Multiple Lexicalisation - A Java Based Study

Anonymous Author(s)

Abstract

We consider the possibility of making the lexicalisation phase of compilation more powerful by avoiding the need for the lexer to return a single token string from the input character string. This has the potential to empower language design by softening the boundaries between lexical and phrase level specification. The large number of lexicalisations makes it impractical to parse each one individually, but it is possible to share the parsing of common subparts, reducing the number of tokens parsed from the product of the token numbers associated with the components to their sum. We report total numbers of lexicalisations of example Java strings, and the impact on these numbers of various lexical disambiguation strategies, and we introduce a new generalised parsing technique that can efficiently parse multiple lexicalisations of character string simultaneously. We then use this technique on Java, reporting on the number of lexicalisations that correspond to syntactically correct Java strings and the degree to which the standard Java lexer is safe in the sense that it does not remove all the syntactically correct lexicalisations of an input character string. Our multi-lexer parser is an alternative to scannerless parsing of a character level grammar, retaining the separation between grammar terminals and the corresponding lexical tokens. This has the advantages of allowing the parser to use terminal level lookahead and keeping lexical level disambiguation separate from the context free grammar.

Keywords lexicalisation, generalised parsing, syntax analysis

1 Introduction

Conventionally compiler front ends include a lexical analyser (lexer), which takes an input string of characters and returns a string of tokens, and a syntax analyser (parser), which determines the syntactical structure of the token string with respect to a given grammar. In almost all cases an input character string can be lexicalised in many different ways, and in the conventional set-up the lexer 'decides' which lexicalisation to return before the subsequent compilation stages are carried out. This can sometimes be inconvenient, and imposes certain language design constraints.

So called 'scannerless' parsers are built from grammars that are specified at character level, with nonterminals whose languages are the patterns of the conventional lexer-returned terminals. This allows all lexicalisations to be considered, but has some drawbacks. The lexical parts of the character level grammar are typically highly ambiguous and disambiguation techniques, such as longest match, which address this are not safe (in the sense that they can reject all derivations of a string) for context free grammars in general. Furthermore, traditional parsing techniques gain efficiency by using lookahead, and lookahead is not effective at character level.

In this paper we present an approach in which a separate lexer is used but it is permitted to return multiple lexicalisations of the character string to the parser. This allows the lexical level disambiguation techniques to remain separate from the context free grammar and keywords to be used in parser lookahead, while also allowing lexical disambiguation decisions to be made at any stage in the compiler front end. Thus syntax and semantic information can be used for lexical disambiguation if desired.

The approach is based on a version of GLL parsing which is capable of efficiently parsing a set of input strings concurrently. In the version presented in this paper our parser will call the lexer each time it requires an input terminal. This allows additional use of lookahead because the parser has grammar defined information which may identify restrictions on the possible next terminals (i.e. the so-called local follow sets for each grammar position). This information can be passed to the lexer, limiting the potential matches it has to attempt. We note, however, that the underlying parser can be written so that it accepts a precomputed set of strings of terminals, and in fact sets of strings that correspond to the lexicalisations of many different character strings. But we shall not discuss this further in the current paper.

We will present a formal description of our algorithm but this paper is practically focused, based on the analysis of this approach when applied to Java. First we consider some data related to the choice of lexicalisations of strings in Java, in particular the impact of 'longest match within a token' and 'longest match with priority' disambiguations. We then introduce indexed token strings that allow sharing of parsing between lexicalisations with common substrings. In Section 3 we give our new multi-parsing algorithm, and in Section 4 we run the parser on Java examples to investigate the impact and consequences of using a softer boundary between lexicalisation and syntax analysis. We conclude with a brief discussion of related work.

Note on terminology: the terms 'token' and 'terminal' are to some degree used interchangeably when discussing compiler front ends. The set of terminal names is part of the specification of the particular grammar on which the compiler is based, and they are simply symbols. Lexical analysers take character strings and test them for membership against specified sets. Commonly these sets are specified by regular

SLE 2019, October 20–22, 2019, Athens
2019.

expressions over the character set, and are often referred to as tokens. The sets of terminal names and token names in a particular compiler coincide and thus either term can be used. In this paper, to support easier reading, we will mainly use 'token' when focusing on lexical activity and 'terminal' when discussing parsers.

2 Lexer/parser interface in Java

The particular token sequence delivered to a standard Java parser is determined by longest match and token priority rules that can be incorporated in the lexer, and in most cases this is adequate. However, even in Java there are some lexicalisations that are rejected but which would have corresponded to syntactically correct sentences, while the selected lexicalisation is not syntactically correct and is thus rejected by the parser. For example, Java is a layout insensitive language in the sense that whitespace can normally be inserted for readability without changing the semantics of a program and we expect to be able to leave out whitespace surrounding operators, brackets and punctuation symbols. However, the expression `x=y--z`, for example, will be rejected by a Java compiler, while the expression `x=y- -z` will be accepted. Furthermore, `x=y+-z` *will* be accepted by the compiler. The rejection of `x=y--z` is a result of a lexer disambiguation decision which returned the postfix decrement operator `--` rather than the binary subtraction operator followed by the unary minus operator.

The most extreme alternative approach to lexical disambiguation is to have none at all and to return all lexicalisations. While this is typically an impractically large number of strings, it serves as a bench mark against which partial disambiguation strategies which eliminate some of the potential lexicalisations can be evaluated. A more viable approach is to allow partial lexical disambiguation under the control of language designer. In the main part of this paper we will introduce a parsing technology that can accept and efficiently parse multiple input strings, thus returning a set of lexicalisations to the parser is a practical possibility.

We begin with an analysis of the scale of the problem: how impractical would it be to simply produce all the lexicalisations of an input character string and parse them, and to what extent do various common disambiguation conventions reduce the number of lexicalisations? To help illustrate our discussions we shall refer to the following Java program fragment, Ex1.java,

```
int x=3;
while(x<10){x+=1}
```

2.1 Lexicalisation counts

We shall use general rather than Java specific terminology, and assume we have a specified set of characters, $\mathcal{A}$, and a set of *token* (or terminal) names. Each token t denotes a set of strings of characters, the *pattern* of t. A string in t is a *lexeme* of t. A *lexicalisation* of a character string u is a string $t_1 \dots t_k$ of tokens such that $u = u_1 \dots u_k$ where $u_i \in t_i$, $1 \le i \le k$.

A character string can be partitioned into substrings whose lexicalisations are independent of each other. For example, in Java an opening brace `{` is not a character in a lexeme of any token other than itself, and the special cases of string literals and comments. Thus, in most cases, the positions immediately before and immediately after a `{` are partition divisions.

For a character string $u = u_1 \dots u_k$ where the position between each u_i and u_{i+1} is a lexical partition division, the number of lexicalisations of u is the product of the numbers of lexicalisations of the u_i. If each lexicalisation were to be parsed separately this product gives an upper bound on the number of parses required for u. However, if the parsing of each u_i can be shared across the different lexicalisations then the number of parses can be reduced to the sum of the numbers required for each u_i, and each of these parses only a substring not the full token string.

For the Java example above we have u_1 is the string *int* which, using `ID` for the identifier token and `int` for the single lexeme keyword token, has four lexicalisations as

`(i) ID` `(ii) ID ID` `(iii) ID ID ID` `(iv) int`

Then u_2 is a single space character, u_3 is the lexeme x from `ID` and so on. We have that u_8 is the string *while* which has six lexicalisations, u_{12} is 10 and u_{16} is + =, both of which have two lexicalisations. This gives a total of 96 lexicalisations of the example string.

The lexicalisations of a character string can have different numbers of elements so simply counting the number of lexicalisations does not reflect the fact that parser complexity is thought of in terms of input string length. In this paper we will consider the total number of tokens a parser has to process as a measure of the parsing cost. In particular this is independent of the efficiency of the parsing technique itself. For the naive 'parse each string separately' approach this total number of tokens is the sum of the numbers of the lengths of each lexicalisation. The 96 lexicalisations of the above example contain 2056 tokens to be processed.

In the next section we shall consider methods which allow tokens in several lexicalisations to be shared, reducing the total number to be processed. First we consider the impact of lexical disambiguation on token numbers.

The number of lexicalisations of u_i, and correspondingly the number of tokens to be parsed, can be reduced by applying disambiguation criteria. We can simply return the longest lexeme choice within each token, for example returning `ID` but not `ID ID` or `ID ID ID` against *int*. This reduces the number of lexicalisations in our example to eight, with a total of 148 tokens.

We can also declare priority when the same lexeme belongs to two tokens, for example returning just the 'plus and

becomes' token against += and not also the two tokens + and =, and returning just `int` and not `ID` against *int*.

We report lexicalisation data for several examples: Ex1.java above, for Life.java which is a 217 line, 5859 character is a model solution to an undergraduate assignment on Conway's game of Life, for Linden.java a 40 line, 961 character program which implements a Lindenmayer string rewriting algorithm, and Sand.java which is a parser generator used as a 'sandbox' for exploring backtracking recursive descent parsers. We also report data for Ex2.java below which is rejected by the Java compiler as discussed above.

```
import java.io.*;
class Ex2{
    public static void main(String[] args) {
       int y=3, z=2;
       int x = y--z;
       System.out.print("x = " + x + "\n");
    }
}
```

Table 1 gives the number of lexicalisations and the total number of tokens with no disambiguation applied, with longest match within tokens, and with longest match and priority. For a character string u we use $NoD(u)$ to refer to the set of all lexicalisations. We define the set $LM(u)$ of lexicalisations under longest match within tokens recursively. For a token t and a character string u we denote by u_t the longest prefix of u which is a lexeme of t and we denote the corresponding suffix of u by u'_t, so $u = u_t u'_t$. Then

$$LM(u,t) = \begin{cases} \emptyset & \text{if } u_t = \epsilon \\ \{t\} & \text{if } u_t = u \\ \{tt' \mid t' \in LM(u'_t)\} & otherwise \end{cases}$$

and $LM(u)$ is the union of all the $LM(u,t)$. We also define the set $LP(u)$ of lexicalisations under longest match and priority, assuming that we have a (possibly partial) priority relation $<$ defined on the token set. (In our examples we have used the standard Java lexical priorities.) Then $LP(u)$ is the union of all the $LP(u,t)$ where

$$LP(u,t) = \begin{cases} \emptyset & \text{if } \exists (s > t)\ u_s = u_t \\ LM(u,t) & otherwise \end{cases}$$

All disambiguation specifications have strengths and weaknesses, and the appropriateness of a particular strategy is application specific. $LM(u)$ and $LP(u)$ are not the only possible specifications and we are certainly not claiming that they are necessarily the best. For example, they are defined from the left, which is natural for a left-to-right string processor, but defining from the right would give different outcomes. Our goal here is simply to give an idea of how much the size of a set of lexicalisations may be reduced by various disambiguation approaches.

From Table 1 we can see that the number of lexicalisations is extremely large for reasonably sized programs and parsing each one individually is not likely to be practical. The longest possible length of a lexicalisation is the length of the underlying character string, so the total number of tokens is always bounded by the number of lexicalisations multiplied by the length of the character string. However, we report the actual numbers for comparison, in the next section, with the numbers of tokens parsed by a token-sharing parser.

We also ran all the experiments with longest match across tokens combined with priority.

$$LAP(u,t) = \begin{cases} \emptyset & \text{if } \exists s\ u_s > u_t \\ \emptyset & \text{if } \exists (s > t)\ u_s = u_t \\ LM(u,t) & otherwise \end{cases}$$

The results were always a single lexicalisation, as is to be expected as this mimics the behaviour of a classical Java lexer. The two lexicalisations found for Ex2.java with the LP disambiguation are the ones associated with the choices `'--'` and `'-' '-'` discussed above. The Java compiler selects the first one using longest match across tokens, but only the second one is syntactically valid. With LAP disambiguation, Ex2.java is rejected.

Note on whitespace: our approach permits various treatments of whitespace and comments. However, this paper focuses on the main idea of multi-lexing and we shall not discuss whitespace choices. We simply note that, of course, the separate nature of the lexer allows many alternatives: it is easy to include whitespace specification in the grammar and allow the lexer to pass whitespace tokens to the parser, or to incorporate whitespace into the token immediately to the left or to the right. We can write custom lexers with different tokens for layout sensitive languages, and we can write lexers that handle nested comments. The approach supports modularity, different whitespace conventions can be supported in different parts of language/grammar specification. In this paper, for data reporting purposes, we simply assume that there is a token that matches strings of whitespace characters and that just a single token is returned, not all possible lexicalisations. Effectively sequences of whitespace characters are replaced by a single space. We include these tokens in our token counts but they don't change the number of lexicalisations. For parsing purposes, in our examples, the whitespace tokens are discarded by the lexer and not returned to the parser.

2.2 Indexed token strings

Generalised parsers usually remerge threads that result from non-deterministic choices by synchronising on positions in the input token string. This allows them to achieve polynomial rather than exponential complexity. In the previous section we mentioned that if a character string is partitioned into lexical divisions then each of these can be lexicalised

u	length	lexicalisations			total tokens		
		NoD(u)	LM(u)	LP(u)	NoD(u)	LM(u)	LP(u)
Ex1	26	96	8	2	2056	148	37
Ex2	169	1×10^{11}	256	2	1×10^{13}	19072	149
Life	5859	2×10^{387}	7×10^{65}	1×10^{18}	8×10^{390}	1×10^{69}	2×10^{21}
Linden	961	3×10^{72}	3×10^{13}	4×10^{6}	$\mathbf{1 \times 10^{75}}$	$\mathbf{1 \times 10^{16}}$	$\mathbf{1 \times 10^{9}}$
Sand	5685	7×10^{369}	8×10^{73}	3×10^{28}	2×10^{373}	1×10^{77}	7×10^{31}

Table 1. Lexicalisations

independently and parsed just once. This can be achieved in practice by synchronising the parser on the input positions of the character string. In order to do this we use tokens together with the right hand position of the lexeme to which they correspond.

An *indexed token string* is a sequence of pairs of the form (t, h) where t is a token and h is an integer. An *indexed lexicalisation* of u is a string $(t_1, h_1) \ldots (t_k, h_k)$ such that $u = u_1 \ldots u_k$ where $u_i \in t_i$ and the length of u_i is $h_i - h_{i-1}$, $1 \leq i \leq k$. (We take $h_0 = 0$.)

For Ex1.java above, *int* has indexed lexicalisations

`(i) (int,3)  (ii) (ID,3)  (iii) (ID,2) (ID,3)`
`(iv) (ID,1) (ID,3)  (v) (ID,1)(ID,2)(ID,3)`

Although only the right end index of the lexeme is stored, the left hand end is the right index of the previous element in the string. So each token instance does have a well defined left and right index.

We can see that in general there are more indexed lexicalisations of a string than there are non-indexed ones. In the above example we can take the first lexeme as *in* and the second as *t* or the first as *i* and the second as *nt*. Both choices generate the lexicalisation `ID ID` but they generate different indexed lexicalisations. If each indexed lexicalisation were to be parsed independently this would be more expensive than using the non-indexed lexicalisations. However, the indexing allows common substrings to be parsed only once, which is ultimately more efficient, and when disambiguation is used the numbers of indexed and non-indexed lexicalisations are almost the same.

For the above example, without disambiguation, we have that *int* has five indexed lexicalisations, *while* has 17, and 10 and `+=` both have two. This gives a total of 340 indexed lexicalisations of the example string, and 7452 tokens. However, the processing of tokens whose left and right hand extents are the same is shared in our LCNP parser (and in a standard GLL or GLR character level parser) and thus only 43 are actually processed.

If we use the longest match disambiguation within each token we get 8 indexed lexicalisations with 148 tokens, the same as for the non-indexed case, but with only 22 distinct tokens to be processed.

Table 2 gives the numbers of indexed token strings (ITSs) for the examples considered in Section 2.1. However, rather than giving the total number of tokens in these ITSs we give the number that are actually processed by the parser as a result of the sharing. We note that, as we would expect, the numbers of lexicalisations in the presence of the LM disambiguation are the same as for the non-indexed cases in Section 2.1.

The data clearly shows the importance of the shared parsing approach in making multi-lexing feasible. The number of lexicalisations to be parsed without any prior disambiguation, even in the case of non-indexed lexicalisations discussed in Section 2.1, is very much greater than the current estimated age of the universe (4.32×10^{17} seconds) for normal sized programs such as Life.java. However, with shared parsing the number of tokens that need to be considered is in the low thousands.

3 Parsing with lexical choice

[6] presents clustered nonterminal parsing (CNP), a version of generalised LL (GLL) parsing that returns a set of binary subtree representations (BSRs) which encode all the derivations of an input string. A shared packed parse forest (SPPF) representation of the derivations can be extracted from the BSR set in a straightforward way if desired [6].

In this paper we give a new GLL style algorithm, LCNP, which takes as input a character string and calls a lexer each time it needs an element of an indexed lexicalisation of the string. The lexer is an input to the LCNP algorithm and can be configured in any way a user requires, be based on any lexical technique, and return either all tokenisations or a subset determined by any specified disambiguation rules. LCNP parses all the returned lexicalisations together, sharing common parts, in time which is worst case cubic in the length of the underlying character string. The output is a BSR set with respect to the grammar terminals/tokens, and thus the constructed derivations are, as is conventional, with respect to the user specified token level grammar not the character level grammar.

We note here that GLL parsers can in fact be extended, in a manner similar to LCNP, so that they take as input any set of indexed token strings, without reference to any particular character string, and efficiently parse all of the input strings concurrently [13]. However, in this paper we will focus on the integrated lexer approach. The important point though

u	length	total ITS			tokens (shared)		
		NoD(u)	LM(u)	LP(u)	NoD(u)	LM(u)	LP(u)
Ex1	26	340	8	2	43	22	20
Ex2	169	8×10^{16}	256	2	269	82	75
Life	5859	3×10^{758}	7×10^{65}	1×10^{18}	15183	2370	2210
Linden	961	1×10^{121}	3×10^{13}	4×10^{6}	1961	403	380
Sand	5685	8×10^{714}	8×10^{73}	3×10^{28}	14456	2163	2012

Table 2. Indexed lexicalisations

is that LCNP can accept tokens from different lexicalised strings, not that the lexer is integrated within the parser.

3.1 Notation and BSR sets

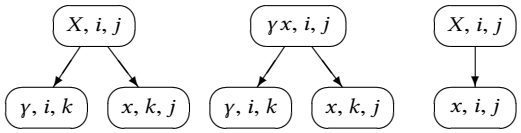
A *context free grammar* (CFG) consists of a set **T** of terminal (token) names, a set **N** of nonterminals disjoint from **T**, a start symbol $S \in \mathbf{N}$, and a set of grammar rules $X ::= \alpha_1 \mid \ldots \mid \alpha_t$, one for each nonterminal $X \in \mathbf{N}$, where each α_k, $1 \leq k \leq t$, 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 $\sigma \stackrel{*}{\Rightarrow} \tau$.

Derivations can be represented as ordered rooted trees. The root node is labelled with the start nonterminal, interior nodes are labelled with nonterminals and the leaf nodes are labelled with terminals or ϵ. The children of a nonterminal node X correspond, in order, to the right hand side of a production rule for X. To ultimately achieve worst case cubic size data structures, the derivation trees are binarised to the left in the standard way by introducing intermediate nodes. A right child is always labelled with a terminal, nonterminal or ϵ, but the left child, if it exists, may be an intermediate node.

In an indexed binarised derivation tree the node labels have additional integer extents, (x, i, j). The left extent of the left-most leaf is 0, the left extent of any other leaf is the right extent of its left sibling, and the right extents of leaves are defined so that the labels are of the form (ϵ, i, i) or $(a, i, i+1)$ if a is a terminal. The left(right) extent of an interior node is the left(right) extent of its left(right) child.

The indexed binarised derivation trees for all derivations of a given string can be merged into what is called a shared packed parse forest, a worst case cubic size representation of the potentially infinite set of such trees. Shared packed parse forests were introduced in [3] and a discussion of binarised SPPFs for GLL parsers can be found in [8].

A binary subtree representation (BSR) is a 4-tuple (Ω, i, k, j), where Ω is either a production rule $X ::= \alpha$ or a string β of length at least two such that there is a production rule of the form $X ::= \beta\gamma$, and $0 \leq i \leq k \leq j$. A BSR element corresponds to a subtree, of depth two, of an indexed binarised derivation tree. The parent of the subtree corresponds to Ω, its extents are i, j and its children have extents i, k and k, j. BSR sets are introduced and discussed in detail in [6] and rather than repeat the definitions we just give illustrative examples. For a terminal or nonterminal x and non-empty string γ, the BSR elements $(X ::= \gamma x, i, k, j)$, $(\gamma x, i, k, j)$ and $(X ::= x, i, i, j)$ correspond, respectively, to the subtrees



In this paper it is only necessary to have a basic understanding of BSR sets as motivation for the functions that construct them as part of the LCNP specification. The functions themselves are simple and are specified below. BSR sets can be used directly for semantic analysis and code generation at later stages in a compiler, but to turn an LCNP parser into an SPPF builder, if that is desired, then the SPPF extraction algorithm given in [6] can be called post-parse.

3.2 LCNP: lexical choice CNP parser specification

We now give the specification for a GLL parser for Γ which takes as input a character string and a lexical function. (The token string input CNP algorithm is described in [6].) The input character string will be held in the variable I and is terminated by the end of string symbol, denoted by \$.

3.2.1 Lexical considerations

We require the lexical function to take as input a character string, $u = a_0 \ldots a_{m-1}$, and a terminal t and to return a set of input indices j such that $a_0 \ldots a_{j-1} \in t$.

$$lex(a_0 \ldots a_{m-1}, t) \subseteq \{j \mid a_0 \ldots a_{j-1}\}$$

We also need a special end-of-string character \$ and a corresponding token and we require

$$lex(\$, \$) = \{1\}$$

The function $lex()$ is called either to compute the next input position(s) or to check for the existence of lexicalisations against some 'lookahead' terminal set, see $lexLKH()$ and $valid()$ below.

One of the attractions of our LCNP algorithm is that it makes no assumptions about how $lex()$ is implemented or

what, if any, filtering (disambiguation) methods it has applied to the set it returns. However, of course, the efficiency of $lex()$ impacts on the efficiency of the LCNP algorithm that calls it. Whether or not lexical disambiguation is applied before $lex()$ returns is a property of the particular function $lex()$ the language specifier chooses to implement. In Section 4 we report on the results of running LCNP with different versions of $lex()$ which apply each of the disambiguation variations discussed in Section 2. For illustration of LCNP, we will use three versions specified as follows.

$$lexFull(a_0 \ldots a_{m-1}, t) = \{ j \mid a_0 \ldots a_{j-1} \in t \}$$

Let $max_{I,t}$ denote the maximum element of $lexFull(I, t)$ if it exists, otherwise $max_{I,t} = -1$. Then

$$lexLong(I, t) = \begin{cases} \{max_{I,t}\} & \text{if } max_{I,t} \neq -1 \\ \emptyset & \text{otherwise} \end{cases}$$

$$lexLongPriority(I, t) = \begin{cases} \emptyset & \text{if, } \exists (t' > t)\ \ max_{I,t} = max_{I,t'} \\ lexLong(I, t) & \text{otherwise} \end{cases}$$

These essentially implement the three disambiguation specifications *NoD*, *LM* and *LP* discussed in Section 2. However, a compiler designer can specify any form of lexer they want, provided it meets the general requirements given at the start of this section.

Our parsers use one terminal symbol lookahead. For a grammar position $X ::= \alpha \cdot \beta$ we define $predict(\beta, X)$, a set of terminals which can be the first symbol of a string derived from this point in the grammar:

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

3.2.2 Parser overview

LCNP algorithm itself is similar to the token string input CNP algorithm and, in recursive descent style, the parsers have a section of code for each alternate of each nonterminal. To handle nested nonterminal calls, the call return positions are recorded in a Call Return Forest (CRF, see below). Global variables, c_I and c_U, hold the current input index and the index of the current CRF node, respectively. The algorithm is written assuming that the input character string is held in a global variable I to which all functions have access. Thus we use a slightly different signature $lex(i, t)$, with the specification that $lex(i, t)$ gives the same result as $lex(I_i, t)$ where I_i is the right (postfix) substring of I which starts at position i.

We also define a predicate $valid(i, T)$ as the result of a lexical test against the token set T.

$$valid(i, T) = \begin{cases} true & \text{if } \exists (t \in T)(lex(i, t) \neq \emptyset) \\ false & otherwise \end{cases}$$

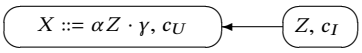
Flow of control is handled using an outer descriptor selection loop together with algorithm line labels.

LCNP descriptors, $(X ::= \alpha \cdot \beta, h, i)$, comprise a grammar slot, $X ::= \alpha \cdot \beta$, that forms a line label, the integer index of a CRF cluster node, and an input position. As descriptors are created they are stored for processing, in a set $\mathcal{R}$. In order to ensure that repeated computations are not performed, the set of all descriptors which have been created are also stored in a set $\mathcal{U}$, and an element is only added to $\mathcal{R}$ if it is not already in $\mathcal{U}$.

When the descriptor $(X ::= \alpha \cdot \beta, h, i)$ is removed from $\mathcal{R}$ the parser recommences its execution at input position $c_I = i$ and at the line of the code for $X ::= \alpha \cdot \beta$.

If $\beta = x\gamma$ where x is a terminal then $lex(c_I, x)$ is called. For each $k \in lex(c_I, x)$ there is a lexeme in the pattern of x between positions c_I and $j = c_I + k$ in the input character string I. To incorporate lookahead, we construct the set J of j such that $j - c_I \in lex(c_I, x)$ and $valid(j, predict(\beta, X))$ is true. This construction is performed by the function $lexLKH()$ defined in Section 3.4.1. For each $j \in J$ a descriptor $(X ::= \alpha x \cdot \gamma, c_U, j)$ is created which, when processed, will cause the parser to resume at grammar position $X ::= \alpha x \cdot \gamma$ and input position j. The parser execution then continues by removing the next descriptor from $\mathcal{R}$.[1]

If $\beta = Z\gamma$ where Z is a nonterminal the return position $X ::= \alpha Z \cdot \gamma$, c_U and c_I are stored. Then, for each rule $Z ::= \tau$ such that $valid(I_{c_I}, predict(\tau, Z))$ is true, a descriptor $(Z ::= \cdot\tau, c_I, c_I)$ is created by $ntAdd()$, and the next descriptor is removed from $\mathcal{R}$. In general it is possible for a call to Z to match more than one substring, and for the same sub-parse from Z to belong to several derivations. So the return positions are stored in records, indexed by Z and c_I, containing the entries $(X ::= \alpha Z \cdot \gamma, c_U)$



To keep the size of the records worst case cubic we share nodes with the same label in a *Call Return Forest*, CRF, which takes the role of the GSS in a classical GLL or GLR parser. An example can be seen in Section 3.3. The building and reading of the CRF is carried out by stand-alone, grammar independent, support functions $call()$ and $rtn()$, which are called from the main parser. These functions are specified in Section 3.4.1.

Finally, if we are at the end of a rule $X ::= \alpha\cdot$ then we have successfully matched α to the input substring $a_{c_U} \ldots a_{c_I - 1}$. For each child $(Y ::= \nu X \cdot \mu, l)$ of (X, c_U), provided that $valid(c_I, predict(\mu, Y))$ is true, a descriptor $(Y ::= \nu X \cdot \mu, l, c_I)$ is created so that execution from the corresponding positions can be continued. There is a potential complication in that it is possible for additional children to be added to (X, c_U)

[1] It is possible to avoid creating a descriptor when $|J| = 1$ by just updating the value of c_I and continuing. However, when there are multiple lexeme matches it is possible to return to the same algorithm position and repeat some computation, which is what the set $\mathcal{U}$ guards against.

after a return action has been carried out, and there are cases where it is not possible to order the descriptor processing order to avoid this situation. To deal with it, the return action is recorded as a triple (X, c_U, c_I) in the *contingent return* set $\mathcal{P}$. This return action is then applied to new children when they are created.

As the parser proceeds it also builds a set, Υ, of BSR elements, using the support function *bsrAdd*(). When a descriptor is created $(X ::= \alpha \cdot \beta, k, j)$, if $\beta = \epsilon$ or, if $\beta \neq \epsilon$ and $|\alpha| > 1$, a BSR element $(X ::= \alpha, k, i, j)$ or (β, k, i, j), respectively, is created. The integer i is either the current value of c_I or the index of a CRF node (Y, i) where $\alpha = \gamma Y$. For more discussion on the BSR sets constructed by a CNP algorithm see [6], and for an example see Section 3.3.

LCNP parsers are specified via a set of templates into which grammar symbols and production rules are substituted; these are given in Section 3.4. First we give an example.

3.3 Example

To give a small example we consider the alphabet $\{a, b, c\}$ which has just three characters, and we consider tokens s, t where the lexemes of s are the two strings aa and cc and the lexemes of t are all the nonempty strings of a and b

$$s = (\ aa \mid cc\ ) \qquad t = (\ a \mid b\ )^+$$

The LCNP parser for the grammar

$$S ::= \ s\ S \ \mid \ B \qquad B ::= \ t\ B \ \mid \ \epsilon$$

whose terminals are s, t is as follows.

let $I = a_0 \ldots a_{m-1}$ denote the input character string and $a_m = \$$
let m denote the height of Σ
create CRF node $u_0 = (S, 0)$
$\mathcal{U} := \emptyset$; $\mathcal{R} := \emptyset$; $\mathcal{P} := \emptyset$; $\Upsilon := \emptyset$
ntAdd$(S, 0)$
while $\mathcal{R} \neq \emptyset$ {
  remove a descriptor, (L, i, k) say, from $\mathcal{R}$
  $c_U := i$; $c_I := k$; **goto** L
$S ::= \cdot sS$:
  for each $j \in$ *lexLKH*(s, c_I, S, S) {
    bsrAdd$(S ::= s \cdot S, c_U, c_I, j)$
    dscAdd$(S ::= s \cdot S, c_U, j)$ }
  goto L_0
$S ::= s \cdot S$:
  call$(S ::= sS\cdot, c_U, c_I)$; **goto** L_0
$S ::= sS\cdot$:
  rtn(S, c_U, c_I); **goto** L_0
$S ::= \cdot B$:
  call$(S ::= B\cdot, c_U, c_I)$; **goto** L_0
$S ::= B\cdot$:
  rtn(S, c_U, c_I); **goto** L_0
$B ::= \cdot tB$:
  for each $j \in$ *lexLKH*(t, c_I, B, B) {
    bsrAdd$(B ::= t \cdot B, c_U, c_I, j)$
    dscAdd$(B ::= t \cdot B, c_U, j)$ }
  goto L_0
$B ::= t \cdot B$:
  call$(B ::= tB\cdot, c_U, c_I)$; **goto** L_0
$B ::= tB\cdot$:
  rtn(B, c_U, c_I); **goto** L_0
$B ::= \cdot$:
  $\Upsilon := \Upsilon \cup \{(B ::= \epsilon, c_I, c_I, c_I)\}$
  rtn(B, c_U, c_I); **goto** L_0
L_0: }
if (for some α and l, $(S ::= \alpha, 0, l, m) \in \Upsilon$) {report success}
else {report failure}

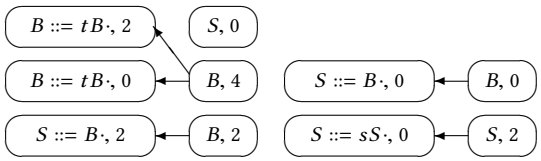
We can run this algorithm on the character string aaab using the lexers *lexLong*() and *lexFull*() defined above.

3.3.1 Input aaab and *lexLong*()

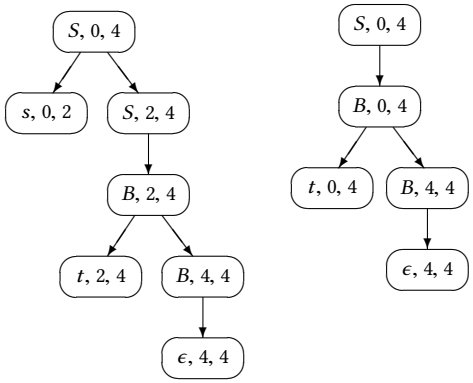
Effectively when the parser runs *lexLong*() will generate the following two indexed token sequences from aaab.

$$(t, 4) \quad (s, 2)(t, 4)$$

The parser constructs the descriptor set
$U = \{(S ::= .sS, 0, 0), (S ::= .B, 0, 0), (S ::= s.S, 0, 2),$
$(S ::= .B, 2, 2), (B ::= .tB, 0, 0), (B := t.B, 0, 4), (B ::= \epsilon., 4, 4),$
$(B ::= tB., 0, 4), (S ::= B., 0, 4), (B ::= .tB, 2, 2), (B := t.B, 2, 4),$
$(B ::= tB., 2, 4), (S ::= B., 2, 4), (S ::= sS., 0, 4)\}$
and CRF



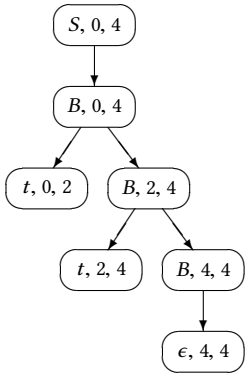
It also returns the BSR set
$\Upsilon = \{(B ::= \epsilon, 4, 4, 4), (B ::= tB, 0, 4, 4), (S ::= B, 0, 0, 4),$
$(S ::= sS, 0, 2, 4), (S ::= B, 2, 2, 4), (B ::= tB, 2, 2, 4)\}$
which embeds precisely the indexed derivation trees for the two ITSs above.



Note the syntax grammar is not ambiguous so there is only one derivation tree for each ITS. If there had been more than one derivation the BSR set would have embedded all of the corresponding derivation trees.

3.3.2 Input aaab and *lexFull*()

Effectively when the parser runs *lexFull*() will generate the following 11 indexed token sequences from aaab.
$(t,4)$ $(t,3)(t,4)$ $(t,2)(t,4)$ $(t,1)(t,4)$ $(t,2)(t,3)(t,4)$
$(t,1)(t,3)(t,4)$ $(t,1)(t,2)(t,4)$ $(t,1)(t,2)(t,3)(t,4)$
$(s,2)(t,4)$ $(s,2)(t,3)(t,4)$ $(t,1)(s,3)(t,4)$
All but the last of these correspond to strings in the grammar. The parser will reject the last one. The set $\mathcal{U}$ and the CRF are too large to diplay here. For compactness we have combined some BSRs, so (Ω, i, K, j) represents all the BSRs (Ω, i, k, j) where $k \in K$.
$\Upsilon = \{(B ::= \epsilon, 4, 4, 4), (B ::= tB, 0, \{1,2,3,4\}, 4),$
$(B ::= tB, 1, \{2,3,4\}, 4), (B ::= tB, 2, \{3,4\}, 4), (B ::= tB, 3, 4, 4),$
$(S ::= B, 0, 0, 4), (S ::= sS, 0, 2, 4), (S ::= B, 2, 2, 4)\}$
This set has 14 elements and embeds precisely the derivation trees of the ten syntactically correct ITSs. Of course, the same BSR element can contribute to several derivation trees, this is why the BSR set is more efficient than just the set of trees. For example, the BSR elements $(B ::= tB, 2, 4, 4)$ and $(S ::= B, 0, 0, 4)$ contribute to the derivations on the left and right above, respectively, and to the derivation tree for $(t,2)(t,4)$



$(B ::= \epsilon, 4, 4)$ contributes to all three trees above. It would take too much space here to show the steps the parser takes, but if the reader walks through the algorithm using *lexFull*() they will see that how the parsings of the ITSs are shared. We can identify the position of a terminal in the string from the index of the terminal to its left, so $(t,2)$ appears at positions 0 and 1 and so it is only considered twice even though it appears fours times in the ITSs.

3.4 LCNP generator specification

Throughout this section the following notation is used.
I: a constant variable containing the input string
m: a constant integer whose value is the length of the input
c_I, c_U: integer variables
CRF: a digraph whose nodes are labelled (L, j)
where L is either a nonterminal or a grammar slot
$\mathcal{P}$: set of CRF return actions represented as triples (X, k, j)
Υ: set of BSRs, $(X ::= \mu, i, k, j)$ and (μ, i, k, j)
$\mathcal{R}$: set of descriptors waiting to be processed
$\mathcal{U}$: set of all descriptors constructed so far
\$: end-of-string character

The parser has two global variables c_U and c_I, that hold the current CRF node index and input position, respectively, and a set of support functions. The functions *ntAdd*() and *predict*(β, X) have to be constructed for a given grammar Γ by the parser generator, and the latter makes use of the standard FIRST and FOLLOW sets [1] which must also be constructed by the parser generator. The other functions are grammar independent. All functions assume the existence of a global input string I, global sets Υ, $\mathcal{P}$, $\mathcal{U}$, $\mathcal{R}$, and a global CRF graph.

3.4.1 LCNP support functions

```
ntAdd(X, j) {
    for all(grammar rules X ::= τ)) {
      if valid(j, predict(τ, X)) { dscAdd(X ::= ·τ, j, j); } } }

dscAdd(L, i, k) { if (L, i, k) ∉ U {add (L, i, k) to U and R}}

lexLKH(t, i, β, X) {
  let J = ∅
  for each k ∈ lex(i, t) {
    if (valid(i + k, predict(β, X))) add i + k to J }
  return J }

rtn(X, k, j) {
  if ((X, k, j) ∉ P) {
      add (X, k, j) to P
      for each child u of (X, k) in the CRF {
          let (Y ::= vX · μ, i) be the label of u
          if valid(j, predict(μ, Y)) {
            dscAdd(Y ::= vX · μ, i, j)
            bsrAdd(Y ::= vX · μ, i, k, j) } } }

call(L, i, j) {
  suppose that L is Y ::= vX · μ
  if there is no CRF node labelled (L, i) create one
  let u be the CRF node labelled (L, i)
  if there is no CRF node labelled (X, j) {
        create a CRF node v labelled (X, j)
        create an edge from v to u
        ntAdd(X, j) }
  else { let v be the CRF node labelled (X, j)
        if there is not an edge from v to u {
          create an edge from v to u
          for all ((X, j, h) ∈ P) {
            if valid(h, predict(μ, Y)) {
              dscAdd(L, i, h); bsrAdd(L, i, j, h)  } } } }
```

$bsrAdd(X ::= \alpha \cdot \beta, i, k, j)$ {
  if($\beta = \epsilon$){ insert $(X ::= \alpha, i, k, j)$ into Υ }
  else if($|\alpha| > 1$){ insert (α, i, k, j) into Υ }}

3.4.2 The LCNP templates

Now we give the code templates which specify the LCNP parser. A parser is obtained by substituting the specific grammar production rules into the templates.

For each nonterminal X in the grammar there is a section of the algorithm, $code(X)$, which will be defined below. In addition to the grammar slot labels, we require a label L_0 which labels the end of the controlling while loop, then the statement `goto` L_0 is equivalent to `break` in C-style programming languages.

When the descriptors have all been dealt with, the test for acceptance is made by checking for the existence of a BSR of the form $(S ::= \alpha, 0, l, m)$, for some α and l, where m is the length of the input string.

We suppose that the nonterminals of the grammar Γ are $A, \ldots, Z$, with start symbol S. Then the LCNP parser for Γ is given by:

let $I = a_0 \ldots a_{m-1}$ denote the input character string and $a_m = \$$
create CRF node $u_0 = (S, 0)$
$\mathcal{U} := \emptyset$; $\mathcal{R} := \emptyset$; $\mathcal{P} := \emptyset$; $\Upsilon := \emptyset$
$ntAdd(S, 0)$
while $\mathcal{R} \neq \emptyset$ {
  remove a descriptor, (L, k, j) say, from $\mathcal{R}$
  $c_U := k$; $c_I := j$; **goto** L
  $code(A)$
  ...
  $code(Z)$
L_0: }
if (for some α and l, $(S ::= \alpha, 0, l, m) \in \Upsilon$) {report success}
else {report failure}

We give the specification for $code(X)$ in terms of functions $code(X ::= \alpha \cdot \beta)$. We refer to the specifications of $code(X ::= \alpha \cdot \beta)$ as the *LCNP templates*. Suppose that the grammar rule for X is $X ::= \tau_1 \mid \ldots \mid \tau_p$, we define:

$code(X) =$ $X ::= \cdot\tau_1$:
  $code(X ::= \cdot\tau_1)$
  $rtn(X, c_U, c_I)$; **goto** L_0
  ...
$X ::= \cdot\tau_p$:
  $code(X ::= \cdot\tau_p)$
  $rtn(X, c_U, c_I)$; **goto** L_0

Given a slot E we define $code(E)$ as follows, where t is any terminal and Y is any nonterminal, α and β are (possibly empty) strings of terminals and nonterminals, and L denotes the label corresponding to the slot $X ::= \alpha Y \cdot \beta$.

$code(X ::= \cdot) = \Upsilon := \Upsilon \cup \{(X ::= \epsilon, c_I, c_I, c_I)\}$

$code(X ::= \alpha t \cdot \beta) =$ **for** each $j \in lexLKH(t, c_I, I, \beta, X)$ {
  $bsrAdd(X ::= \alpha t \cdot \beta, c_U, c_I, j)$
  $dscAdd(X ::= \alpha t \cdot \beta, c_U, j)$ }
  goto L_0
$X ::= \alpha t \cdot \beta$:

$code(X ::= \alpha Y \cdot \beta) =$ $call(X ::= \alpha Y \cdot \beta, c_U, c_I)$; **goto** L_0
$X ::= \alpha Y \cdot \beta$:

$code(X ::= \cdot x_1 \ldots x_d) =$
  $code(X ::= x_1 \cdot x_2 \ldots x_d)$
  $code(X ::= x_1 x_2 \cdot x_3 \ldots x_d)$
  ...
  $code(X ::= x_1 \ldots x_d \cdot)$

4 Java case study

In this section we use an LCNP multi-parser for Java to investigate both the numbers of potential lexicalisations that are syntactically correct and the work required to parse them.

We consider again four of the example programs considered in Section 2. Ex1.java is not a complete program so we instead we use Ex3.java which is the same as Ex2.java except that we replace −− with − − −.

```
int x = y---z;
```

In Table 3, we give the number of those lexicalisations that were successfully parsed (sentences). This gives a measure of how much lexical disambiguation could be 'left' to a post parse process to resolve.

For Ex3.java the two lexical choices, `'--' '-'` and `'-' '--'`, for − − − both give rise to syntactically correct lexicalisations.

The high numbers for the case of no disambiguation arise because in many places in Java where an assignment `ID = exp` is legal then so is a declaration `ID ID = exp`. So without longest match within `ID` an assignment statement also lexicalises to a declaration. Because of the nature of Java the number of instances of ambiguity is approximately $log(s)$ where s is the number of sentences. So although the number of sentences is large, the number of instances of ambiguity could be manageble. Thus it may be possible, for reasonably well behaved languages, to dispense with most lexical disambiguation and pass the valid sentences to a post-parse disambiguator which could use context information to resolve any ambiguity. This may be useful for domain specific language applications. We will discuss disambiguation before parsing is complete further in Section 4.2.

4.1 Parser data structure cardinalities

Wall clock times and actual memory usage are implementation and hardware dependent. We have implemented our

u	total ITS				sentences			
	NoD(u)	LM(u)	LP(u)	LAP(u)	NoD(u)	LM(u)	LP(u)	LAP(u)
Ex2	8×10^{16}	256	2	1	8	8	1	0
Ex3	1×10^{17}	384	3	1	16	16	2	1
Life	3×10^{758}	7×10^{65}	1×10^{18}	1	2.0×10^{39}	1.4×10^{20}	1	1
Linden	1×10^{121}	3×10^{13}	4×10^{6}	1	512	32	1	1
Sand	8×10^{714}	8×10^{73}	3×10^{28}	1	4.5×10^{27}	1.1×10^{20}	1	1

Table 3. Syntactically correct lexicalisations

LCNP parser exactly as described in Section 3.2 and we have not focused on runtime or memory efficiency, (although our implementation does parse Life.java in under 0.099 seconds). An implementation independent measure of the work done by a GLL-style parser is the number of descriptors created, as there is an execution of the outmost loop of the parser for each descriptor. Memory usage is dominated by the size of the output derivation representation, which is worst case cubic in the length of the input string. So we report the size of the descriptor sets, $|\mathcal{U}|$, and the size of the output BSR sets, $|\Upsilon|$.

As we have mentioned, an alternative approach to multi-lexing is to use a character level grammar and require the parser to directly fulfil the lexer role. So we also report here the size of the descriptor sets created by a classical GLL parser, SGLLJ, for the character level Java grammar. Standard GLL parsers construct SPPF outputs. The size of an SPPF is worst case cubic if the SPPF is binarised. We give the number of SPPF nodes for comparison with the size of the BSR sets produced by the LCNP parser. This data is presented in Table 4. For the GLL parser, the disambiguation is carried out post-parse by removing nodes from the SPPF. So the number $|\mathcal{U}|$ of descriptors created is the same for all the disambiguation strategies and is just quoted once in Table 4. As we would expect, the number of LCNP descriptors is much less than for SGLLJ. This is partly due to the node clustering in the LCNP CRF, but also because the LCNP grammar is at 'token' level and thus has fewer nonterminals and grammar rules than the SGLLJ grammar.

The size of the LCNP BSR sets is much less that the corresponding SPPF node numbers. This is partly because, as well as the packed nodes which essentially correspond to BSR elements, the SPPF has nonterminal and terminal nodes. However, this only accounts for a difference of about factor two. The much greater difference is again because the SPPF has nodes for the character level parts of the derivation.

4.2 Disambiguation before parsing is complete

In a sense, the primary research question addressed in this paper is what is the impact of allowing more powerful approaches to lexicalisation? Tables 3 and 4 give data which can be used to inform our conclusions.

The example LCNP lexical disambiguation specifications we have considered are all applied before the parsing is complete. In contrast the SGLLJ versions are applied after the parser has constructed the SPPF. The latter is safer in the sense that we can ensure that lexical disambiguation does not remove all possible syntactically correct lexicalisations. Table 3 shows that, for the example Java programs and LM or LP, the former approach is also safe in this sense, but with LAP it is not. Of course, LNCP can also take the latter approach by using the NoD specification and carrying out the required disambiguation on the constructed BSR set.

In general, there are common cases in which 'on-the-fly' lexicalisation disambiguation such as LM is safe. For example, many programming language specifications require that in the input character string no identifier can be immediately followed by a character that can be in an identifier. So in Java, $ifx < 3$ cannot be interpreted as beginning with the keyword token `if`. In this situation longest match within the identifier token is safe and, as we can see from the data in Table 4, applying LM disambiguation during the parse considerably reduces both the number of parser execution steps and the size of the data structures, whilst not removing all possible lexicalisations. The flexibility of the multi-lexer approach gives LCNP parsers the power to take advantage of such efficiency options.

In certain circumstances it is possible to modify a GLL character level parser, such as SGGLJ, so that it carries out lexical disambiguation during the parse. We say a token t has the *suffix property* if, whenever for some $u \in t$ we have $ua \in t$ then for all $v \in t$ we have $va \in t$. For example, in Java the ID token has the suffix property. In the case of a token t with the suffix property, LM disambiguation can be applied by a character level GLL parser by preventing returns from the nonterminal t in the presence of certain lookahead input characters.

A GLR parser for a character level grammar can similarly perform LM disambiguation during the parse for tokens t with the suffix property by removing the reductions associated with t from certain entries in the LR table.

However, these GLL and GLR modifications are subtle, somewhat ad hoc, and require changes to the generated parser or parse table that are not easy to reason about in general. The multi-lexing approach explicitly separates the

u	LCNP $\|\mathcal{U}\|$			LCNP $\|\Upsilon\|$			SGLLJ $\|\mathcal{U}\|$	SGLLJ SPPF nodes		
	NoD(u)	LM(u)	LP(u)	NoD(u)	LM(u)	LP(u)	all	NoD(u)	LM(u)	LP(u)
Ex2	1346	882	696	421	230	202	7219	2876	1136	1117
Ex3	1367	903	717	433	242	214	7307	2914	1178	1153
Life	48187	33981	28433	15719	8131	7392	303585	114923	46828	46521
Linden	7198	4935	44215	2357	1197	1109	427000	16660	6403	6283
Sand	43435	31234	26010	13841	7468	6830	276080	102681	39063	38812

Table 4. Data structure sizes

lexical activity, allowing it to be specified and reasoned about independently of the parsing technology.

There are also many cases in which on the fly LP disambiguation is safe. However, there are languages, such as PL/1 and the Fortran family, where keywords can be used as identifiers if the phrase level context ensures that the keyword would not be valid at that point.

There are also languages where a keyword must be returned even if it occurs in what in Java would be an identifier. For example *DOFRED* should be lexicalised as `'DO' ID`. The data in Table 3 suggests that using LM and not LP would not result in many sentences that would need disambiguation post-parse, and Table 4 indicates that the efficiency gain of LP over LM is less significant than that of LM over NoD. With the multi-lexing approach a language implementer can investigate these trade-offs and make informed decisions about an optimal place to set the divide between lexical and phrase level disambiguation.

5 Concluding remarks

5.1 Related work

An initial version of multi-lexing and associated GLL parsing is presented in [13]. The thesis includes a discussion of lexical disambiguation and the GLL parser described accepts a separately precomputed set of tokens with extents and constructs an SPPF.

Aycock and Horspool [2] described an approach for dealing with the specific case when two or more tokens share a common lexeme, the situation that is commonly addressed by token priority. Their motivating example was the language PL/I in which keywords such as `IF` can also be identifier names. The idea is that when a lexeme that matches more than one token is found, a so-called Schrodinger token is returned. When the parser reaches that token it decides, if it can, which of the actual tokens to use based on the grammar context. A general parser is used but only one token string is actually parsed, so this is not multi-parsing.

In [4] and [5] a non-deterministic lexer for French is described. The primary motivation is dealing with lexemes, such as *a priori*, which can include spaces and may have more than one lexicalisation. There is some investigation into handling multiple lexicalisations but there is no formal treatment; the methods used are specific to the French translation application.

Scannerless parsing, using a grammar defined at character level, has also been explored in depth the literature [10]. The tokens from the traditional representation appear as non-terminals in the character level grammar and, unless on-the-fly disambiguation is applied, the parser effectively constructs and parses all the original lexicalisations. The resulting grammar is highly ambiguous but the emergence of practical general parsing algorithms has allowed this approach to implemented. For example, it is used in ASF+SDF [9] and implemented in an SGLR parser [11] which is used in Stratego/XT [12]. Rascal [7] also provides support for character level parsing.

5.2 Summary

In this paper we have considered the possibility of making the lexicalisation phase of compilation more powerful by avoiding the need for the lexer to return a single token string from the input character string. The naive approach is to allow the lexer to return all possible lexicalisations, and to parse each of them. However, the large number of lexicalisations makes it impractical to parse each one individually, but it is possible to share the parsing of common subparts. This has the effect of reducing the number of tokens parsed from the product of the number of lexicalisations of the components to their sum. Our LCNP implementation used in Section 4, which has not been optimised, parsed the 3×10^{758} non-disambiguated indexed lexicalisations of Life.java in 0.099 seconds.

A simple way to achieve shared parsing is to specify the grammar at character level, and effectively use the parser as the lexer. As we have practical worst case cubic parsers, this makes the process worst case cubic in the length of the character string, but in reality this is still very large. To improve execution time some lexical disambiguation could be applied so only a subset of all lexicalisations has to be parsed, allowing a trade-off between flexibility and efficiency. This is somewhat uncomfortable in the character level grammar approach because lexical disambiguation techniques such as longest match are hard to reason about in the case of context

free grammars. In practice what happens is that some non-terminals are designated as 'lexical' and the disambiguation is applied only to their sublanguages. This illustrates the discomfort associated with merging the lexical and phrase level syntax of a language. Our approach retains the separation between grammar terminals and the corresponding lexical tokens but still permits the shared parsing obtained in the character level parser. This has the advantage of allowing the parser to use terminal level lookup, keeping lexical level disambiguation separate from the context free grammar, and allowing potentially more efficient lexer implementations, such as linear time finite state automata based techniques.

References

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

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

[3] Sylvie Billot and Bernard Lang. 1989. The Structure of Shared Forests in Ambiguous Parsing. In *Proceedings of the 27th conference on Association for Computational Linguistics.* Association for Computational Linguistics, 143–151.

[4] J.-P. Chanod and P. Tapananeinen. 1996. A Non-deterministic Tokeniser for Finite-State Parsing. In *ECAI 96. 12th European Conference on Artificial Intelligence*, W. Wahlster (Ed.). JohnWiley & Sons, Ltd., 10–12.

[5] J.-P. Chanod and P. Tapananeinen. 1999. Finite State Based Reductionist Parsing for French. In *Extended Finite State Models of Languages*, A. Kornai (Ed.). Cambridge University Press, 72–85.

[6] A. Johnstone E. Scott and L.T. van Binsbergen. 2018. Derivation representation using binary subtree sets. *Science of Computer Programming* (2018).

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

[8] Elizabeth Scott and Adrian Johnstone. 2013. GLL parse-tree generation. *Science of Computer Programming* 78 (2013), 1828–1844. https://doi.org/10.1016/j.scico.2012.03.005

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

[10] Eelco Visser. 1997. *Scannerless Generalised-LR Parsing.* Technical Report P9707. University of Amsterdam.

[11] Eelco Visser. 1997. *Syntax definition for language prototyping.* Ph.D. Dissertation. University of Amsterdam.

[12] Eelco Visser. 2004. Program transformation with Stratego/XT: rules, strategies, tools, and systems in StrategoXT-0.9. In *Domain-Specific Program Generation*, C.Lengauer et. al (Ed.). Lecture Notes in Computer Science, Vol. 3016. Springer-Verlag, Berlin, 216–238.

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