Multiple input parsing and lexical analysis

ELIZABETH SCOTT, ADRIAN JOHNSTONE, and ROBERT WALSH, Royal Holloway, University of London, UK

CCS Concepts: • **Theory of computation → Grammars and context-free languages**; • **Software and its engineering → Syntax**.

Additional Key Words and Phrases: Lexical analysis, programming language syntax specification, lexical disambiguation

ACM Reference Format:
Elizabeth Scott, Adrian Johnstone, and Robert Walsh. 2018. Multiple input parsing and lexical analysis. 1, 1 (April 2018), 45 pages. https://doi.org/10.1145/1122445.1122456

Abstract

This paper introduces two new approaches in the areas of lexical analysis and context free parsing. We present an extension, MGLL, of generalised parsing which allows multiple input strings to be parsed together efficiently, and we present an enhanced approach to lexical analysis which exploits this multiple parsing capability. The work provides new power to formal language specification and disambiguation, and brings new techniques into the historically well studied areas of lexical and syntax analysis. It encompasses character level parsing at one extreme and the classical LEX/YACC style division at the other, allowing the advantages of both approaches.

1 INTRODUCTION

In this paper we present a modern alternative to the traditional approach to formal language specification in which lexical and syntax analysis are separate procedures. Separation of 'words' from the 'sentences' they are composed into matches human perception and allows efficient implementations of language analysers. However, the particular meaning of a word may be dependent on the sentence in which it appears and separate lexical and syntax specifications do not easily support this. The problem is that the lexical phase returns a single lexicalised string to the syntax analyser. For example, the Java language specifications have moved away from traditional LEX/YACC style specifications as the language has become more baroque. The first Java Language specification document gives two grammars, one of which is intended to be suitable for use with Yacc-like LALR parser generators. The more recent JLS18 version does not provide an LALR formal grammar, and the grammar that is given would pose significant challenges to a traditional parser generator. The technical contributions of this paper are two new approaches in the areas of context free parsing and lexical analysis. We present an extension, MGLL, of generalised parsing which allows multiple input strings to be parsed together efficiently, and we present an enhanced approach to lexical analysis which exploits this multiple

Authors' address: Elizabeth Scott, e.scott@rhul.ac.uk; Adrian Johnstone, a.johnstone@rhul.ac.uk; Robert Walsh, Royal Holloway, University of London, Egham Hill, Egham, Surrey, UK, TW20 0EX.

Manuscript submitted to ACM

parsing capability. The work provides new power to formal language specification and disambiguation, and brings new techniques into the historically well studied areas of lexical and syntactic analysis.

The paper is divided into three parts: Part 1 addresses TWE (Tokens With Extents) sets which represent sets of lexicalisations, Part 2 introduces MGLL, and Part 3 looks at practicalities, including a Java case study. Parts 1 and 2 can be approached independently. Part 1 (Section 2) discusses 'indexed' lexicalisations and sets up the associated 'extents' machinery and can be seen as motivation for the multi-parser presented in Part 2. Conversely Part 2 (Sections 3, 4) can be seen as the primary contribution, with lexical flexibility as an application. Multi-parsing is novel, and it is easier to understand if the application to parsing multiple lexicalisations is thought of as a concrete example. Thus we present the multi-lexing material first, although multi-parsing is more fundamental.

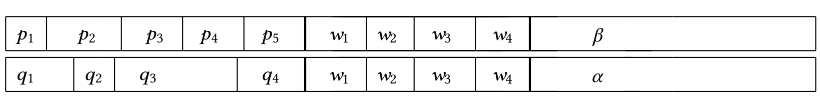
In Part 3 (Sections 5 – 9) we look at practicalities including constructing TWE sets from a character string, whitespace handling and lexical disambiguation. Part 3 provides a formal basis for the experimental results of a Java case study we initially reported in [SJ19] and have updated here. A full evaluation of implementation strategies and run-time costs will only emerge as the technique is used more widely in the community. To establish base line practicality we have carried out initial investigations, using the Java language specification as an exemplar.

1.1 Words and sentences

Language analysis typically involves grouping a set of characters into words (lexical analysis) and then structuring the words into sentences (phrase level analysis) from which semantics are extracted. Lexical analysis is commonly held to be a solved problem: theoretically the input character strings are grouped into words and recognised by linear time finite state automata which are automatically constructed from the regular expression specifications [ASU86, ALSU06]. This approach is embodied in the venerable LEX and FLEX tools [ME90]. In practice however, lexical analysers for real languages are often constructed by hand to facilitate features which do not fit easily into the domain of regular expressions and a strict lexer/parser divide.

Furthermore, for programming languages one particular lexicalisation of an input character string is usually selected, independently of the phrase level structure, before parsing is attempted. Permitting several lexicalisations to be parsed allows syntactic context to be used before ultimately rejecting lexicalisations, making the boundary between lexical and phrase level specification more fluid and giving a programming language designer more freedom.

The key to the practicality of allowing multiple lexicalisations lies in the efficient parsing of multiple input strings. In principle, if two sentences have a common subsequence of words, for example $p_1p_2p_3p_4p_5w_1w_2w_3w_4\alpha$ and $q_1q_2q_3q_4w_1w_2w_3w_4\beta$, then the parsing of the sequence $w_1w_2w_3w_4$ could be shared. (An illustration of how such a situation may arise from multiple lexicalisations is given in Section 1.2.2.) However, most formal parsing techniques are based on the position of each word in the sequence (the words p_i and q_i are parsed at step i etc), and since the lengths of the initial sequences are different (5 and 4 respectively) the parser cannot synchronise on w_1. In the MGLL paradigm we add a pair of integer 'extents' to each word; these are the left and right index positions of the word in the input character string. Then the MGLL parser steps correspond to the left extents, and by making the left extent of w_1 the same in both sequences a parser can synchronise on w_1 and parse $w_1w_2w_3w_4$ concurrently for both sequences.



1.2 Language structure and ambiguity

We can view a programming language $\mathcal{L}$ as being a set of strings over some finite alphabet $\mathcal{A}$, together with a semantics (meaning) for each element of $\mathcal{L}$. The job of a language designer is to specify the set of strings and their semantics. One rather extreme approach would be to make $\mathcal{L}$ the set $\mathcal{A}^*$ of all the strings over $\mathcal{A}$ and to include `invalid` as a possible semantic value. However, specifying semantics formally is difficult and splitting the language analysis task into several stages is usually helpful. Specified syntax rules restrict $\mathcal{L}$ to a much smaller subset of the set of all strings, and semantics are specified only for those strings.

1.2.1 Syntax and semantics. In detail, the words of $\mathcal{L}$ are specified as a set of subsets of $\mathcal{A}^*$, and each of these subsets (patterns) is identified by a unique token. The sentences are then specified via a context free grammar whose terminals are these tokens. The point is to take advantage of well understood structural specifications such as regular expressions and context free grammars, both to reduce the number of strings for which semantics must be specified and to provide a structure on which the semantic definitions can be based. In this sense, the syntactic part of a language specification is the first stage of the semantic specification.

In general, given a sequence of characters q and a set of tokens, there will be more than one way of converting q into a token sequence. Even in natural languages this is the case; for example `greenhouse` corresponds both to a single token `noun` and to a sequence of length two, `adjective noun`. In English this is handled by requiring that words are separated by spaces in a character string. Programming languages do not usually require a user to separate the words in this way so other mechanisms are needed to identify individual words. Furthermore, some context free grammars can structure a token sequence into a sentence in more than one way. For example `read and write or mark` can be structured as `(read and write) or mark` or as `read and (write or mark)`. Selecting from among several token sequences and sentential structures is referred to as disambiguation. Context free ambiguity is hard to reason about in practice, and is undecidable in general.

1.2.2 Lexical ambiguity. It is usual to design phrase level grammars to limit potential phrase level ambiguity, but lexical level ambiguity is universal and extensive. Lexical disambiguation is primarily carried out according to two principles: longest match from the left and designer-specified priority. In the first case, the character string q is read from the left and the longest string q_1 which matches some token is selected: $q = q_1q'$. The process is then repeated with the string q', and continues until the whole string has been lexicalised. If q_1 belongs to the sets of two tokens, then the token with the highest priority is selected.

Mostly this lexical disambiguation strategy works well, but it is not always powerful enough. Consider the following English 'sentence' in which the words have not been identified using spaces:

```
painthegreenhousered
```

Using the longest match disambiguation strategy, a painting contractor would interpret this as

```
paint  the  greenhouse  red
```

This may result in unexpected behaviour if one of the houses had a greenhouse in the garden! If there were no greenhouse then the contractor may reject the instruction as invalid, despite the fact that one of the houses was painted green.

This slightly artificial seeming situation actually does arise in C-style languages, including Java. Given input `x-----y` a Java lexical analyser will lexicalise the string as `(x)(--)(--)(-)(y)` which will then be rejected as invalid by the semantic analyser because the postdecrement operator `--` returns a value and the second `--` cannot be applied. However, there is of course a lexicalisation (x)(--)(-)(--)(y) which does have a valid semantic interpretation. A similar example is x--y which is lexicalised as (x)(--)(y) rather than the syntactically valid (x)(-)(-)(y). Perhaps a little confusingly, the unique lexicalisation of x+-y is the syntactically correct (x)(+)(-)(y), so x+-y is legal Java while x--y is not.

The greenhouse example provides a concrete illustration of the sharing remarks made in Section 1.1. After parsing the two lexicalisations of the phrase, an MGLL parse can synchronise the parsing of any subsequent text:

p_1(paint)	p_2(the)	p_3(green)	p_4(house)	$p_5(red)$	w_1(and)	w_2(then)	w_3(go)	w_4(home)	$\cdots$

q_1(paint)	q_2(the)	q_3(greenhouse)	$q_4(red)$	w_1(and)	w_2(then)	w_3(go)	w_4(home)	$\cdots$

In fact we have $p_1 = q_1$, $p_2 = q_2$ and $p_5 = q_4$ and an MGLL parser will actually concurrently parse $p_1 p_2 = q_1 q_2$, and $p_5 w_1 w_2 w_3 w_4 = q_4 w_1 w_2 w_3 w_4$.

1.3 Our approach

The need to decide on a single token string before parsing commences is a significant drawback of the classical approach to language analysis. We now have efficient general parsers which can cope with ambiguity and, as we shall show in this paper, multiple input strings. Thus we can allow the lexer to pass some selection decisions, such as the choice between (x)(--)(y) and (x)(-)(-)(y) above, to the parser.

In this paper we (i) give a new version, MGLL, of the GLL parsing approach which parses multiple input strings and represents the resulting derivations efficiently in an extended form of shared packed parse forest (SPPF), (ii) describe how to specify multiple lexicalisations which form the input to an MGLL parser, and (iii) give lexical disambiguation mechanisms.

To motivate this somewhat technically detailed paper we highlight now that our technique is practical, and performs better than the alternative character level specification approach; it constitutes a paradigm shift in the handling of the traditional lexer/parser interface of a compiler. For example, as discussed in Section 8.3, our Java prototype MGLL implementation can parse all the 10^{26323} (indexed) lexicalisations of an example 64,537 character Java program, returning all the derivations of the syntactically correct lexicalisations in an extended shared packed parse forest (ESPPF) with 1,077,525 nodes. A corresponding character level GLL parser produces a shared packed parse forest with 2,735,250 nodes. With the standard longest match and priority lexical disambiguation enabled our MGLL parser returns all the derivations of the syntactically correct lexicalisations in an ESPPF with 301,920 nodes.

1.4 Related work

Production compilers typically use hand crafted front ends because as languages get richer, the lack of generality of the classical parsing approaches as described in the textbooks forces a shift to ad hoc mechanisms for dealing with non-determinisms.

The Java Language specifications have moved away from their initial LEX/YACC based approach and JLS18 does not provide an LALR formal grammar, posing significant challenges to a traditional parser generator. The actual parser used in Oracle's open-sourced javac is a collection of manually crafted parsing functions. We do not know what the formal relationship is between the published grammar and the parser, or how the hand-written parser deals with the ambiguities we have encountered. The same trend is visible within the GNU team: the LALR grammar for gcc was dropped some years ago in favour of a handcrafted parser. Clang has always had a hand-crafted parser as far as we know. In the software engineering world, the need for code quality measurement tools presents a particular parsing challenge since these systems need to include parsers for a broad set of languages and dialects. All this demonstrates need for more powerful general techniques.

The multi-parsing approach presented in the paper is completely new, but there are relationships to generalised parsing, character level parsing, and certain forms of lexical decision postponement.

Aycock and Horspool [AH01] described an approach primarily targeted at the problem of overlapping token sets, that is when two or more tokens share some lexemes and the choice of token depends on the context. Their motivating example was the language PL/I in which keywords such as `IF` are also permitted to be identifier names. Aycock and Horspool introduced what they called a Schrodinger token which is returned when lexemes which match more than one token are found. The values of instances of the Schrodinger token are determined at the time they are parsed. A general parser is used but only one token string is actually parsed. At the end of the paper there is a brief discussion of a possible extension of the technique to apply to more general lexical ambiguity. The example given is from C-like languages in which >> could be treated as two instances of the closing bracket > or as the binary shift operator. The idea, which is only sketched, is to introduce a further padding token `NULL` which matches just the empty string lexeme and which is ignored by the parser.

In [CT96] and [CT99] a non-deterministic lexical analyser for the French language is presented. Chanod and Tapanainen's focus is particularly on lexemes, such as *a priori* or *de meñe* or *240 000*, which can include spaces and may have more than one lexicalisation. Their system contains two lexical analysers; the first 'knows' about lexemes with spaces and identifies these then the second runs to match the remaining space delimited lexemes. The issue of more than one lexicalisation of a string is discussed but no formal treatment is given. Such cases are handled by methods specific to the French translation application.

Another way of passing lexicalisation decisions to the parser, often referred to as scannerless parsing, has also been explored in the literature [Vis97a]. It is possible to collapse the lexical/phrase level structure of a language specification by taking the tokens to be the alphabet characters $\mathcal{A}$, and taking the pattern of $y \in \mathcal{A}$ to be the set which contains just the string `'y'` of length one. The context free grammar then has $\mathcal{A}$ as its terminal set. The tokens from the traditional representation appear as non-terminals in the character level grammar, and the parser effectively constructs and parses all the original lexicalisations. The emergence of practical general parsing algorithms has allowed this approach to be implemented. For example, it is used in ASF+SDF [vdBHKO02] and implemented in an SGLR parser [Vis97b] which is used in Stratego/XT [Vis04]. Rascal [KvdSV09] also provides support for character level parsing.

However, character level grammars are nearly always ambiguous and using them needs care. There are two particular problems. Firstly the resulting derivation tree representation will require a lot of space and generation time, and it leaves a bigger disambiguation task for the semantic phase. Secondly, many parsing algorithms are made more efficient by using the next input (lookahead) symbol to determine the parse action. Character level tokens are not, in general, particularly good in this role. Filtering most of the lexicalisations before the parsing stage reduces the work for the parser, and parsing is, in general, a more computationally expensive operation than lexicalisation. Furthermore, phrase level error reporting is usually more helpful if it is presented at token rather than character level.

We have already presented [SJ19] a preliminary multi-parsing GLL-style algorithm, LCNP. This parser only parses multiple lexicalisations of a single character string, and the lexer is called by the parser during its execution. The main focus of the study was an examination of lexicalisation issues in Java. In this paper we review those results and give some further data on the relative sizes of the parser structures for MGLL and character level GLL Java parsers.

Part 1 - Multiple Lexicalisations

The key to efficient multi-parsing is ultimately to replace the input token strings with a set of *tokens with extents*. Conceptually the tokens in the string have an integer added. In a classical parsing approach this integer is the position of the token in the input string, and it is implicit in the indexed notation $t_0 t_1 \ldots t_m$ used to refer to the input string. Using more varied indexing gives the machinery needed for MGLL; the input string is modified to have the form $(t_0, i_0)(t_1, i_1) \ldots (t_m, i_m)$. For the multi-lexer parsing applications the integer i_j is the end position of the lexeme associated with t_j in the underlying character string. Then an indexed token string can be represented with a set of triples (t_j, i_{j-1}, i_j), and taking the union of sets for several input strings allows them to be parsed together with shared subsequences parsed concurrently. In this part of the paper we define and analyse these sets of triples.

We formally define the notions of indexed token strings and token with extents sets, identify properties of sets of tokens with extents which are important for the multi-lexer parsing application, and discuss a representation which does not increase the worst case asymptotic complexity order of the parser.

The reader whose is interested the generality of multi-parsers can just read Sections 2.1 and 2.2 and then move on to Part 2.

2 TOKENS WITH EXTENTS (TWE) SETS

To gain insight into the potential cost of identifying and processing all possible lexicalisations of an input string, we examined a 5859 character Java implementation of Conway's game of Life. Even using the original, relatively small, lexical and syntax specification for Java [GJS96], the total number of different possible lexicalisations is 1.92×10^{387}, far too large a number for them to be dealt with individually. Perhaps more surprisingly, 2.0×10^{39} of these lexicalisations are syntactically valid Java sentences (see Section 8.1). A practical technique which is going to handle such input numbers needs a representation which allows common parts to be shared and processed together. First we give the machinery which replaces classical strings of lexical tokens with TWE sets. Lexical disambiguation can be performed by removing elements from the TWE set, but there are some limitations in terms of the properties of the reduced sets and these are analysed in Sections 2.2 and 2.3. In Section 2.4 we discuss the 'parser lookahead' sets of a TWE set that have the role that the 'next input symbol' has in a recursive descent parser.

2.1 Indexed token strings - ITS

Definition 2.1 The definition of a language $\mathcal{L}$ includes a set of fundamental characters $\mathcal{A}$ and a set of tokens; each token denotes a set of strings of characters, the token's *pattern*. We call a string of elements of $\mathcal{A}$ a *character string* and a character string in the pattern of a token is called a *lexeme* of that token. A *lexicalisation* of a character string q is a string $t_0 \ldots t_m$ of tokens with the property that there exist $q_i \in t_i$, $0 \le i \le m$, such that $q = q_0 \ldots q_m$. We say that $q_0, \ldots, q_m$ is a *sequence of lexemes corresponding to the lexicalisation* $t_0 \ldots t_m$.

In fact, the notion of lexicalisation is not quite adequate for capturing the full spectrum of lexical outcomes for a given character string. Suppose that the token t has a pattern which consists of all non-empty strings from the character set $\{a, b, c\}$. Then the lexicalisation tt can correspond to $aba = q_0 q_1$ in two ways, with $q_0 = ab$ or $q_1 = ba$.

To capture these alternatives we consider pairs (t, i), where t is a token and i is an integer position in the character string, the right extent of the lexeme matched to t. This allows the two lexicalisations above to be represented

$$(t, 2)(t, 3) \qquad (t, 1)(t, 3)$$

If we replace the pairs with triples, then each string can be represented as a set rather than a string:

$$\{(t, 0, 2), (t, 2, 3)\}, \qquad \{(t, 0, 1), (t, 1, 3)\}$$

We called these triples *tokens with extents*.

Of course there are two other lexicalisations of aba as t and ttt. These have token with extents representations, respectively,

$$\{(t, 0, 3)\} \qquad \text{and} \qquad \{(t, 0, 1), (t, 1, 2), (t, 2, 3)\}$$

A parser that is required to parse all the lexicalisations, t, tt and ttt, may improve efficiency by parsing the strings together and only processing the shared triples $(t, 0, 1)$ and $(t, 2, 3)$, once. More significantly, as mentioned above, this approach allows shared parsing of, say, substrings v and w in the nine token strings of the form $t^i w t^j v$, $1 \le i, j \le 3$, even though these strings do not even have the same number of tokens, because the extents in w and v will always be the same.

Definition 2.2 We call a sequence $u = (t_0, i_0)(t_1, i_1) \ldots (t_k, i_k)$, $0 < i_1 < \ldots < i_k$, of pairs in which each token has a right extent (the end position of the corresponding lexeme in the character string) an *indexed token string*. We call $t_0 t_1 \ldots t_k$ the *underlying token string* of u; we call $\{(t_0, 0, i_1), (t_1, i_1, i_2), \ldots, (t_k, i_{k-1}, i_k)\}$ the *token with extents set* of u; and we say that each triple (t_j, i_{j-1}, i_j) *belongs* to u.

For parsing purposes t_k will be the end of string character, \$, i_{k-1} will be the length, m, of the underlying character string, and $i_k = m + 1$.

As already mentioned, we can represent each indexed token string as a set of token-with-extents triples, and parsing the union of these sets rather than each set individually is where the efficiency is gained. However, the relationship between the set of indexed token strings and the union of the sets of corresponding triples is somewhat complex, as we discuss next.

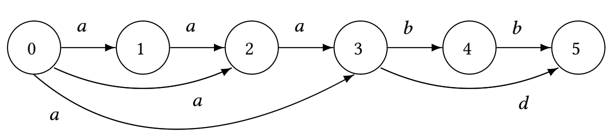
2.2 Sets of tokens with extents

Definition 2.3 A *set of tokens with extents* (TWE set) is a finite set of triples of the form (t, i, j), where t is an element of some specified set of tokens and i, j are integers with $0 \le i < j$. The *height* of a TWE set is the largest integer, m, such that there is a triple (t, i, m) in the set. If the set is empty then the height is 0. For example,

$$\Sigma_0 = \{(a, 0, 1), (a, 0, 2), (a, 0, 3), (a, 1, 2), (a, 2, 3), (b, 3, 4), (b, 4, 5), (d, 3, 5)\}$$

is a TWE set of height 5.

It can help to visualise a TWE set as a directed acyclic graph. The *TWE graph* of a TWE set Σ has a node labelled k for each k such there is a triple of the form (t, i, k) or a triple of the form (t, k, j) in Σ. For each triple $(t, i, j) \in \Sigma$, there is an edge labelled t from node i to node j. For example, the TWE set Σ_0 has TWE graph



Definition 2.4 We say that the indexed token string $(t_0, i_0)(t_1, i_1) \ldots (t_k, i_k)$ is *embedded* in the TWE set Σ if i_k is the height of Σ and the triples $(t_1, 0, i_0), (t_1, i_0, i_1), \ldots, (t_k, i_{k-1}, i_k)$ all belong to Σ. The empty string is the only string

embedded in the empty TWE set, and the empty string is not embedded in any nonempty TWE set. We denote the set of all indexed token strings embedded in Σ by $strings(\Sigma)$.

Definition 2.5 We refer to a set of indexed token strings in which all the strings have the same rightmost index as an ITS (*indexed token string*) set. For an ITS set X, we define the associated TWE set, Σ_X, to be the union of the TWE sets of the strings in X. Note that the height of Σ_X is the rightmost index of the strings in X.

For example, Σ_0 above embeds the set of indexed token strings

$$X = \{ (a,1)(a,2)(a,3)(b,4)(b,5), \quad (a,2)(a,3)(d,5), \quad (a,1)(a,2)(a,3)(d,5), \quad (a,2)(a,3)(b,4)(b,5), \quad (a,3)(b,4)(d,5), \quad (a,3)(d,5) \}$$

X is an ITS set, $\Sigma_X = \Sigma_0$, and $X = strings(\Sigma_0)$.

The rest of this section addresses issues needed to establish the correspondence between a partially disambiguated TWE set and the ITS set expected to be parsed. The reader who wants to focus on the multi-parsing technique can pass over this and proceed to Part 2 of the paper.

Tight And Consistent ITS Sets

A TWE set built from all the lexicalisations of a single character string has well behaved properties. However, once these sets are manipulated, for example when elements are removed for ambiguity reduction or unions of sets are taken to allow wider potential parser sharing, these properties can become compromised. We define the properties that are required for TWE set parsing to correspond to parsing a set of token sequences.

Definition 2.6 A TWE set Σ is *tight* if every triple in Σ belongs to some indexed token string embedded in Σ. An ITS set X is *consistent* if every string embedded in Σ_X is an element of X.

Tightness of the TWE set ensures that a parser does not have to process triples that can never be part of a sentence and consistency of the underlying ITS set ensures that only the sentences in that set are accepted by the parser.

It follows from the structure of the TWE graph that for a TWE set Σ, with height m, the set of strings embedded in Σ is precisely the set of paths from the node labelled 0 to the node m in its TWE graph. Furthermore, the TWE is tight if and only if node 0 is the only source node (node with no in-edges), and node m is the only sink node (node with no out-edges).

2.3 Properties Of Σ_X And $strings(\Sigma)$

Not all TWE sets we may construct are tight, and for some ITS sets X the associated TWE sets embed more that just the strings in X, i.e. X is not consistent.

For example, consider the TWE set $\{(a,0,1),\ (a,0,2),\ (b,1,3)\ (b,2,3)\}$. If we remove $(a,0,1)$ the resulting set $\Sigma = \{(a,0,2),\ (b,1,3),\ (b,2,3)\}$ embeds only the string $(a,2)(b,3)$, which has associated TWE set $\{(a,0,2),\ (b,2,3)\}$. Thus Σ is not tight.

Conversely the set with two indexed lexicalisations $X = \{ (a,2)(b,3),\ (c,2)(d,3) \}$ has TWE set

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

which also embeds $(a,2)(d,3)$ and $(c,2)(b,3)$. Thus X is not consistent.

It is an immediate consequence of the definition that every string in an ITS set, X say, is embedded in the TWE set associated with X, i.e. $X \subseteq strings(\Sigma_X)$.

In the case that the TWE set has redundant triples which do not belong to any embedded indexed token string, i.e. it is not tight, these triples will take unnecessary parse-time space, and possibly parse-time activity. Furthermore, given input Σ, an MGLL parser will parse all the strings in $strings(\Sigma)$, not just those from the ITS set from which Σ was created. Thus we need to reason about these situations.

The following lemmas, whose proofs are given in Appendix A, give a definition of a tight TWE set in terms of the triples rather than the embedded strings and then in terms of the sets Σ_X and $strings(\Sigma)$.

Lemma 1. *A TWE set Σ with height m is tight if and only if, for every element $(t, i, j) \in \Sigma$, (a) $i = 0$ or there is an element $(t', i', i) \in \Sigma$, and (b) $j = m$ or there is an element $(t', j, j') \in \Sigma$.*

Lemma 2. *(i) A TWE set Σ is tight if $\Sigma \subseteq \Sigma_{strings(\Sigma)}$.*
(ii) If the TWE set Σ is tight then $\Sigma_{strings(\Sigma)} = \Sigma$.
(iii) For any ITS set X, Σ_X is tight.
(iv) An ITS set X is consistent if and only if $strings(\Sigma_X) = X$.
(v) For any TWE set Σ, $strings(\Sigma)$ is consistent.

2.4 Representing TWE sets

For parsers whose input is a single token string, the string can be held in an input array and when an input symbol is matched by the parser the next input symbol is obtained simply by incrementing the index pointer. For multiple input parsing, for each token a, say, at position i, the next input tokens are all the tokens of the form (b, k, j) such that there is a token (a, i, k). We need to be able to find these 'next' tokens efficiently. The parser will only need to consider the token and left extent (a, i) and will only need to find the next tokens with their left extents, (b, k). We use the *parser lookahead sets* of a TWE set Σ which are defined as

$$lk\Sigma_{a,i} = \{k \mid (a, i, k) \in \Sigma \text{ and for some } b, j, (b, k, j) \in \Sigma\}$$

To maintain the complexity bound of the parsers, and to ensure that the space taken by the parser lookahead sets is linear in the case where there is only one embedded string, it is important to be able represent and construct these lookahead sets efficiently. To demonstrate that this is possible we discuss one particular representation.

We represent Σ as an array *input*, of dimension m, the height of Σ. The ith element of *input* is a set of pairs of the form $(a, \Sigma_{a,i})$, whose first element is a token and whose second element is a set of integers, $\Sigma_{a,i} = \{k \mid (a, i, k) \in \Sigma\}$. We do not include $(a, \Sigma_{a,i})$ in $input[i]$ if $\Sigma_{a,i} = \emptyset$.

We construct a second array, also of dimension m, whose elements are sets of tokens,

$$t\Sigma_k = \{b \mid \text{for some } j, (b, k, j) \in \Sigma\}$$

Then we have

$$lk\Sigma_{a,i} = \{k \in \Sigma_{a,i} \mid t\Sigma_k \neq \emptyset\}$$

As we mentioned above, when running an MGLL parser we assume that the strings embedded in Σ all end with the end-of-string symbol \$. Thus we include the triple $(\$, m, m + 1)$ in Σ, and so $\{m + 1\} = lk\Sigma_{\$,m}$ and $\{\$\} = t\Sigma_m$.

The worst case size of *input* is $O(m^2)$. In fact, the size of the set $\Sigma_{a,i}$ is the number of edges labelled a, in the graphical representation of Σ, whose source node is i. So the size of the data structures is $O(m \times d)$ where d is the largest number of out-edges of any node in the graphical representation of Σ. Furthermore, d is bounded by the number of strings embedded in Σ and by $O(m)$. Since there is at most one element $(a, i, k) \in \Sigma$ for each a, i, k, elements are added to $\Sigma_{a,i}$ only once, so there is no search associated with element insertion. Thus the construction time for the data structures is $O(m \times d)$, and the size and construction time when there is only one input string is linear in m.

If the graphical representation of the TWE set for Σ has an edge between every pair of nodes then Σ embeds $O(2^m)$ strings, but an MGLL parser parses all these strings together in worst case $O(m^3)$ time and space.

Part 2 - Parsing Multiple Input Strings

We now introduce a fully general parsing technique, MGLL, which will concurrently parse any set of token strings that can be written as a consistent set of indexed token strings (see Section 2.2). The application of MGLL that we have is for parsing multiple lexicalisations of a given character string but the algorithm can take as input any TWE set, Σ. The only assumption that we need to make is that Σ is tight, and then the MGLL parser will parse all the sentences embedded in $strings(\Sigma)$. Even in our applications in this paper, Σ may be a subset of a TWE set corresponding to all lexicalisations of some character string because of ambiguity reduction (see Section 7). However, if the reader has in mind the case where the input TWE set is of the form X_Σ, where X is the set of all indexed lexicalisations of some underlying character string, then this is sufficient to allow a full understanding of the parsing technique.

GLL is an extension of recursive descent parsing in which parse functions are replaced with labelled blocks of code, and calls and returns from these blocks are handled directly using an explicit stack. This allows points of non-determinism in the recursive descent parser to be put on a worklist so that they are all explored by the parser. For this to be effective the call stacks associated with each element on the worklist are combined into a Tomita-style graph structured stack [Tom91, Tom86]. To turn this into a technique that can handle multiple input strings, what in a recursive descent parser are matches to input symbols are treated in MGLL as additional points of non-determinism, and all possible matches are put onto the worklist for subsequent exploration. The details are discussed in Section 4.

Before describing the MGLL parsing algorithm we need to consider how the output derivations will be represented. A common approach is to use a shared packed parse forest (SPPF), [Tom86, BL89]. GLL parsers construct a binarised SPPF that is worst case cubic in size. In Section 3 we give an extended representation that can embed derivations of more than one sentence.

3 REPRESENTING MULTIPLE DERIVATIONS

Generalised parsers typically construct a packed graphical representation of the derivation fragments constructed from a nondeterministic grammar. This representation embeds all the derivations in the case of an ambiguous grammar. We now describe an extension of the representation which embeds derivations of sets of sentences. For a context free grammar Γ and TWE set Σ, we denote by $sen(\Sigma, \Gamma)$ the set of strings embedded in Σ whose underlying token sequences are sentences in Γ. For input Σ, an MGLL parser for Γ will generate an extended shared packed parse forest (ESPPF) which embeds precisely the derivations of the strings in $sen(\Sigma, \Gamma)$. In fact, the standard GLL SPPF [SJ13] construction extends without any additional machinery to generate an ESPPF, we simply need to show how to identify the embedded sentences and to show that they are the elements of $sen(\Sigma, \Gamma)$.

3.1 ESPPF – extended SPPFs for multiple input strings

We begin by describing classical SPPFs to establish notation and to create a base point for the extended definition.

Definition 3.1 A *context free grammar* (CFG) consists of a set $\mathbf{T}$, of terminals, a set $\mathbf{N}$, of nonterminals disjoint from $\mathbf{T}$, a start symbol $S \in \mathbf{N}$, and a set of grammar rules $X ::= \alpha_1 \mid \ldots \mid \alpha_t$, one for each nonterminal $X \in \mathbf{N}$, where each α_k, $1 \leq k \leq t$, is a string over the alphabet $\mathbf{T} \cup \mathbf{N}$. We refer to the α_k as the *production alternates*, or just *alternates*, of X, and to $X ::= \alpha_k$ as a *production rule*, or just a *production*. A *derivation step* is an expansion $\gamma Y \beta \Rightarrow \gamma \alpha \beta$ where $\gamma, \beta \in (\mathbf{T} \cup \mathbf{N})^*$ and α is an alternate of Y. A *derivation* of τ from σ is a sequence $\sigma \Rightarrow \beta_1 \Rightarrow \ldots \Rightarrow \beta_{n-1} \Rightarrow \tau$, also written $\sigma \stackrel{*}{\Rightarrow} \tau$. We say α is *nullable* if $\alpha \stackrel{*}{\Rightarrow} \epsilon$.

Note, the use of terms terminal and nonterminal is standard for context free grammars. The term token is used in lexical analysis and the set of tokens constructed by a lexical analyser form the terminals of the phrase level grammar. Thus 'terminal' and 'token' are used interchangeably. In this part of the paper we shall use the word terminal for consistency with traditional context free grammar terminology, and also because MGLL does not require terminals to have associated lexemes from an underlying character sequence.

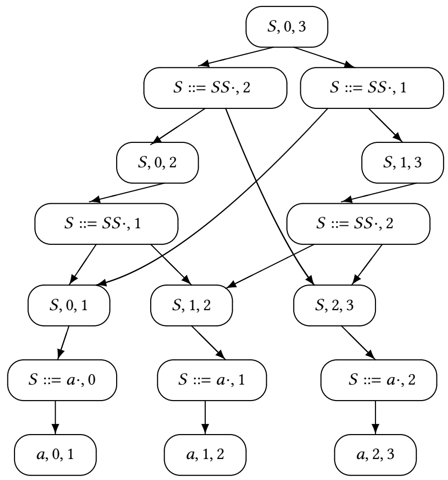
3.1.1 Annotated derivation trees. A *derivation tree* is a graphical representation of a derivation of a sentence in a CFG Γ. It is an ordered tree whose root node is labelled with the start symbol and leaf nodes are labelled with a terminal or ϵ. An interior node is labelled with a nonterminal, X say, and its children are labelled with the symbols of an alternate of X. In order to ultimately share nodes, derivation tree nodes are annotated with integer *extents* to ensure that they are uniquely identified by their labels. *Symbol nodes* are are labelled with triples (x, i, j) where x is a terminal, nonterminal or ϵ. For a classical SPPF, the extents (i, j) correspond to the substring generated by the node, so $x \stackrel{*}{\Rightarrow} a_{i+1} \ldots a_j$.

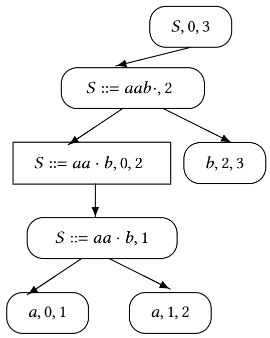
In order to ensure that the parser has worst-case cubic runtime and to allow process descriptors (see below) to contain just one SPPF node, the derivation trees are binarised from the left in the natural way by introducing *intermediate nodes*, as shown in Example 3.1 below. An intermediate node is labelled with a *grammar slot*, a position before or after a terminal or nonterminal on the right hand side of a production rule. We use a 'dot' to indicate a grammar slot, $A ::= \mu \cdot \nu$.

3.1.2 SPPFs. An SPPF is a representation of all of the annotated derivation trees of a string $a_1 \ldots a_n$ with respect to Γ. It is the result of merging all the annotated derivation trees, sharing nodes with the same label. For ambiguous grammars, a symbol or intermediate node can have different families of children in different derivation trees. In the SPPF each family is grouped together under a *packed node*. Packed nodes are labelled with a grammar slot, $X ::= \alpha x \cdot \beta$, and an integer k, the *pivot*. The right child of the packed node will be a node labelled (x, k, j) and the left child, if it exists, will be an intermediate node labelled $(X ::= \alpha \cdot x\beta, i, k)$, or a symbol node (α, i, k), if α has length 1. The yield of the SPPF, the leaves read in left to right order, corresponds to the input string.

For a production of length zero or one, an SPPF node will have only one child. Rather than writing special cases of various functions, we use a special 'dummy' node, denoted by Δ, for the missing child. By convention Δ will always be the left child and it will usually be omitted from displayed graphs.

Example 3.1 The following are the SPPFs for the strings *aaa* and *aab* in the grammar, Γ_1: $S ::= S\ S \mid a \mid a\ a\ b$





The rectangular nodes are the intermediate nodes, and the nodes with one integer in their label are the packed nodes. The SPPF on the right above is also the binarised derivation tree for *aab* in Γ_1. The SPPF on the left is obtained by merging the two annotated binarised derivation trees for *aaa*.

3.1.3 ESPPFs. We now describe the extended SPPF for a TWE set with respect to a grammar. We suppose that we have a context free grammar, Γ say, which has terminal set $\mathbf{T}$. We consider an input TWE set Σ, of height m, and we suppose that

$$\{\ (a_{11}, i_{11})(a_{12}, i_{12})\dots(a_{1j_1}, m), \quad (a_{21}, i_{21})(a_{22}, i_{22})\dots(a_{2j_2}, m), \quad \dots, \quad (a_{d1}, i_{d1})(a_{d2}, i_{d2})\dots(a_{dj_p}, m)\ \}$$

is the corresponding ITS set $strings(\Sigma)$.

For each indexed token string

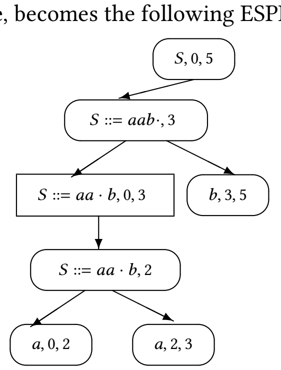
$$(a_{h1}, i_{h1})(a_{h2}, i_{h2})\dots(a_{hj_1}, m)$$

we obtain an ESPPF by simply replacing each extent and pivot value, k, in the labels of the SPPF nodes with i_{hk} (or with 0 or m as appropriate).

For example, if we index the string *aab* as

$$(a, 2)(a, 3)(b, 5)$$

(so $m = 5$) then the SPPF for *aab* in Γ_1, above, becomes the following ESPPF for this ITS:



We combine the ESPPFs for each string in an ITS set into a single ESPPF by sharing nodes.

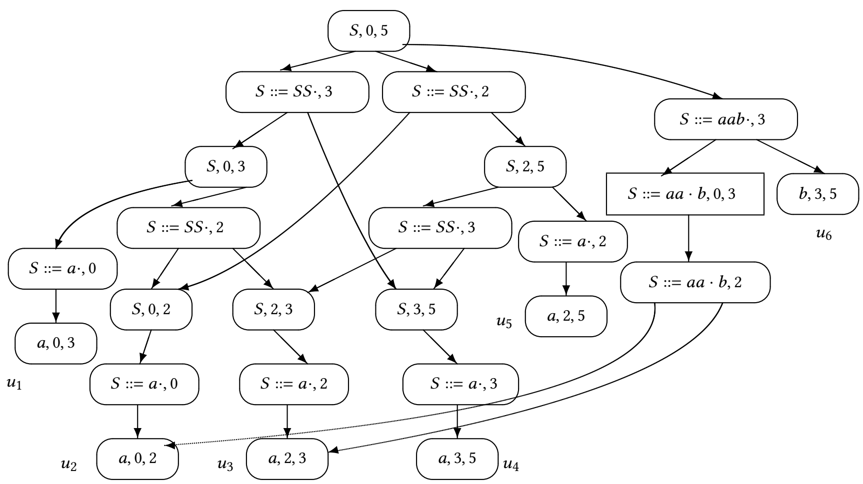
Definition 3.2 The *extended shared packed parse forest (ESPPF)* for Σ in Γ is the graph obtained by taking the union of the ESPPFs for each string in $strings(\Sigma)$. Symbol and intermediate nodes with the same labels are merged as are packed nodes which have the same label and the same parent label. Of course, the ESPPF for Σ contains only derivations of strings in the language of the grammar, the other strings have empty ESPPF.

3.1.4 Example. For the grammar Γ_1, in Example 3.1 above, we consider the TWE set

$$\Sigma \;=\; \{(a,0,2),(a,0,3),(a,0,5),(a,2,3),(a,2,5),(a,3,5),(b,3,5)\}$$

(We can imagine that a is a token whose pattern is all nonempty strings composed from `x`, `yy` and `zz`, and that b is a token whose pattern contains the single string `yy`. Then Σ is the TWE set constructed from all the strings which are indexed lexicalisations of `zzxyy`.)

Then the ESPPF for Σ in Γ_1 is



Of the six indexed token strings in $strings(\Sigma)$ only four have underlying strings which are sentences in the grammar, two corresponding to aa, one to aaa and one to aab. The ESPPFs for these are combined to form the ESPPF above for Σ.

Our ESPPF definition is deliberately declarative. The method by which an ESPPF (and any SPPF) is constructed depends on the parsing algorithm being used. In most cases the construction is bottom up with leaf nodes constructed as input symbols are read, and parent nodes are constructed after the processing of the corresponding grammar rule is completed.

3.1.5 ESPPF sentence finding algorithm. For a traditional SPPF the yield, the sequence of leaf node labels read left to right, gives the input string which generated the SPPF. For an ESPPF it is not quite so easy to read off the set of strings it has parsed. We cannot simply construct strings of leaf nodes by matching the extents. In the example in Section 3.1.4, $(a,0,3)$ and $(b,3,5)$ are leaf nodes whose extents match and end at 5, but ab is not a sentence in the grammar.

We now give a constructive definition of the set $iSentences(\chi)$ of indexed token strings whose derivations are captured in the ESPPF χ of Σ in Γ and the corresponding set $sentences(\chi)$ of underlying sentences in Γ. Of course, $sentences(\chi)$ can be obtained from $iSentences(\chi)$ by just removing the extents from the triples. However, $iSentences(\chi)$ and its subsets will be larger that the corresponding $sentences(\chi)$. So if only the latter is required then the direct construction will be more efficient.

Suppose that we are given a nonempty ESPPF, χ say, with root node w_S, labelled $(S, 0, m)$. For each ESPPF node w, including the packed nodes, we define associated sets iPS_w and PS_w, of strings of triples and terminals (tokens), respectively, which are sets of partial sentences. The sets iPS_w and PS_w are constructed from the corresponding sets of for w's children, so in this sense the algorithm walks the ESPPF from the leaves up.

- For a leaf node, w say, labelled (a, i, j), if $i < j$ set $iPS_w = \{w\}$ and $PS_w = \{a\}$. If $i = j$ set $iPS_w = PS_w = \{\epsilon\}$, where ϵ denotes the empty string.
- For a packed node w, with two children, $y = (t, i, k)$ and $z = (s, k, j)$ where $i \leq k \leq j$, set iPS_w to be $iPS_y \cdot iPS_z$, the set of all strings which are concatenations of some element in iPS_y with some element in iPS_z. Similarly set PS_w to be $PS_y \cdot PS_z$.
- For a packed node w, with one child, $y = (t, i, j)$, set $iPS_w = iPS_y$ and $PS_w = PS_y$.
- For an internal node w, with packed node children $w_1, ..., w_p$, set iPS_w to be the union of iPS_{w_i}, for $i = 1, \dots, p$, and set PS_w to be the union of PS_{w_i}, for $i = 1, \dots, p$.
- Let iPS_{w_S} and PS_{w_S} be the sets associated with the root node $w_S = (S, 0, m)$ of χ. The ESPPF *captures* the derivations of an ITS $(a_1, n_1) \dots (a_{m-1}, n_{m-1})(a_m, m)$ if and only if $(a_1, 0, n_1) \dots (a_{m-1}, n_{m-2}, n_{m-1})(a_m, n_{m-1}, m) \in iPS_{w_S}$. The set of captured strings is denoted by $iSentences(\chi)$, and by $iSentences(w_S)$ and $iSentences(S, 0, m)$, as convenient.

Note that it is easy to see by structural induction that if $u = (x, i, j)$, or if $u = (x, k)$ with parent (x', i, j), and if $i < j$ then the strings in iPS_u are all of the form $(a, i, l) \dots (b, l', j)$, and that PS_u is the set of underlying strings of the indexed token strings in iPS_u.

If we apply the sentence finding procedure to the ESPPF in Example 3.1.4 we get that

$$iPS_{w_S} = \{u_1u_4,\ \ u_2u_3u_4,\ \ u_2u_5,\ \ u_2u_3u_6\}$$

This gives indexed lexicalisations

$$(a, 3)(a, 5), \quad (a, 2)(a, 3)(a, 5), \quad (a, 2)(a, 5), \quad (a, 2)(a, 3)(b, 5)$$

and sentences

$$PS_{w_S} \ = \ \{\ aaa,\ \ aa,\ \ aab\ \}$$

The next theorem follows from the definition of the ESPPF of a TWE set with respect to a grammar, and its proof is given in Appendix A.

Theorem 1. *Let Σ be a tight TWE set and χ be the ESPPF for Σ with respect to a grammar Γ. The indexed token strings encoded in χ are precisely the strings embedded in Σ whose underlying token strings are in the language of the grammar, i.e. $iSentences(\chi) = sen(\Sigma, \Gamma)$.*

4 PARSING MULTIPLE LEXICALISATIONS

We now describe the MGLL parsing algorithm for a context free grammar Γ, which takes as input a TWE set representation, Σ say, of a consistent set of indexed token strings and constructs an ESPPF representation of the derivations of these strings. An MGLL parser can output the set $sen(\Sigma, \Gamma)$, which contains precisely the indexed strings which were successfully parsed. Detailed expositions of the GLL algorithm may be found in [SJ10a] and [SJ13] with supporting material in [JS11a] and [JS11b]. The basic approach is a generalisation of recursive descent parsing.

4.1 Parsing as grammar traversal

The structure of a recursive descent parser follows closely the form of the underlying grammar: terminals are matched to the next input symbol and nonterminals trigger a call to a corresponding parse function. A parse function for a nonterminal, X say, is comprised of a block of code for each alternate of the grammar rule for X, and the standard function call stack handles the return from nonterminal calls. For a general grammar, more than one of the alternates may be valid at the same point in a parse. An MGLL parser captures all the possible choices and explores each in turn. To achieve this the parse functions associated with a recursive descent parser are replaced with algorithm line labels, goto statements and an explicit stack which replaces the function call stack. The stack elements are pairs (L, j), where L is the algorithm line label to be returned to when the stack element is popped, and j is the current input position when (L, j) is created. We call j the *level* of the node (L, j).

The parser configurations are stored as *process descriptors*, which record the current line of the algorithm, the stack-top, input position and ESPPF node. The parser is made efficient by representing all of the separate stacks in a single structure, a graph structured stack (GSS). The individual stacks are merged into the GSS by sharing nodes with the same return label if they are at the same level. The descriptors are stored in a set $\mathcal{U}$, to make sure that the same descriptor is not processed more than once and there is a 'worklist' $\mathcal{R}$, which contains those descriptors in $\mathcal{U}$ which have not yet been processed.

Positions in the parsing algorithm and GSS nodes are labelled with grammar slots as defined above for ESPPF intermediate nodes. A special slot, denoted by L_0, labels the end of the outer loop of the parsing algorithm. We think of L_0 as corresponding to the end of an augmented grammar start rule $S' ::= S \cdot \$$.

When executing, an MGLL parser is essentially traversing the grammar and the ITS strings embedded in the input TWE set. Each traversal has its own associated stack embedded in the GSS which is being constructed. The stack elements are nodes of the graph and there is a directed edge from node u to node v if u is immediately above v on a stack. The edges of the GSS are labelled with an ESPPF node or the dummy node, denoted by Δ (see Section 3.1.2). This node will be the left child of the ESPPF node constructed when the associated subparse is complete.

The parser employs three variables, c_U which holds the current stack top (a GSS node), c_I which holds the current ITS index (i.e TWE element left index) and c_N which holds the current ESPPF node. When a process descriptor is created the values of c_U, c_I and c_N are recorded in the descriptor and when a descriptor is processed in order to continue a traversal, these variables are set using the values in the descriptor. The outer loop of an MGLL parser selects the next descriptor (L, u, i, w), and the parse continues from the line L, with $c_U = u$, $c_I = i$ and $c_N = w$.

The GSS and ESPPF are built using support functions, formally defined in Section 4.5, whose definition is independent of the grammar for which the parser has been built. The function $add()$ creates descriptors and adds them to $\mathcal{U}$ and $\mathcal{R}$, $create()$ pushes return labels onto the stack and $pop()$ takes a GSS node $u = (L, j)$ and 'pops' it: for each edge (u, v) in the GSS it creates a descriptor with code label L and stack node v.

It is possible for new edges to be added to a GSS node, u, after a $pop()$ action has been applied. Thus, when $pop(u, z)$ is called, this action is recorded in a set, $\mathcal{P}$. If a later new edge is added to u, by the function $create()$, then the set $\mathcal{P}$ is inspected and any earlier pop actions associated with u are applied down the new edge.

The ESPPF is built by the functions $getNodeE(i)$ and $getNodeT(a, i, j)$, also defined in Section 4.5, which construct and return ESPPF nodes labelled (ϵ, i, i) and (a, i, j), respectively, and $getNode(L, w, z)$, which creates a parent node with grandchildren w and z.

By calling the sentence finding algorithm described in Section 3.1.5 an MGLL parser can output precisely the ITSs and sentences that were embedded in the input TWE set (and for which it has constructed derivations). However, these sets can in some cases have exponential size, so we do not include sentence reporting in the basic MGLL algorithm. We simply either output the ESPPF constructed or report failure.

For efficiency the algorithm uses the following precomputed subsets of Σ, described in Section 2.4.

$$t\Sigma_k = \{b \mid \text{for some } j, (b,k,j) \in \Sigma\} \qquad lk\Sigma_{a,i} = \{k \in \Sigma_{a,i} \mid t\Sigma_k \neq \emptyset\}$$

The function $testSelect()$, defined in Section 4.4, uses $t\Sigma_k$ with the standard FIRST and FOLLOW sets to limit descriptor creation, and $lk\Sigma_{a,i}$ is used for input symbol matching, as described in Section 4.3.

4.2 Example - an MGLL parser for $S ::= \ S\ S \mid a \mid a\ a\ b$

construct the sets $lk\Sigma_{a,i}$ and $t\Sigma_i$ from Σ
create GSS node $u_0 = (L_0, 0)$
$\mathcal{U} := \emptyset$; $\mathcal{R} := \emptyset$; $\mathcal{P} := \emptyset$
$add(J_S, u_0, 0, \Delta)$
while $\mathcal{R} \neq \emptyset$ {
  remove a descriptor, (L, u, i, w) say, from $\mathcal{R}$
  $c_U := u$; $c_N := w$; $c_I := i$; **goto** L
J_S: **if**$(testSelect(c_I, SS, S, \Sigma))$ { $add(S ::= \cdot SS, c_U, c_I, \Delta)$ }
  if$(testSelect(c_I, a, S, \Sigma))$ { $add(S ::= \cdot a, c_U, c_I, \Delta)$ }
  if$(testSelect(c_I, aab, S, \Sigma))$ { $add(S ::= \cdot aab, c_U, c_I, \Delta)$ }
  goto L_0
$S ::= \cdot SS$:
  $c_U := create(S ::= S \cdot S, c_U, c_I, c_N)$; **goto** J_S
$S ::= S \cdot S$:
  $c_U := create(S ::= SS\cdot, c_U, c_I, c_N)$; **goto** J_S
$S ::= SS\cdot$:
  $pop(c_U, c_N)$; **goto** L_0
$S ::= \cdot a$:
  for each $k \in \Sigma_{a,c_I}$ {
    if$(testSelect(k, \epsilon, S, \Sigma))$ {
      $c_R := getNodeT(a, c_I, k)$
      $c_T := getNode(S ::= a\cdot, c_N, c_R)$
      $add(S ::= a\cdot, c_U, k, c_T)$ } }
  goto L_0
$S ::= a\cdot$:
  $pop(c_U, c_N)$; **goto** L_0
$S ::= \cdot aab$:
  for each $k \in \Sigma_{a,c_I}$ {
    if$(testSelect(k, ab, S, \Sigma))$ {

$c_R := getNodeT(a, c_I, k)$
$c_T := getNode(S ::= a \cdot ab, c_N, c_R)$
$add(S ::= a \cdot ab, c_U, k, c_T)$ } }
goto L_0
$S ::= a \cdot ab$:
for each $k \in \Sigma_{a,c_I}$ {
if($testSelect(k, b, S, \Sigma)$) {
$c_R := getNodeT(a, c_I, k)$
$c_T := getNode(S ::= aa \cdot b, c_N, c_R)$
$add(S ::= aa \cdot b, c_U, k, c_T)$ } }
goto L_0
$S ::= aa \cdot b$:
for each $k \in \Sigma_{b,c_I}$ {
if($testSelect(k, \epsilon, S, \Sigma)$) {
$c_R := getNodeT(b, c_I, k)$
$c_T := getNode(S ::= aab\cdot, c_N, c_R)$
$add(S ::= aab\cdot, c_U, k, c_T)$ } }
goto L_0
$S ::= aab\cdot$:
$pop(c_U, c_N)$; **goto** L_0
L_0: }
if (there exists an ESPPF node $(S, 0, m)$) { return the ESPPF }
else { report failure }

4.3 Terminal matching

The main change required from the original GLL specification is in the ‘matching’ of terminals. In the classical GLL algorithm, the next input symbol is obtained simply by incrementing the input pointer. For the multi-input parser there may be several next terminals and these may have different ‘lengths’ (right extents or next input indexes). In an MGLL parser, process descriptors are created for each possibility.

In order to avoid creating descriptors that will terminate as soon their processing begins, we add a test against the next input symbol. As a consequence, readers who are familiar with classical GLL will notice that we have also moved the positions, in the templates, of other tests to avoid unnecessary repetition.

For a given terminal a, and left extent i, the parser matches all of the triples $(a, i, k) \in \Sigma$ at the same time. There is a potential search cost associated with finding all of the input triples which can follow these triples, i.e. the elements (b, k, j). So initially parser lookahead sets, $lk\Sigma_{a,i}$, are computed; we assume that an efficient data representation and construction process are used, see Section 2.2. Because a lookahead test is performed before creating a descriptor, it is only necessary to store the next index, k, in the descriptors.

We also note here that, although a descriptor is a 4-tuple (L, u, i, w) where $u = (L', k)$ is a GSS node, we do not need the full ESPPF node $w = (L'', p, q)$ because it will always be the case that $p = k$ and $q = i$. Thus there are at most $O(m)$ possible descriptors.

The MGLL parser specification is a set of 'templates' and a parser generator constructs an MGLL parser from the templates by substituting actual grammar symbols, alternates and sets of terminals into the templates. The GSS and ESPPF construction is done by support functions which are independent of the grammar and can be used by any MGLL parser. Finally, there is a function that handles the parser lookahead.

4.4 Lookahead testing functions

The function $testSelect()$ is used for efficiency to guard certain parser actions. It checks whether, at input index i, there is a TWE element whose left extent is i and which lies in the predictor set for the current grammar position. The predict set is based on the standard first and follow sets:

$$\textsc{first}_T(\alpha) = \{t \in \mathbf{T} \mid \alpha \overset{*}{\Rightarrow} t\alpha'\} \qquad \textsc{follow}_T(X) = \{t \in \mathbf{T} \mid S \overset{*}{\Rightarrow} \alpha X t \beta\}$$

$$\textsc{first}(\alpha) = \begin{cases} \textsc{first}_T(\alpha) \cup \{\epsilon\} & \text{if } \alpha \overset{*}{\Rightarrow} \epsilon \\ \textsc{first}_T(\alpha) & \text{otherwise} \end{cases}$$

$$\textsc{follow}(X) = \begin{cases} \textsc{follow}_T(X) \cup \{\$\} & \text{if } S \overset{*}{\Rightarrow} \gamma X \\ \textsc{follow}_T(X) & \text{otherwise} \end{cases}$$

$$predict(\beta, X) = \{b \mid b \in \textsc{first}(\beta) \text{ or } (\epsilon \in \textsc{first}(\beta) \text{ and } b \in \textsc{follow}(X))\}$$

$$testSelect(i, \beta, X, \Sigma) = \begin{cases} true & \text{if } (predict(\beta, X) \cap t\Sigma_i) \neq \emptyset \\ false & \text{otherwise} \end{cases}$$

4.5 GSS and ESPPF constructing functions

We now give the support functions for the MGLL parsers.

$add(L, u, i, w)$ {
  if $((L, u, i, w) \notin \mathcal{U})$ { add (L, u, i, w) to $\mathcal{U}$ and to $\mathcal{R}$ } }

$pop(u, z)$ {
  if $(u \neq u_0$ and $(u, z) \notin \mathcal{P})$ {
    let $L = (Y ::= \nu X \cdot \mu, k)$ be the label of u
    let i be the right extent of z
    add (u, z) to $\mathcal{P}$
    for each GSS edge (u, w, v) {
      let y be the node returned by $getNode(L, w, z)$
      if$(testSelect(i, \mu, Y, \Sigma))$ $add(L, v, i, y)$) } } }

$create(Y ::= \nu X \cdot \mu, u, i, w)$ {
  let L be $Y ::= \nu X \cdot \mu$
  if there is not already a GSS node labelled (L, i) create one

```
    let v be the GSS node labelled (L, i)
    if there is not an edge from v to u labelled w {
        create an edge from v to u labelled w
        for all z such that (v, z) ∈ 𝒫 {
            let y be the node returned by getNode(L, w, z)
            let j be the right extent of z
            if(testSelect(j, μ, Y, Σ)) add(L, v, j, y)) } }
    return v }
```

The functions that build the ESPPF take a grammar slot, L say, as a parameter. The nodes constructed are labelled with slots related to L and the specific construction depends on the type of slot. The following notation is used for the required properties. We say that L is *eoR*, end-of-rule, if L is the end of a production, i.e. of the form $X ::= \alpha\cdot$. We say that L is *fiR*, first-in-rule, if L is not *eoR* and it is of the form $X ::= x.\tau$ where x is a terminal or a nonterminal.[1] Finally, lhs_L denotes the nonterminal on the left hand side of L.

```
getNodeE(i) {
    if there is no ESPPF node y labelled (ϵ, i, i) create one
    return y }

getNodeT(a, i, j) {
    if there is no ESPPF node y labelled (a, i, j) create one
    return y }

getNode(L, z, w) {
    if (L is fiR) { return z }
    else {
        suppose that w has label (q, k, j)
        if (L is eoR)  set  Ω := lhs_L   else  set Ω := L
        if (z = Δ) let i := k  else suppose that z has label (Ω′, i, k)
        if there does not exist an ESPPF node y labelled (Ω, i, j) create one
        if y does not have a child labelled (L, k) {
                    create one with right child w and, if z ≠ Δ, left child z }
    return y } }
```

4.6 MGLL parser templates

MGLL parsers are specified using a set of code templates. The parser for a specific grammar, Γ, is obtained by substituting the nonterminals, terminals and grammar rules of Γ into the templates. The template for the main function $MGLLparse()$ assumes the start nonterminal of Γ is S and the set of nonterminals of Γ is $\{X_1, \ldots, X_p\}$ (the operation of the parser is independent of the order nonterminal code templates $code(X_i)$).

[1]Note this will result in the ESPPF being a multigraph in the case of grammar rules of the form $X ::= AA\gamma$ where A is a nullable nonterminal. The definition of fiR can be modified to exclude slots of the form $X ::= A \cdot A\gamma$ to avoid this if desired.

We assume that there is a TWE set Σ, comprised of triples (a, i, j), where a is a terminal and $0 \leq i < j \leq m$, and a final triple, $(\$, m, m+1)$. The set Σ and the sets $lk\Sigma_{a,i}$ and $t\Sigma_i$ are computed using an efficient representation such as that described in Section 2.2. We use the following notation

m is a constant integer, the height of Σ
c_I is an integer variable whose value is in $\{0, \ldots, m\}$
GSS is a weighted digraph whose nodes are labelled with elements of the form (L, j), where L is a grammar slot or L_0
c_U is a GSS node variable, c_N, c_T and c_R are ESPPF node variables
$\mathcal{P}$ is a set of (GSS node, ESPPF node, integer) triples
$\mathcal{R}$ is a set of descriptors (Grammar slot, GSS node, integer, ESPPF node) yet to be processed
$\mathcal{U}$ is the set of all descriptors constructed so far
$sentence(w)$ is the result of running the sentence finding algorithm on the ESPPF subgraph rooted at w

Template for the main function $MGLLparse(\Sigma)$

$MGLLparse(\Sigma)$=
  construct the sets $lk\Sigma_{a,i}$ and $t\Sigma_i$ from Σ
  create GSS node $u_0 = (L_0, 0)$
  $\mathcal{U} := \emptyset;\ \mathcal{R} := \emptyset;\ \mathcal{P} := \emptyset$
  $add(J_S, u_0, 0, \Delta)$
  while$(\mathcal{R} \neq \emptyset)$ {
    remove a descriptor, (L, u, i, w) say, from $\mathcal{R}$
    $c_U := u;\ c_N := w;\ c_I := i;$ **goto** L
    $code(X_1)$
    ...
    $code(X_p)$
L_0: }
  if (there exists an ESPPF node $(S, 0, m)$) { return the ESPPF }
  else { report failure }

The template for grammar rules

Consider the grammar rule $X ::= \tau_1 \mid \ldots \mid \tau_p,\ 1 \leq i \leq p$. We give the template for $code(X)$ in terms of functions $code(X ::= \cdot\tau_i)$, which will be specified below.

$code(X)$ =
  J_X : **if**$(testSelect(c_I, \tau_1, X, \Sigma))$ { $add(X ::= \cdot\tau_1, c_U, c_I, \Delta)$ }
     ...
     if$(testSelect(c_I, \tau_p, X, \Sigma))$ { $add(X ::= \cdot\tau_p, c_U, c_I, \Delta)$ }
     goto L_0
  $X ::= \cdot\tau_1$: $code(X ::= \cdot\tau_1);\ \ pop(c_U, c_N);$ **goto** L_0
     ...
  $X ::= \cdot\tau_p$: $code(X ::= \cdot\tau_p);\ \ pop(c_U, c_N);$ **goto** L_0

The templates for alternates

In the following X, Y are nonterminals, t is a terminal, and α and β are (possibly empty) strings of terminals and nonterminals.

$$code(X ::= \cdot) \quad = \quad c_R := getNodeE(c_I);\ \ c_N := getNode(X ::= \cdot, c_N, c_R)$$

$$\begin{aligned} code(X ::= \alpha t \cdot \beta) \quad = \quad & \textbf{for}\ \text{each}\ k \in \Sigma_{t,c_I}\ \{ \\ & \quad \textbf{if}(testSelect(k, \beta, X, \Sigma))\ \{ \\ & \quad c_R := getNodeT(t, c_I, k) \\ & \quad c_T := getNode(X ::= \alpha t \cdot \beta, c_N, c_R) \\ & \quad add(X ::= \alpha t \cdot \beta, c_U, k, c_T)\ \}\ \} \\ & \textbf{goto}\ L_0 \\ & X ::= \alpha t \cdot \beta : \end{aligned}$$

$$\begin{aligned} code(X ::= \alpha Y \cdot \beta) \quad = \quad & c_U := create(X ::= \alpha Y \cdot \beta, c_U, c_I, c_N);\ \textbf{goto}\ J_Y \\ & X ::= \alpha Y \cdot \beta : \end{aligned}$$

$$\begin{aligned} code(X ::= \cdot x_1 \dots x_d) \quad = \quad & code(X ::= x_1 \cdot x_2 \dots x_d) \\ & code(X ::= x_1 x_2 \cdot x_3 \dots x_d) \\ & \dots \\ & code(X ::= x_1 x_2 \dots x_d \cdot) \end{aligned}$$

Part 3 - Using The Multi-Parsing Approach

We now look at the application of MGLL parsing to processing multiple lexicalisations of a given character string. We discuss various approaches to constructing a TWE set from the character string. Compiler front ends can treat whitespace in various different ways; we review some of these and discuss their incorporation into the multi-lexer parsing environment. We also provide a formal discussion of lexical ambiguity reduction in a TWE set.

We have implemented all the techniques described in this paper in our ART toolset [JS11b]. We report in Section 8 on data structure sizes and on the impact of various disambiguation choices, using examples from Java. In Section 9 we give some preliminary comparative performance evaluations of the techniques.

The advantage of MGLL is increased expressive power and flexibility, not improved efficiency. The multi-lexer parser approach is fully general and comparisons to other techniques that place restrictions on either the lexicalisations constructed or on the phrase level grammar used require care. The closest correspondence that can be achieved to determinism limited techniques is to construct the TWE set using the standard DFA lexicalisation style, with longest match and priority disambiguation applied, and then to parse the single resulting lexicalisation with a general parser. This models the traditional approach and we can compare examples using this and the full MGLL approach. The results (Section 9) show that the additional overheads from the MGLL approach are not significant.

5 TWE SET CONSTRUCTION FROM CHARACTER STRINGS

Our description of MGLL above does not make any assumptions about where the ITS set and corresponding TWE set are derived from. Currently, our primary application is the lexical phase of compilation, and in this section we briefly discuss the construction of the TWE set Σ_γ which corresponds to the set, LX(γ), of all lexicalisations of the character string $\gamma = x_1 \ldots x_h$.

We shall assume we have a given set of characters $\mathcal{A}$, a set of tokens T, and a specification for the pattern of each token. We assume that no pattern contains the empty string. We want to produce Σ_γ directly from the character string γ, not via the ITS set. Since Σ_γ is the TWE set of LX(γ), if it has an element of the form (t, i, j) then either $i = 0$ or Σ_γ also has an element of the form (t', l, i). So we can begin by creating all elements, $(x_1 \ldots x_j, 0, j)$, such that $x_1 \ldots x_j$ is a lexeme of some token. Then, for values of j from 1 to $m - 1$, if there is already an element of the form (t, i, j), add all elements $(x_{j+1} \ldots x_h, j, h)$ such that $x_{j+1} \ldots x_h$ is a lexeme of some token. Once the construction is complete the set is pruned, as described in Appendix B, which eliminates partial lexicalisations that did not extend to full lexicalisations.

The TWE elements themselves can be constructed using a variety of techniques. Here we briefly discuss both finite state automata and GLL based approaches, and in Section 9.2 we report some corresponding experimental data.

If the patterns of the tokens are defined by regular expressions, we can build the finite state automata in the usual way [ALSU06] and it is easy to use the automata to construct a TWE set which embeds all the ITS set of all lexicalisations of a character string. An example algorithm is given in Appendix B.

Limiting the specification of token patterns to regular expressions makes handling comments, and whitespace in general, harder. If the patterns of the tokens are context free we can construct a grammar whose terminals are the elements of $\mathcal{A}$ and for each token $t \in T$ there is a nonterminal and corresponding grammar rules which generate its pattern. The start symbol, S, then has productions of the form $S ::= t \mid t\ S$, for each $t \in T$. A generalised parser for this grammar can be used to parse γ, and the set Σ_γ is then the set of SPPF node labels of the form (t, j, i).

Of course, parsers are generally more computationally expensive than recognisers as the latter are not required to find all derivations or to construct an output structure. To construct a GLL recogniser from any of the GLL parser family, and thus to get nearer to the efficiency of an automata based TWE constructor, we can simply remove the SPPF construction functionality. We do not need the $getNode()$ functions at all, the GSS does not need to have labelled edges and the descriptors are triples (L, u, i) which do not include an SPPF node. This means that the data structures are smaller and that the parser runs more quickly. To add the TWE generating functionality, the TWE elements are associated with grammar nonterminals that represent the tokens and the TWE elements are output on the return from pop actions on these nonterminals.

These modifications can be made to the MGLL algorithm described above. However, in [SJ18] we have presented an EBNF GLL algorithm which directly implements regular expression constructs such as Kleene and positive closure using iteration rather than recursive grammar rules. This significantly reduces function call stack (GSS) activity and also reduces the number of descriptors that have to be processed. The TWE constructing modifications can be applied to the EBNF GLL templates and this is what we have done to produce the TWE set construction algorithm that we used for the results quoted in this paper. An informal description of the algorithm can be found in Appendix B.

We also note that in a production compiler the lexer does more than just lexicalisation; for example it may retain formatting information for use in error messages. Our presentation assumes that the input has been buffered into a character string and that the left and right extents are indices into that string: any application that retains formatting information can generate such indices at the same time.

6 WHITESPACE HANDLING

Our formulation above gives a clean model for lexical and syntax analysis in cases where all the characters in the input character string are used in lexemes of tokens which are passed to the parser. However, traditional lexers often treat whitespace characters and comments differently, effectively suppressing them from the character string. We briefly discuss some approaches that are possible in a multi-lexer parser. In this section we shall use ws to denote a designated whitespace token which we assume is specified in the same way as the other tokens in T.

6.1 Explicit whitespace handling

Explicit whitespace handling can been achieved cleanly in a multi-lexer parser by treating ws in the same way as the other tokens, simply passing them on to the parser. The grammar is modified to include an optional ws after each instance of the other terminals in the grammar. One advantage is that, by using different whitespace tokens which are inserted in the appropriate places in the grammar, the issues associated with different whitespace conventions in embedded languages can be handled safely. In many cases it is possible to have the whitespace tokens inserted automatically; this approach is worked through in detail in [JSvdB14]. However, the whitespace tokens increase the size of the parser and the size of the data structures it produces, and the approach does not have the clean lexer/parser interface that whitespace suppression achieves.

6.2 Character level grammars

Character level grammars treat whitespace characters like any other character and whitespace-matching nonterminals are defined in the grammar. This is essentially equivalent to the explicit whitespace handling approach for token level grammars. However, whitespace ambiguity significantly increases the size of the parser output structures if they include the derivations from the whitespace nonterminals. The ambiguity has to be resolved using syntax level disambiguation or the whitespace has to be handled using on-the-fly mechanisms that are outwith the pure parsing approaches.

As we shall discuss below, the multi-lexer parser approach can include cleanly defined lexical level disambiguation. Thus multi-lexer parser specification with explicit whitespace handling can provide the power of a character level specification in a more efficient way.

6.3 Whitespace suppression

For indexed token strings whitespace suppression is easy, tokens of the form (ws, j) are simply removed. So, for example,

$$(t_0, i_0)(ws, i_1)(t_2, i_2)(t_3, i_3)(ws, i_4)(t_5, i_5) \quad \text{becomes} \quad (t_0, i_0)(t_2, i_2)(t_3, i_3)(t_5, i_5)$$

To apply the suppression directly on the TWE set we just have to update the extents. For each triple, $(ws, 0, k)$, and for each triple of the form (t, k, j) add the triple $(t, 0, j)$ and delete $(ws, 0, k)$. Then, let j be the smallest integer for which there is a triple (ws, j, k). For each triple of the form (t, i, j) add the triple (t, i, k) and delete (ws, j, k). Continue in this way until there are no triples with a whitespace token and then prune the set. Note, this process preserves the height of the TWE set being modified.

The highly ambiguous nature of whitespace could make the absorption process costly, for example in terms of the number of different extents and the need to avoid the redundancy of absorbing one whitespace token into another. In Section 7 we discuss lexical disambiguation techniques. For languages such as Java, these techniques can be used to eliminate all but at most one whitespace token between non-whitespace tokens, making subsequent whitespace absorption straightforward.

Alternatively, if whitespace is well behaved in the sense that whitespace characters cannot appear at the start of lexemes for other tokens, we could carry out suppression as the TWE set is being constructed, essentially achieving the classical approach of having the lexer suppress whitespace. For a TWE constructor built from an EBNF GLL recogniser this is easy to implement. We simply add the Kleene closure of the set of whitespace characters to the end of the production rules for each token. If whitespace characters cannot appear at the start of a lexeme of any other token then the lookahead in the GLL recogniser will ensure that only lexemes that include all whitespace to their right are matched.

6.4 Embedded whitespace conventions

Of course, the use of whitespace suppression is not always straightforward. For example for embedded languages which have different whitespace and comment conventions from their host language, tokens may be suppressible in some contexts but not others. This is difficult for traditional parsers in which one lexicalisation is selected before parsing, because the context required to decide on suppressibility is not available. In the multi-lexer environment for each whitespace matching substring two lexicalisations can be generated, one in which the token is seen as whitespace and suppressed, and the other in which the token is retained.

Suppose that in the host language comments are enclosed in `(* *)` bracket pairs but in the embedded language these denote strings. We can define two comment and two string tokens, c_h, s_h and c_e, s_e, for the host and embedded languages, respectively. The tokens c_h and c_e are suppressed as described above, while the tokens s_h, s_e are retained. So, for example, `(*end*)` will generate two triples, $(c_h, i, i+7)$ and $(s_e, i, i+7)$, and the former will be suppressed as whitespace. If the 'correct' lexicalisation was the comment then the token s_e will not be valid and the parser will (correctly) reject derivations which attempt to include it. Conversely, if the string was the correct lexicalisation then the parser will reject derivations which omit it, unless the instance of the string token in the embedded grammar was optional.

In the latter case, where there is a token, t say, such that the patterns of ws and t have a common lexeme and there exist sentences of the forms utv and uy, suppression of the token ws is unsafe. Thus, although whitespace suppression in a multi-lexer environment is widely applicable, it cannot be used in all cases.

6.5 Separate whitespace processing

A similar effect to whitespace disambiguation could be achieved by having an initial procedure that takes the input character string, identifies whitespace sequences and replaces them with a single whitespace character such as `\n`. The lexical definition of ws is just the string of length one whose character is `\n`, making either explicit whitespace handling or whitespace absorption relatively simple to adopt. The processing is outwith the theory we have developed for TWE sets and can take any form a user desires.

This is the approach that was used to implement whitespace suppression in the C# case study reported in Walsh's thesis [Wal15].

7 LEXICAL AMBIGUITY REDUCTION

Although MGLL parsers can parse all lexicalisations of an input string in worst case cubic time, parsing all of them and then using syntax level disambiguation to select from those for which the parse succeeds is likely to be slower than is desirable. In contrast to character level parsing, the multi-lexer parser approach allows the number of lexicalisations to be reduced prior to parsing via user-specified lexical disambiguation rules which are applied to the TWE set. This allows the classical case in which only one token string is presented to the parser to be modelled, whilst retaining the flexibility of passing some lexical ambiguity on to the parser if appropriate. The rules we consider suppress some elements from a given TWE set, but may not remove all ambiguities. For this reason we refer to them as lexical ambiguity reduction rules.

7.1 ITS versus TWE set ambiguity redution

Conceptually lexical ambiguity reduction is the removal of strings from an ITS set of lexicalisations. However, as we discussed above, for efficiency reasons we work with TWE sets and our rules will remove triples from a TWE set.

Removing strings from an ITS set is not always equivalent to removing triples from the corresponding TWE set. Removing strings from an ITS set may make it inconsistent, but the corresponding TWE set will embed the smallest enclosing consistent set so some removed strings will be reinstated. Furthermore, removing a triple from a TWE set will remove all the strings which contain that triple. If the ambiguity removal is complete in the sense that only one lexicalisation remains at the end then the same outcome can be achieved by removing either strings from the ITS set or triples from the TWE set. However, it is important to recognise that there are some ITS sets which cannot be constructed by TWE element removal.

7.2 Identifying TWE set ambiguities

Ambiguity in a tight TWE set, Σ say, corresponds exactly to the existence of two or more distinct triples with the same left extent, $(a, i, j), (b, i, k) \in \Sigma$, and, equivalently, to nodes in the graphical representation with more than one out edge. Of course, it also corresponds to the existence of nodes with more than one in-edge, and there are dual 'in-edge' ambiguity reduction rules but we will not consider these here.

7.3 Ambiguity reduction rules

A lexical disambiguation rule is specified in two parts: a relation R on the set of tokens and a condition, *cond*, on the extents. A rule $\{R, cond\}$ is applied to a pair of distinct triples (u, v), The purpose of a disambiguation rule is to remove elements from a TWE set. If $u = (a, i, j)$, $v = (b, i, k)$, and if aRb and $cond(j, k)$ hold, then the rule $\{R, cond\}$ applies to (u, v) and if v is in the TWE set then u is marked for removal (see below).

In practice, a rule may be better thought of as being applied to the left element, u, with the parameter v, because only u is affected by the rule application. We think of R as a priority relation, so bRa may be read as a has priority over b under R.

When specifying the extent condition *cond* we use the convention that a triple will represent its right extent. For example, $\{R, u < v\}$ denotes the rule in which $cond(u, v)$ is the condition that the right extent of u is less than the right extent of v, and if R is defined to be the identity relation aRa this gives longest match (see Section 7.4).

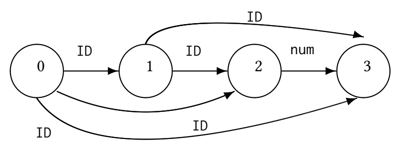
If $u = (b, i, k)$ and $v = (a, i, j)$ then, informally, the rule $\{R, cond\}$ is applies to (u, v) if a has priority over b under R and $cond(u, v)$ is true. It may seem natural to delete the triple u in this case. However this can lead to results which are dependent on the order of application. For example, consider triples $(a, 0, 1), (b, 0, 2), (c, 0, 3)$ and a rule $\{R, u < v\}$ where bRa and cRb but not cRa. Then removing $(b, 0, 2)$ before $(c, 0, 3)$ would leave $(c, 0, 3)$ whereas removing $(c, 0, 3)$ first would ultimately leave just $(a, 0, 1)$. To ensure that the result is independent of the order of application of rules, triples are just marked for removal. Disambiguation rules are applied to the pair (u, v) if u is unmarked and regardless of whether or not v is marked.

Lexical disambiguation Given a TWE set, Σ, and set, $\mathcal{LDR}$, of lexical disambiguation rules, a triple $u = (b, i, k) \in \Sigma$ is marked for deletion if there is a triple $v = (a, i, j) \in \Sigma$ and a rule $\{R, cond\} \in \mathcal{LDR}$ such that bRa and $cond(u, v)$ is true.

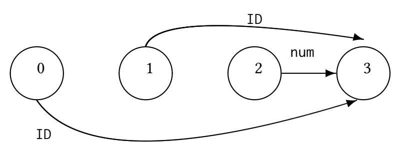
This formalism does restrict disambiguation decisions to being local in the sense that the specification is in terms of only pairs of triples with the same left extent. It cannot include any other contextual information from other parts of the TWE graph. But it is general enough to model the standard longest match and priority style specifications, and other things such as shortest match can also be specified.

7.4 Longest match disambiguation

We can specify a local form of longest match lexical disambiguation for a token, a say, as $\{R^a, u < v\}$, where sR^at if and only if $s = t = a$. Then (b, i, k) is marked for deletion if $b = a$ and there is a triple (a, i, j) such that $k < j$. This rule is commonly used to disambiguate identifier tokens. Suppose that `ID` is a token whose pattern is the set of C-style identifiers. The string *xy1* has a TWE set which is represented graphically as



Applying the lexical disambiguation rule $\{R^{\mathtt{ID}}, u < v\}$ first at node 0, marks $(\mathtt{ID}, 0, 1)$ and $(\mathtt{ID}, 0, 2)$ for deletion. Then applying the rule at node 1 marks $(\mathtt{ID}, 1, 2)$. All the possible applications of all rules have now been applied so the marked triples (edges) are deleted, leaving the TWE set

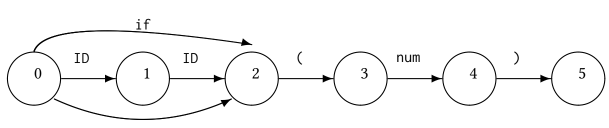


Pruning this set leaves the single triple $(\mathtt{ID}, 0, 3)$ corresponding to the lexicalisation of *ab1* as `ID`.

We refer to this local longest match as Longest Within disambiguation. It is common for longest match to apply across tokens, so that for example, *if1* tokenises to an identifier even in the case that *if* is a keyword. We can specify longest match across all the tokens as $(R^{tot}, u < v)$ where R^{tot} is the total relation over all tokens, $sR^{tot}t$ is true for all tokens s, t. We refer to this as Longest Across disambiguation.

7.5 Token priority based disambiguation

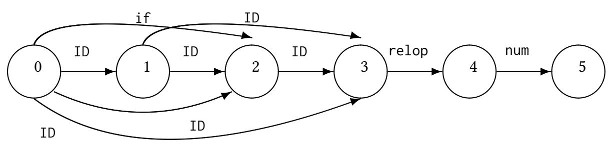
Longest Across does not resolve the situation in which the same lexeme belongs to the pattern of two or more tokens. To give the keyword token `if` priority over the identifier token we can define a lexical disambiguation rule $\{R_1, u = v\}$, where R_1 contains the single element $\mathtt{ID}\, R_1\, \mathtt{if}$. Given tokens `(`, `num` and `)`, the TWE set for *if(1)* is



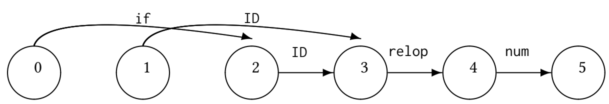
Applying both $\{R^{\mathtt{ID}}, u < v\}$ and $\{R_1, u = v\}$ at node 0 leaves the TWE set which embeds the single string, `if ( num )`.

7.6 Other disambiguation possibilities

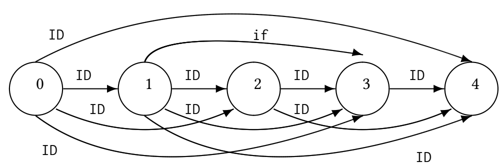
As a further illustration we consider early versions of FORTRAN in which spaces were not significant and, in particular, keywords took priority over longer identifiers. So *ifx>1* was interpreted as `if ID relop num`. We can model this using the disambiguation rule $\{R^{\mathtt{ID}}, u < v\}$ together with the rule $\{R_2, u \leq v\}$ where R_2 has just one element `ID` R_2 `if`. Then *ifx>1* has TWE set



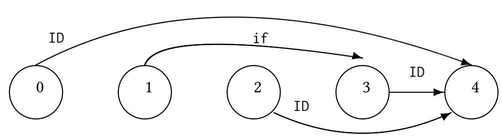
Applying $\{R^{\mathtt{ID}}, u < v\}$ at node 1 and $\{R_2, u \leq v\}$ at node 0, marks the triples $(ID, 0, 2)$, $(ID, 0, 3)$ and $(ID, 1, 2)$. Applying $\{R^{\mathtt{ID}}, u < v\}$ at node 0 marks $(ID, 0, 1)$ and then removing marked edges, gives



Pruning this set leaves the tight TWE set which embeds just the lexicalisation `if ID relop num`. Of course, these disambiguation rules still cause words such as *sift* to be lexicalised as `ID`. The TWE set for *sift* is



Applying the disambiguation rules and removing marked edges gives



Pruning this leaves the TWE set $\{(ID, 0, 4)\}$.

The $\{R, cond\}$ paradigm is powerful because we can specify any condition *cond* that we wish. For example, we could include the condition $i = 2m$ to specify that only even numbered nodes should have the rule applied to them. We can also easily extend the mechanism to look at follow or preceding tokens. For example, $\{R, rextent(u) = lextent(v)\}$ could specify that $u = (b, i, k)$ is marked for deletion if there is a triple $v = (a, k, j)$ such that bRa. In this paper we have just discussed the rules that implement the familiar longest match and priority disambiguations to show how to obtain the standard Lex functionality and to indicate the flexibility of our approach.

The cardinality reductions which result from applying the longest match and priority ambiguity reduction rules to the TWE sets generated for some example Java programs are discussed in Section 8.2.

8 MULTI-LEXER PARSING FOR JAVA

The goal of our work is increased power and flexibility in the lexical specification of formal languages. We are not claiming that our approach is more efficient than classical ‘lex then parse’ techniques. However, the increased power must not compromise the practicality of the technique for real languages. This is clearly a potential cause for concern because the ‘size’ of the problem would be very large if the ‘solution’ was to simply generate all the possible lexicalisations of an input string and parse all of them individually. In this section we examine the lexicalisations of some example Java programs, and report the size of data structures created when these programs are parsed using the MGLL approach and the character level approach.

We report, in Table 1, the total number of lexicalisations of three example strings, showing that parsing all of them is not possible using any of the current single input token-level parsing techniques, but the corresponding TWE set is small and easily parsed using MGLL.

We report, in Table 2, the lower total number of lexicalisations and TWE set sizes that are generated if lexical disambiguation is applied, showing that efficiency gains can be made because of the flexibility of MGLL over character level parsing.

We also compare the MGLL approach with character level approach at the parsing level, reporting, in Table 3, the size of the parser related data structures produced.

We have previously carried out an initial study using a provisional implementation of the approach [SJ19]. This is both updated and extended in this paper, including a large example and the JAVA JLS18 standard specification. We use the following Java programs.

Sample programs

`Life.java` – a 217 line, 5859 character implementation of Conway’s Game of Life.

`Linden.java` – a 40 line, 961 character program that implements a Lindenmayer string rewriter.

`Sand.java` – a 276 line, 5685 character parser generator used for exploring backtracking recursive descent parsing.

`ListViewTest.java` – a 1601 line, 64537 character program from a JavaFX open source library.

Whitespace treatment

To illustrate the flexibility of whitespace handling, in our initial experiments [SJ19] whitespace was handled explicitly, as described in Section 6.1, but disambiguated so that only a single token was constructed for each contiguous sequence of whitespace characters.

In this paper we have additionally generated the token numbers for the case where whitespace is discarded, as described in Section 6.3, as this is the more typical use case, and for the full whitespace character lists (i.e. explicit whitespace handling without the disambiguation) because this models the character level parsing approach.

Java specification versions

The lexicalisations depend on the language specification, and for the data in Sections 8.1 and 8.2 we have primarily used the original Java specification [GJS96]. The specification is written in BNF without closure operators, so we could use it almost directly with a BNF MGLL parser, and also the division between lexical and parse levels in this version makes the total number of lexicalisations for an input program calculatable (see Section 8.1).

We have also implemented the 2022, JLS18, Java specification [GJS+22], in which we have replaced the EBNF constructs with BNF ones. In the original Java JLS1 specification there is only one Identifier terminal, but in JLS18 there are three. This creates a large increase in the number of lexicalisations, and these cannot be resolved by lexical disambiguation. However, the corresponding TWE sets are still small and can be parsed by an MGLL parser for JLS18. This shows the importance of the TWE/MGLL approach as no parser that requires a single token string input can use the JLS18 specification directly.

8.1 Lexicalisation data

In this section we show data for the total number of lexicalisations and the total number of tokens that would have to be processed if the lexicalisations were parsed independently. The sizes make it clear that such an approach is not possible. We have also shown the corresponding TWE set sizes, which are small and manageable for even a prototype MGLL implementation.

The data in Tables 1 and 2 are for the Java JLS1 specification [GJS96]. We have computed the numbers of lexicalisations by building the graphical representation of the TWE set and then computing from that the number of paths from the start node to the end node. We cannot count these paths individually, but the nature of lexicalisations is that they re-converge at intervals along the graph. We can count the number of paths in each of these so-called 'segments' and then the total number of paths is the product of the sizes of the segments.

The total numbers of lexicalisations and indexed lexicalisations are shown in the third and fifth columns of Table 1. The fourth column gives the total number of tokens in all the strings in the third column. These numbers illustrate that simply parsing all lexicalisations is not a practical option. The TWE size, sixth column, is very much smaller than the corresponding total number of tokens, and is clearly tractable.

The Java JLS18 specification [GJS+22] has three versions of identifier, two of which are subsets of the full identifier class. This creates a lexical ambiguity for every identifier in our input examples and makes the number of lexicalisations so large that even our segments based approach cannot compute the total number. However, as shown in the rightmost column (JLS18 TWE set size), the corresponding TWE sets are still small, and they can be parsed by an MGLL parser for JLS18.

ListViewTest.java is not accepted by the original Java grammar, so we cannot generate lexicalistion counts. But the impracticality is sufficiently demonstrated by the smaller strings. We note, however, that the size for the whitespace discarded TWE set for ListViewTest.java is 980,999, which is easily handled by the MGLL parser, as shown in Table 3.

8.2 Lexical ambiguity

Perhaps surprisingly, as shown in the sentences column of Table 1, a large number of the alternative lexicalisations are syntactically correct. Effectively, in the traditional approach, the lexical analyser is carrying out quite a lot of syntactic disambiguation. For a character level parser, the equivalent disambiguation has to be carried out by the parser and this is not straightforward. Whitespace is a particular problem because, unlike identifiers, any sequence of whitespace tokens is legal where one is legal, so each different lexicalisation of each whitespace string creates a further sentence. The MGLL parser built the full whitespace undisambiguated SPPF but the recursive sentence counting procedure we used was not able to complete the required multiple traversals.

The MGLL approach allows a split between disambiguation at the lexical (TWE set generation) level and at the parse level. This flexibility allows the compiler designer to use more efficient lexical level disambiguation where they can but to have the power to perform lexical disambiguation at the syntax level where it is appropriate. For example, whitespace and identifiers can be disambiguated at TWE set level using a longest match strategy but disambiguation of

input	input length	token strings	tokens to be handled	indexed token strings	TWE set size	sentences	JLS18 TWE set size
Whitespace Compressed							
Life	5859	2×10^{387}	7.5×10^{390}	1×10^{759}	15197	2.0×10^{39}	41946
Linden	961	3×10^{72}	1.9×10^{75}	1×10^{121}	1961	512	5305
Sand	5685	2×10^{371}	6.5×10^{374}	3×10^{728}	14834	4.5×10^{27}	41085
Whitespace Discarded							
Life	5859	2×10^{387}	5.9×10^{390}	1×10^{759}	14538	2.0×10^{39}	41287
Linden	961	3×10^{72}	1.6×10^{75}	1×10^{121}	1866	512	5210
Sand	5685	2×10^{371}	5.3×10^{374}	3×10^{728}	14264	4.5×10^{27}	40515
Whitespace Not Pre-disambiguated							
Life	5859	9×10^{527}	3.7×10^{531}	9×10^{1069}	20652	–	47401
Linden	961	2×10^{96}	1.8×10^{99}	5×10^{174}	2867	–	6211
Sand	5685	2×10^{530}	7.7×10^{533}	8×10^{1065}	19991	–	46242

Table 1. Java lexicalisations

input	token strings	tokens to be handled	indexed token strings	TWE set size	sentences	JLS18 TWE set size
Life (LM)	7×10^{65}	1.0×10^{69}	7×10^{65}	1711	1.5×10^{20}	2910
Life (P)	1×10^{370}	4.8×10^{373}	1×10^{745}	14325	2985984	40648
Life (LP)	1×10^{18}	1.7×10^{21}	1×10^{18}	1551	1	2430
Linden (LM)	3×10^{13}	8.8×10^{15}	3×10^{13}	308	32	548
Linden (P)	2×10^{70}	9.0×10^{72}	3145728	1822	4	5078
Linden (LP)	4×10^{6}	1.1×10^{9}	4×10^{6}	285	1	479
Sand (LM)	8×10^{73}	1.1×10^{77}	8×10^{73}	1593	1.1×10^{18}	2685
Sand (P)	1×10^{357}	2.5×10^{360}	7×10^{718}	14061	3145728	39906
Sand (LP)	3×10^{28}	5.1×10^{31}	3×10^{28}	1442	1	2232

Table 2. Java lexicalisations with partial disambiguation (whitespace discarded)

keywords and identifiers could be passed to the parser, allowing keywords to be used as identifiers where there is no syntactic conflict.

Table 2 shows the data for the cases where longest match within tokens is applied to the TWE set (LM) and both longest match and keyword priority are applied (LP), as described in Section 7. In the Java specification the Identifier token explicitly excludes the keywords, and we have shown the impact of this by applying only keyword priority to the TWE set (P). We have only shown the data for the whitespace discarded case. Longest match fully disambiguates whitespace, and the whitespace compressed data is only significantly different from the discarded case in the size of the TWE sets.

The reader may be surprised that so many syntactically correct sentences remain after longest match is applied. This is because identifiers can appear in the same context as certain Java keywords such as `this`. Keyword priority also does not resolve all ambiguity. For example there are places in a Java program where both one and two identifiers can appear. Using both longest match and keyword priority results in a single sentence in each input string.

We also remark that in data reported here the effect of replacing a whitespace sequence with a single lexeme of length one was achieved in the lexer using longest match disambiguation rather than using an initial processor. The code examples used had no comments so longest match disambiguation was sufficient. We used the same specification grammar for all the whitespace variations; our implementation includes an automatic whitespace nonterminal insertion capability.

8.3 MGLL versus character level Java parsing

As we have already discussed, it is possible to dispense completely with separate lexical analysis and to write a grammar whose terminals are effectively the ASCII characters. If the grammar contains nonterminals corresponding to the tokens of a traditional grammar specification then the character level SPPF effectively embeds all the lexicalisations of the input and all their derivations.

Many of the lexicalisations do not correspond to syntactically correct strings and these will be rejected by the parser. However there will also be many syntactically correct lexicalisations. For example, syntactically, an identifier can appear anywhere that the keyword `this` can appear in a Java program. Under the character level grammar approach, these ambiguities have to be removed using syntax level disambiguation.

Syntactic ambiguities can all be identified as nodes in the SPPF that have more than one packed node child. However, the position of the multiple packed nodes does not always indicate the cause of the problem, particularly when the conflict is between identifiers and keywords.

For example, in the Java character level grammar the interchangeability of the keyword `void` and an identifier may ultimately appear as an ambiguity under the nonterminal `ClassBodyDeclaration` which has grammar rule

```
ClassBodyDeclaration::= ConstructorDeclaration | ClassMemberDeclaration
```

Thus identifying the place to apply, in a character level SPPF, what would be lexical level disambiguation in a standard lexer/parse set-up can be difficult.

Of course we can carry out the disambiguation for specific inputs, and we have done so in order to compare the sizes of the parser data structures required for a character level Java parser with a corresponding TWE based parser.

In order to allow us to report data for a large program taken from an external source, we have used the Java JLS18 specification for the data reported, in Table 3, for the Life.java and ListViewTest.java strings. In each case, the first line displays the number of SPPF nodes generated by a character level Java grammar (SPPF full/nodes), the number of remaining nodes after complete disambiguation (SPPF disambig/nodes) and the size of the descriptor set $\mathcal{U}$ (descriptor/set size). The full SPPF node number is, together with the descriptor set size, a measure of the space required by the parser. The disambiguated nodes number is a measure of the size of the data structure that will be output to a downstream process. A GLL style parser has an outer loop that processes each descriptor and so the size of the descriptor set is a measure of the relative run-time cost of the process.

With the TWE approach we have many choices over where disambiguation is carried out and how whitespace is handled. For our data reporting we have used two white space options: whitespace compression in a way that corresponds to the character level model and discarding whitespace in the way normally deployed by compilers with separated lexers. The latter option is what we would normally expect to use, however this is not an option for a character level parser so we have included the former option to demonstrate that even with this approach the TWE/MGLL data structures are smaller than the character level ones.

As we have said, the only option for a character level parser is syntax level disambiguation. For the TWE/MGLL parser we have correspondingly collected data for the case where no TWE disambiguation is applied. We have also applied 'full' TWE disambiguation using longest match and priority. In the latter case the disambiguated SPPF constructed

	SPPF full nodes	SPPF disambig nodes	descriptor set size
Life.java (217 lines, 5859 characters)			
Character level GLL	164462	39984	356820
MGLL, compressed WS			
undisambiguated TWE set	91348	23238	382265
disambiguated TWE set	38077	23238	221540
MGLL, discarded WS			
undisambiguated TWE set	69996	17217	328224
disambiguated TWE set	28689	17217	204363
ListViewTest.java (1601 lines, 64537 characters)			
Character level GLL	2735250	495896	4932221
MGLL, compressed WS			
undisambiguated TWE set	1405062	248779	4757653
disambiguated TWE set	406689	248439	2178706
MGLL, discarded WS			
undisambiguated TWE set	1077525	183133	3900840
disambiguated TWE set	301920	182823	2003859

Table 3. SPPF and descriptor set sizes for Life.java and ListViewTest.java

has the same size as the disambiguated SPPF for the case where the TWE set is not disambiguated, but the size of the descriptor set is much smaller, i.e. the parser is doing less work to produce the same output. The size of the full SPPF for the TWE disambiguated cases is slightly larger than the disambiguated one. This is not due to disambiguation. It arises because the disambiguated count includes only those nodes that are reachable from the root while the full SPPF count includes those nodes constructed by parse threads that ultimately did not succeed. Compared to the character level approach, the TWE based approaches all generate significantly lower SPPF sizes and descriptor numbers (the former drives the data storage requirements and the latter drives the number of executions of the parser outer loop).

9 PERFORMANCE EVALUATION ISSUES

In this section we present an initial evaluation of the utility of the multi-lexer parsing approach. A complete evaluation of our approach requires two multi-dimensional spaces to be explored: (i) the application space and (ii) the engineering optimisation space. In Section 9.2 we present some early stages of (i). We briefly discuss (ii) in Section 10.2.

We cannot compare the multi-lexer approach with traditional Lex/Yacc technology, or with more general but still limited techniques such as PEGs [For04] or the extended lookahead LL(*) [PF11] approach. These do not even provide multiple derivations of one input sentence: at each step phrase level ambiguity is 'resolved' by selecting one derivation to proceed with, for example by removing conflicts in the case of an LR parser or by selecting the first successful match in the case of PEGs. Ultimately only one derivation is constructed, and therefore only one lexicalisation would be parsed even if several could be input. Parsing multiple sentences will always create multiple derivations even when the sentences themselves are unambiguous.

However, we can experimentally compare the GLL recogniser TWE set construction techniques with one based on the traditional Lex-style DFA approach. We can also use the latter, with longest match and priority disambiguation, and with a GLL parser applied to the single resulting lexicalisation, for comparison with the full MGLL approach. We return to this in Section 9.3.

9.1 Pragmatics

For compilation, the application space comprises different programming languages and their partitioning into lexical and parse level components. As we have already described in some detail, the precise choice of interface between lexical and parsing stages can have a significant impact on the size of TWE sets and SPPFs constructed in the overall parsing chain. In the 'interface' we include the nomination of tokens, choosers applied to the TWE set, choosers applied to the SPPF and, ultimately, the design of the language itself and its grammar.

At present we have an implementation which allows exploration of the application space. Some parts are tightly engineered, many (especially the TWE set and chooser processes) are not, relying as they do on generic Java API functionality. As a trivial example: in several places we want to perform a pairwise comparison over a set of tuples. A tight implementation would represent these sets as arrays of tuples, allowing all pairs to be checked in time $n^2/2$ (where n is the cardinality of the set). In our present implementation we are constrained by the API iterators to perform two nested complete iterations, which executes in time n^2 and also suffers the overhead of the iterator.

Although the current implementation is designed for experimental flexibility in the application space, we can of course take performance measurements from it, and we report some here.

9.2 Experimental scheme and prototype measurements

A GLL parser working on a character level grammar offers full generality. As we have discussed in Section 8.3, an MGLL parser working on a TWE set built by some multi-lexing technology offers three advantages over character-level GLL: (i) smaller data structures, (ii) higher throughput and (iii) the opportunity to specify lexical disambiguations in a way that is quite natural, but which would be hard to express as operations within a character level SPPF. In this section we describe an experimental model which can be used for measuring throughput for the multi-lexer parsing approach.

We have five computational processes that we consider: (i) lexicalisation using a GLL recogniser which builds a TWE set, (ii) MGLL parsing using a full TWE set, (iii) lexical choice to selectively remove elements of a TWE set, (iv) MGLL parsing using that reduced TWE set and (vi) lexicalisation using a Deterministic Finite Automaton which builds a TWE set. This last is only applicable to languages that have a regular lexical specification: it would, for instance, be unable to handle languages which have nested comments.

9.2.1 Experimental software. The software used is part of our ART tool [JS11b] which encourages Ambiguity Resilient Translation – our term for systems that allow ambiguities to be 'carried forward' in the translation process to the point where they are most naturally resolved. Full source code along with the corpora of grammars and source examples that we have used may be found at https://github.com/AJohnstone2007/MultipleInputParsingSnapshot

For this paper, we use Java JLS1 and JLS18 grammars which have been automatically extracted from the specification documents using a handcrafted converter from typeset text to ART specification: the converter is itself part of ART and a detailed description of our extraction process is part of the online corpus.

9.2.2 Test corpus. We use two styles of inputs when testing parsers: a small set of standard programs that we have used in many previous publications and which provide comparability of results concerning the size of SPPFs and other data structures; and bulk testing using complete sources for major open source software packages which we use to increase confidence in our grammar extraction processes. For this paper we have thus used our standard set of small programs for continuity, and also tested all lexer/parser combinations on the source code for Open-JFX.

Java FX is one of the most complex Java APIs supporting, as it does, 2D and 3D graphics and a large collection of GUI widgets, observable data structures and even a complete web browser. Although now open source, the original code base was developed as part of Oracle's main Java offering, and makes extensive use of more modern parts of Java such as lambda expressions.

Our JFX corpus comprises 4,588 files totalling 43,568,050 bytes of source. We maintain two versions: the full version, and a version in which each file has been whitespace-normalised by replacing each run of whitespace with either a single space, or a single newline character depending on whether the run extends beyond a single line. The purpose of the whitespace-normalised files is to side-step the issues concerning disambiguation of whitespace runs, as discussed in Section 6.5, so that the timings for our MGLL experiments illustrate only the impact of language level lexical disambiguation. The total size of the whitespace-normalised corpus is 23,634,044 bytes.

Each of our lexer/parser combinations successfully parses all files in both versions of the JFX corpus except for one file (`JavaScriptBridgeTest.java`) which contains an illegal character constant that is two characters long. We note that this file is also rejected by the Oracle javac compiler.

We have selected seven test inputs with lengths ascending from less than 1kByte to over 200kBytes for this study: test scripts and the rest of the corpus are available online. Test data reported here refers to the whitespace-normalised versions of the files.

9.2.3 Experimental hardware, system software and timing regime. Measurements were made using a DELL XPS 15 9510 laptop with 16GByte of installed memory and an Intel Core i7-11800H eight-core processor running at 2.3GHz. The experiments were run from the command line under Microsoft Windows 10 Enterprise version 10.0.19042 using Oracle's Java HotSpot(TM) 64-Bit Server VM (build 14.0.2+12-46, mixed mode, sharing).

The nanosecond timing routines in the Java System API do not accurately reflect computational load in multicore systems and can even return negative values. As a result we used the System.currentTimeMillis() to measure runtimes. Timings under Windows based JVMs can display a broad distribution with variations of ±10% being typical. This variation does reflect day-to-day behaviour of these kinds of systems. Therefore, for each experiment we made 30 runs and report here the max, median and min run times in milliseconds.

9.2.4 Results. Table 4 shows the runtimes in milliseconds for each of our inputs and each of the five computational processes. In each cell we show the maximum, median and minimum times from 30 iterations of the process. Each iteration comprised a cold start for the JVM, so for short strings we expect Java warmup effects to be in play, and that is particularly evident for the DFA lexer column. Conversely, general parsing algorithms like MGLL have a cubic worst case runtime on highly ambiguous grammars. However, programming language grammars, even modern ones like JLS18 which display nontrivial lexical ambiguity, are unlikely to trigger the worst case.

An MGLL parser comprises a three stage pipeline: multilex to TWE set; apply choosers to TWE set; multiparse from TWE set to ESPPF. An important performance requirement is that the overall process scales up to large strings. To demonstrate this, the rightmost column in Table 4 shows the median throughput, calculated as the sum of the median DFA lex, choice and MGLL parse times divided by the length of the string. Since timings are in milliseconds, this gives throughput in kByte per second.

Throughput is around 20kByte s^{-1} for all inputs except CssParser which is significantly better. We speculate that this is because the CssParser source code is dominated by simple test expressions which induce less lexical ambiguity

Test input	length	GLL lex	MGLL full	Choose	MGLL dis	DFA lex	Throughput
		26	47	9	24	39	
LindenMayer	961	25	43	9	23	10	22.9
		23	39	8	21	8	
		193	252	44	170	134	
Sandbox	5685	188	244	37	161	69	21.3
		165	217	33	153	65	
		218	292	38	204	226	
LifeStrFix	5859	211	273	36	188	66	20.2
		203	219	33	151	65	
		302	366	88	296	377	
MultipleGradientPaintContext	10314	294	360	85	263	186	19.3
		273	326	82	245	182	
		2172	2099	626	1001	1453	
ListViewTest	64540	2142	1894	616	917	1168	23.9
		2121	1797	560	843	1142	
		3441	3855	551	1475	2184	
CssParser	122534	3162	3743	525	1438	1840	32.2
		3130	3375	510	1419	1811	
		15749	8870	2262	2556	5305	
TreeTableViewTest	216893	15521	8702	2090	2397	5054	22.7
		15428	8545	1991	2354	4918	

Table 4. Runtime in milliseconds and throughput in kByte per second for DFA based lexing and MGLL parsing

than the other inputs. This hypothesis is supported by the relatively short lexical choice runtime which indicates that the full TWE set for CssParser has low ambiguity.

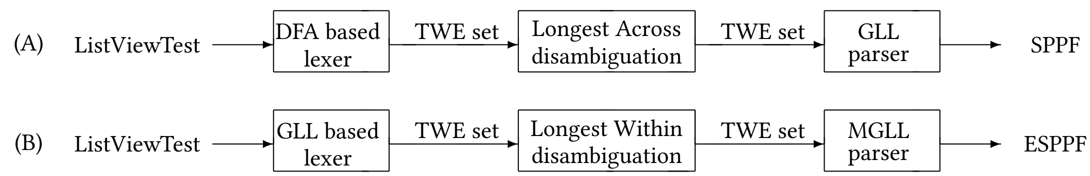
There are some other effects displayed in the table that merit further analysis. The GLL based recogniser does not have to create SPPF nodes and only uses the subgrammars whose start symbols are the non-terminals that correspond to tokens, so it will be significantly faster than a GLL character level parser which has to carry out the equivalent lexical analysis. When we compare the GLL based lexer with the DFA based lexer we can see that its performance is good for inputs up to at least 10kbytes. However for longer strings, the performance of GLL Lex reduces. We hypothesise that is because of congestion in the hash tables that store our GSS structures leading to saturation; further engineering work is required to investigate this. Conversely, the speedup of MGLL running on the disambiguated TWE set relative to MGLL on the full set increases for very long strings. It is possible that the reduced size of the TWE set is keeping the disambiguated MGLL parse small enough to avoid the postulated saturation effect.

9.3 The multi-lexer parsing landscape

The multi-lexer parsing approach allows the specifier to make choices at three points: the method used to construct the TWE set, the amount of disambiguation applied to the TWE set before parsing, and the choice of parsing technique. Using a DFA-based TWE set constructor (lexer) is closest to the traditional techniques, but requires the specification of token patterns via regular languages. If lexical disambiguation leaves a single lexicalisation then this can be input to any parser which is applicable to the syntax level grammar. In particular any fully general parsing technique such as GLL, GLR or Earley parser can be used.

However, as we discussed in Section 1.2.2, full lexical disambiguation may leave only a lexicalisation which is not syntactically valid. The MGLL based multi-lexer parser allows some lexical pre-disambiguation while permitting more than one lexicalisation to remain prior to parsing. This allows the parser to resolve some lexical ambiguity using the syntactic context. Furthermore, applying some degree of TWE set disambiguation can reduce the work required of the parser, and can also reduce post parse disambiguation requirements.

In this section we compare (A) a DFA TWE set builder producing a single lexicalisation which is input to a GLL parser, to (B) a GLL TWE set builder with only partical lexical disambiguation and an MGLL parser, on the Java JLS18 specification and the ListViewTest.java program.



We have used a GLL parser because the JLS18 grammar as specified in the standard is ambiguous, so we need a generalised parser, and MGLL degrades to GLL when there is only one lexicalisation embedded in the TWE set. We have used Longest Across lexicial disambiguation, which applies longest match both within and across tokens in the classical LEX style, for the GLL parser input to reduce the TWE set to contain a single lexicalisation[2]. We have used Longest Within for the MGLL parser input so that longest match is applied only to lexemes of the same token, leaving lexical ambiguities between tokens where they cannot be resolved correctly by priority specification.

The Java JLS18 specification has three identifier terminals and it is not possible to apply systematic lexical level disambiguation to get a single lexicalisation, but this is required if we want to use an existing non-MGLL parsing technology. So we have modified the JLS18 specification so that `TypeIdentifier` and `UnqualifiedMethodIdentifier` are nonterminals that derive the terminal `Identifier`. We refer to this as the identifier merged specification.

MGLL can parse the original specification and 'select' the correct identifier from the TWE set using the syntax context. So we have taken the runtime data quoted in the previous section and added data structure information to also provide the GLL lexer/MGLL parser configuration with the unmodified JLS18 specification for comparison. The three lines in Table 5 thus represent:

(i) modified input string (» replaced with > >) and modified JLS18 specification (identifier terminals merged)

(ii) unmodified input string and modified JLS18 specification

(ii) unmodified input string and unmodified JLS18 specification.

The three Identifier tokens have almost identical patterns, so of course the number of lexicalisations and the size of the full TWE set for (iii) are higher, corresponding to the higher level of lexical ambiguity. The ambiguity is retained after the TWE set disambiguation and resolved by the parser. So the disambiguated TWE set is larger and the parse time is slightly longer. However, unlike (i) and (ii), the full JLS18 specification is being used, and no modification to the input string is needed. We also note that, of course, we can use the DFA based TWE set builder with the MGLL parser and any of the disambiguation choices, and the DFA and MGLL numbers reported in Table 5 will remain the same.

[2]Longest match disambiguation is not correct for some instances of » in a java program. We have added spaces between the seven instances of » in ListViewTest.java to get a correct single lexicalisation. This is not required for, or applied to, the case where we use Longest Within and MGLL.

	full TWE set	indexed lexicalisations	lexer runtime	disambig TWE set	disambig indexed lexicalisations	disambig runtime	SPPF nodes	descriptor set size	parse runtime
(i)	337231	6×10^{12115}	693ms	14952	1	160ms	293148	1233158	673ms
DFA Lexer, Longest Across disambiguation, GLL parser, identifier merged JLS18 specification									
(ii)	337231	6×10^{12115}	1044ms	56415	1×10^{187}	145ms	321078	1419114	770ms
GLL ENBF Lexer, Longest Within disambiguation, MGLL parser, identifier merged JLS18 spefication									
(iii)	980999	1×10^{26323}	2142ms	143795	2×10^{2317}	616ms	301920	2003859	917ms
GLL ENBF Lexer, Longest Within disambiguation, MGLL parser, JLS18 standard specification									

Table 5. Data for ListViewTest.java

10 DISCUSSION AND FURTHER WORK

In this section we note some generalisations that, for simplicity, were not detailed in the presentation above and we also highlight some potential extensions and applications.

10.1 General MGLL application

Multi-parsing without multi-lexing As we have said, parsing multiple lexicalisations of an underlying character string is our primary application for multi-parsing. However, MGLL can parse any set of sentences that are exactly the syntactically correct strings in some representation as a consistent ITS set. Although we do not currently have any substantial examples (the technique is still quite new) we give a simple illustrative example.

Consider the set of strings of the form $ca^k b^h d$, where k, h are integers greater than 0. This set is the language, $L(\Gamma)$, defined by the grammar, Γ, whose rules are

$$S ::= c\ A\ B\ d \qquad A ::= a\ A \ \mid\ a \qquad B ::= b\ B\ \mid\ b$$

Suppose that we wish to parse all the strings in $L(\Gamma)$ in which the embedded string of a's and b's has length at most 4.

$$L_4 \ = \ \{caaabd, cabbbd, cabd, caabbd, caabd, cabbd\}$$

We can choose indexed token strings corresponding to the required strings as follows

$$X = \{\ (c,1)(a,2)(a,3)(a,4)(b,5)(d,6), \quad (c,1)(a,2)(b,3)(b,4)(b,5)(d,6), \quad (c,1)(a,4)(b,5)(d,6),$$
$$(c,1)(a,2)(a,3)(b,4)(b,5)(d,6), \quad (c,1)(a,3)(a,4)(b,5)(d,6), \quad (c,1)(a,3)(b,4)(b,5)(d,6)\ \}$$

The associated TWE set Σ_X is

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

In fact X is not consistent, Σ_X also embeds the indexed token string $(c,1)(a,2)(b,3)(a,4)(b,5)(d,6)$ which is not in X. However *cababd* is not in the language of Γ and so will be rejected by the MGLL parser.

More generally, if L_n is the set of $n(n-1)/2$ sentences of the form $ca^k b^h d$, where $k + h \le n$, then there is a corresponding indexed token set X whose TWE set is

$$\Sigma_X = \{(c,0,1), (a,i,i+1), (b,i+1,i+2), (a,1,i+1), (d,n+1,n+2) \ \mid\ 1 \le i \le n-1\}$$

The elements of $strings(\Sigma_X)$ that are also sentences in Γ are precisely the elements of L_n, that is

$$strings(\Sigma_X) \cap L(\Gamma) = L_n.$$

The size of Σ_X is $3n-2$, and so the input to an MGLL parser is linear in n, but the size $|L_n|$ of the set of sentences it parses is quadratic in n.

Embedded languages – island parsing For multi-lexer parsing, the advantage that we have highlighted in this paper is the ability to handle lexical disambiguation flexibly. The same approach may also have advantages for compiling systems that have embedded languages, for example SQL embedded in Java. In such cases the lexical tokens depend on the context, i.e. on whether the statement being processed belongs to the outer or the embedded language. For example, something which is a keyword in SQL may be an identifier in Java. A common approach is to have separate lexers for outer and embedded language and to have the parser call which ever is appropriate for the context. With the MGLL approach all of the tokens can be constructed using a single lexer, and the parser will only use those that are correct for the context.

We have previously carried out a case study in the so-called 'island parsing' domain where we used a GLL parser to parse strings with embedded actions targeted at the Tom language analysis tool [JSvdB$^+$13]. It would be interesting to do the same thing with the MGLL approach to see what can be improved by employing a multi-lexer parser.

User specified extents In our discussions in this paper we have focused on TWE elements whose extents are constructed from an underlying character string. However, the MGLL technique does not require this. The extents can be user specified and need only to form a monotonically increasing sequence. When parsing multiple strings, synchronising on any desired subsequence can then be achieved by ensuring that the extents on their symbols match in all strings.

We note that the left-most left extent does not have to be 0, it can be any integer which is less than all the other extents. Using a base left extent which is not 0 could allow sets of triples to be combined. However, for simplicity, throughout this paper we have taken the left-most left extent to be 0.

It is an important feature of our technique that it can parse sets comprised of token strings of different lengths, but the MGLL parser needs a fixed length to be able to determine whether the whole string has been parsed. Thus we require that in any given set, the indexed token strings all have the same final right extent.

We also note that the choice of extents for the indexed token strings has no effect on whether the string is parsed successfully. The MGLL parser will parse all the strings embedded in the TWE set and accept precisely those whose underlying string is in the language of the grammar. However, the extents do impact on the efficiency of the parser. At one extreme the extents can be chosen so that they are all different, except for the rightmost extents. This will ensure that the ITS set is consistent, but the parser will effectively parse each string separately and the size of the input TWE set would be as big as it could be, with one triple for every terminal instance in every input string. At the other extreme we could try to use extents to make the TWE set as small as possible. This will improve efficiency in one direction but it will also increase the number of strings embedded in the input set and thus the number of redundant derivations represented in the output.

The construction of TWE sets which are not based on underlying character strings, and the selection of extents to tune parser efficiency, is likely to require experience which will only be built up over time. The indexing in the previous ca^kb^hd example was chosen using human judgment, not a principled process. However, the MGLL technique does not require application specific modification. It works in the same way on any TWE set, raising the potential for efficient, tunable multi-parsing to improve efficiency in any application which requires the structural analysis of a large number of strings. The remaining application specific challenge is to find a suitable indexing choice, not the subsequent MGLL multi-parsing.

10.2 Engineering optimisation

Generalised parsing algorithms provide many opportunities for tight optimisation when computing their internal data structures. In our previous work on GLR style algorithms [SJ06, SJE07] we developed very efficient implementations in ANSI-C that mapped grammar positions onto small integers and performed their own memory management: we achieved speedups of around an order of magnitude during this optimisation work. Our work on the GLL algorithm has been less tightly engineered, but we have provided elsewhere [SJ16] results of a comparison between a conventional Object Oriented implementation of a simplified GLL algorithm and a version that used a similar style of low level memory management. Both were written in Java: on a highly ambiguous grammar we reported speedup factors of between 10 and 20 for examples large enough to trigger Java's warmup behaviour. We also reported there that a naive transcription of the low level Java code to C resulted in a speedup of over 50 compared to the pure OO Java implementation. Now, care is required when interpreting those reports: the grammar was pathologically ambiguous, and the performance gap between Java and compiled low-level C has narrowed in the intervening years. Nevertheless, our experience is that careful low level implementations of parsing algorithms even in Java can yield significant speed ups over code that leverages the standard Java APIs.

The MGLL approach also offers interesting opportunities to exploit multi-core processors. For instance, the construction of TWE sets proceeds left to right, and the 'consumption' of the TWE set by an MGLL parser also proceeds left to right. Hence there is a natural producer-consumer relationship between the two parts which could be run on separate processors. This will be explored in future work.

10.3 Multi-lexer parsing with other techniques

We have developed multi-parsing with GLL because the close relationship between the GLL technique and grammar traversal makes it relatively easy to handle the situation in which there may be many 'next input' tokens at a given step in the algorithm.

The difference between a recogniser and a parser is that the former simply checks that the input is syntactically correct. A parser returns a representation of the derivation of the input and a fully general parser returns a representation of all derivations of the input. In its original form, Earley's algorithm [Ear70] is a recogniser that is correct for all context free grammars. However, Earley's proposed extension to a general parser contained an error. GLR [Tom86] is a generalisation of LR parsing that was designed to produce derivations, with the aim of correcting the error in Earley's derivation construction proposal. Tomita's algorithm also contained an error in the case of grammars with 'hidden left recursion'. We can create versions of both these recognisers that take as input TWE sets, but this is more complicated than the GLL adaptation, particularly because both Earley and Tomita's algorithms treat the matching of input symbols in a separate 'scanner' or 'shifter' phase at the end of each algorithm step. For multiple input strings the worklists associated with this phase needs to be modified.

However, any recogniser, or any parser, that makes full disambiguation choices during the parse, will only determine that at least one of the tokenisations embedded in the TWE set is syntactically correct. It is the construction of the ESPPF and the existence of the sentence finding algorithm that allow us to parse, and recover, multiple input sentences embedded in a TWE set. Extending other generalised parsing techniques to admit multi-parsing will thus include modifying them to output an ESPPF, or a similar form of representation, from an input TWE set.

We have given a corrected version of the general parser version proposed by Earley for his algorithm, that generates an SPPF [SJ10b]. We also have produced a corrected version, RNGLR, of GLR based on so called 'right nulled' LR parse tables [SJ06]. It is likely that both these algorithms could be modified to produce an ESPPF from a TWE set. But we have focused on MGLL for this paper as implementation as grammar traversal makes the extension of GLL relatively simple.

10.4 Disambiguation in a character level grammar

For a character level grammar there is no notion of lexical ambiguity. The 'tokens' are characters whose patterns are pairwise disjoint singleton sets. All ambiguity is syntactic, i.e. a property of the grammar. The lexical notions of longest match and priority for a standard separated lexical/phrase level specification need to be converted to syntactic disambiguation rules for the corresponding character level grammar.

In fact, the natural syntactic versions of longest match and priority (choosing the longest possible sequence of input tokens and choosing one grammar rule over another) are not sufficient, in general, to implement the lexical versions (which are in terms of the character string and token priority). The point at which syntactic ambiguity becomes apparent (the position of multiple packed nodes) is not always the point at which the longest matching character substring can be determined. Thus not only is the multi-lexer parser approach potentially more efficient than character level parsing, it can allow more natural and effective lexical disambiguation.

We do not formally consider syntactic disambiguation in this paper, but the relationship between the lexical disambiguation rules we have introduced and the syntactic disambiguation strategies used in a character level parser merits further investigation.

10.5 Conclusions

In this paper we have introduced MGLL, a general parsing technique that can efficiently parse a set of input strings together, sharing the processing of common parts. We have also introduced its application to multi-lexer parsing, giving a language specifier the power of a character level grammar specification whilst retaining the advantages of a token level grammar. These advantages include: (i) the lexical disambiguation strategy can be specified independently of the syntax level, but disambiguation decisions can also be passed on to the parser or even to a post parse processor if desired, (ii) patterns of tokens can be defined and recognised without full parsing, and this is more efficient than character level parsing, particularly when the patterns are regular languages, and (iii) lookahead and error reporting in the parser can be at token rather than character level.

We thus achieve a spectrum of possibilities for the lexical/phrase level divide, with character level parsing at one extreme, and the classical LEX/YACC-style division at the other. The language designer is free to choose the lexical/phrase level divide, while our concrete separation avoids the (mis-)use of lexical disambiguation techniques at phrase level.

11 ACKNOWLEDGEMENTS

This work was partially supported by EPSRC funded project EP/I032509/1 PLanCompS: Programming Language Components and Specifications.

We thank the anonymous referees for their detailed work and for the many helpful suggestions they made.

REFERENCES

[AH01] John Aycock and R. Nigel Horspool. Schrodinger's token. *Software: practice and experience*, 31:803 – 814, 2001.

[ALSU06] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: principles, techniques, and tools, 2nd edition.* Addison-Wesley, 2006.

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: principles, techniques, and tools.* Addison-Wesley, 1986.

[BL89] Sylvie Billot and Bernard Lang. The structure of shared forests in ambiguous parsing. In *Proceedings of the 27th conference on Association for Computational Linguistics*, pages 143–151. Association for Computational Linguistics, 1989.

[CT96] J.-P. Chanod and P. Tapananeinen. A non-deterministic tokeniser for finite-state parsing. In W. Wahlster, editor, *ECAI 96. 12th European Conference on Artificial Intelligence*, pages 10–12. JohnWiley & Sons, Ltd., 1996.

[CT99] J.-P. Chanod and P. Tapananeinen. Finite state based reductionist parsing for french. In A. Kornai, editor, *Extended Finite State Models of Languages*, pages 72–85. Cambridge University Press, 1999.

[Ear70] J Earley. An efficient context-free parsing algorithm. *Communications of the ACM*, 13(2):94–102, February 1970.

[For04] Bryan Ford. Parsing expression grammars: a recognition-based syntactic foundation. In Neil D. Jones and Xavier Leroy, editors, *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2004, Venice, Italy, January 14-16, 2004*, pages 111–122. ACM, 2004.

[GJS96] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification.* Addison-Wesley, 1996.

[GJS+22] James Gosling, Bill Joy, Guy Steele, Gilad Bracha, Alex Bucklkey, Daniel Smith, and Gavin Bierman. *The Java Language Specification Java SE 18 Edition.*
`https://docs.oracle.com/javase/specs/jls/se18/jls18.pdf`, 2022.

[JS11a] Adrian Johnstone and Elizabeth Scott. Modelling GLL parser implementations. In M.van den Brand B.Malloy, S.Staab, editor, *SLE 2010*, volume 6563 of *Lecture Notes in Computer Science*, pages 42–61. Springer-Verlag, 2011.

[JS11b] Adrian Johnstone and Elizabeth Scott. Translator generation using ART. In M.van den Brand B.Malloy, S.Staab, editor, *SLE 2010*, volume 6563 of *Lecture Notes in Computer Science*, pages 306–315. Springer-Verlag, 2011.

[JSvdB+13] Adrian Johnstone, Elizabeth Scott, Mark van den Brand, Ali Afroozeh, Maarten Manders, and Jean-Christophe Moreau, Pierre-Etienneand Bach. Island grammar-based parsing using gll and tom. In *Software Language Engineering Lecture Notes in Computer Science 5th International Conference, SLE 2012, Revised Selected Papers*, volume 7745, 2013.

[JSvdB14] Adrian Johnstone, Elizabeth Scott, and Mark van den Brand. Modular grammar specification. *Science of Computer Programming*, 87:23 – 43, 2014.

[KvdSV09] P. Klint, T. van der Storm, and J. Vinju. Rascal: A domain specific language for source code analysis and manipulation. In *Source Code Analysis and Manipulation*, pages 108–177. IEEE, 2009.

[ME90] M.E.Lesk and E.Schmidt. Lex—-a lexical analyzer generator. In *UNIX Vol. II*, pages 375–387, Philadelphia, 1990. W.B.Saunders.

[PF11] Terence Parr and Kathleen Fisher. Ll(*): the foundation of the antlr parser generator. In *PLDI*, pages 425–436, 2011.

[SJ06] Elizabeth Scott and Adrian Johnstone. Right nulled GLR parsers. *ACM Transactions on Programming Languages and Systems*, 28(4):577–618, July 2006.

[SJ10a] Elizabeth Scott and Adrian Johnstone. GLL parsing. In *9th Workshop on Language Descriptions Tools and Applications (LDTA 2009)*, volume 253 of *Electronic Notes in Theoretical Computer Science*, pages 177–189. Elsevier, 2010.

[SJ10b] Elizabeth Scott and Adrian Johnstone. Recognition is not parsing – SPPF-style parsing from cubic recognisers. 75:55–70, 2010.

[SJ13] Elizabeth Scott and Adrian Johnstone. GLL parse-tree generation. *Science of Computer Programming*, 78:1828–1844, 2013.

[SJ16] E. Scott and A. Johnstone. Structuring the GLL parsing algorithm for performance. *Science of Computer Programming*, 125:1–22, 2016.

[SJ18] Elizabeth Scott and Adrian Johnstone. GLL syntax analysers for EBNF grammars. *Science of Computer Programming*, 166:120–145, 2018.

[SJ19] Elizabeth Scott and Adrian Johnstone. Multiple lexicalisation - A Java based case study. In *Proceedings of the 12th ACM SIGPLAN International Conference on Software Language Engineering, SLE'19*. ACM, 2019.

[SJE07] Elizabeth Scott, Adrian Johnstone, and Giorgios Economopoulos. A cubic Tomita style GLR parsing algorithm. *Acta Informatica*, 44(6):427–461, 2007.

[Tom86] Masaru Tomita. *Efficient parsing for natural language.* Kluwer Academic Publishers, Boston, 1986.

[Tom91] Masaru Tomita. *Generalized LR parsing.* Kluwer Academic Publishers, The Netherlands, 1991.

[vdBHKO02] M.G.J. van den Brand, J. Heering, P. Klint, and P.A. Olivier. Compiling language definitions: the ASF+SDF compiler. *ACM Transactions on Programming Languages and Systems*, 24(4):334–368, 2002.

[Vis97a] Eelco Visser. Scannerless generalised-LR parsing. Technical Report P9707, University of Amsterdam, 1997.

[Vis97b] Eelco Visser. *Syntax definition for language prototyping.* PhD thesis, University of Amsterdam, 1997.

[Vis04] Eelco Visser. Program transformation with Stratego/XT: rules, strategies, tools, and systems in StrategoXT-0.9. In C.Lengauer et. al, editor, *Domain-Specific Program Generation*, volume 3016 of *Lecture Notes in Computer Science*, pages 216–238. Springer-Verlag, Berlin, June 2004.

[Wal15] R. M. Walsh. *Adapting Compiler Front Ends for Generalised Parsing, PhD Thesis.* Royal Holloway, University of London, 2015.

A APPENDIX - PROOFS

Proof of Lemma 1

If Σ is tight then (t, i, j) belongs to some string u embedded in Σ. If $i \neq 0$ then this string is of the form $u'(t', i)(t, j)v$ and thus, for some k, $(t', k, i) \in \Sigma$. Similarly, if $j \neq h$ then the string is of the form $u'(t, j)(t', k)v$ and thus, for some k, $(t', j, k) \in \Sigma$.

Now suppose that (a) and (b) hold and suppose that $(t, i, j) \in \Sigma$. Repeatedly using (a), we can choose elements $(t_0, i_1, i), (t_1, i_2, i_1), \dots, (t_p, 0, i_p)$ in Σ. Repeatedly using (b), we can choose elements $(s_1, j, j_1), (s_2, j_1, j_2), \dots, (s_f, i_f, h)$ in Σ. Then the string $(t_p, i_p) \dots (t_1, i_1)(t_0, i)(t, j)(s_1, j_1) \dots (s_f, h)$ is embedded in Σ. So Σ is tight.

Proof of Lemma 2

(i) If $\Sigma \subseteq \Sigma_{strings(\Sigma)}$ and $g \in \Sigma$ then $g \in \Sigma_{strings(\Sigma)}$ and so g belongs to some string embedded in Σ. Thus Σ is tight.

(ii) By definition, if Σ is tight then every triple $g \in \Sigma$ belongs to the TWE set of some string in $strings(\Sigma)$ so $g \in \Sigma_{strings(\Sigma)}$. By the definition of $strings(\Sigma)$ we have $\Sigma_{strings(\Sigma)} \subseteq \Sigma$, so $\Sigma_{strings(\Sigma)} = \Sigma$.

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

(iv) This is an immediate consequence of the definitions: by definition of Σ_x and $strings(\Sigma_X)$ we have $X \subseteq strings(\Sigma_X)$, and the definition of the consistency of X is equivalent to $strings(\Sigma_X) \subseteq X$.

(v) Let $X = strings(\Sigma)$. We show that $strings(\Sigma_X) = X$ then the result follows from (iii). As we remarked in the proof of (iii), we have $X \subseteq strings(\Sigma_X)$, for any X. So we need to show that $strings(\Sigma_X) \subseteq X$.

Let h be the height of Σ_X. We prove, by induction, that if $(t_1, 0, i_1), (t_2, i_1, i_2), \dots, (t_k, i_{k-1}, i_k)$ are elements of Σ_X then there are elements $(t_{k+1}, i_k, i_{k+1}), \dots, (t_l, i_{l-1}, h)$ in Σ_X such that $(t_1, i_1) \dots (t_k, i_k)(t_{k+1}, i_{k+1}) \dots (t_l, h)$ is in X. Then, if $u \in strings(\Sigma_X)$ we have, by definition, that u is of the form $(t_1, i_1), (t_2, i_2), \dots, (t_l, h)$, where the triples (t_p, i_{p-1}, i_p) all belong to Σ_X, so we will have $u \in X$ by induction.

For $(t_1, 0, i_1) \in \Sigma_X$, by construction there must be a string $u \in X$ of the form $(t_1, i_1)(t_2, i_2) \dots (t_l, h)$ and the triples $(t_2, i_1, i_2), \dots, (t_l, i_{l-1}, h)$ also belong to Σ_X, as required.

Now suppose that $(t_1, 0, i_1), (t_2, i_1, i_2), \dots, (t_k, i_{k-1}, i_k), (s, i_k, j)$, are elements of Σ_X and, by induction, that there are elements $(t_{k+1}, i_k, i_{k+1}), \dots, (t_l, i_{l-1}, h)$ in Σ_X such that $(t_1, i_1)(t_2, i_2) \dots (t_k, i_k)(t_{k+1}, i_{k+1}) \dots (t_l, h)$ is in X. By construction, since $(s, i_k, j) \in \Sigma_X$, there is a string of the form $u(s', i_k)(s, j)v$ in X. Since X is consistent we have that $(t_1, i_1)(t_2, i_2) \dots (t_k, i_k)(s, j)v$ is in X. The TWE set corresponding to this string is a subset of Σ_X and is a set of triples of the required form.

Proof of Theorem 1

Suppose that $\gamma = (a_1, n_1) \dots (a_m, n_m) \in strings(\Sigma, \Gamma)$, so $(a_j, n_{j-1}, n_j) \in \Sigma$ for $1 \leq j \leq m$ and there is an SPPF for $a_1 \dots a_m$ in Γ. We construct the ESPPF χ_γ for γ as above by replacing each extent and pivot value, k, in the labels of the SPPF nodes with n_k. Let $u_j = (a_j, n_{j-1}, n_j)$ for $1 \leq j \leq m$ and $n_0 = 0$. By definition, all of the nodes in χ_γ are nodes in χ and the root node $(S, 0, n_m)$ of χ_γ is also the root noded of χ. We show by structural induction that if $u = (x, i, j) \in \chi_\gamma$ then $u_{i+1} \dots u_j \in PS_u$, where, if $i = j$, $u_{i+1} \dots u_j = \epsilon$. If u is a leaf node then this is true by definition. So suppose that u has a packed node child w. If w has two children $y = (t, i, k)$ and $z = (s, k, j)$ then by induction $u_{i+1} \dots u_k \in PS_y$ and $u_{k+1} \dots u_j \in PS_z$ so $u_{i+1} \dots u_j \in PS_w \subseteq PS_u$ by definition. If w has only one child $y = (t, i, j)$ then again the result

$u_{i+1}\dots u_j \in PS_y = PS_w \subseteq PS_u$ follows immediately by induction and definition. Thus, since χ_γ is an ESPPF for γ, we have $u_1\dots u_{n_m} \in PS_{w_S}$ and thus $\gamma \in sentences(\chi)$.

Now suppose that $\gamma = (a_1, n_1)\dots(a_m, n_m) \in sentences(\chi)$. By construction $u_i = (a_i, n_{i-1}, n_i)$ are leaf nodes in χ and so, by definition, $(a_i, n_{i-1}, n_i) \in \Sigma$ and $\gamma \in strings(\Sigma)$. Also by definition, $u_1\dots u_{n_m} \in PS_{w_S}$. We show by structural induction that if $u = (x, i, j) \in \chi$ and $u_{i+1}\dots u_j \in PS_u$ then x derives $a_{i+1}\dots a_j$ in Γ. From this it follows that $\gamma \in strings(\Sigma, \Gamma)$ as required. To construct the proof we show at the same time that if $u = (X ::= \alpha x \cdot \beta, i, j) \in \chi$ and $u_{i+1}\dots u_j \in PS_u$ then αx derives $a_{i+1}\dots a_j$ in Γ.

If u is a leaf node then $x = a_j$ or, if $i = j$, $x = \epsilon$ so the result is trivially true. So suppose that u has a packed node child w such that $u_{i+1}\dots u_j \in PS_w \subseteq PS_u$. If w has only one child $y = (t, i, j)$ then $u = (X, i, j)$, $w = (X ::= t\cdot, i)$, $u_{i+1}\dots u_j \in PS_y = PS_w$ and by induction t, and hence X, derives $a_{i+1}\dots a_j$ in Γ. If w has two children $y = (t, i, k)$ and $z = (x, k, j)$ then $w = (X ::= \alpha x \cdot \beta, k)$ where t is either α or $X ::= \alpha \cdot x\beta$. We have $PS_w = PS_y \cdot PS_z$ and $u_{i+1}\dots u_k \in PS_y$, $u_{ik+1}\dots u_j \in PS_z$. By induction, α derives $u_{i+1}\dots u_k$ and x derives $u_{ik+1}\dots u_j \in PS_z$, so αx derives $u_{i+1}\dots u_j$. Finally, either $\beta = \epsilon$ and $u = (X, i, j)$, so X derives $u_{i+1}\dots u_j$, or $u = (X ::= \alpha x \cdot \beta, i, j)$. This proves the result.

B APPENDIX - ADDITIONAL ALGORITHMS

B.1 TWE Set Pruning - $prune(\Sigma)$

To allow us to ensure that a set is tight, we define a procedure which removes redundant triples, constructing the maximum size tight TWE set, $prune(\Sigma)$, contained in Σ. The definition interacts naturally with consistency in the sense that $prune(\Sigma)$ constructed has the same set of embedded strings as Σ.

We define $prune(\Sigma)$, where Σ is a TWE set of height h, as follows:

- construct Σ' by taking all the elements in Σ of the form $(t, 0, i)$ for any t, i
- form the closure, Σ'', of Σ' under the property that (i) $\Sigma' \subseteq \Sigma''$ and (ii) if $(t', j, i) \in \Sigma''$ then $(s, i, f) \in \Sigma''$ for all $(s, i, f) \in \Sigma$
- define $prune(\Sigma)$ to be the smallest set which contains all the elements of the form (t, j, h) in Σ'', and which has the property that if $(t', j, i) \in prune(\Sigma)$ then $(s, f, j) \in prune(\Sigma)$ for all $(s, f, j) \in \Sigma''$.

The following lemma shows that $prune(\Sigma)$ has the required properties.

Lemma 3. *For any TWE set Σ we have $prune(\Sigma) = \Sigma_{strings(\Sigma)}$. In particular, $prune(\Sigma)$ is tight and the indexed token strings embedded in Σ are precisely those embedded in Σ, i.e. $strings(\Sigma) = strings(prune(\Sigma))$.*

Let h be the height of Σ. From the construction of Σ'', if there is an element of the form $(t, i, h) \in \Sigma''$ then there is a string in $strings(\Sigma)$. Thus, if $strings(\Sigma) = \emptyset$ we have $prune(\Sigma) = \emptyset$ and the result holds.

So we suppose that $strings(\Sigma) \neq \emptyset$ and then, from the construction of Σ'', Σ'' is non-empty and has height h, and so $strings(\Sigma'') \subseteq strings(\Sigma)$. Also by construction we have $strings(\Sigma) \subseteq strings(\Sigma'')$. Similarly, $strings(\Sigma'') = strings(prune(\Sigma))$.

Consider an element $(t, i, j) \in prune(\Sigma)$. Since $(t, i, j) \in \Sigma''$ there is a sequence $(t_0, 0, i_1), \dots, (t_l, i_l, i), (t, i, j)$ of elements in Σ''. Since $(t, i, j) \in prune(\Sigma)$ we have $(t_l, i_l, i) \in prune(\Sigma)$ and thus part (a) of Lemma 2 holds for $prune(\Sigma)$.

Since $(t, i, j) \in prune(\Sigma)$, if $j < h$ then, by construction of $prune(\Sigma)$ there must be an element $(t', j, f) \in prune(\Sigma)$, as required for part (b) of Lemma 2. Thus, by Lemma 2, $prune(\Sigma)$ is tight.

Finally then, since $strings(\Sigma) = strings(\Sigma'') = strings(prune(\Sigma))$, we get

$$\Sigma_{strings(\Sigma)} = \Sigma_{strings(prune(\Sigma))} = prune(\Sigma)$$

B.2 Automata based TWE set generation

$lexicalise(\gamma, T)$ {
  for $t \in T$ build a DFA Δ_t which accepts precisely the pattern of t
  if $\gamma = \epsilon$ set $h := 0$ otherwise let $\gamma = x_1 \dots x_h$
  initialise $\Sigma' := \{(\$, h, h + 1)\}$
  if $\gamma = \epsilon$ initialise $J := \emptyset$ otherwise initialise $J := \{1\}$
  while $J \neq \emptyset$ {
    remove the smallest integer i from J
    for each $t \in T$ {
      traverse Δ_t with input $x_i \dots x_h$
      each time an accepting state is reached {
        if the remaining input is $x_{j+1} \dots x_h$ where $j \leq h$
        add (t, i, j) to Σ' and j to J } } }
  return $\Sigma_\gamma := prune(\Sigma')$ }

The construction produces all the lexicalisations from the left. Since any complete lexicalisation can be produced by starting at the left of the character string, this produces all lexicalisations, but not, in most cases, lexicalisations of all the substrings of γ.

B.3 TWE set generation using a GLL EBNF recogniser

An EBNF grammar is a grammar in which the right hand sides of the grammar rules are regular expressions over the set of terminals and nonterminals. We define an *EBNF lexical grammar* as a restricted form of EBNF grammar with a start rule S, and identified lexical nonterminals $T_1, \dots, T_f$ which are distinct from each other and from S. Each token nonerminal T_q has a specified associated token t_q. The grammar rule for S must be a single Kleene closure of the form

$$S ::= (T_1 \mid \dots \mid T_f)*$$

and S and T_j, $1 \leq j \leq f$ must not appear on the right hand side of any other rule in the grammar. The method allows for any context free specification of the token patterns. However, in the most straightforward case, EBNF lexical grammar just has a single rule for each T_q whose right hand side is a regular expression over the characters $\mathcal{A}$. This will result in an efficient GLL parser as there will be very little GSS activity and can be directly compared to the automata based approach described in Section B.2.

The EBNF GLL algorithm [SJ18] differs from the MGLL algorithm above in that it has templates for regular expressions, and also in that it has more complicated $getNode()$ functions to build the SPPF. However, the modification we make to generate the TWE elements is in the template for $code(X)$, which is essentially the same for EBNF GLL as for MGLL above (the difference is that the alternates τ_i can be regular expressions rather than just strings but this is not visible at the level of the $code(X)$ template).

The required additional functionality is to create a TWE element (t_q, i, j) when the parser 'matches' $x_{i-1} \dots x_j$ to the token nonterminal T_q, and then to effectively start a new parse from input position j. As long as the grammar is a lexical grammar this happens exactly when the parser carries out a pop action associated with token nonterminal. The current GSS node, c_U, will have a label of the form $(S ::= (T_1| \dots |T_q \cdot | \dots |T_f)*, i)$ and the required TWE element will be $(t_q, level(c_U), c_I)$, where $level(L, i)$ denotes i, the integer index of the GSS node (L, i).

Denoting the TWE set being built by the parser by Σ, all we have to do is modify the $code(X)$ template for token nonterminals to add an element to Σ after a return from a pop action.

TWE building template for $code(X)$

If T_q is a token nonerminal with the grammar rule $T_q ::= \tau_1 \mid \dots \mid \tau_p$ and associated token t_q then

$$
\begin{array}{rl}
code(T_q) = & \\
J_{T_q}: & \textbf{if}(testSelect(c_I, \tau_1, T_q, \Sigma)) \{ \ add(T_q ::= \cdot\tau_1, c_U, c_I) \ \} \\
 & \dots \\
 & \textbf{if}(testSelect(c_I, \tau_p, T_q, \Sigma)) \{ \ add(T_q ::= \cdot\tau_p, c_U, c_I) \ \} \\
 & \textbf{goto } L_0 \\
T_q ::= \cdot\tau_1 : & code(T_q ::= \cdot\tau_1); \ \ pop(c_U, c_I); \ \ \text{add } (t_q, level(c_U), c_I) \text{ to } \Sigma; \ \ \textbf{goto } L_0 \\
 & \dots \\
T_q ::= \cdot\tau_p : & code(T_q ::= \cdot\tau_p); \ \ pop(c_U, c_I); \ \ \text{add } (t_q, level(c_U), c_I) \text{ to } \Sigma; \ \ \textbf{goto } L_0
\end{array}
$$

Note that the ITS set of all lexicalisations of a character string will always be consistent, as required for MGLL input. As for the automata method, this algorithm produces lexicalisations from the left. For most programming languages all partial left hand lexicalisations can be extended to full lexicalisations so the TWE set will also be tight. In specifications for which not all initial left segment lexicalisations can be extended to completion the $prune()$ procedure described above can be used to extract a tight subset.