializer* **;** *for-condition* **;)** *embedded-statement* |
    for (*for-initializer* **;** *for-condition* **;** *for-iterator* **)** *embedded-statement*

for-initializer ::= *local-variable-declaration* | *statement-expression-list*

for-condition ::= *boolean-expression*

for-iterator ::= *statement-expression-list*

statement-expression-list ::= *statement-expression* | *statement-expression-list* **,**
    statement-expression

foreach-statement ::= **foreach (** *type* **identifier in** *expression* **)** *embedded-statement*

jump-statement ::= *break-statement* | *continue-statement* | *goto-statement* |
    return-statement | *throw-statement*

break-statement ::= **break ;**

continue-statement ::= **continue ;**

goto-statement ::= **goto identifier ;** | **goto case** *constant-expression* **;** |
    goto default ;

return-statement ::= **return ;** | **return** *expression* **;**

throw-statement ::= **throw** **;** | **throw** *expression* **;**

try-statement ::= **try** *block catch-clauses* |
    try *block finally-clause* |
    try *block catch-clauses finally-clause*

catch-clauses ::= *specific-catch-clauses* |
    general-catch-clause |
    specific-catch-clauses general-catch-clause

specific-catch-clauses ::= *specific-catch-clause* | *specific-catch-clauses*
    specific-catch-clause

specific-catch-clause ::= **catch** **(** *class-type* **)** *block* |
    catch **(** *class-type* **identifier** **)** *block*

general-catch-clause ::= **catch** *block*

finally-clause ::= **finally** *block*

checked-statement ::= **checked** *block*

unchecked-statement ::= **unchecked** *block*

lock-statement ::= **lock** **(** *expression* **)** *embedded-statement*

using-statement ::= **using** **(** *resource-acquisition* **)** *embedded-statement*

resource-acquisition ::= *local-variable-declaration* | *expression*

yield-statement ::= **yield** **return** *expression* **;** |
    yield **break** **;**

namespace-declaration ::= **namespace** *qualified-identifier namespace-body* |
    namespace *qualified-identifier namespace-body* **;**

qualified-identifier ::= **identifier** | *qualified-identifier* **.** **identifier**

namespace-body ::= **{** **}** |
    { *namespace-member-declarations* **}** |
    { *using-directives* **}** |
    { *using-directives namespace-member-declarations* **}** |
    { *extern-alias-directives* **}** |
    { *extern-alias-directives namespace-member-declarations* **}** |
    { *extern-alias-directives using-directives* **}** |

{ *extern-alias-directives using-directives namespace-member-declarations* **}**

extern-alias-directives ::= *extern-alias-directive* |
extern-alias-directives extern-alias-directive

extern-alias-directive ::= **extern alias identifier ;**

using-directives ::= *using-directive* | *using-directives using-directive*

using-directive ::= *using-alias-directive* | *using-namespace-directive*

using-alias-directive ::= **using identifier =** *namespace-or-type-name* **;**

using-namespace-directive ::= **using** *namespace-name* **;**

namespace-member-declarations ::= *namespace-member-declaration* |
namespace-member-declarations namespace-member-declaration

namespace-member-declaration ::= *namespace-declaration* | *type-declaration*

type-declaration ::= *class-declaration* | *struct-declaration* |
interface-declaration | *enum-declaration* | *delegate-declaration*

qualified-alias-member ::= **identifier :: identifier** |
identifier :: identifier *type-argument-list*

class-declaration ::= **class identifier** *class-body* |
class identifier *class-body* **;** |
class identifier *type-parameter-constraints-clauses class-body* |
class identifier *type-parameter-constraints-clauses class-body* **;** |
class identifier *class-base class-body* |
class identifier *class-base class-body* **;** |
class identifier *class-base type-parameter-constraints-clauses class-body* |
class identifier *class-base type-parameter-constraints-clauses class-body* **;** |
class identifier *type-parameter-list class-body* |
class identifier *type-parameter-list class-body* **;** |
class identifier *type-parameter-list type-parameter-constraints-clauses*
class-body |
class identifier *type-parameter-list type-parameter-constraints-clauses*
class-body **;** |
class identifier *type-parameter-list class-base class-body* |
class identifier *type-parameter-list class-base class-body* **;** |
class identifier *type-parameter-list class-base*
type-parameter-constraints-clauses class-body |
class identifier *type-parameter-list class-base*
type-parameter-constraints-clauses class-body **;** |

partial class identifier *class-body* |
partial class identifier *class-body* **;** |
partial class identifier *type-parameter-constraints-clauses class-body* |
partial class identifier *type-parameter-constraints-clauses class-body* **;** |
partial class identifier *class-base class-body* |
partial class identifier *class-base class-body* **;** |
partial class identifier *class-base type-parameter-constraints-clauses class-body* |
partial class identifier *class-base type-parameter-constraints-clauses class-body* **;** |
partial class identifier *type-parameter-list class-body* |
partial class identifier *type-parameter-list class-body* **;** |
partial class identifier *type-parameter-list type-parameter-constraints-clauses class-body* |
partial class identifier *type-parameter-list type-parameter-constraints-clauses class-body* **;** |
partial class identifier *type-parameter-list class-base class-body* |
partial class identifier *type-parameter-list class-base class-body* **;** |
partial class identifier *type-parameter-list class-base type-parameter-constraints-clauses class-body* |
partial class identifier *type-parameter-list class-base type-parameter-constraints-clauses class-body* **;** |
class-modifiers **class identifier** *class-body* |
class-modifiers **class identifier** *class-body* **;** |
class-modifiers **class identifier** *type-parameter-constraints-clauses class-body* |
class-modifiers **class identifier** *type-parameter-constraints-clauses class-body* **;** |
class-modifiers **class identifier** *class-base class-body* |
class-modifiers **class identifier** *class-base class-body* **;** |
class-modifiers **class identifier** *class-base type-parameter-constraints-clauses class-body* |
class-modifiers **class identifier** *class-base type-parameter-constraints-clauses class-body* **;** |
class-modifiers **class identifier** *type-parameter-list class-body* |
class-modifiers **class identifier** *type-parameter-list class-body* **;** |
class-modifiers **class identifier** *type-parameter-list type-parameter-constraints-clauses class-body* |
class-modifiers **class identifier** *type-parameter-list type-parameter-constraints-clauses class-body* **;** |
class-modifiers **class identifier** *type-parameter-list class-base class-body* |
class-modifiers **class identifier** *type-parameter-list class-base class-body* **;** |
class-modifiers **class identifier** *type-parameter-list class-base type-parameter-constraints-clauses class-body* |
class-modifiers **class identifier** *type-parameter-list class-base type-parameter-constraints-clauses class-body* **;** |
class-modifiers **partial class identifier** *class-body* |

class-modifiers **partial class identifier** *class-body* **;** |
class-modifiers **partial class identifier** *type-parameter-constraints-clauses*
    class-body |
class-modifiers **partial class identifier** *type-parameter-constraints-clauses*
    class-body **;** |
class-modifiers **partial class identifier** *class-base class-body* |
class-modifiers **partial class identifier** *class-base class-body* **;** |
class-modifiers **partial class identifier** *class-base*
    type-parameter-constraints-clauses class-body |
class-modifiers **partial class identifier** *class-base*
    type-parameter-constraints-clauses class-body **;** |
class-modifiers **partial class identifier** *type-parameter-list class-body* |
class-modifiers **partial class identifier** *type-parameter-list class-body* **;** |
class-modifiers **partial class identifier** *type-parameter-list*
    type-parameter-constraints-clauses class-body |
class-modifiers **partial class identifier** *type-parameter-list*
    type-parameter-constraints-clauses class-body **;** |
class-modifiers **partial class identifier** *type-parameter-list class-base*
    class-body |
class-modifiers **partial class identifier** *type-parameter-list class-base*
    class-body **;** |
class-modifiers **partial class identifier** *type-parameter-list class-base*
    type-parameter-constraints-clauses class-body |
class-modifiers **partial class identifier** *type-parameter-list class-base*
    type-parameter-constraints-clauses class-body **;** |
attributes **class identifier** *class-body* |
attributes **class identifier** *class-body* **;** |
attributes **class identifier** *type-parameter-constraints-clauses class-body* |
attributes **class identifier** *type-parameter-constraints-clauses class-body* **;** |
attributes **class identifier** *class-base class-body* |
attributes **class identifier** *class-base class-body* **;** |
attributes **class identifier** *class-base type-parameter-constraints-clauses*
    class-body |
attributes **class identifier** *class-base type-parameter-constraints-clauses*
    class-body **;** |
attributes **class identifier** *type-parameter-list class-body* |
attributes **class identifier** *type-parameter-list class-body* **;** |
attributes **class identifier** *type-parameter-list*
    type-parameter-constraints-clauses class-body |
attributes **class identifier** *type-parameter-list*
    type-parameter-constraints-clauses class-body **;** |
attributes **class identifier** *type-parameter-list class-base class-body* |
attributes **class identifier** *type-parameter-list class-base class-body* **;** |
attributes **class identifier** *type-parameter-list class-base*
    type-parameter-constraints-clauses class-body |

attributes **class identifier** *type-parameter-list class-base*
type-parameter-constraints-clauses class-body **;** |
attributes **partial class identifier** *class-body* |
attributes **partial class identifier** *class-body* **;** |
attributes **partial class identifier** *type-parameter-constraints-clauses*
class-body |
attributes **partial class identifier** *type-parameter-constraints-clauses*
class-body **;** |
attributes **partial class identifier** *class-base class-body* |
attributes **partial class identifier** *class-base class-body* **;** |
attributes **partial class identifier** *class-base*
type-parameter-constraints-clauses class-body |
attributes **partial class identifier** *class-base*
type-parameter-constraints-clauses class-body **;** |
attributes **partial class identifier** *type-parameter-list class-body* |
attributes **partial class identifier** *type-parameter-list class-body* **;** |
attributes **partial class identifier** *type-parameter-list*
type-parameter-constraints-clauses class-body |
attributes **partial class identifier** *type-parameter-list*
type-parameter-constraints-clauses class-body **;** |
attributes **partial class identifier** *type-parameter-list class-base class-body* |
attributes **partial class identifier** *type-parameter-list class-base class-body* **;** |
attributes **partial class identifier** *type-parameter-list class-base*
type-parameter-constraints-clauses class-body |
attributes **partial class identifier** *type-parameter-list class-base*
type-parameter-constraints-clauses class-body **;** |
attributes class-modifiers **class identifier** *class-body* |
attributes class-modifiers **class identifier** *class-body* **;** |
attributes class-modifiers **class identifier** *type-parameter-constraints-clauses*
class-body |
attributes class-modifiers **class identifier** *type-parameter-constraints-clauses*
class-body **;** |
attributes class-modifiers **class identifier** *class-base class-body* |
attributes class-modifiers **class identifier** *class-base class-body* **;** |
attributes class-modifiers **class identifier** *class-base*
type-parameter-constraints-clauses class-body |
attributes class-modifiers **class identifier** *class-base*
type-parameter-constraints-clauses class-body **;** |
attributes class-modifiers **class identifier** *type-parameter-list class-body* |
attributes class-modifiers **class identifier** *type-parameter-list class-body* **;** |
attributes class-modifiers **class identifier** *type-parameter-list*
type-parameter-constraints-clauses class-body |
attributes class-modifiers **class identifier** *type-parameter-list*
type-parameter-constraints-clauses class-body **;** |
attributes class-modifiers **class identifier** *type-parameter-list class-base*
class-body |

attributes class-modifiers **class identifier** *type-parameter-list class-base*
class-body **;** |
attributes class-modifiers **class identifier** *type-parameter-list class-base*
type-parameter-constraints-clauses class-body |
attributes class-modifiers **class identifier** *type-parameter-list class-base*
type-parameter-constraints-clauses class-body **;** |
attributes class-modifiers **partial class identifier** *class-body* |
attributes class-modifiers **partial class identifier** *class-body* **;** |
attributes class-modifiers **partial class identifier**
type-parameter-constraints-clauses class-body |
attributes class-modifiers **partial class identifier**
type-parameter-constraints-clauses class-body **;** |
attributes class-modifiers **partial class identifier** *class-base class-body* |
attributes class-modifiers **partial class identifier** *class-base class-body* **;** |
attributes class-modifiers **partial class identifier** *class-base*
type-parameter-constraints-clauses class-body |
attributes class-modifiers **partial class identifier** *class-base*
type-parameter-constraints-clauses class-body **;** |
attributes class-modifiers **partial class identifier** *type-parameter-list*
class-body |
attributes class-modifiers **partial class identifier** *type-parameter-list*
class-body **;** |
attributes class-modifiers **partial class identifier** *type-parameter-list*
type-parameter-constraints-clauses class-body |
attributes class-modifiers **partial class identifier** *type-parameter-list*
type-parameter-constraints-clauses class-body **;** |
attributes class-modifiers **partial class identifier** *type-parameter-list*
class-base class-body |
attributes class-modifiers **partial class identifier** *type-parameter-list*
class-base class-body **;** |
attributes class-modifiers **partial class identifier** *type-parameter-list*
class-base type-parameter-constraints-clauses class-body |
attributes class-modifiers **partial class identifier** *type-parameter-list*
class-base type-parameter-constraints-clauses class-body **;**

class-modifiers ::= *class-modifier* | *class-modifiers class-modifier*

class-modifier ::= **new** | **public** | **protected** | **internal** | **private** | **abstract** |
sealed | **static**

class-base ::= **:** *class-type* | **:** *interface-type-list* |
: *class-type* **,** *interface-type-list*

interface-type-list ::= *interface-type* | *interface-type-list* **,** *interface-type*

class-body ::= **{ }** | **{** *class-member-declarations* **}**

```
class-member-declarations ::= class-member-declaration |
    class-member-declarations class-member-declaration

class-member-declaration ::= constant-declaration | field-declaration |
    method-declaration | property-declaration | event-declaration |
    indexer-declaration | operator-declaration | constructor-declaration |
    finalizer-declaration | static-constructor-declaration | type-declaration

constant-declaration ::= const type constant-declarators ; |
    constant-modifiers const type constant-declarators ; |
    attributes const type constant-declarators ; |
    attributes constant-modifiers const type constant-declarators ;

constant-modifiers ::= constant-modifier | constant-modifiers constant-modifier

constant-modifier ::= new | public | protected | internal | private

field-declaration ::= type variable-declarators ; |
    field-modifiers type variable-declarators ; |
    attributes type variable-declarators ; |
    attributes field-modifiers type variable-declarators ;

field-modifiers ::= field-modifier | field-modifiers field-modifier

field-modifier ::= new | public | protected | internal | private | static |
    readonly | volatile

variable-declarators ::= variable-declarator |
    variable-declarators , variable-declarator

variable-declarator ::= identifier | identifier = variable-initializer

variable-initializer ::= expression | array-initializer

method-declaration ::= method-header method-body

method-header ::= return-type member-name ( ) |
    return-type member-name ( ) type-parameter-constraints-clauses |
    return-type member-name ( formal-parameter-list ) |
    return-type member-name ( formal-parameter-list )
        type-parameter-constraints-clauses |
    return-type member-name type-parameter-list ( ) |
    return-type member-name type-parameter-list ( )
        type-parameter-constraints-clauses |
    return-type member-name type-parameter-list ( formal-parameter-list ) |
```

return-type member-name type-parameter-list **(** *formal-parameter-list* **)**
type-parameter-constraints-clauses |
method-modifiers return-type member-name **(** **)** |
method-modifiers return-type member-name **(** **)** *type-parameter-constraints-clauses* |
method-modifiers return-type member-name **(** *formal-parameter-list* **)** |
method-modifiers return-type member-name **(** *formal-parameter-list* **)**
type-parameter-constraints-clauses |
method-modifiers return-type member-name type-parameter-list **(** **)** |
method-modifiers return-type member-name type-parameter-list **(** **)**
type-parameter-constraints-clauses |
method-modifiers return-type member-name type-parameter-list **(**
formal-parameter-list **)** |
method-modifiers return-type member-name type-parameter-list **(**
formal-parameter-list **)** *type-parameter-constraints-clauses* |
attributes return-type member-name **(** **)** |
attributes return-type member-name **(** **)** *type-parameter-constraints-clauses* |
attributes return-type member-name **(** *formal-parameter-list* **)** |
attributes return-type member-name **(** *formal-parameter-list* **)**
type-parameter-constraints-clauses |
attributes return-type member-name type-parameter-list **(** **)** |
attributes return-type member-name type-parameter-list **(** **)**
type-parameter-constraints-clauses |
attributes return-type member-name type-parameter-list **(** *formal-parameter-list* **)**
|
attributes return-type member-name type-parameter-list **(** *formal-parameter-list* **)**
type-parameter-constraints-clauses |
attributes method-modifiers return-type member-name **(** **)** |
attributes method-modifiers return-type member-name **(** **)**
type-parameter-constraints-clauses |
attributes method-modifiers return-type member-name **(** *formal-parameter-list* **)** |
attributes method-modifiers return-type member-name **(** *formal-parameter-list* **)**
type-parameter-constraints-clauses |
attributes method-modifiers return-type member-name type-parameter-list **(** **)** |
attributes method-modifiers return-type member-name type-parameter-list **(** **)**
type-parameter-constraints-clauses |
attributes method-modifiers return-type member-name type-parameter-list **(**
formal-parameter-list **)** |
attributes method-modifiers return-type member-name type-parameter-list **(**
formal-parameter-list **)** *type-parameter-constraints-clauses*

method-modifiers ::= *method-modifier* | *method-modifiers method-modifier*

method-modifier ::= **new** | **public** | **protected** | **internal** | **private** | **static** |
virtual | **sealed** | **override** | **abstract** | **extern**

return-type ::= *type* | **void**

member-name ::= **identifier** | *interface-type* **.** **identifier**

method-body ::= *block* | **;**

formal-parameter-list ::= *fixed-parameters* | *fixed-parameters* **,** *parameter-array* |
    parameter-array

fixed-parameters ::= *fixed-parameter* | *fixed-parameters* **,** *fixed-parameter*

fixed-parameter ::= *type* **identifier** |
    parameter-modifier type **identifier** |
    attributes type **identifier** |
    attributes parameter-modifier type **identifier**

parameter-modifier ::= **ref** | **out**

parameter-array ::= **params** *array-type* **identifier** |
    attributes **params** *array-type* **identifier**

property-declaration ::= *type member-name* **{** *accessor-declarations* **}** |
    property-modifiers type member-name **{** *accessor-declarations* **}** |
    attributes type member-name **{** *accessor-declarations* **}** |
    attributes property-modifiers type member-name **{** *accessor-declarations* **}**

property-modifiers ::= *property-modifier* | *property-modifiers property-modifier*

property-modifier ::= **new** | **public** | **protected** | **internal** | **private** | **static** |
    virtual | **sealed** | **override** | **abstract** | **extern**

accessor-declarations ::= *get-accessor-declaration* |
    get-accessor-declaration set-accessor-declaration |
    set-accessor-declaration |
    set-accessor-declaration get-accessor-declaration

get-accessor-declaration ::= **get** *accessor-body* |
    accessor-modifier **get** *accessor-body* |
    attributes **get** *accessor-body* |
    attributes accessor-modifier **get** *accessor-body*

set-accessor-declaration ::= **set** *accessor-body* |
    accessor-modifier **set** *accessor-body* |
    attributes **set** *accessor-body* |
    attributes accessor-modifier **set** *accessor-body*

accessor-modifier ::= **protected** | **internal** | **private** | **protected internal** |
    internal protected

accessor-body ::= *block* | **;**

event-declaration ::= **event** *type variable-declarators* **;** |
    event-modifiers **event** *type variable-declarators* **;** |
    attributes **event** *type variable-declarators* **;** |
    attributes event-modifiers **event** *type variable-declarators* **;** |
    event *type member-name* **{** *event-accessor-declarations* **}** |
    event-modifiers **event** *type member-name* **{** *event-accessor-declarations* **}** |
    attributes **event** *type member-name* **{** *event-accessor-declarations* **}** |
    attributes event-modifiers **event** *type member-name* **{** *event-accessor-declarations* **}**

event-modifiers ::= *event-modifier* | *event-modifiers event-modifier*

event-modifier ::= **new** | **public** | **protected** | **internal** | **private** | **static** |
    virtual | **sealed** | **override** | **abstract** | **extern**

event-accessor-declarations ::= *add-accessor-declaration remove-accessor-declaration*
    | *remove-accessor-declaration add-accessor-declaration*

add-accessor-declaration ::= **add** *block* | *attributes* **add** *block*

remove-accessor-declaration ::= **remove** *block* | *attributes* **remove** *block*

indexer-declaration ::= *indexer-declarator* **{** *accessor-declarations* **}** |
    indexer-modifiers indexer-declarator **{** *accessor-declarations* **}** |
    attributes indexer-declarator **{** *accessor-declarations* **}** |
    attributes indexer-modifiers indexer-declarator **{** *accessor-declarations* **}**

indexer-modifiers ::= *indexer-modifier* | *indexer-modifiers indexer-modifier*

indexer-modifier ::= **new** | **public** | **protected** | **internal** | **private** | **virtual** |
    sealed | **override** | **abstract** | **extern**

indexer-declarator ::= *type* **this** **[** *formal-parameter-list* **]** |
    type interface-type **.** **this** **[** *formal-parameter-list* **]**

operator-declaration ::= *operator-modifiers operator-declarator operator-body* |
    attributes operator-modifiers operator-declarator operator-body

operator-modifiers ::= *operator-modifier* | *operator-modifiers operator-modifier*

operator-modifier ::= **public** | **static** | **extern**

operator-declarator ::= *unary-operator-declarator* | *binary-operator-declarator* |
    conversion-operator-declarator

unary-operator-declarator ::= *type* **operator** *overloadable-unary-operator* **(** *type*
    identifier)

overloadable-unary-operator ::= **+** | **-** | **!** | **~** | **++** | **--** | **true** | **false**

binary-operator-declarator ::= *type* **operator** *overloadable-binary-operator*
    (*type* **identifier ,** *type* **identifier)**

overloadable-binary-operator ::= **+** | **-** | ***** | **/** | **%** | **&** | **|** | **^** | **<<** | **>>** | **==** |
    != | **>** | **<** | **>=** | **<=**

conversion-operator-declarator ::= **implicit operator** *type* **(** *type* **identifier)** |
    explicit operator *type* **(** *type* **identifier)**

operator-body ::= *block* | **;**

constructor-declaration ::= *constructor-declarator constructor-body* |
    constructor-modifiers constructor-declarator constructor-body |
    attributes constructor-declarator constructor-body |
    attributes constructor-modifiers constructor-declarator constructor-body

constructor-modifiers ::= *constructor-modifier* | *constructor-modifiers*
    constructor-modifier

constructor-modifier ::= **public** | **protected** | **internal** | **private** | **extern**

constructor-declarator ::= **identifier ( )** |
    identifier ( ) *constructor-initializer* |
    identifier (*formal-parameter-list* **)** |
    identifier (*formal-parameter-list* **)** *constructor-initializer*

constructor-initializer ::= **: base ( )** |
    : base (*argument-list* **)** |
    : this ( ) |
    : this (*argument-list* **)**

constructor-body ::= *block* | **;**

static-constructor-declaration ::= *static-constructor-modifiers* **identifier ( )**
    static-constructor-body |
    attributes static-constructor-modifiers **identifier ( )** *static-constructor-body*

static-constructor-modifiers ::= **static** | **extern static** | **static extern**

static-constructor-body ::= *block* | **;**

finalizer-declaration ::= **extern ~ identifier ( )** *finalizer-body* |
    attributes **extern ~ identifier ( )** *finalizer-body* |
    ~ identifier ( ) *finalizer-body* |
    attributes **~ identifier ( )** *finalizer-body*

finalizer-body ::= *block* | **;**

struct-declaration ::= **struct identifier** *struct-body* |
    struct identifier *struct-body* **;** |
    struct identifier *type-parameter-constraints-clauses struct-body* |
    struct identifier *type-parameter-constraints-clauses struct-body* **;** |
    struct identifier *struct-interfaces struct-body* |
    struct identifier *struct-interfaces struct-body* **;** |
    struct identifier *struct-interfaces type-parameter-constraints-clauses*
        struct-body |
    struct identifier *struct-interfaces type-parameter-constraints-clauses*
        struct-body **;** |
    struct identifier *type-parameter-list struct-body* |
    struct identifier *type-parameter-list struct-body* **;** |
    struct identifier *type-parameter-list type-parameter-constraints-clauses*
        struct-body |
    struct identifier *type-parameter-list type-parameter-constraints-clauses*
        struct-body **;** |
    struct identifier *type-parameter-list struct-interfaces struct-body* |
    struct identifier *type-parameter-list struct-interfaces struct-body* **;** |
    struct identifier *type-parameter-list struct-interfaces*
        type-parameter-constraints-clauses struct-body |
    struct identifier *type-parameter-list struct-interfaces*
        type-parameter-constraints-clauses struct-body **;** |
    partial struct identifier *struct-body* |
    partial struct identifier *struct-body* **;** |
    partial struct identifier *type-parameter-constraints-clauses struct-body* |
    partial struct identifier *type-parameter-constraints-clauses struct-body* **;** |
    partial struct identifier *struct-interfaces struct-body* |
    partial struct identifier *struct-interfaces struct-body* **;** |
    partial struct identifier *struct-interfaces type-parameter-constraints-clauses*
        struct-body |
    partial struct identifier *struct-interfaces type-parameter-constraints-clauses*
        struct-body **;** |
    partial struct identifier *type-parameter-list struct-body* |
    partial struct identifier *type-parameter-list struct-body* **;** |
    partial struct identifier *type-parameter-list type-parameter-constraints-clauses*
        struct-body |

partial struct identifier *type-parameter-list type-parameter-constraints-clauses*
    struct-body **;** |
partial struct identifier *type-parameter-list struct-interfaces struct-body* |
partial struct identifier *type-parameter-list struct-interfaces struct-body* **;** |
partial struct identifier *type-parameter-list struct-interfaces*
    type-parameter-constraints-clauses struct-body |
partial struct identifier *type-parameter-list struct-interfaces*
    type-parameter-constraints-clauses struct-body **;** |
struct-modifiers **struct identifier** *struct-body* |
struct-modifiers **struct identifier** *struct-body* **;** |
struct-modifiers **struct identifier** *type-parameter-constraints-clauses*
    struct-body |
struct-modifiers **struct identifier** *type-parameter-constraints-clauses*
    struct-body **;** |
struct-modifiers **struct identifier** *struct-interfaces struct-body* |
struct-modifiers **struct identifier** *struct-interfaces struct-body* **;** |
struct-modifiers **struct identifier** *struct-interfaces*
    type-parameter-constraints-clauses struct-body |
struct-modifiers **struct identifier** *struct-interfaces*
    type-parameter-constraints-clauses struct-body **;** |
struct-modifiers **struct identifier** *type-parameter-list struct-body* |
struct-modifiers **struct identifier** *type-parameter-list struct-body* **;** |
struct-modifiers **struct identifier** *type-parameter-list*
    type-parameter-constraints-clauses struct-body |
struct-modifiers **struct identifier** *type-parameter-list*
    type-parameter-constraints-clauses struct-body **;** |
struct-modifiers **struct identifier** *type-parameter-list struct-interfaces*
    struct-body |
struct-modifiers **struct identifier** *type-parameter-list struct-interfaces*
    struct-body **;** |
struct-modifiers **struct identifier** *type-parameter-list struct-interfaces*
    type-parameter-constraints-clauses struct-body |
struct-modifiers **struct identifier** *type-parameter-list struct-interfaces*
    type-parameter-constraints-clauses struct-body **;** |
struct-modifiers **partial struct identifier** *struct-body* |
struct-modifiers **partial struct identifier** *struct-body* **;** |
struct-modifiers **partial struct identifier** *type-parameter-constraints-clauses*
    struct-body |
struct-modifiers **partial struct identifier** *type-parameter-constraints-clauses*
    struct-body **;** |
struct-modifiers **partial struct identifier** *struct-interfaces struct-body* |
struct-modifiers **partial struct identifier** *struct-interfaces struct-body* **;** |
struct-modifiers **partial struct identifier** *struct-interfaces*
    type-parameter-constraints-clauses struct-body |
struct-modifiers **partial struct identifier** *struct-interfaces*
    type-parameter-constraints-clauses struct-body **;** |

struct-modifiers **partial struct identifier** *type-parameter-list struct-body* |
struct-modifiers **partial struct identifier** *type-parameter-list struct-body* **;** |
struct-modifiers **partial struct identifier** *type-parameter-list*
    type-parameter-constraints-clauses struct-body |
struct-modifiers **partial struct identifier** *type-parameter-list*
    type-parameter-constraints-clauses struct-body **;** |
struct-modifiers **partial struct identifier** *type-parameter-list struct-interfaces*
    struct-body |
struct-modifiers **partial struct identifier** *type-parameter-list struct-interfaces*
    struct-body **;** |
struct-modifiers **partial struct identifier** *type-parameter-list struct-interfaces*
    type-parameter-constraints-clauses struct-body |
struct-modifiers **partial struct identifier** *type-parameter-list struct-interfaces*
    type-parameter-constraints-clauses struct-body **;** |
attributes **struct identifier** *struct-body* |
attributes **struct identifier** *struct-body* **;** |
attributes **struct identifier** *type-parameter-constraints-clauses struct-body* |
attributes **struct identifier** *type-parameter-constraints-clauses struct-body* **;** |
attributes **struct identifier** *struct-interfaces struct-body* |
attributes **struct identifier** *struct-interfaces struct-body* **;** |
attributes **struct identifier** *struct-interfaces*
    type-parameter-constraints-clauses struct-body |
attributes **struct identifier** *struct-interfaces*
    type-parameter-constraints-clauses struct-body **;** |
attributes **struct identifier** *type-parameter-list struct-body* |
attributes **struct identifier** *type-parameter-list struct-body* **;** |
attributes **struct identifier** *type-parameter-list*
    type-parameter-constraints-clauses struct-body |
attributes **struct identifier** *type-parameter-list*
    type-parameter-constraints-clauses struct-body **;** |
attributes **struct identifier** *type-parameter-list struct-interfaces struct-body* |
attributes **struct identifier** *type-parameter-list struct-interfaces struct-body* **;**
    |
attributes **struct identifier** *type-parameter-list struct-interfaces*
    type-parameter-constraints-clauses struct-body |
attributes **struct identifier** *type-parameter-list struct-interfaces*
    type-parameter-constraints-clauses struct-body **;** |
attributes **partial struct identifier** *struct-body* |
attributes **partial struct identifier** *struct-body* **;** |
attributes **partial struct identifier** *type-parameter-constraints-clauses*
    struct-body |
attributes **partial struct identifier** *type-parameter-constraints-clauses*
    struct-body **;** |
attributes **partial struct identifier** *struct-interfaces struct-body* |
attributes **partial struct identifier** *struct-interfaces struct-body* **;** |

attributes **partial struct identifier** *struct-interfaces*
    type-parameter-constraints-clauses struct-body |
attributes **partial struct identifier** *struct-interfaces*
    type-parameter-constraints-clauses struct-body **;** |
attributes **partial struct identifier** *type-parameter-list struct-body* |
attributes **partial struct identifier** *type-parameter-list struct-body* **;** |
attributes **partial struct identifier** *type-parameter-list*
    type-parameter-constraints-clauses struct-body |
attributes **partial struct identifier** *type-parameter-list*
    type-parameter-constraints-clauses struct-body **;** |
attributes **partial struct identifier** *type-parameter-list struct-interfaces*
    struct-body |
attributes **partial struct identifier** *type-parameter-list struct-interfaces*
    struct-body **;** |
attributes **partial struct identifier** *type-parameter-list struct-interfaces*
    type-parameter-constraints-clauses struct-body |
attributes **partial struct identifier** *type-parameter-list struct-interfaces*
    type-parameter-constraints-clauses struct-body **;** |
attributes struct-modifiers **struct identifier** *struct-body* |
attributes struct-modifiers **struct identifier** *struct-body* **;** |
attributes struct-modifiers **struct identifier** *type-parameter-constraints-clauses*
    struct-body |
attributes struct-modifiers **struct identifier** *type-parameter-constraints-clauses*
    struct-body **;** |
attributes struct-modifiers **struct identifier** *struct-interfaces struct-body* |
attributes struct-modifiers **struct identifier** *struct-interfaces struct-body* **;** |
attributes struct-modifiers **struct identifier** *struct-interfaces*
    type-parameter-constraints-clauses struct-body |
attributes struct-modifiers **struct identifier** *struct-interfaces*
    type-parameter-constraints-clauses struct-body **;** |
attributes struct-modifiers **struct identifier** *type-parameter-list struct-body* |
attributes struct-modifiers **struct identifier** *type-parameter-list struct-body* **;** |
attributes struct-modifiers **struct identifier** *type-parameter-list*
    type-parameter-constraints-clauses struct-body |
attributes struct-modifiers **struct identifier** *type-parameter-list*
    type-parameter-constraints-clauses struct-body **;** |
attributes struct-modifiers **struct identifier** *type-parameter-list*
    struct-interfaces struct-body |
attributes struct-modifiers **struct identifier** *type-parameter-list*
    struct-interfaces struct-body **;** |
attributes struct-modifiers **struct identifier** *type-parameter-list*
    struct-interfaces type-parameter-constraints-clauses struct-body |
attributes struct-modifiers **struct identifier** *type-parameter-list*
    struct-interfaces type-parameter-constraints-clauses struct-body **;** |
attributes struct-modifiers **partial struct identifier** *struct-body* |
attributes struct-modifiers **partial struct identifier** *struct-body* **;** |

attributes *struct-modifiers* **partial** **struct** **identifier**
type-parameter-constraints-clauses *struct-body* |
attributes *struct-modifiers* **partial** **struct** **identifier**
type-parameter-constraints-clauses *struct-body* **;** |
attributes *struct-modifiers* **partial** **struct** **identifier** *struct-interfaces*
struct-body |
attributes *struct-modifiers* **partial** **struct** **identifier** *struct-interfaces*
struct-body **;** |
attributes *struct-modifiers* **partial** **struct** **identifier** *struct-interfaces*
type-parameter-constraints-clauses *struct-body* |
attributes *struct-modifiers* **partial** **struct** **identifier** *struct-interfaces*
type-parameter-constraints-clauses *struct-body* **;** |
attributes *struct-modifiers* **partial** **struct** **identifier** *type-parameter-list*
struct-body |
attributes *struct-modifiers* **partial** **struct** **identifier** *type-parameter-list*
struct-body **;** |
attributes *struct-modifiers* **partial** **struct** **identifier** *type-parameter-list*
type-parameter-constraints-clauses *struct-body* |
attributes *struct-modifiers* **partial** **struct** **identifier** *type-parameter-list*
type-parameter-constraints-clauses *struct-body* **;** |
attributes *struct-modifiers* **partial** **struct** **identifier** *type-parameter-list*
struct-interfaces *struct-body* |
attributes *struct-modifiers* **partial** **struct** **identifier** *type-parameter-list*
struct-interfaces *struct-body* **;** |
attributes *struct-modifiers* **partial** **struct** **identifier** *type-parameter-list*
struct-interfaces *type-parameter-constraints-clauses* *struct-body* |
attributes *struct-modifiers* **partial** **struct** **identifier** *type-parameter-list*
struct-interfaces *type-parameter-constraints-clauses* *struct-body* **;**

struct-modifiers ::= *struct-modifier* | *struct-modifiers* *struct-modifier*

struct-modifier ::= **new** | **public** | **protected** | **internal** | **private**

struct-interfaces ::= **:** *interface-type-list*

struct-body ::= **{** **}** | **{** *struct-member-declarations* **}**

struct-member-declarations ::= *struct-member-declaration* |
struct-member-declarations *struct-member-declaration*

struct-member-declaration ::= *constant-declaration* | *field-declaration* |
method-declaration | *property-declaration* | *event-declaration* |
indexer-declaration | *operator-declaration* | *constructor-declaration* |
static-constructor-declaration | *type-declaration*

array-initializer ::= **{** *variable-initializer-list* **,** **}** |

{ } | **{** *variable-initializer-list* **}**

variable-initializer-list ::= *variable-initializer* | *variable-initializer-list* **,** *variable-initializer*

interface-declaration ::= **interface identifier** *interface-body* |
interface identifier *interface-body* **;** |
interface identifier *type-parameter-constraints-clauses interface-body* |
interface identifier *type-parameter-constraints-clauses interface-body* **;** |
interface identifier *interface-base interface-body* |
interface identifier *interface-base interface-body* **;** |
interface identifier *interface-base type-parameter-constraints-clauses interface-body* |
interface identifier *interface-base type-parameter-constraints-clauses interface-body* **;** |
interface identifier *type-parameter-list interface-body* |
interface identifier *type-parameter-list interface-body* **;** |
interface identifier *type-parameter-list type-parameter-constraints-clauses interface-body* |
interface identifier *type-parameter-list type-parameter-constraints-clauses interface-body* **;** |
interface identifier *type-parameter-list interface-base interface-body* |
interface identifier *type-parameter-list interface-base interface-body* **;** |
interface identifier *type-parameter-list interface-base type-parameter-constraints-clauses interface-body* |
interface identifier *type-parameter-list interface-base type-parameter-constraints-clauses interface-body* **;** |
partial interface identifier *interface-body* |
partial interface identifier *interface-body* **;** |
partial interface identifier *type-parameter-constraints-clauses interface-body* |
partial interface identifier *type-parameter-constraints-clauses interface-body* **;** |
partial interface identifier *interface-base interface-body* |
partial interface identifier *interface-base interface-body* **;** |
partial interface identifier *interface-base type-parameter-constraints-clauses interface-body* |
partial interface identifier *interface-base type-parameter-constraints-clauses interface-body* **;** |
partial interface identifier *type-parameter-list interface-body* |
partial interface identifier *type-parameter-list interface-body* **;** |
partial interface identifier *type-parameter-list type-parameter-constraints-clauses interface-body* |
partial interface identifier *type-parameter-list type-parameter-constraints-clauses interface-body* **;** |
partial interface identifier *type-parameter-list interface-base interface-body* |

partial interface identifier *type-parameter-list interface-base interface-body* **;**
    |
partial interface identifier *type-parameter-list interface-base*
    type-parameter-constraints-clauses interface-body |
partial interface identifier *type-parameter-list interface-base*
    type-parameter-constraints-clauses interface-body **;** |
interface-modifiers **interface identifier** *interface-body* |
interface-modifiers **interface identifier** *interface-body* **;** |
interface-modifiers **interface identifier** *type-parameter-constraints-clauses*
    interface-body |
interface-modifiers **interface identifier** *type-parameter-constraints-clauses*
    interface-body **;** |
interface-modifiers **interface identifier** *interface-base interface-body* |
interface-modifiers **interface identifier** *interface-base interface-body* **;** |
interface-modifiers **interface identifier** *interface-base*
    type-parameter-constraints-clauses interface-body |
interface-modifiers **interface identifier** *interface-base*
    type-parameter-constraints-clauses interface-body **;** |
interface-modifiers **interface identifier** *type-parameter-list interface-body* |
interface-modifiers **interface identifier** *type-parameter-list interface-body* **;** |
interface-modifiers **interface identifier** *type-parameter-list*
    type-parameter-constraints-clauses interface-body |
interface-modifiers **interface identifier** *type-parameter-list*
    type-parameter-constraints-clauses interface-body **;** |
interface-modifiers **interface identifier** *type-parameter-list interface-base*
    interface-body |
interface-modifiers **interface identifier** *type-parameter-list interface-base*
    interface-body **;** |
interface-modifiers **interface identifier** *type-parameter-list interface-base*
    type-parameter-constraints-clauses interface-body |
interface-modifiers **interface identifier** *type-parameter-list interface-base*
    type-parameter-constraints-clauses interface-body **;** |
interface-modifiers **partial interface identifier** *interface-body* |
interface-modifiers **partial interface identifier** *interface-body* **;** |
interface-modifiers **partial interface identifier**
    type-parameter-constraints-clauses interface-body |
interface-modifiers **partial interface identifier**
    type-parameter-constraints-clauses interface-body **;** |
interface-modifiers **partial interface identifier** *interface-base interface-body* |
interface-modifiers **partial interface identifier** *interface-base interface-body* **;**
    |
interface-modifiers **partial interface identifier** *interface-base*
    type-parameter-constraints-clauses interface-body |
interface-modifiers **partial interface identifier** *interface-base*
    type-parameter-constraints-clauses interface-body **;** |

interface-modifiers **partial interface identifier** *type-parameter-list*
    interface-body |
interface-modifiers **partial interface identifier** *type-parameter-list*
    interface-body **;** |
interface-modifiers **partial interface identifier** *type-parameter-list*
    type-parameter-constraints-clauses interface-body |
interface-modifiers **partial interface identifier** *type-parameter-list*
    type-parameter-constraints-clauses interface-body **;** |
interface-modifiers **partial interface identifier** *type-parameter-list*
    interface-base interface-body |
interface-modifiers **partial interface identifier** *type-parameter-list*
    interface-base interface-body **;** |
interface-modifiers **partial interface identifier** *type-parameter-list*
    interface-base type-parameter-constraints-clauses interface-body |
interface-modifiers **partial interface identifier** *type-parameter-list*
    interface-base type-parameter-constraints-clauses interface-body **;** |
attributes **interface identifier** *interface-body* |
attributes **interface identifier** *interface-body* **;** |
attributes **interface identifier** *type-parameter-constraints-clauses*
    interface-body |
attributes **interface identifier** *type-parameter-constraints-clauses*
    interface-body **;** |
attributes **interface identifier** *interface-base interface-body* |
attributes **interface identifier** *interface-base interface-body* **;** |
attributes **interface identifier** *interface-base*
    type-parameter-constraints-clauses interface-body |
attributes **interface identifier** *interface-base*
    type-parameter-constraints-clauses interface-body **;** |
attributes **interface identifier** *type-parameter-list interface-body* |
attributes **interface identifier** *type-parameter-list interface-body* **;** |
attributes **interface identifier** *type-parameter-list*
    type-parameter-constraints-clauses interface-body |
attributes **interface identifier** *type-parameter-list*
    type-parameter-constraints-clauses interface-body **;** |
attributes **interface identifier** *type-parameter-list interface-base*
    interface-body |
attributes **interface identifier** *type-parameter-list interface-base*
    interface-body **;** |
attributes **interface identifier** *type-parameter-list interface-base*
    type-parameter-constraints-clauses interface-body |
attributes **interface identifier** *type-parameter-list interface-base*
    type-parameter-constraints-clauses interface-body **;** |
attributes **partial interface identifier** *interface-body* |
attributes **partial interface identifier** *interface-body* **;** |
attributes **partial interface identifier** *type-parameter-constraints-clauses*
    interface-body |

attributes **partial interface identifier** *type-parameter-constraints-clauses*
interface-body **;** |
attributes **partial interface identifier** *interface-base interface-body* |
attributes **partial interface identifier** *interface-base interface-body* **;** |
attributes **partial interface identifier** *interface-base*
type-parameter-constraints-clauses interface-body |
attributes **partial interface identifier** *interface-base*
type-parameter-constraints-clauses interface-body **;** |
attributes **partial interface identifier** *type-parameter-list interface-body* |
attributes **partial interface identifier** *type-parameter-list interface-body* **;** |
attributes **partial interface identifier** *type-parameter-list*
type-parameter-constraints-clauses interface-body |
attributes **partial interface identifier** *type-parameter-list*
type-parameter-constraints-clauses interface-body **;** |
attributes **partial interface identifier** *type-parameter-list interface-base*
interface-body |
attributes **partial interface identifier** *type-parameter-list interface-base*
interface-body **;** |
attributes **partial interface identifier** *type-parameter-list interface-base*
type-parameter-constraints-clauses interface-body |
attributes **partial interface identifier** *type-parameter-list interface-base*
type-parameter-constraints-clauses interface-body **;** |
attributes interface-modifiers **interface identifier** *interface-body* |
attributes interface-modifiers **interface identifier** *interface-body* **;** |
attributes interface-modifiers **interface identifier**
type-parameter-constraints-clauses interface-body |
attributes interface-modifiers **interface identifier**
type-parameter-constraints-clauses interface-body **;** |
attributes interface-modifiers **interface identifier** *interface-base*
interface-body |
attributes interface-modifiers **interface identifier** *interface-base*
interface-body **;** |
attributes interface-modifiers **interface identifier** *interface-base*
type-parameter-constraints-clauses interface-body |
attributes interface-modifiers **interface identifier** *interface-base*
type-parameter-constraints-clauses interface-body **;** |
attributes interface-modifiers **interface identifier** *type-parameter-list*
interface-body |
attributes interface-modifiers **interface identifier** *type-parameter-list*
interface-body **;** |
attributes interface-modifiers **interface identifier** *type-parameter-list*
type-parameter-constraints-clauses interface-body |
attributes interface-modifiers **interface identifier** *type-parameter-list*
type-parameter-constraints-clauses interface-body **;** |
attributes interface-modifiers **interface identifier** *type-parameter-list*
interface-base interface-body |

attributes interface-modifiers **interface identifier** *type-parameter-list*
interface-base interface-body **;** |
attributes interface-modifiers **interface identifier** *type-parameter-list*
interface-base type-parameter-constraints-clauses interface-body |
attributes interface-modifiers **interface identifier** *type-parameter-list*
interface-base type-parameter-constraints-clauses interface-body **;** |
attributes interface-modifiers **partial interface identifier** *interface-body* |
attributes interface-modifiers **partial interface identifier** *interface-body* **;** |
attributes interface-modifiers **partial interface identifier**
type-parameter-constraints-clauses interface-body |
attributes interface-modifiers **partial interface identifier**
type-parameter-constraints-clauses interface-body **;** |
attributes interface-modifiers **partial interface identifier** *interface-base*
interface-body |
attributes interface-modifiers **partial interface identifier** *interface-base*
interface-body **;** |
attributes interface-modifiers **partial interface identifier** *interface-base*
type-parameter-constraints-clauses interface-body |
attributes interface-modifiers **partial interface identifier** *interface-base*
type-parameter-constraints-clauses interface-body **;** |
attributes interface-modifiers **partial interface identifier** *type-parameter-list*
interface-body |
attributes interface-modifiers **partial interface identifier** *type-parameter-list*
interface-body **;** |
attributes interface-modifiers **partial interface identifier** *type-parameter-list*
type-parameter-constraints-clauses interface-body |
attributes interface-modifiers **partial interface identifier** *type-parameter-list*
type-parameter-constraints-clauses interface-body **;** |
attributes interface-modifiers **partial interface identifier** *type-parameter-list*
interface-base interface-body |
attributes interface-modifiers **partial interface identifier** *type-parameter-list*
interface-base interface-body **;** |
attributes interface-modifiers **partial interface identifier** *type-parameter-list*
interface-base type-parameter-constraints-clauses interface-body |
attributes interface-modifiers **partial interface identifier** *type-parameter-list*
interface-base type-parameter-constraints-clauses interface-body **;**

interface-modifiers ::= *interface-modifier* | *interface-modifiers interface-modifier*

interface-modifier ::= **new** | **public** | **protected** | **internal** | **private**

interface-base ::= **:** *interface-type-list*

interface-body ::= **{ }** | **{** *interface-member-declarations* **}**

interface-member-declarations ::= *interface-member-declaration* |
    interface-member-declarations interface-member-declaration

interface-member-declaration ::= *interface-method-declaration* |
    interface-property-declaration | *interface-event-declaration* |
    interface-indexer-declaration

interface-method-declaration ::= *return-type* **identifier** **(** **)** **;** |
    return-type **identifier** **(** **)** *type-parameter-constraints-clauses* **;** |
    return-type **identifier** **(** *formal-parameter-list* **)** **;** |
    return-type **identifier** **(** *formal-parameter-list* **)**
        type-parameter-constraints-clauses **;** |
    return-type **identifier** *type-parameter-list* **(** **)** **;** |
    return-type **identifier** *type-parameter-list* **(** **)**
        type-parameter-constraints-clauses **;** |
    return-type **identifier** *type-parameter-list* **(** *formal-parameter-list* **)** **;** |
    return-type **identifier** *type-parameter-list* **(** *formal-parameter-list* **)**
        type-parameter-constraints-clauses **;** |
    new *return-type* **identifier** **(** **)** **;** |
    new *return-type* **identifier** **(** **)** *type-parameter-constraints-clauses* **;** |
    new *return-type* **identifier** **(** *formal-parameter-list* **)** **;** |
    new *return-type* **identifier** **(** *formal-parameter-list* **)**
        type-parameter-constraints-clauses **;** |
    new *return-type* **identifier** *type-parameter-list* **(** **)** **;** |
    new *return-type* **identifier** *type-parameter-list* **(** **)**
        type-parameter-constraints-clauses **;** |
    new *return-type* **identifier** *type-parameter-list* **(** *formal-parameter-list* **)** **;** |
    new *return-type* **identifier** *type-parameter-list* **(** *formal-parameter-list* **)**
        type-parameter-constraints-clauses **;** |
    attributes return-type **identifier** **(** **)** **;** |
    attributes return-type **identifier** **(** **)** *type-parameter-constraints-clauses* **;** |
    attributes return-type **identifier** **(** *formal-parameter-list* **)** **;** |
    attributes return-type **identifier** **(** *formal-parameter-list* **)**
        type-parameter-constraints-clauses **;** |
    attributes return-type **identifier** *type-parameter-list* **(** **)** **;** |
    attributes return-type **identifier** *type-parameter-list* **(** **)**
        type-parameter-constraints-clauses **;** |
    attributes return-type **identifier** *type-parameter-list* **(** *formal-parameter-list* **)**
        ; |
    attributes return-type **identifier** *type-parameter-list* **(** *formal-parameter-list* **)**
        type-parameter-constraints-clauses **;** |
    attributes **new** *return-type* **identifier** **(** **)** **;** |
    attributes **new** *return-type* **identifier** **(** **)** *type-parameter-constraints-clauses* **;** |
    attributes **new** *return-type* **identifier** **(** *formal-parameter-list* **)** **;** |
    attributes **new** *return-type* **identifier** **(** *formal-parameter-list* **)**
        type-parameter-constraints-clauses **;** |

attributes **new** *return-type* **identifier** *type-parameter-list* **(** **)** **;** |
attributes **new** *return-type* **identifier** *type-parameter-list* **(** **)**
type-parameter-constraints-clauses **;** |
attributes **new** *return-type* **identifier** *type-parameter-list* **(**
formal-parameter-list **)** **;** |
attributes **new** *return-type* **identifier** *type-parameter-list* **(**
formal-parameter-list **)** *type-parameter-constraints-clauses* **;**

interface-property-declaration ::= *type* **identifier** **{** *interface-accessors* **}** |
new *type* **identifier** **{** *interface-accessors* **}** |
attributes type **identifier** **{** *interface-accessors* **}** |
attributes **new** *type* **identifier** **{** *interface-accessors* **}**

interface-accessors ::= **get** **;** | *attributes* **get** **;** |
set **;** | *attributes* **set** **;** |
get **;** **set** **;** | **get** **;** *attributes* **set** **;** |
attributes **get** **;** **set** **;** | *attributes* **get** **;** *attributes* **set** **;** |
set **;** **get** **;** | **set** **;** *attributes* **get** **;** |
attributes **set** **;** **get** **;** | *attributes* **set** **;** *attributes* **get** **;**

interface-event-declaration ::= **event** *type* **identifier** **;** |
new event *type* **identifier** **;** |
attributes **event** *type* **identifier** **;** |
attributes **new event** *type* **identifier** **;**

interface-indexer-declaration ::= *type* **this** **[** *formal-parameter-list* **]** **{**
interface-accessors **}** |
new *type* **this** **[** *formal-parameter-list* **]** **{** *interface-accessors* **}** |
attributes type **this** **[** *formal-parameter-list* **]** **{** *interface-accessors* **}** |
attributes **new** *type* **this** **[** *formal-parameter-list* **]** **{** *interface-accessors* **}**

enum-declaration ::= **enum** **identifier** *enum-body* |
enum identifier *enum-body* **;** |
enum identifier *enum-base enum-body* |
enum identifier *enum-base enum-body* **;** |
enum-modifiers **enum identifier** *enum-body* |
enum-modifiers **enum identifier** *enum-body* **;** |
enum-modifiers **enum identifier** *enum-base enum-body* |
enum-modifiers **enum identifier** *enum-base enum-body* **;** |
attributes **enum identifier** *enum-body* |
attributes **enum identifier** *enum-body* **;** |
attributes **enum identifier** *enum-base enum-body* |
attributes **enum identifier** *enum-base enum-body* **;** |
attributes enum-modifiers **enum identifier** *enum-body* |
attributes enum-modifiers **enum identifier** *enum-body* **;** |
attributes enum-modifiers **enum identifier** *enum-base enum-body* |

attributes *enum-modifiers* **enum** **identifier** *enum-base* *enum-body* **;**

enum-base ::= **:** *integral-type*

enum-body ::= **{** *enum-member-declarations* **,** **}** |
{ **}** | **{** *enum-member-declarations* **}**

enum-modifiers ::= *enum-modifier* | *enum-modifiers* *enum-modifier*

enum-modifier ::= **new** | **public** | **protected** | **internal** | **private**

enum-member-declarations ::= *enum-member-declaration* |
enum-member-declarations **,** *enum-member-declaration*

enum-member-declaration ::= **identifier** | *attributes* **identifier** |
identifier **=** *constant-expression* | *attributes* **identifier** **=** *constant-expression*

delegate-declaration ::= **delegate** *return-type* **identifier** **(** **)** **;** |
delegate *return-type* **identifier** **(** **)** *type-parameter-constraints-clauses* **;** |
delegate *return-type* **identifier** **(** *formal-parameter-list* **)** **;** |
delegate *return-type* **identifier** **(** *formal-parameter-list* **)**
type-parameter-constraints-clauses **;** |
delegate *return-type* **identifier** *type-parameter-list* **(** **)** **;** |
delegate *return-type* **identifier** *type-parameter-list* **(** **)**
type-parameter-constraints-clauses **;** |
delegate *return-type* **identifier** *type-parameter-list* **(** *formal-parameter-list* **)** **;** |
delegate *return-type* **identifier** *type-parameter-list* **(** *formal-parameter-list* **)**
type-parameter-constraints-clauses **;** |
delegate-modifiers **delegate** *return-type* **identifier** **(** **)** **;** |
delegate-modifiers **delegate** *return-type* **identifier** **(** **)**
type-parameter-constraints-clauses **;** |
delegate-modifiers **delegate** *return-type* **identifier** **(** *formal-parameter-list* **)** **;** |
delegate-modifiers **delegate** *return-type* **identifier** **(** *formal-parameter-list* **)**
type-parameter-constraints-clauses **;** |
delegate-modifiers **delegate** *return-type* **identifier** *type-parameter-list* **(** **)** **;** |
delegate-modifiers **delegate** *return-type* **identifier** *type-parameter-list* **(** **)**
type-parameter-constraints-clauses **;** |
delegate-modifiers **delegate** *return-type* **identifier** *type-parameter-list* **(**
formal-parameter-list **)** **;** |
delegate-modifiers **delegate** *return-type* **identifier** *type-parameter-list* **(**
formal-parameter-list **)** *type-parameter-constraints-clauses* **;** |
attributes **delegate** *return-type* **identifier** **(** **)** **;** |
attributes **delegate** *return-type* **identifier** **(** **)**
type-parameter-constraints-clauses **;** |
attributes **delegate** *return-type* **identifier** **(** *formal-parameter-list* **)** **;** |

attributes **delegate** *return-type* **identifier** **(** *formal-parameter-list* **)**
type-parameter-constraints-clauses **;** |
attributes **delegate** *return-type* **identifier** *type-parameter-list* **(** **)** **;** |
attributes **delegate** *return-type* **identifier** *type-parameter-list* **(** **)**
type-parameter-constraints-clauses **;** |
attributes **delegate** *return-type* **identifier** *type-parameter-list* **(**
formal-parameter-list **)** **;** |
attributes **delegate** *return-type* **identifier** *type-parameter-list* **(**
formal-parameter-list **)** *type-parameter-constraints-clauses* **;** |
attributes *delegate-modifiers* **delegate** *return-type* **identifier** **(** **)** **;** |
attributes *delegate-modifiers* **delegate** *return-type* **identifier** **(** **)**
type-parameter-constraints-clauses **;** |
attributes *delegate-modifiers* **delegate** *return-type* **identifier** **(**
formal-parameter-list **)** **;** |
attributes *delegate-modifiers* **delegate** *return-type* **identifier** **(**
formal-parameter-list **)** *type-parameter-constraints-clauses* **;** |
attributes *delegate-modifiers* **delegate** *return-type* **identifier**
type-parameter-list **(** **)** **;** |
attributes *delegate-modifiers* **delegate** *return-type* **identifier**
type-parameter-list **(** **)** *type-parameter-constraints-clauses* **;** |
attributes *delegate-modifiers* **delegate** *return-type* **identifier**
type-parameter-list **(** *formal-parameter-list* **)** **;** |
attributes *delegate-modifiers* **delegate** *return-type* **identifier**
type-parameter-list **(** *formal-parameter-list* **)**
type-parameter-constraints-clauses **;**

delegate-modifiers ::= *delegate-modifier* | *delegate-modifiers delegate-modifier*

delegate-modifier ::= **new** | **public** | **protected** | **internal** | **private**

global-attributes ::= *global-attribute-sections*

global-attribute-sections ::= *global-attribute-section* |
global-attribute-sections global-attribute-section

global-attribute-section ::= **[** *global-attribute-target-specifier attribute-list* **]** |
[*global-attribute-target-specifier attribute-list* **,** **]**

global-attribute-target-specifier ::= *global-attribute-target* **:**

global-attribute-target ::= **identifier** | **abstract** | **as** | **base** | **bool** | **break** |
byte | **case** | **catch** | **char** | **checked** | **class** | **const** | **continue** | **decimal** |
default | **delegate** | **do** | **double** | **else** | **enum** | **event** | **explicit** | **extern** |
false | **finally** | **fixed** | **float** | **for** | **foreach** | **goto** | **if** | **implicit** | **in** |
int | **interface** | **internal** | **is** | **lock** | **long** | **namespace** | **new** | **null** |
object | **operator** | **out** | **override** | **params** | **private** | **protected** | **public** |

readonly | **ref** | **return** | **sbyte** | **sealed** | **short** | **sizeof** | **stackalloc** |
static | **string** | **struct** | **switch** | **this** | **throw** | **true** | **try** | **typeof** | **uint**
| **ulong** | **unchecked** | **unsafe** | **ushort** | **using** | **virtual** | **void**| **volatile** |
while

attributes ::= *attribute-sections*

attribute-sections ::= *attribute-section* | *attribute-sections attribute-section*;

attribute-section ::= **[** *attribute-list* **]** |
[*attribute-target-specifier attribute-list* **]** |
[*attribute-list* **,** **]** | **[** *attribute-target-specifier attribute-list* **,** **]**

attribute-target-specifier ::= *attribute-target* **:**

attribute-target ::= **identifier** | **abstract** | **as** | **base** | **bool** | **break** | **byte** |
case | **catch** | **char** | **checked** | **class** | **const** | **continue** | **decimal** | **default** |
delegate | **do** | **double** | **else** | **enum** | **event** | **explicit** | **extern** | **false** |
finally | **fixed** | **float** | **for** | **foreach** | **goto** | **if** | **implicit** | **in** | **int** |
interface | **internal** | **is** | **lock** | **long** | **namespace** | **new** | **null** | **object** |
operator | **out** | **override** | **params** | **private** | **protected** | **public** | **readonly** |
ref | **return** | **sbyte** | **sealed** | **short** | **sizeof** | **stackalloc** | **static** | **string** |
struct | **switch** | **this** | **throw** | **true** | **try** | **typeof** | **uint** | **ulong** |
unchecked | **unsafe** | **ushort** | **using** | **virtual** | **void** | **volatile** | **while**

attribute-list ::= *attribute* | *attribute-list* **,** *attribute*

attribute ::= *attribute-name* | *attribute-name attribute-arguments*

attribute-name ::= *type-name*

attribute-arguments ::= **(** *positional-argument-list* **,** *named-argument-list* **)** |
(*named-argument-list* **)** | **(** **)** | **(** *positional-argument-list* **)**

positional-argument-list ::= *positional-argument* |
positional-argument-list **,** *positional-argument*

positional-argument ::= *attribute-argument-expression*

named-argument-list ::= **identifier** **=** *attribute-argument-expression*

attribute-argument-expression ::= *expression*

type-parameter-list ::= **<** *type-parameters* **>**

type-parameters ::= *type-parameter* | *attributes type-parameter* |

```
    type-parameters , type-parameter |
    type-parameters , attributes type-parameter

type-parameter ::= identifier

type-argument-list ::= < type-arguments >

type-arguments ::= type-argument | type-arguments , type-argument

type-argument ::= type

type-parameter-constraints-clauses ::= type-parameter-constraints-clause |
    type-parameter-constraints-clauses type-parameter-constraints-clause

type-parameter-constraints-clause ::= where type-parameter :
    type-parameter-constraints

type-parameter-constraints ::= primary-constraint | secondary-constraints |
    constructor-constraint | primary-constraint , secondary-constraints |
    primary-constraint , constructor-constraint |
    secondary-constraints , constructor-constraint |
    primary-constraint , secondary-constraints , constructor-constraint

primary-constraint ::= class-type | class | struct

secondary-constraints ::= interface-type | type-parameter |
  secondary-constraints , interface-type |
  secondary-constraints , type-parameter

constructor-constraint ::= new ( )
```

Appendix B

C# 1.2 Language Specification

This appendix describes the C# 1.2 language specification used in 6.2. The first part of this appendix is the lexical specification. The second part is the GIFT-annotated parsing grammar.

B.1 Lexical specification

This lexical specification is based on the lexical syntax given in the appendix of the language specification [HCC02]. Tokens are described using EBNF rules. A string is in the pattern of the token if there is a derivation of the string in the grammar, whose start symbol is the non-terminal labelled as the token. For readability, non-token non-terminals will be listed in italics and token non-terminals will be in bold. Characters will be underlined. Additionally, $[\alpha]$ will be interpreted to mean one symbol in the sequence depicted by α.

identifier ::= (*letter-character* | _) *identifier-part-character** |
    @ (*letter-character* | _) *identifier-part-character**
integer-literal ::= [1-9]+ *integer-type-suffix*? |
    0 ((x|X) [0-9a-fA-F]+)? *integer-type-suffix*?
real-literal ::= [0-9]* . [0-9]+ *exponent-part*? *real-type-suffix*? |
    [0-9]+ (*exponent-part real-type-suffix*? | *real-type-suffix*)
character-literal ::= ' *character* '
string-literal ::= " *regular-string-literal-character* " |
    @ " (*single-verbatim-string-literal* | " ")* "
null-literal ::= null
letter-character := [a-zA-Z_]
identifier-part-character ::= [a-zA-Z0-9_] | *unicode-character-escape-sequence*
unicode-character-escape-sequence ::= \ (u [0-9a-fA-F] [0-9a-fA-F] [0-9a-fA-F] [0-
    9a-fA-F] | U [0-9a-fA-F] [0-9a-fA-F] [0-9a-fA-F] [0-9a-fA-F] [0-9a-fA-F] [0-
    9a-fA-F] [0-9a-fA-F] [0-9a-fA-F])

```
integer-type-suffix ::= (u|U) (l|L)? | (l|L) (u|U)?
exponent-part ::= (e|E) (+|-)? [0-9]+
real-type-suffix ::= f|F|d|D|m|M
character ::= single-character | simple-escape-sequence |
    hexadecimal-escape-sequence | unicode-character-escape-sequence
single-character ::= any unicode character except \ ' \n \r
simple-escape-sequence ::= \ ('|"|\|0|a|b|f|n|r|t|t|v
hexadecimal-escape-sequence ::= \ x [0-9a-fA-F] [0-9a-fA-F]? [0-9a-fA-F]? [0-9a-
    fA-F]?
regular-string-literal-character ::= single-regular-string-literal-character |
    simple-escape-sequence | hexadecimal-escape-sequence |
    unicode-character-escape-sequence
single-regular-string-literal-character ::= any unicode character except " \ \n
    \r
single-verbatim-string-literal-character ::= any unicode character except "
```

Additionally there are three layout tokens in the lexical specification

```
new-line ::= \r \n? | \n
whitespace ::= ␣ | \t | \v | \f
comment ::= // input-character* | /* (not-asterisk | *+ not-slash)* *+ /
input-character ::= any unicode character except \n \r
not-slash ::= any unicode character except /
not-asterisk ::= any unicode character except *
```

As with many languages, C# has a number of tokens whose patterns are the same as their label - known as the keywords. The following list of such tokens are *reserved*, that is these patterns should be recognised as matching only these tokens.

```
{ } [ ] ( ) . , : ; + - * / % & | ^ ! ~ = < >
? ++ -- && || << >> == != <= >= += -= *= /= %= &=
|= ^= <<= >>= -> abstract as base bool break byte case
catch char checked class const continue decimal default
delegate double do else enum event explicit extern
false finally fixed float foreach for goto if implicit interface
internal int in is lock long namespace new object operator
out override params private protected public readonly ref
return sbyte sealed short sizeof stackalloc static string
struct switch this throw true try typeof uint ulong unchecked
unsafe ushort using virtual void volatile while
```

There are also keywords that are not reserved - known as the *contextual* keywords.

The patterns for these tokens can also be matched by other tokens

add **assembly** **field** **get** **method** **module** **param** **property** **remove** **set** **type**

B.2 GIFT-annotated Parsing Grammar

The parsing grammar is based on the grammar given in the appendix of [HCC02]. This grammar is given as a set of BNF grammar rules. Additionally, the grammar is annotated with GIFT operators designed to transform the trees generated by the parser of this grammar into trees in the abstract syntax given in Appendix C. The terminals of the grammar are in bold. The start symbol of this grammar is the non-terminal *compilation-unit*.

namespace-name ::= (*namespace-or-type-name*^)!*qualified-identifier*

type-name ::= *namespace-or-type-name*^

namespace-or-type-name ::=
    identifier |
    (*namespace-or-type-name*^)!*qualified-identifier* . **identifier**

type ::= *value-type*^ | *reference-type*^

value-type ::= *struct-type*^ | *enum-type*^

struct-type ::= (*type-name*^)!*qualified-identifier* |
    simple-type^

simple-type ::= *numeric-type*^ | **bool**!*predefined-type*

numeric-type ::=
    integral-type!*predefined-type* |
    floating-point-type^ |
    decimal!*predefined-type*

integral-type ::= **sbyte** | **byte** | **short** | **ushort** | **int** | **uint** | **long** | **ulong** | **char**

floating-point-type ::= **float**!*predefined-type* | **double**!*predefined-type*

enum-type ::= (*type-name*^)!*qualified-identifier*

reference-type ::= *class-type*^ | (*interface-type*^)!*qualified-identifier* |
    array-type | *delegate-type*^

```
class-type ::=
   (type-name^)!qualified-identifier |
   object!predefined-type |
   string!predefined-type

interface-type ::= type-name^

delegate-type ::= (type-name^)!qualified-identifier

variable-reference ::= expression

argument-list ::= argument | argument-list^ , argument

argument ::=
   expression |
   ref variable-reference^ |
   out variable-reference^

primary-expression ::=
   primary-no-array-creation-expression^ |
   array-creation-expression^

primary-no-array-creation-expression ::=
   literal |
   simple-name^ |
   parenthesized-expression^ |
   member-access^ |
   invocation-expression^ |
   element-access^ |
   this-access^ |
   base-access^ |
   post-increment-expression^ |
   post-decrement-expression^ |
   object-creation-expression^ |
   delegate-creation-expression^ |
   typeof-expression^ |
   checked-expression^ |
   unchecked-expression^

simple-name ::= identifier

parenthesized-expression ::= ( expression )

member-access ::= (primary-expression^)!expression . identifier |
   predefined-type . identifier
```

```
predefined-type ::= bool | byte!integral-type | char!integral-type | decimal |
    double | float | int!integral-type | long!integral-type | object | sbyte!
    integral-type | short!integral-type | string | uint!integral-type | ulong!
    integral-type | ushort!integral-type

invocation-expression ::=
   (primary-expression^)!expression ( argument-list ) |
   (primary-expression^)!expression ( )

element-access ::= (primary-no-array-creation-expression^)!expression [
    expression-list ]

expression-list ::= expression | expression-list^ , expression

this-access ::= this

base-access ::= base . identifier | base [ expression-list ]

post-increment-expression ::= (primary-expression^)!expression ++!
    unary-assignment-operator

post-decrement-expression ::= (primary-expression^)!expression --!
    unary-assignment-operator

object-creation-expression ::= new type ( argument-list ) | new type ( )

array-creation-expression ::=
   new non-array-type^ [ expression-list ] rank-specifiers^ array-initializer |
   new non-array-type^ [ expression-list ] |
   new non-array-type^ [ expression-list ] rank-specifiers^ |
   new non-array-type^ [ expression-list ] array-initializer |
   new array-type array-initializer

delegate-creation-expression ::= new (delegate-type^)!type ( expression!argument )

typeof-expression ::=
   typeof ( type!return-type ) |
   typeof ( void!return-type )

checked-expression ::= checked ( expression )

unchecked-expression ::= unchecked ( expression )

unary-expression ::=
   primary-expression^ |
```

```
    +!unary-operator (unary-expression^)!expression |
    -!unary-operator (unary-expression^)!expression |
    !!unary-operator (unary-expression^)!expression |
    ~!unary-operator (unary-expression^)!expression |
    pre-increment-expression^ |
    pre-decrement-expression^ |
    cast-expression^

pre-increment-expression ::= ++!unary-assignment-operator (unary-expression^)!
     expression

pre-decrement-expression ::= --!unary-assignment-operator (unary-expression^)!
     expression

cast-expression ::= ( type ) (unary-expression^)!expression

multiplicative-expression ::= unary-expression^ |
    (multiplicative-expression^)!expression (*!overloadable-binary-operator)!
        binary-operator (unary-expression^)!expression |
    (multiplicative-expression^)!expression (/!overloadable-binary-operator)!
        binary-operator (unary-expression^)!expression |
    (multiplicative-expression^)!expression (%!overloadable-binary-operator)!
        binary-operator (unary-expression^)!expression

additive-expression ::= multiplicative-expression^ |
    (additive-expression^)!expression (+!overloadable-binary-operator)!
        binary-operator (multiplicative-expression^)!expression |
    (additive-expression^)!expression (-!overloadable-binary-operator)!
        binary-operator (multiplicative-expression^)!expression

shift-expression ::= additive-expression^ | (shift-expression^)!expression (<<!
     overloadable-binary-operator)!binary-operator (additive-expression^)!expression |
      (shift-expression^)!expression (>>!overloadable-binary-operator)!
     binary-operator (additive-expression^)!expression

relational-expression ::= shift-expression^ |
    (relational-expression^)!expression (<!overloadable-binary-operator)!
        binary-operator (shift-expression^)!expression |
    (relational-expression^)!expression (>!overloadable-binary-operator)!
        binary-operator (shift-expression^)!expression |
    (relational-expression^)!expression (<=!overloadable-binary-operator)!
        binary-operator (shift-expression^)!expression |
    (relational-expression^)!expression (>=!overloadable-binary-operator)!
        binary-operator (shift-expression^)!expression |
    (relational-expression^)!expression is type |
    (relational-expression^)!expression as type
```

```
equality-expression ::= relational-expression^ |
    (equality-expression^)!expression (==!overloadable-binary-operator)!
        binary-operator (relational-expression^)!expression |
    (equality-expression^)!expression (!=!overloadable-binary-operator)!
        binary-operator (relational-expression^)!expression

and-expression ::= equality-expression^ |
    (and-expression^)!expression (&!overloadable-binary-operator)!binary-operator (
        equality-expression^)!expression

exclusive-or-expression ::= and-expression^ |
    (exclusive-or-expression^)!expression (^!overloadable-binary-operator)!
        binary-operator (and-expression^)!expression

inclusive-or-expression ::=
    exclusive-or-expression^ |
    (inclusive-or-expression^)!expression (|!overloadable-binary-operator)!
        binary-operator (exclusive-or-expression^)!expression

conditional-and-expression ::= inclusive-or-expression^ |
    (conditional-and-expression^)!expression &&!binary-operator
         (inclusive-or-expression^)!expression

conditional-or-expression ::= conditional-and-expression^ |
    (conditional-or-expression^)!expression ||!binary-operator
         (conditional-and-expression^)!expression

conditional-expression ::=
    conditional-or-expression^ |
    (conditional-or-expression^)!expression ? expression : expression

assignment ::= (unary-expression^)!expression assignment-operator expression

assignment-operator ::= = | += | -= | *= | /= | %= | &= | |= | ^= | <<= | >>=

expression ::= conditional-expression^ | assignment^

constant-expression ::= expression

boolean-expression ::= expression

statement ::=
    labeled-statement^ |
    declaration-statement |
    embedded-statement
```

```
embedded-statement ::=
   block |
   empty-statement^ |
   expression-statement^ |
   selection-statement^ |
   iteration-statement^ |
   jump-statement^ |
   try-statement^ |
   checked-statement^ |
   unchecked-statement^ |
   lock-statement^ |
   using-statement^

block ::= { statement-list^ } | { }

statement-list ::= statement | statement-list^ statement

empty-statement ::= ;

labeled-statement ::= identifier : statement

declaration-statement ::= local-variable-declaration ; | local-constant-declaration ;

local-variable-declaration ::= type local-variable-declarators^

local-variable-declarators ::= local-variable-declarator^ |
   local-variable-declarators^ , local-variable-declarator^

local-variable-declarator ::=
   identifier!variable-declarator |
   (identifier = local-variable-initializer^)!variable-declarator

local-variable-initializer ::=
   expression!variable-initializer |
   array-initializer!variable-initializer

local-constant-declaration ::= const type constant-declarators

constant-declarators ::= constant-declarator | constant-declarators^ ,
    constant-declarator

constant-declarator ::= identifier = constant-expression^

expression-statement ::= statement-expression^ ;
```

```
statement-expression ::=
    (invocation-expression^)!expression |
    (object-creation-expression^)!expression |
    (assignment^)!expression |
    (post-increment-expression^)!expression |
    (post-decrement-expression^)!expression |
    (pre-increment-expression^)!expression |
    (pre-decrement-expression^)!expression

selection-statement ::= if-statement^ | switch-statement^

if-statement ::=
    if ( boolean-expression^ ) embedded-statement |
    if ( boolean-expression^ ) embedded-statement else embedded-statement

switch-statement ::= switch ( expression ) switch-block^

switch-block ::= { } | { switch-sections^ }

switch-sections ::= switch-section | switch-sections^ switch-section

switch-section ::= switch-labels^ statement-list^

switch-labels ::= switch-label | switch-labels^ switch-label

switch-label ::= case constant-expression^ : | default :

iteration-statement ::=
    while-statement^ |
    do-statement^ |
    for-statement^ |
    foreach-statement^

while-statement ::=
    while ( boolean-expression^ ) embedded-statement

do-statement ::=
    do embedded-statement while ( boolean-expression^ ) ;

for-statement ::=
    for ( ; ; ) embedded-statement |
    for ( ; ; for-iterator^ ) embedded-statement |
    for ( ; for-condition^ ; ) embedded-statement |
    for ( ; for-condition^ ; for-iterator^ ) embedded-statement |
    for ( for-initializer ; ; ) embedded-statement |
    for ( for-initializer ; ; for-iterator^ ) embedded-statement |
```

```
    for ( for-initializer ; for-condition^ ; ) embedded-statement |
    for ( for-initializer ; for-condition^ ; for-iterator^ ) embedded-statement

for-initializer ::= local-variable-declaration |
    (statement-expression-list^)!expression-list

for-condition ::= boolean-expression^

for-iterator ::=
    (statement-expression-list^)!expression-list

statement-expression-list ::= statement-expression^ |
    statement-expression-list^ , statement-expression^

foreach-statement ::=
    foreach ( type identifier in expression ) embedded-statement

jump-statement ::= break-statement^ | continue-statement^ | goto-statement^ |
     return-statement^ | throw-statement^

break-statement ::= break ;

continue-statement ::= continue ;

goto-statement ::=
    goto identifier ; |
    goto case constant-expression^ ; |
    goto default ;

return-statement ::= return ; | return expression ;

throw-statement ::= throw ; | throw expression ;

try-statement ::=
    try block catch-clauses |
    try block finally-clause^ |
    try block catch-clauses finally-clause^

catch-clauses ::=
    specific-catch-clauses^ |
    general-catch-clause |
    specific-catch-clauses^ general-catch-clause

specific-catch-clauses ::= specific-catch-clause | specific-catch-clauses^
     specific-catch-clause
```

specific-catch-clause ::= **catch** **(** (*class-type*^)!*type* **)** *block* |
catch **(** (*class-type*^)!*type* **identifier** **)** *block*

general-catch-clause ::= **catch** *block*

finally-clause ::= **finally** *block*

checked-statement ::= **checked** *block*

unchecked-statement ::= **unchecked** *block*

lock-statement ::= **lock** **(** *expression* **)** *embedded-statement*

using-statement ::= **using** **(** *resource-acquisition* **)** *embedded-statement*

resource-acquisition ::= *local-variable-declaration* | *expression*

compilation-unit ::=
ϵ |
namespace-member-declarations^ |
global-attributes^ |
global-attributes^ *namespace-member-declarations*^ |
using-directives^ |
using-directives^ *namespace-member-declarations*^ |
using-directives^ *global-attributes*^ |
using-directives^ *global-attributes*^ *namespace-member-declarations*^

namespace-declaration ::=
namespace *qualified-identifier* *namespace-body*^ |
namespace *qualified-identifier* *namespace-body*^ **;**

qualified-identifier ::= **identifier** | *qualified-identifier*^ **.** **identifier**

namespace-body ::=
{ **}** |
{ *namespace-member-declarations*^ **}** |
{ *using-directives*^ **}** |
{ *using-directives*^ *namespace-member-declarations*^ **}**

using-directives ::= *using-directive* | *using-directives*^ *using-directive*

using-directive ::= *using-alias-directive*^ | *using-namespace-directive*^

using-alias-directive ::=
using **identifier** **=** (*namespace-or-type-name*^)!*qualified-identifier* **;**

using-namespace-directive ::= **using** *namespace-name*ˆ **;**

namespace-member-declarations ::= *namespace-member-declaration* |
*namespace-member-declarations*ˆ *namespace-member-declaration*

namespace-member-declaration ::= *namespace-declaration*ˆ | *type-declaration*

type-declaration ::=
class-declaration |
struct-declaration |
interface-declaration |
enum-declaration |
delegate-declaration

class-declaration ::=
class identifier *class-body*ˆ |
class identifier *class-body*ˆ **;** |
class identifier *class-base class-body*ˆ |
class identifier *class-base class-body*ˆ **;** |
*class-modifiers*ˆ **class identifier** *class-body*ˆ |
*class-modifiers*ˆ **class identifier** *class-body*ˆ **;** |
*class-modifiers*ˆ **class identifier** *class-base class-body*ˆ |
*class-modifiers*ˆ **class identifier** *class-base class-body*ˆ **;** |
*attributes*ˆ **class identifier** *class-body*ˆ |
*attributes*ˆ **class identifier** *class-body*ˆ **;** |
*attributes*ˆ **class identifier** *class-base class-body*ˆ |
*attributes*ˆ **class identifier** *class-base class-body*ˆ **;** |
*attributes*ˆ *class-modifiers*ˆ **class identifier** *class-body*ˆ |
*attributes*ˆ *class-modifiers*ˆ **class identifier** *class-body*ˆ **;** |
*attributes*ˆ *class-modifiers*ˆ **class identifier** *class-base class-body*ˆ |
*attributes*ˆ *class-modifiers*ˆ **class identifier** *class-base class-body*ˆ **;**

class-modifiers ::= *class-modifier*ˆ | *class-modifiers*ˆ *class-modifier*ˆ

class-modifier ::= **new**!*modifier* | **public**!*modifier* | **protected**!*modifier* | **internal**!
modifier | **private**!*modifier* | **abstract**!*modifier* | **sealed**!*modifier*

class-base ::= **:** *class-type*ˆ | **:** *interface-type-list*ˆ | **:** *class-type*ˆ **,**
*interface-type-list*ˆ

interface-type-list ::= (*interface-type*ˆ)!*qualified-identifier* |
*interface-type-list*ˆ **,** (*interface-type*ˆ)!*qualified-identifier*

class-body ::= **{ }** | **{** *class-member-declarations*ˆ **}**

class-member-declarations ::= *class-member-declaration*ˆ |

```
    class-member-declarations^ class-member-declaration^

class-member-declaration ::=
    constant-declaration!member-declaration |
    field-declaration!member-declaration |
    method-declaration!member-declaration |
    property-declaration!member-declaration |
    event-declaration!member-declaration |
    indexer-declaration!member-declaration |
    operator-declaration!member-declaration |
    constructor-declaration!member-declaration |
    destructor-declaration!member-declaration |
    static-constructor-declaration!member-declaration |
    type-declaration!member-declaration

constant-declaration ::=
    const type constant-declarators ; |
    constant-modifiers^ const type constant-declarators ; |
    attributes^ const type constant-declarators ; |
    attributes^ constant-modifiers^ const type constant-declarators ;

constant-modifiers ::= constant-modifier^ | constant-modifiers^ constant-modifier^

constant-modifier ::=
    new!modifier |
    public!modifier |
    protected!modifier |
    internal!modifier |
    private!modifier

field-declaration ::=
    type variable-declarators ; |
    field-modifiers^ type variable-declarators ; |
    attributes^ type variable-declarators ; |
    attributes^ field-modifiers^ type variable-declarators ;

field-modifiers ::= field-modifier^ | field-modifiers^ field-modifier^

field-modifier ::=
    new!modifier |
    public!modifier |
    protected!modifier |
    internal!modifier |
    private!modifier |
    static!modifier |
    readonly!modifier |
```

```
    volatile!modifier

variable-declarators ::= variable-declarator |
    variable-declarators^ , variable-declarator

variable-declarator ::= identifier | identifier = variable-initializer

variable-initializer ::= expression | array-initializer

method-declaration ::= method-header^ method-body^

method-header ::=
    return-type member-name^ ( ) |
    return-type member-name^ ( formal-parameter-list ) |
    method-modifiers^ return-type member-name^ ( ) |
    method-modifiers^ return-type member-name^ ( formal-parameter-list ) |
    attributes^ return-type member-name^ ( ) |
    attributes^ return-type member-name^ ( formal-parameter-list ) |
    attributes^ method-modifiers^ return-type member-name^ ( ) |
    attributes^ method-modifiers^ return-type member-name^ ( formal-parameter-list )

method-modifiers ::= method-modifier^ | method-modifiers^ method-modifier^

method-modifier ::= new!modifier | public!modifier | protected!modifier | internal!
     modifier | private!modifier | static!modifier | virtual!modifier | sealed!
     modifier | override!modifier | abstract!modifier | extern!modifier

return-type ::= type | void

member-name ::= identifier!qualified-identifier |
    (interface-type^ . identifier)!qualified-identifier

method-body ::= block!body | ;!body

formal-parameter-list ::= fixed-parameters^ |
    fixed-parameters^ , parameter-array | parameter-array

fixed-parameters ::= fixed-parameter | fixed-parameters^ , fixed-parameter

fixed-parameter ::=
    type identifier |
    parameter-modifier^ type identifier |
    attributes^ type identifier |
    attributes^ parameter-modifier^ type identifier

parameter-modifier ::= ref | out
```

```
parameter-array ::= params array-type identifier |
    attributes^ params array-type identifier

property-declaration ::=
    type member-name^ { accessor-declarations } |
    property-modifiers^ type member-name^ { accessor-declarations } |
    attributes^ type member-name^ { accessor-declarations } |
    attributes^ property-modifiers^ type member-name^ { accessor-declarations }

property-modifiers ::= property-modifier^ | property-modifiers^ property-modifier^

property-modifier ::=
    new!modifier |
    public!modifier |
    protected!modifier |
    internal!modifier |
    private!modifier |
    static!modifier |
    virtual!modifier |
    sealed!modifier |
    override!modifier |
    abstract!modifier |
    extern!modifier

accessor-declarations ::=
    get-accessor-declaration^ |
    get-accessor-declaration^ set-accessor-declaration^ |
    set-accessor-declaration^ |
    set-accessor-declaration^ get-accessor-declaration^

get-accessor-declaration ::=
    get accessor-body^ |
    attributes^ get accessor-body^

set-accessor-declaration ::=
    set accessor-body^ |
    attributes^ set accessor-body^

accessor-body ::= block!body | ;!body

event-declaration ::=
    event (type variable-declarators)!local-variable-declaration ; |
    event-modifiers^ event (type variable-declarators)!local-variable-declaration ; |
    attributes^ event (type variable-declarators)!local-variable-declaration ; |
```

```
    attributes^ event-modifiers^ event (type variable-declarators)!
        local-variable-declaration ; |
    event type member-name^ { event-accessor-declarations } |
    event-modifiers^ event type member-name^ { event-accessor-declarations } |
    attributes^ event type member-name^ { event-accessor-declarations } |
    attributes^ event-modifiers^ event type member-name^ {
        event-accessor-declarations }

event-modifiers ::=
    event-modifier^ |
    event-modifiers^ event-modifier^

event-modifier ::=
    new!modifier |
    public!modifier |
    protected!modifier |
    internal!modifier |
    private!modifier |
    static!modifier |
    virtual!modifier |
    sealed!modifier |
    override!modifier |
    abstract!modifier |
    extern!modifier

event-accessor-declarations ::=
    add-accessor-declaration remove-accessor-declaration |
    remove-accessor-declaration add-accessor-declaration

add-accessor-declaration ::=
    add block |
    attributes add block

remove-accessor-declaration ::=
    remove block |
    attributes^ remove block

indexer-declaration ::=
    indexer-declarator { accessor-declarations } |
    indexer-modifiers^ indexer-declarator { accessor-declarations } |
    attributes^ indexer-declarator { accessor-declarations } |
    attributes^ indexer-modifiers^ indexer-declarator { accessor-declarations }

indexer-modifiers ::=
    indexer-modifier^ |
    indexer-modifiers^ indexer-modifier^
```

```
indexer-modifier ::= new!modifier | public!modifier | protected!modifier | internal!
    modifier | private!modifier | virtual!modifier | sealed!modifier | override!
    modifier | abstract!modifier | extern!modifier

indexer-declarator ::=
   type this [ formal-parameter-list ] |
   type (interface-type^)!qualified-identifier . this [ formal-parameter-list ]

operator-declaration ::=
   operator-modifiers^ operator-declarator operator-body^ |
   attributes^ operator-modifiers^ operator-declarator operator-body^

operator-modifiers ::=
   operator-modifier^ |
   operator-modifiers^ operator-modifier^

operator-modifier ::=
   public!modifier |
   static!modifier |
   extern!modifier

operator-declarator ::=
   unary-operator-declarator^ |
   binary-operator-declarator^ |
   conversion-operator-declarator^

unary-operator-declarator ::=
   type operator overloadable-unary-operator ( type identifier )

overloadable-unary-operator ::= +!unary-operator | -!unary-operator | !!
    unary-operator | ~!unary-operator | ++!unary-assignment-operator | --!
    unary-assignment-operator | true | false

binary-operator-declarator ::=
   type operator overloadable-binary-operator ( type identifier , type identifier )

overloadable-binary-operator ::= + | - | * | / | % | & | | | ^ | << | >> | == |
    != | > | < | >= | <=

conversion-operator-declarator ::= implicit operator type ( type identifier ) |
   explicit operator type ( type identifier )

operator-body ::=
   block!body |
   ;!body
```

```
constructor-declaration ::=
   constructor-declarator^ constructor-body^ |
   constructor-modifiers^ constructor-declarator^ constructor-body^ |
   attributes^ constructor-declarator^ constructor-body^ |
   attributes^ constructor-modifiers^ constructor-declarator^ constructor-body^

constructor-modifiers ::=
   constructor-modifier^ |
   constructor-modifiers^ constructor-modifier^

constructor-modifier ::= public!modifier | protected!modifier | internal!modifier |
    private!modifier | extern!modifier

constructor-declarator ::=
   identifier ( ) |
   identifier ( ) constructor-initializer |
   identifier ( formal-parameter-list ) |
   identifier ( formal-parameter-list ) constructor-initializer

constructor-initializer ::=
   : base!constructor-order ( ) |
   : base!constructor-order ( argument-list ) |
   : this!constructor-order ( ) |
   : this!constructor-order ( argument-list )

constructor-body ::=
   block!body |
   ;!body

static-constructor-declaration ::=
   static-constructor-modifiers identifier ( ) static-constructor-body^ |
   attributes^ static-constructor-modifiers identifier ( ) static-constructor-body^

static-constructor-modifiers ::=
   static |
   extern static |
   static extern

static-constructor-body ::= block!body | ;!body

destructor-declaration ::=
   ~ identifier ( ) destructor-body^ |
   extern ~ identifier ( ) destructor-body^ |
   attributes^ ~ identifier ( ) destructor-body^ |
   attributes^ extern ~ identifier ( ) destructor-body^
```

```
destructor-body ::=
    block!body |
    ;!body

struct-declaration ::= struct identifier struct-body^ |
    struct identifier struct-body^ ; |
    struct identifier (struct-interfaces^)!interface-base struct-body^ |
    struct identifier (struct-interfaces^)!interface-base struct-body^ ; |
    struct-modifiers^ struct identifier struct-body^ |
    struct-modifiers^ struct identifier struct-body^ ; |
    struct-modifiers^ struct identifier (struct-interfaces^)!interface-base
        struct-body^ |
    struct-modifiers^ struct identifier (struct-interfaces^)!interface-base
        struct-body^ ; |
    attributes^ struct identifier struct-body^ |
    attributes^ struct identifier struct-body^ ; |
    attributes^ struct identifier (struct-interfaces^)!interface-base struct-body^ |
    attributes^ struct identifier (struct-interfaces^)!interface-base struct-body^ ;
        |
    attributes^ struct-modifiers^ struct identifier struct-body^ |
    attributes^ struct-modifiers^ struct identifier struct-body^ ; |
    attributes^ struct-modifiers^ struct identifier (struct-interfaces^)!
        interface-base struct-body^ |
    attributes^ struct-modifiers^ struct identifier (struct-interfaces^)!
        interface-base struct-body^ ;

struct-modifiers ::=
    struct-modifier^ |
    struct-modifiers^ struct-modifier^

struct-modifier ::=
    new!modifier |
    public!modifier |
    protected!modifier |
    internal!modifier |
    private!modifier

struct-interfaces ::= : interface-type-list

struct-body ::=
    { } |
    { struct-member-declarations^ }

struct-member-declarations ::=
    struct-member-declaration^ |
```

*struct-member-declarations*ˆ *struct-member-declaration*ˆ

struct-member-declaration ::=
constant-declaration!*member-declaration* |
field-declaration!*member-declaration* |
method-declaration!*member-declaration* |
property-declaration!*member-declaration* |
event-declaration!*member-declaration* |
indexer-declaration!*member-declaration* |
operator-declaration!*member-declaration* |
constructor-declaration!*member-declaration* |
static-constructor-declaration!*member-declaration* |
type-declaration!*member-declaration*

array-type ::= *non-array-type*ˆ *rank-specifiers*ˆ

non-array-type ::= *type*

rank-specifiers ::= *rank-specifier* | *rank-specifiers*ˆ *rank-specifier*

rank-specifier ::= **[ ]** | **[** *dim-separators*ˆ **]**

dim-separators ::= **,** | *dim-separators*ˆ **,**

array-initializer ::=
{ *variable-initializer-list* **,** **}** |
{ } |
{ *variable-initializer-list* **}**

variable-initializer-list ::= *variable-initializer* |
*variable-initializer-list*ˆ **,** *variable-initializer*

interface-declaration ::=
interface identifier *interface-body*ˆ |
interface identifier *interface-body*ˆ **;** |
interface identifier *interface-base interface-body*ˆ |
interface identifier *interface-base interface-body*ˆ **;** |
*interface-modifiers*ˆ **interface identifier** *interface-body*ˆ |
*interface-modifiers*ˆ **interface identifier** *interface-body*ˆ **;** |
*interface-modifiers*ˆ **interface identifier** *interface-base interface-body*ˆ |
*interface-modifiers*ˆ **interface identifier** *interface-base interface-body*ˆ **;** |
*attributes*ˆ **interface identifier** *interface-body*ˆ |
*attributes*ˆ **interface identifier** *interface-body*ˆ **;** |
*attributes*ˆ **interface identifier** *interface-base interface-body*ˆ |
*attributes*ˆ **interface identifier** *interface-base interface-body*ˆ **;** |
*attributes*ˆ *interface-modifiers*ˆ **interface identifier** *interface-body*ˆ |

```
    attributes^ interface-modifiers^ interface identifier interface-body^ ; |
    attributes^ interface-modifiers^ interface identifier interface-base
        interface-body^ |
    attributes^ interface-modifiers^ interface identifier interface-base
        interface-body^ ;

interface-modifiers ::= interface-modifier^ | interface-modifiers^
     interface-modifier^

interface-modifier ::=
    new!modifier |
    public!modifier |
    protected!modifier |
    internal!modifier |
    private!modifier

interface-base ::= : interface-type-list

interface-body ::=
    { } |
    { interface-member-declarations^ }

interface-member-declarations ::=
    interface-member-declaration |
    interface-member-declarations^ interface-member-declaration

interface-member-declaration ::=
    interface-method-declaration^ |
    interface-property-declaration^ |
    interface-event-declaration^ |
    interface-indexer-declaration^

interface-method-declaration ::=
    return-type identifier ( ) ; |
    return-type identifier ( formal-parameter-list ) ; |
    new return-type identifier ( ) ; |
    new return-type identifier ( formal-parameter-list ) ; |
    attributes^ return-type identifier ( ) ; |
    attributes^ return-type identifier ( formal-parameter-list ) ; |
    attributes^ new return-type identifier ( ) ; |
    attributes^ new return-type identifier ( formal-parameter-list ) ;

interface-property-declaration ::=
    type identifier { interface-accessors } |
    new type identifier { interface-accessors } |
    attributes^ type identifier { interface-accessors } |
```

*attributes*ˆ **new** *type* **identifier** **{** *interface-accessors* **}**

interface-accessors ::=
get **;** |
*attributes*ˆ **get** **;** |
set **;** |
*attributes*ˆ **set** **;** |
get **;** **set** **;** |
get **;** *attributes*ˆ **set** **;** |
*attributes*ˆ **get** **;** **set** **;** |
*attributes*ˆ **get** **;** *attributes*ˆ **set** **;** |
set **;** **get** **;** |
set **;** *attributes*ˆ **get** **;** |
*attributes*ˆ **set** **;** **get** **;** |
*attributes*ˆ **set** **;** *attributes*ˆ **get** **;**

interface-event-declaration ::=
event *type* **identifier** **;** |
new **event** *type* **identifier** **;** |
*attributes*ˆ **event** *type* **identifier** **;** |
*attributes*ˆ **new** **event** *type* **identifier** **;**

interface-indexer-declaration ::=
type **this** **[** *formal-parameter-list* **]** **{** *interface-accessors* **}** |
new *type* **this** **[** *formal-parameter-list* **]** **{** *interface-accessors* **}** |
*attributes*ˆ *type* **this** **[** *formal-parameter-list* **]** **{** *interface-accessors* **}** |
*attributes*ˆ **new** *type* **this** **[** *formal-parameter-list* **]** **{** *interface-accessors* **}**

enum-declaration ::=
enum **identifier** *enum-body*ˆ |
enum **identifier** *enum-body*ˆ **;** |
enum **identifier** *enum-base* *enum-body*ˆ |
enum **identifier** *enum-base* *enum-body*ˆ **;** |
*enum-modifiers*ˆ **enum** **identifier** *enum-body*ˆ |
*enum-modifiers*ˆ **enum** **identifier** *enum-body*ˆ **;** |
*enum-modifiers*ˆ **enum** **identifier** *enum-base* *enum-body*ˆ |
*enum-modifiers*ˆ **enum** **identifier** *enum-base* *enum-body*ˆ **;** |
*attributes*ˆ **enum** **identifier** *enum-body*ˆ |
*attributes*ˆ **enum** **identifier** *enum-body*ˆ **;** |
*attributes*ˆ **enum** **identifier** *enum-base* *enum-body*ˆ |
*attributes*ˆ **enum** **identifier** *enum-base* *enum-body*ˆ **;** |
*attributes*ˆ *enum-modifiers*ˆ **enum** **identifier** *enum-body*ˆ |
*attributes*ˆ *enum-modifiers*ˆ **enum** **identifier** *enum-body*ˆ **;** |
*attributes*ˆ *enum-modifiers*ˆ **enum** **identifier** *enum-base* *enum-body*ˆ |
*attributes*ˆ *enum-modifiers*ˆ **enum** **identifier** *enum-base* *enum-body*ˆ **;**

enum-base ::= **:** *integral-type*

enum-body ::=
    { *enum-member-declarations* **,** **}** |
    { **}** |
    { *enum-member-declarations* **}**

enum-modifiers ::= *enum-modifier*ˆ | *enum-modifiers*ˆ *enum-modifier*ˆ

enum-modifier ::=
    new!*modifier* |
    public!*modifier* |
    protected!*modifier* |
    internal!*modifier* |
    private!*modifier*

enum-member-declarations ::=
    enum-member-declaration |
    *enum-member-declarations*ˆ **,** *enum-member-declaration*

enum-member-declaration ::=
    identifier |
    *attributes*ˆ **identifier** |
    identifier **=** *constant-expression*ˆ |
    *attributes*ˆ **identifier** **=** *constant-expression*ˆ

delegate-declaration ::=
    delegate *return-type* **identifier** **(** **)** **;** |
    delegate *return-type* **identifier** **(** *formal-parameter-list* **)** **;** |
    *delegate-modifiers*ˆ **delegate** *return-type* **identifier** **(** **)** **;** |
    *delegate-modifiers*ˆ **delegate** *return-type* **identifier** **(** *formal-parameter-list* **)** **;** |
    *attributes*ˆ **delegate** *return-type* **identifier** **(** **)** **;** |
    *attributes*ˆ **delegate** *return-type* **identifier** **(** *formal-parameter-list* **)** **;** |
    *attributes*ˆ *delegate-modifiers*ˆ **delegate** *return-type* **identifier** **(** **)** **;** |
    *attributes*ˆ *delegate-modifiers*ˆ **delegate** *return-type* **identifier** **(**
        formal-parameter-list **)** **;**

delegate-modifiers ::= *delegate-modifier*ˆ | *delegate-modifiers*ˆ *delegate-modifier*ˆ

delegate-modifier ::=
    new!*modifier* |
    public!*modifier* |
    protected!*modifier* |
    internal!*modifier* |
    private!*modifier*

global-attributes ::= *global-attribute-sections*ˆ

global-attribute-sections ::= *global-attribute-section* | *global-attribute-sections*ˆ
global-attribute-section

global-attribute-section ::=
[*global-attribute-target-specifier*ˆ *attribute-list* **]** |
[*global-attribute-target-specifier*ˆ *attribute-list* **,** **]**

global-attribute-target-specifier ::= *global-attribute-target* **:**

global-attribute-target ::= **assembly** | **module**

attributes ::= *attribute-sections*ˆ

attribute-sections ::= *attribute-section* | *attribute-sections*ˆ *attribute-section*

attribute-section ::=
[*attribute-list* **]** |
[*attribute-target-specifier*ˆ *attribute-list* **]** |
[*attribute-list* **,** **]** |
[*attribute-target-specifier*ˆ *attribute-list* **,** **]**

attribute-target-specifier ::= *attribute-target* **:**

attribute-target ::= **field** | **method** | **param** | **property** | **type** | **return** | **event**

attribute-list ::= *attribute* | *attribute-list*ˆ **,** *attribute*

attribute ::= (*attribute-name*ˆ)!*qualified-identifier* |
(*attribute-name*ˆ)!*qualified-identifier attribute-arguments*

attribute-name ::= *type-name*ˆ

attribute-arguments ::=
((*positional-argument-list*ˆ)!*expression-list* **,** *named-argument-list* **)** |
(*named-argument-list* **)** | **(** **)** |
((*positional-argument-list*ˆ)!*expression-list* **)**

positional-argument-list ::= *positional-argument*ˆ | *positional-argument-list*ˆ **,**
*positional-argument*ˆ

positional-argument ::= *attribute-argument-expression*ˆ

named-argument-list ::= *named-argument* | *named-argument-list*ˆ **,** *named-argument*

named-argument ::= **identifier** = *attribute-argument-expression*ˆ

attribute-argument-expression ::= *expression*

literal ::= **false** | **true** |
integer-literal |
real-literal |
character-literal |
string-literal |
null-literal

Appendix C

Abstract Syntax Grammar for C# 1.2

This appendix describes the abstract syntax grammar for C# used in Chapter 6. This grammar is given as a set of EBNF grammar rules. The start symbol of this grammar is the non-terminal `compilation-unit`.

literal ::= **false** | **true** | **integer-literal** | **real-literal** | **character-literal** | **string-literal** | **null-literal**

type ::= *predefined-type* | *qualified-identifier* | *array-type*

predefined-type ::= **object** | **string** | *integral-type* | **float** | **double** | **decimal** | **bool**

integral-type ::= **sbyte** | **byte** | **short** | **ushort** | **int** | **uint** | **long** | **ulong** | **char**

expression ::= *literal* |
identifier |
(*expression* **)** |
expression **.** **identifier** |
predefined-type **.** **identifier** |
expression **(** *argument-list*? **)** |
expression **[** *expression-list* **]** |
this |
base **.** **identifier** |
base **[** *expression-list* **]** |
expression unary-assignment-operator |
new *type* **(** *argument-list*? **)** |
new *type* **[** *expression-list* **]** *rank-specifier** *array-initializer*? |
new *type rank-specifier*+ *array-initializer* |
typeof **(** *return-type* **)** |

```
    checked ( expression ) |
    unchecked ( expression ) |
    unary-operator expression |
    unary-assignment-operator expression |
    ( type ) expression |
    expression binary-operator expression |
    expression is type |
    expression as type |
    expression ? expression : expression |
    expression assignment-operator expression

unary-operator ::= + | - | ! | ~

unary-assignment-operator ::= ++ | --

binary-operator ::= overloadable-binary-operator | || | &&

overloadable-unary-operator ::= unary-operator | unary-assignment-operator | true |
    false

overloadable-binary-operator ::= + | - | * | / | % | & | | | ^ | << | >> | == |
    != | > | < | >= | <=

assignment-operator ::= = | += | -= | *= | /= | %= | &= | |= | ^= | <<= | >>=

argument-list ::= argument (, argument)*

argument ::= expression | ref expression | out expression

expression-list ::= expression (, expression)*

statement ::= embedded-statement | declaration-statement | identifier : statement

embedded-statement ::= block |
    ; |
    expression ; |
    if ( expression ) embedded-statement (else embedded-statement)? |
    switch ( expression ) { switch-section* } |
    while ( expression ) embedded-statement |
    do embedded-statement while ( expression ) ; |
    for ( for-initializer? ; expression? ; expression-list? ) embedded-statement |
    foreach ( type identifier in expression ) embedded-statement |
    break ; |
    continue ; |
    goto identifier ; |
    goto case expression ; |
```

```
    goto default ; |
    return expression? ; |
    throw expression? ; |
    try block catch-clauses? (finally block)? |
    checked block |
    unchecked block |
    lock ( expression ) embedded-statement |
    using ( resource-acquisition ) embedded-statement

block ::= { statement* }

declaration-statement ::= local-variable-declaration ; | local-constant-declaration ;

local-variable-declaration ::= type variable-declarators

local-constant-declaration ::= const type constant-declarators

switch-section ::= switch-label+ statement+

switch-label ::= case expression : | default :

for-initializer ::= local-variable-declaration | expression-list

catch-clauses ::= specific-catch-clause+ |
    specific-catch-clause* general-catch-clause

specific-catch-clause ::= catch ( type identifier? ) block

general-catch-clause ::= catch block

resource-acquisition ::= local-variable-declaration | expression

compilation-unit ::= using-directive* global-attribute-section*
    namespace-member-declaration*

namespace-member-declaration ::= namespace qualified-identifier { using-directive*
    namespace-member-declaration* } ;? type-declaration ;

qualified-identifier ::= identifier (. identifier)*

using-directive ::= using (identifier =)? qualified-identifier ;

type-declaration ::= class-declaration | struct-declaration | interface-declaration
    | enum-declaration | delegate-declaration
```

class-declaration ::= *attribute-section** *modifier** **class** **identifier** *class-base*? **{**
    *member-declaration** **}** **;**?

class-base ::= **:** (**object** | **string** | *qualified-identifier*) (**,** *qualified-identifier*)*

modifier ::= **new** | **public** | **protected** | **internal** | **private** | **abstract** | **sealed** |
    static | **readonly** | **volatile** | **virtual** | **override** | **extern**

member-declaration ::= *constant-declaration* | *field-declaration* |
    method-declaration | *property-declaration* | *event-declaration* |
    indexer-declaration | *operator-declaration* | *constructor-declaration* |
    static-constructor-declaration | *destructor-declaration* | *type-declaration*

body ::= *block* | **;**

constant-declaration ::= *attribute-section** *modifier** **const** *type*
    constant-declarators **;**

constant-declarators ::= *constant-declarator* (**,** *constant-declarator*)*

constant-declarator ::= **identifier** **=** *expression*

field-declaration ::= *attribute-section** *modifier** *type* *variable-declarators* **;**

variable-declarators ::= *variable-declarator* (**,** *variable-declarator*)*

variable-declarator ::= **identifier** (**=** *variable-initializer*)?

variable-initializer ::= *expression* | *array-initializer*

method-declaration ::= *attribute-section** *modifier** *return-type*
    qualified-identifier **(** *formal-parameter-list*? **)** *body*

return-type ::= **void** | *type*

formal-parameter-list ::= *fixed-parameter* (**,** *fixed-parameter*)* (**,** *parameter-array*)?
    | *parameter-array*

fixed-parameter ::= *attribute-section** (**ref** | **out**)? *type* **identifier**

parameter-array ::= *attribute-section** **params** *type* *rank-specifier*+ **identifier**

property-declaration ::= *attribute-section** *modifier** *type* *qualified-identifier* **{**
    accessor-declarations **}**

accessor-declarations ::= *attribute-section** **get** *body* (*attribute-section** **set** *body*)?
    |
    *attribute-section** **set** *body* (*attribute-section** **get** *body*)?

event-declaration ::= *attribute-section** *modifier** **event** *local-variable-declaration*
    ; |
    *attribute-section** *modifier** **event** *type qualified-identifier* **{**
        event-accessor-declarations **}**

indexer-declaration ::= *attribute-section** *modifier** *indexer-declarator* **{**
    accessor-declarations **}**

indexer-declarator ::= *type* (*qualified-identifier* **.**)? **this** **[** *formal-parameter-list* **]**

operator-declaration ::= *attribute-section** *modifier** *operator-declarator body*

operator-declarator ::= (**implicit** | **explicit**) **operator** *type* **(** *type* **identifier** **)** |
    type **operator** *overloadable-unary-operator* **(** *type* **identifier** **)** |
    type **operator** *overloadable-binary-operator* **(** *type* **identifier** **,** *type* **identifier** **)**

constructor-declaration ::= *attribute-section** *modifier** **identifier** **(**
    formal-parameter-list? **)** *constructor-initializer*? *body*

constructor-initializer ::= **:** *constructor-order* **(** *argument-list*? **)**

constructor-order ::= **base** | **this**

static-constructor-declaration ::= *attribute-section** *static-constructor-modifiers*
    identifier **(** **)** *body*

static-constructor-modifiers ::= **static** | **extern** **static** | **static** **extern**

destructor-declaration ::= *attribute-section** **extern**? **~** **identifier** **(** **)** *body*

struct-declaration ::= *attribute-section** *modifier** **struct** **identifier** *interface-base*?
    { *member-declaration** **}** **;**?

array-type ::= *type rank-specifier*+

rank-specifier ::= **[** **,*** **]**

array-initializer ::= **{** *variable-initializer-list*? **,**? **}**

variable-initializer-list ::= *variable-initializer* (**,** *variable-initializer*)*

interface-declaration ::= *attribute-section** *modifier** **interface** **identifier**
    interface-base? **{** *interface-member-declaration** **}** **;**?

interface-base ::= **:** *interface-type-list*

interface-type-list ::= *qualified-identifier* (**,** *qualified-identifier*)*

interface-member-declaration ::= *attribute-section** **new**? **event** *type* **identifier** **;** |
    *attribute-section** **new**? *type* **identifier** **{** *interface-accessors* **}** |
    *attribute-section** **new**? *return-type* **identifier** **(** *formal-parameter-list*? **)** **;** |
    *attribute-section** **new**? *type* **this** **[** *formal-parameter-list* **]** **{**
        interface-accessors **}**

interface-accessors ::= *attribute-section** **get** **;** |
    *attribute-section** **set** **;** |
    *attribute-section** **get** **;** *attribute-section** **set** **;** |
    *attribute-section** **set** **;** *attribute-section** **get** **;**

enum-declaration ::= *attribute-section** *modifier** **enum** **identifier** *enum-base*? **{**
    enum-member-declarations? **,**? **}** **;**?

enum-base ::= **:** *integral-type*

enum-member-declarations ::= *enum-member-declaration* (**,** *enum-member-declaration*)*

enum-member-declaration ::= *attribute-section** **identifier** |
    *attribute-section** **identifier** **=** *expression*

delegate-declaration ::= *attribute-section** *modifier** **delegate** *return-type*
    identifier **(** *formal-parameter-list*? **)** **;**

event-accessor-declarations ::=
    add-accessor-declaration remove-accessor-declaration |
    remove-accessor-declaration add-accessor-declaration

add-accessor-declaration ::= *attribute-section** **add** *block*

remove-accessor-declaration ::= *attribute-section** **remove** *block*

global-attribute-section ::= **[** (**assembly** | **module**) **:** *attribute-list* **,**? **]**

attribute-list ::= *attribute* (**,** *attribute*)*

attribute ::= *qualified-identifier attribute-arguments*?

attribute-arguments ::= **(** *expression-list*? **)** |

```
    ( expression-list , named-argument-list ) |
    ( named-argument-list )

named-argument-list ::= named-argument (, named-argument)*

named-argument ::= identifier = expression

attribute-section ::= [ (attribute-target :)? attribute-list ,? ]

attribute-target ::= field | event | method | param |  property | return | type
```

Bibliography

[AC76] American National Standards Institute, Committee on Computers and Information Processing X3 and Computer and Business Equipment Manufacturers Association. *American National Standard programming language PL/I: approved August 9, 1976, American National Standards Institute, Inc.: ANSI X3.53-1976*. 1430 Broadway, New York, NY 10018, USA: American National Standards Institute, 1976, p. 403 (cit. on p. 27).

[Afr+13] Ali Afroozeh et al. "Safe Specification of Operator Precedence Rules". In: *Software Language Engineering*. Ed. by Martin Erwig, Richard F. Paige, and Eric Wyk. Vol. 8225. Lecture Notes in Computer Science. Springer International Publishing, 2013, pp. 137–156. ISBN: 9783319026534. DOI: 10.1007/978-3-319-02654-1_8. URL: http://dx.doi.org/10.1007/978-3-319-02654-1_8 (cit. on p. 184).

[AH01] John Aycock and R. Nigel Horspool. "Schrödinger's token". In: *Software: Practice and Experience* 31.8 (2001), pp. 803–814. ISSN: 1097-024X. DOI: 10.1002/spe.390. URL: http://dx.doi.org/10.1002/spe.390 (cit. on pp. 50, 141).

[AH99] John Aycock and Nigel Horspool. "Faster Generalized LR Parsing". English. In: *Compiler Construction*. Ed. by Stefan Jähnichen. Vol. 1575. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 1999, pp. 32–46. ISBN: 9783540657170. DOI: 10.1007/978-3-540-49051-7_3. URL: http://dx.doi.org/10.1007/978-3-540-49051-7_3 (cit. on p. 8).

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1986. ISBN: 0201100886 (cit. on p. 9).

[B14] Leonardo B. *litjson*. GitHub, Inc. 2014. URL: https://github.com/lbv/litjson (cit. on p. 153).

[BJ78] Dines Bjørner and Cliff B. Jones. *The Vienna Development Method: The Meta-Language*. Vol. 61. Lecture notes in computer science, vol. 61. Springer Berlin Heidelberg, 1978. URL: http://link.springer.com/book/10.1007/3-540-08766-4 (cit. on p. 87).

[BN98] Franz Baader and Tobias Nipkow. *Term Rewriting and All That*. New York, NY, USA: Cambridge University Press, 1998. ISBN: 0521455200 (cit. on p. 88).

[Bra+01] M.G.J. van den Brand et al. "The Asf+Sdf Meta-environment: A Component-Based Language Development Environment". English. In: *Compiler Construction*. Ed. by Reinhard Wilhelm. Vol. 2027. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2001, pp. 365–370. ISBN: 9783540418610. DOI: 10.1007/3-540-45306-7_26. URL: http://dx.doi.org/10.1007/3-540-45306-7_26 (cit. on p. 51).

[BV12] Hendrikus J.S. Basten and Jurgen J. Vinju. "Parse Forest Diagnostics with Dr. Ambiguity". English. In: *Software Language Engineering*. Ed. by Anthony Sloane and Uwe Aßmann. Vol. 6940. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2012, pp. 283–302. ISBN: 9783642288296. DOI: 10.1007/978-3-642-28830-2_16. URL: http://dx.doi.org/10.1007/978-3-642-28830-2_16 (cit. on p. 184).

[Cho56] N. Chomsky. "Three models for the description of language". In: *Information Theory, IRE Transactions on* 2.3 (1956), pp. 113–124. ISSN: 0096-1000. DOI: 10.1109/TIT.1956.1056813 (cit. on p. 12).

[Chu+15] Martin Churchill et al. "Reusable Components of Semantic Specifications". English. In: *Transactions on Aspect-Oriented Software Development XII*. Ed. by Shigeru Chiba et al. Vol. 8989. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2015, pp. 132–179. ISBN: 9783662467336. DOI: 10.1007/978-3-662-46734-3_4. URL: http://dx.doi.org/10.1007/978-3-662-46734-3_4 (cit. on pp. 140, 157, 158, 172).

[CT96] Jean-Pierre Chanod and Pasi Tapanainen. "A non-deterministic tokeniser for finite-state parsing". In: *Proceedings of the Workshop on Extended finite state models of language (ECAI'96)*. 1996 (cit. on p. 54).

[D04] Mosses Peter D. "Modular structural operational semantics". In: *The Journal of Logic and Algebraic Programming* 60-61 (2004), pp. 195–228. URL: http://www.sciencedirect.com/science/article/pii/S156783260400027X (cit. on p. 158).

[DFH04] L. Duan, A. Franz, and K. Horiguchi. *Method and system for reducing lexical ambiguity*. US Patent 6,721,697. 2004. URL: http://www.google.com/patents/US6721697 (cit. on p. 54).

[Ear70] Jay Earley. "An Efficient Context-free Parsing Algorithm". In: *Commun. ACM* 13.2 (1970), pp. 94–102. ISSN: 0001-0782. DOI: 10.1145/362007.362035. URL: http://doi.acm.org/10.1145/362007.362035 (cit. on p. 8).

[GJ08] D. Grune and C. Jacobs. *Parsing Techniques: A Practical Guide*. Monographs in Computer Science. Springer, 2008. ISBN: 9780387202488 (cit. on p. 10).

[Gos+15] James Gosling et al. *The Java Language Specification, Java SE 8 Edition*. 2015. URL: https://docs.oracle.com/javase/specs/jls/se8/jls8.pdf (cit. on p. 140).

[HCC02] Hewlett-Packard, Intel Corporation, and Microsoft Corporation. *C# Language Specification. Standard ECMA-334 2nd Edition*. ECMA International, 2002. URL: http://www.ecma-international.org/publications/files/ECMA-ST-WITHDRAWN/ECMA-334,%202nd%20edition,%20December%202002.pdf (cit. on pp. 23, 140, 157, 166, 225, 227).

[HCC06] Hewlett-Packard, Intel Corporation, and Microsoft Corporation. *C# Language Specification. Standard ECMA-334 4th Edition*. ECMA International, 2006. URL: http://www.ecma-international.org/publications/standards/Ecma-334.htm (cit. on pp. 22, 140, 141, 189, 191).

[Hee+89] J. Heering et al. "The Syntax Definition Formalism SDF—Reference Manual—". In: *SIGPLAN Not.* 24.11 (1989), pp. 43–75. ISSN: 0362-1340. DOI: 10.1145/71605.71607. URL: http://doi.acm.org/10.1145/71605.71607 (cit. on p. 97).

[IA78] American National Standards Institute and Computer and Business Equipment Manufacturers Association. *American National Standard Programming Language FORTRAN: approved April 3, 1978: American National Standards Institute, Inc.: ANSI X3.9-1978*. 1430 Broadway, New York, NY 10018, USA: American National Standards Institute, 1978. URL: https://www.fortran.com/F77_std/rjcnf-0.html (cit. on p. 40).

[ISO11] ISO/IEC. *ISO/IEC 9899:2011 - Information technology – Programming languages – C*. Tech. rep. International Organization for Standardization (ISO), 2011. URL: http://www.open-std.org/jtc1/sc22/wg14/www/standards.html (cit. on pp. 137, 138).

[JS10] Adrian Johnstone and Elizabeth Scott. "Tear-Insert-Fold Grammars". In: *Proceedings of the Tenth Workshop on Language Descriptions, Tools and Applications*. LDTA '10. Paphos, Cyprus: ACM, 2010, 6:1–6:8. ISBN: 9781450300636. DOI: 10.1145/1868281.1868287. URL: http://doi.acm.org/10.1145/1868281.1868287 (cit. on pp. 87, 94, 99).

[JS11] Adrian Johnstone and Elizabeth Scott. "Translator Generation Using ART". In: *Proceedings of the Third International Conference on Software Language Engineering*. SLE'10. Eindhoven, The Netherlands: Springer-Verlag, 2011, pp. 306–315. ISBN: 9783642194399. URL: http://dl.acm.org/citation.cfm?id=1964571.1964599 (cit. on p. 140).

[JS98] Adrian Johnstone and Elizabeth Scott. "RDP—an iterator-based recursive descent parser generator with tree promotion operators". In: *ACM SIGPLAN Notices* 33.9 (1998), pp. 87–94 (cit. on pp. 94, 97).

[JSB11] Adrian Johnstone, Elizabeth Scott, and Mark van den Brand. "LDT: A Language Definition Technique". In: *Proceedings of the Eleventh Workshop on Language Descriptions, Tools and Applications*. LDTA '11. Saarbrucken, Germany: ACM, 2011, 9:1–9:8. ISBN: 9781450306652. DOI: 10.1145/1988783.1988792. URL: http://doi.acm.org/10.1145/1988783.1988792 (cit. on p. 135).

[KBV01] J. W. Klop, Marc Bezem, and R. C. De Vrijer, eds. *Term Rewriting Systems*. New York, NY, USA: Cambridge University Press, 2001. ISBN: 0521391156 (cit. on pp. 88, 181).

[Knu68] Donald E. Knuth. "Semantics of context-free languages". In: *Theory of Computing Systems* 2.2 (1968), pp. 127–145. ISSN: 0025-5661. DOI: 10.1007/BF01692511. URL: http://dx.doi.org/10.1007/BF01692511 (cit. on p. 86).

[KSV11] Paul Klint, Tijs van der Storm, and Jurgen Vinju. "EASY Meta-programming with Rascal". In: *Proceedings of the 3rd International Summer School Conference on Generative and Transformational Techniques in Software Engineering III*. GTTSE'09. Braga, Portugal: Springer-Verlag, 2011, pp. 222–289. ISBN: 3-642-18022-1, 978-3-642-18022-4. URL: http://dl.acm.org/citation.cfm?id=1949925.1949932 (cit. on p. 51).

[KV10] Lennart CL Kats and Eelco Visser. "The spoofax language workbench: rules for declarative specification of languages and IDEs". In: *ACM sigplan notices*. Vol. 45. 10. ACM. 2010, pp. 444–463 (cit. on p. 51).

[Lan74] Bernard Lang. "Deterministic Techniques for Efficient Non-Deterministic Parsers". In: *Proceedings of the 2nd Colloquium on Automata, Languages and Programming*. London, UK: Springer-Verlag, 1974, pp. 255–269. ISBN: 3540068414. URL: http://dl.acm.org/citation.cfm?id=646230.681872 (cit. on p. 8).

[Ler+12] Xavier Leroy et al. *The OCaml system release 4.00*. 2012. URL: http://caml.inria.fr/pub/docs/manual-ocaml-4.00/ (cit. on p. 188).

[Lip10] Eric Lippert. *No backtracking, Part One*. Microsoft. 2010. URL: http://blogs.msdn.com/b/ericlippert/archive/2010/10/04/no-backtracking-part-one.aspx (cit. on p. 25).

[LS75] Michael E Lesk and Eric Schmidt. *Lex: A lexical analyzer generator*. Bell Laboratories. Murray Hill, New Jersey 07974: Bell Laboratories, 1975. URL: https://ken-cc.googlecode.com/svn/trunk/doc/lex.pdf (cit. on p. 34).

[Mic15a] Robert Michael Walsh. *CSharpFrontEnd*. GitHub, Inc. 2015. URL: https://github.com/robertmichaelwalsh/CSharpFrontEnd (cit. on pp. 141, 157).

[Mic15b] Robert Michael Walsh. *Multilex*. GitHub, Inc. 2015. URL: https://github.com/robertmichaelwalsh/Multilex (cit. on pp. 141, 142, 155).

[Mos+15] Peter Mosses et al. *Programming Language Components and Specifications*. 2011-2015. URL: http://www.plancomps.org/ (cit. on pp. 3, 23, 86, 94, 140, 157, 185).

[Mos08] Peter D. Mosses. "Component-Based Description of Programming Languages". In: *Visions of Computer Science. BCS International Academic Conference*. (Imperial College, London, UK). Ed. by Erol Gelenbe, Samson Abramsky, and Vladimiro Sassone. BCS. 2008. URL: http://ewic.bcs.org/content/ConWebDoc/22912 (cit. on p. 158).

[Noz91] Rahman Nozohoor-Farshi. "GLR Parsing for ϵ-grammars". English. In: *Generalized LR Parsing*. Ed. by Masaru Tomita. Springer US, 1991, pp. 61–75. ISBN: 9781461368045. DOI: 10.1007/978-1-4615-4034-2_5. URL: http://dx.doi.org/10.1007/978-1-4615-4034-2_5 (cit. on p. 8).

[OJ09] Jeffrey L. Overbey and Ralph E. Johnson. "Software Language Engineering". In: ed. by Dragan Gašević, Ralf Lämmel, and Eric Wyk. Berlin, Heidelberg: Springer-Verlag, 2009. Chap. Generating Rewritable Abstract Syntax Trees, pp. 114–133. ISBN: 9783642004339. DOI: 10.1007/978-3-642-

00434-6_8. URL: http://dx.doi.org/10.1007/978-3-642-00434-6_8 (cit. on p. 88).

[Plo04] Gordon D Plotkin. "A structural approach to operational semantics". In: *The Journal of Logic and Algebraic Programming* 60–61 (2004). Structural Operational Semantics, pp. 17–139. ISSN: 1567-8326. DOI: http://dx.doi.org/10.1016/j.jlap.2004.05.001. URL: http://www.sciencedirect.com/science/article/pii/S1567832604000402 (cit. on p. 87).

[Pro06] Mono Project. *mono-1-2*. GitHub, Inc. 2006. URL: https://github.com/mono/mono/tree/mono-1-2 (cit. on p. 153).

[Pro11] Mono Project. *mono-2-0*. GitHub, Inc. 2011. URL: https://github.com/mono/mono/tree/mono-2-0 (cit. on p. 153).

[Ros91] James A. Roskind. *A YACC-able C++ 2.1 GRAMMAR, AND THE RESULTING AMBIGUITIES*. Version 2.0. 1991. URL: ftp://ftp.iecc.com/pub/file/c++grammar/grammar5.txt (cit. on p. 54).

[RS59] Michael O. Rabin and Dana Scott. "Finite Automata and Their Decision Problems". In: *IBM Journal of Research and Development* 3.2 (1959), pp. 114–125. ISSN: 0018-8646. DOI: 10.1147/rd.32.0114 (cit. on pp. 11, 34).

[San+14] Bram van der Sanden et al. "Parse Forest Disambiguation". PhD thesis. Master's thesis, Eindhoven University of Technology, the Netherlands, 2014 (cit. on p. 184).

[SG15] Richard Stallman and the GCC developer Community. "GNU Compiler Collection Internals". In: *Free Software Foundation, Inc.* (1988-2015) (cit. on p. 87).

[SJ10a] Elizabeth Scott and Adrian Johnstone. "GLL Parsing". In: *Electronic Notes in Theoretical Computer Science* 253.7 (2010). Proceedings of the Ninth Workshop on Language Descriptions Tools and Applications (LDTA 2009), pp. 177–189. ISSN: 1571-0661. DOI: http://dx.doi.org/10.1016/j.entcs.2010.08.041. URL: http://www.sciencedirect.com/science/article/pii/S1571066110001209 (cit. on p. 187).

[SJ10b] Elizabeth Scott and Adrian Johnstone. "Recognition is Not Parsing - SPPF-style Parsing from Cubic Recognisers". In: *Sci. Comput. Program.* 75.1-2 (2010), pp. 55–70. ISSN: 0167-6423. DOI: 10.1016/j.scico.2009.07.001. URL: http://dx.doi.org/10.1016/j.scico.2009.07.001 (cit. on p. 19).

[SJ13] Elizabeth Scott and Adrian Johnstone. “GLL parse-tree generation”. In: *Science of Computer Programming* 78.10 (2013), pp. 1828–1844 (cit. on pp. 8, 55, 59).

[Tom85] Masaru Tomita. “An Efficient Context-free Parsing Algorithm for Natural Languages”. In: *Proceedings of the 9th International Joint Conference on Artificial Intelligence - Volume 2*. IJCAI’85. Los Angeles, California: Morgan Kaufmann Publishers Inc., 1985, pp. 756–764. ISBN: 0-934613-02-8, 978-0-934-61302-6. URL: http://dl.acm.org/citation.cfm?id=1623611.1623625 (cit. on pp. 8, 16).

[Vis97] Eelco Visser. *Scannerless generalized-LR parsing*. Universiteit van Amsterdam. Programming Research Group, 1997 (cit. on pp. 51, 52, 127).

[Wir77] Niklaus Wirth. “What Can We Do About the Unnecessary Diversity of Notation for Syntactic Definitions?” In: *Commun. ACM* 20.11 (1977), pp. 822–823. ISSN: 0001-0782. DOI: 10.1145/359863.359883. URL: http://doi.acm.org/10.1145/359863.359883 (cit. on p. 14).

[Wir96] Niklaus Wirth. “Extended Backus-Naur Form (EBNF)”. In: *ISO/IEC* 14977 (1996), p. 2996 (cit. on p. 14).