Better Parsing Through Lexical Conflict Resolution

Nathan Keynes

November 2, 2007

Contents

Chapter 1

Introduction

1.1 Introduction

The standard technique today for automatic generation of parsers is to use tools such as yacc and lex, to produce an LALR(1) parser and DFA-based scanner respectively. The scanner is responsible for reading the input character stream, and separating it into a sequence of tokens. The parser than operates on the token sequence to perform the actual parsing.

The reason for using this type of two-phase arrangement is primarily for efficiency. By making this separation is is possible to retain single lookahead for both the scanner and parser, whereas a single-phase parser would normally require arbitrary lookahead (in order to retain one token of lookahead).

However, there are problems with this from the point of view of a language implementor - not only is it necessary to manually create an interface between the two phases, but the very act of separating a grammar out into lexical and syntactic components normally results in the creation of conflicts in the lexical sub-language. A language designer is then forced to either deal with these conflicts manually, or to design the language with the limitations of the tools in mind.

1.2 Contributions

The aim of this thesis is to investigate the area of lexical conflicts in general, and to show that automatic conflict resolution is both a useful and feasible technique. To this end, practical algorithms for implementing this are developed, and an example implementation produced for demonstration purposes. In particular, it is shown that this can be done with little or no runtime[1] cost.

[1] Hereinafter "runtime" is assumed to refer to the execution of the generated parser code

1.3 Notation

The following notation is used throughout for grammar extracts:

:	grammar production
=	lexical production
START	the start symbol
WHITESPACE	the special symbol denoting whitespace, which can occur anywhere
identifier	non terminal symbols
'quoted string'	literal string
"double quoted string"	regular expression (à la lex)

A simple grammar might then look like:

```
START: 'begin' expr 'end' ;
expr: IDENT | ``[0-9]+'' | expr '+' expr ;
IDENT = ``[A-Za-z][A-Za-z0-9]*'' ;
WHITESPACE = ``[ \t\r\n]+'' ;
```

Chapter 2

Related Work

This chapter discusses previous work on lexical conflict resolution, and other work which attempts or manages to solve similar problems.

2.1 Conflict Detection and Resolution - Nawrocki

The notion of automatic lexical conflict detection and resolution appears to have first been proposed by Nawrocki in [8]. In that paper the basic types of lexical conflicts are characterized, and some example techniques proposed for detecting and resolving them.

The fundamental idea is that the lexical and syntactical specifications can be analyzed together, and that lexical conflicts can then be resolved at runtime on the basis of the current parser state. As a result, the scanner can effectively make its decisions on the basis of the preceding string, rather than being required to treat each token in isolation. Some of the more useful effects of this are discussed in the following chapter.

However, the algorithms given are not necessarily practical (or sufficient), indeed no mention is given therein as to their efficiency or practicality. Neither is any attempt made to make use of the techniques within a "real" parser generator system, so there is no information as to the relative speed of a scanner using these techniques.

2.2 Scannerless Parsing - NSLR

Scannerless parsing was proposed by Salomon and Cormack in [10]. Their basic notion was to discard the lexical analysis stage altogether, and to operate an LR parser directly upon the character input. The primary motivation for this precise approach is unclear, although it was apparently felt that producing two separate specifications for a language was suboptimal; "The compiler writer suffers in that he must partition the grammar for his programming language into two interelated [sic] grammars, and he must design an interface between the two phases."(ibid).

Without disagreeing with this statement, it does not seem that this necessarily leads to the conclusion that the parsing system should be reduced to a

single phase - merely that the multiple phases, if used, should ideally be produced from the single language specification. Indeed this is the approach taken in this paper.

The authors conclude, however, that the preferred metalanguage is a character level EBNF, and proceed to attempt to produce an efficient character level parser from that input. While their paper does not include any comparative timings with their single-phase parser, it would appear that it should not necessarily be significantly (or at all) slower than the equivalent two-phase parser.

There are definite advantages to this approach - not only is it in principle simpler than a two-phase parser, it also deals with lexical conflicts of the nature discussed in this paper as a matter of course (since the LR parser already has the information needed to resolve such conflicts). However, there are two major difficulties, to which S&C devote the bulk of their paper.

The primary problem is that of lookahead. The majority of languages in use today require (at least) one token of lookahead for correct parsing. A scannerless parser created from a standard LALR(1) parser, however, would only have a single character of lookahead, which would rarely be sufficient. The solution devised by S&C to this problem was to use a non-canonical SLR(1) parser (christened NSLR(1)) developed by Tai[12], which permits the parser to continue ahead after encountering an ambiguity, and can use a following *nonterminal* to guide parsing rather than relying on terminal symbols, thereby escaping the single character lookahead restriction. However it is noted in [14] that this parser "...works only for a limited set of grammars, making grammar development difficult".

At the same time, it suffers from this very non-canonicality - the parse order can be quite unpredictable[1], and may in fact differ depending on which tokens follow immediately after a possible reduction. Whether this presents a major difficulty for the language writer is arguable, but it is not uncommon for semantic actions to be written to take advantage of an ordered, well-known parse order.

The second difficulty is that of lexical ambiguity. Languages are frequently highly ambiguous at the lexical level, for the simple reason that specifying an unambiguous lexical set can be quite difficult. The common keyword/identifier ambiguity, for example, is extraordinarily difficult to resolve in a formal manner using only regular languages. In the single EBNF metalanguage used by S&C, however, this problem becomes more critical, as these conflicts can no longer be readily resolved operationally as they are in lex.

The solution described is to implement two additional type of rules, in addition to grammar rules, to whit "exclusion rules" and "adjacency restrictions". These, respectively, indicate that a given production cannot produce some particular string, and that two symbols cannot occur sequentially without some intervening symbol(s). It is claimed that these are sufficient to disambiguate typical programming languages, which does not seem unreasonable.[2]

[1] ie, difficult for a language writer to determine without examining the resulting automata

[2] As an interesting aside, it was demonstrated in [14] that a grammar with exclusion rules is able to describe some non-context-free languages

2.3 Scannerless Parsing - GLR

Salomon and Cormack's work was later extended by Visser in [14] to use a Generalized LR[13] (GLR) parser, rather than the NSLR(1) parser. This has some significant advantages over the earlier work, in that the GLR parser is both substantially more powerful, and retains (technically) a canonical parse order.

Visser also added a "prefer literals" rule, which permits some simplification of the grammar in the common case (keywords override identifiers) by not requiring explicit exclusion rules for these cases.

While no performance measurements were given, the GLR parser used was based on that of Rekers[9] who found that "...the GLR parser is about three times as slow as the YACC parser, ..." for a Pascal grammar. This was believed to be primarily a result of the GLR parser being based off LR(0) parse tables rather than LALR(1), as well as the fact the the yacc tables were (as required) conflict free. Additionally, the GLR parser was implemented in Lisp, rather than C as for yacc, a fact which makes direct comparison difficult.

It is also noted the the GLR algorithm has exponential worst case performace, although in fairness this occurs only on highly ambiguous grammars and input strings - which a simple LALR(1) parser is unlikely to handle well in any case. It would be interesting to compare performance of a scannerless GLR scanner against a conventional two-phase system, considering that typical scannerless grammars are LALR(∞).

One other issue with GLR parsers is that they are typically designed and optimized for the generation of parse trees, not for the processing of semantic actions (as is the case for yacc and most lalr(1) parser generators). While parse trees are perfectly appropriate in some contexts, for many applications they would represent an unnecessary overhead compared to performing direct processing from grammar actions. It is not clear whether it would in fact be possible to use a GLR parser in this manner, but it is expected that it would be quite difficult due to the many discarded sub-parses which must be backtracked. Also GLR parsers will normally merge back together sub-parses when they reach equal states, something which would likely be impossible with semantic actions.

For these reasons, scannerless GLR parsing has not been considered further in this paper. In circumstances where the language being parsed is not LALR(1), a GLR parser may be appropriate (Fortran77 for example). Otherwise, however, it would appear that at present a two-phase parser is more practical.

2.4 AnaGram

AnaGram is a parser generator developed by Parsifal Software [3]. It produces what are essentially scannerless LALR(1) parsers, but with some added lookahead hacks to handle keywords[4]. It thereby represents something of a compromise, in that there exist many grammars which it cannot correctly disambiguate (as non-keywords have the single character lookahead problem) but can be handled without trouble by lex/yacc.

[3] http://www.parsifalsoft.com/

[4] literal strings

AnaGram also includes a number of other enhancements over lex/yacc, in particular its incorporation of semantically determined actions and automatic whitespace productions (both discussed later in this paper). The input language of AnaGram is similar in spirit to the EBNF proposed in [10], although the actual concrete syntax differs.

Chapter 3

Analysis

3.1 Lexical Conflicts

Lexical conflicts can be broadly categorized as being either inherent in the design of the language (ie START: 'begin' | "[a-z]+"), and those that directly result from the separation of the language processing into lexical and syntactic phases (ie START: 'begin' "[a-z]+"). The first language above is ambiguous as specified - the second language is perfectly well-defined, but does not sit well with the standard lexical/syntactical split. To demonstrate, an equivalent lex/yacc specification (eliding constants, etc) would be:

```
bad.l:
  begin    { return KEYWORD_BEGIN; }
  [a-z]+   { return IDENT; }

bad.y:
  START : KEYWORD_BEGIN IDENT
```

However, this is subtly different - lex now has a conflict between the expressions for 'begin' and "[a-z]+" (on the input string "begin"), which did not exist in the grammar expressed as a whole. The net result is that the string "beginbegin" has become unparseable. There are, of course work-arounds:

```
better.l:
  %x SR
  <0>begin   { BEGIN(SR); return KEYWORD_BEGIN; }
  <SR>[a-z]+ { BEGIN(0); return IDENT; }
```

In other words, the language developer is required to explicitly indicate to the scanner generator that certain tokens can only occur in certain contexts, via the use of "start states" So... What's the problem with this? It works, right? Well yes, but there are a few points worth noting:

1. It increases complexity both in the scanner and in the language as a whole.
2. The correct token to scan may depend on not merely the previous token, but on several further back in the input, resulting in chaining of the start states or other hacks and thereby increasing complexity even further.

3. It's fragile. That is, the scanner and parser can become very tightly coupled, resulting in a situation where changes to the language may require non-obvious changes to the start states.

4. The scanner is duplicating work *already performed* by the parser, in keeping track of left context.

5. The language developer is forced to do work in determining contexts which *should* be able to be determined algorithmically, that is, which is derivable from the parser input.

6. It is possible to do better, at little to no runtime cost and reasonable generation-time cost.

Here endeth the rationale for this thesis.

3.2 Taxonomy

Nawrocki[8] identifies two primary classes of lexical conflict. The Identity conflict is defined as the case where there exists a pair of distinct tokens such that there exists at least one string which is in the language of both tokens. An example of this type of conflict is given above in the previous section.

The second class of conflict is the "Longest Match", or LM, conflict. This occurs wherever the scanner has the option of accepting a token, or continuing to scan, and can be considered to be analogous to a shift-reduce conflict. More formally, an LM conflict occurs when there exists a pair of (possibly identical) tokens such that there exists at least one string in the language generated by one token, which forms a valid prefix of a string in the other token. To give a (contrived for the sake of brevity) example expressed in lex terms:

```
lm.l:
  word    { return KEYWORD_WORD; }
  word2   { return KEYWORD_WORD2; }
...
```

When processing the string 'word2', the scanner has the option of accepting the string 'word' and resuming with the '2' next call, or continuing on to accept a word2. In fact, lex will always continue to make the longest match available, but this is not always ideal behavior. To further illustrate, a variation on a problem demonstrated in Modula-2:

```
  START: '[' ``[0-9]+'' '..' ``[0-9]+'' ']' |
         ``[0-9]+\.[0-9]*''
lm2.l:
  [0-9]+          { return INTEGER; }
  [0-9]+\.[0-9]* { return FLOAT; }
  \[              { return LBRACKET; }
  \]              { return RBRACKET; }
  \.\.            { return RANGE; }
```

The unfortunate result of the longest match rule here is that in the string '[123..456]' will be separated into '[', '123.' '.' '456' ']'. That is, the longest match for the string following the left square bracket is the floating point number 123., even though this is not an acceptable token in context. The typical resolution for this type of problem is either to use start states, again, or append trailing context (ie add manual lookahead) to some of the tokens. The caveats outlined earlier for start states also apply here without change.

Trailing context on the other hand is perhaps preferable from a maintenance point of view, in that it is typically less fragile than start states. However, in cases where a conflict could be resolved by either left or right context, the use of trailing context does incur a performance penalty, especially variable length trailing context[1]. Also, like start states, it would seem likely that right context could also be subject to automatic generation.

3.3 Conflict resolving scanners

This section outlines the desired behavior of a "conflict resolving scanner", ie a scanner which can detect and resolve lexical conflicts as far as possible. In other words, by making use of left context information, such a scanner is able to determine the locally "correct" scan of a string which has more than one possible scan.

In order to do this, the scanner needs to know what the current left context is, and have a way to determine the correct action is for a given context. The first point is easily obtainable from the state of the parser, since the requisite left context is necessarily embedded in the parser. There is one small exception to this, which arises from limitations of the LALR(1) technique - under certain circumstances states may be merged which would otherwise (under full LR(1)) be separate, and some left context is obviously lost in the process. To deal with this, one could move to a full LR(1) parser (for example, that of Spector[11]), but these problem states fortunately seem to be relatively rare in practice.

As for decision mechanisms for the scanner, there are several ways in which this could be implemented. The algorithms presented in this paper are based around inclusive scanning (described in the following section), however it is worthwhile to examine the alternatives, and to show the motivation for the particular techniques used herein.

3.4 Scanning techniques

"Exclusive scanning" is defined to refer to a method by which only tokens which can be accepted in the current parser state are be returned by the scanner - otherwise an error is returned and (most likely) the input pointer is unchanged. This is in contrast to "Inclusive scanning", wherein the scanner will always try to return a token, even if it is not immediately acceptable. In order to do so we define a "default DFA", which acts as a fall-back when no parser state specific token is relevant. For the purposes of this paper it is assumed that the default

[1] According to the flex manual, the most expensive features are REJECT actions, line numbers, arbitrary trailing context, and backtracking, in that order

DFA is that DFA which would have been generated by lex for the lexical set, ie it uses only longest match and declaration order to resolve conflicts.

Generating a scanner for an exclusive scanner is relatively simple - a separate DFA can be constructed for each parser state containing only those tokens which are acceptable in that state. The DFAs can then be "merged" if desired using well-known DFA minimization algorithms. Constructing an inclusive scanner is significantly more complex, and is covered in the following chapter.

The principle behind an inclusive scanner is that it should behave as much as possible like a simple lex scanner, using the standard resolution rules of longest match and declaration order. However, when the grammar would require an exception to these rules in order to achieve correct parsing, the scanner's behavior is altered for those specific parser states.

Indirect lexical analysis, on the other hand, is nothing new[1], and requires the parser to query the scanner for each possible token it expects, and receive a boolean yes/no response for each token. In this way there needs to be exactly one DFA for each token, and the lexical analyzer has no responsibility for conflicts - it is up to the parser to choose an appropriate query order. However this technique has not been widely used, most likely as it is relatively inefficient compared to the common direct lexical analysis (an indirect scanner may be required to scan a string several times for each of the different possibilities, whereas a direct scanner will only do so once).

A related method, "Generalized scanning", coined from a certain similarity to Generalized LR techniques[2], returns a list of every possible next token which can be read from the immediate input, rather than merely a single token. The decision of which to then accept is devolved onto the parser. This can be done by simply building a standard combined DFA, but retaining all conflicts. At runtime the scanner proceeds as normal, but at every accepting state appends the current token(s) to a list, which is returned when the scanner reaches a terminal state. While, to the best knowledge of the author, a scanner of this type has never been implemented, it is believed that its performance would fall substantially below that of a lex scanner, due to the overhead involved with the token lists. Nevertheless it represents the most flexible solution known, short of a full scannerless GLR parser.

Finally, there is the possibility of using a scannerless parser, as discussed in some detail in the previous chapter. While it would be possible to construct a conflict resolving scanner with any of the above techniques, only the first two would appear to operate with approximately the same efficiency as a lex scanner. The main difference between inclusive and exclusive scanning then is their behavior on erroneous input, ie where the immediate input does not form a currently acceptable token. The inclusive scanner will return a token based on the default DFA, whereas the exclusive scanner will return an error. Which is preferable would depend on the error recovery strategy taken by the parser.

3.5 Error recovery

The issues only arises in the first place because unlike a lex scanner which has no knowledge of preceding input, a conflict resolving scanner is able to detect a

[2]Similar in spirit, at least, if not in mechanics

parse error before it has even returned the token (in fact it is possible to detect an error on the first erroneous character).

Arbitrarily complex recovery schemes are possible here, but in truth any error recovery technique constitutes at best a heuristic, since there is no way to know what the input's author's real intentions were[3]. For the sake of this discussion then, and for simplicity, it is assumed that the parser will make use of panic-mode error recovery, as is commonly produced by yacc-derived parser generators. While there have been proposals for a number of automatic error-recovery tactics, (for example, in the GMD lalr generator [5]), their popularity and usefulness seems to be limited.

The question then becomes, if the immediate input string could be scanned as more than one token, neither of which are currently acceptable, then what should the scanner return? The exclusive scanner simply returns an error, and leaves it up to the parser to retry from another parse state. However, this makes it very difficult for the parser to skip tokens once it has reached an error state.[4]

On the other hand, the inclusive scanner will return whatever token is given by the default DFA, either the longer or higher priority token. Unfortunately this will not always be the best token with which to continue parsing, since the acceptable set will be different in the error state as opposed to the current state. The advantage of the inclusive approach, though, is that since it at least returns *something*, it is easier to print out meaningful diagnostics.

3.6 Metalanguage

It is hoped that the benefits of using a single, unified metalanguage to describe the target language(s) is clear, relative to two independent metalanguages - primarily there is a reduction in user-visible complexity, particular in the interface between the phases, but also in terms of learning two separate metalanguages. It is hoped that there would also be a corresponding reduction in the costs of maintenance, since only one source file needs to be maintained, rather than two heavily interrelated sources.

The metalanguage chosen for use in developing the generator discussed in this paper is almost identical to a yacc grammar, except that lexical tokens can occur directly within rules as inline regular expressions. Additionally a special lexical-only production was added for semantic purposes (see 5.1). A simple example of this is as follows:

```
stmts: stmts stmt  |  ;
stmt : IDENT ':=' ``[0-9]+'' ';' ;
IDENT = ``[A-Za-z][A-Za-z0-9]*'' ;
WHITESPACE = ``[ \t\r\n]+'' ;
```

Sample input

```
Counter := 56 ;
temp42 := 4 ;
```

[3] mind reading hardware excluded

[4] This could perhaps be worked-around by having the error states considered to accept anything, ie falling back to the default DFA under these circumstances. Although this does mean that error states cannot take advantage of conflict resolutions

This design was motivated by several factors. Firstly and primarily it provides a minimum change from yacc (reducing learning curve) while still providing the necessary context for lexical conflict resolution. Secondly, the use of embedded regular expressions, rather than a character-level grammar, is mainly predicated on the continued use of a two-phase parser - deriving a lexical set from a character-level grammar is an as-yet unsolved problem in general (Although it has been solved for certain subclasses of CFGs[7]). Even if it were possible to do so, it is felt that this schema is nevertheless clearer and more compact, as well as simpler to attach semantic actions to since a token is "reduced" in a single action, rather than over the course of several reductions [5].

[5]ie, IDENT: IDENT | 'A' | 'B' | ... 'Z' {...}; Unless the mechanics of a semantic action are changed somewhat, the actions will need to accumulate the IDENT string a character at a time. IDENT: "[A-Z]+" {...}; however, allows the action to deal with the full string at once.

Chapter 4

Algorithms

The following chapter describes the major algorithms developed, and/or adapted in the course of implementing the system as previously described. This is broadly separated out into conflict identification - determining where the lexical conflicts (if any) are in the language, conflict resolution - determining the "correct" actions for the lexical analyzer to take upon encountering these conflicts, and runtime - how the lexical analyzer is actually implemented in order to take advantage of this information.

4.1 Conflict Identification

Nawrocki[8] gives formal definitions for the two classes of lexical conflicts as follows:

$$\forall X, Y : \mathit{Terminals} \bullet L(X) \cap L(Y) \neq \phi \Rightarrow X \succ Y \tag{4.1}$$

$$\forall X, Y, Z : \mathit{Terminals} \bullet L(Y) \cdot \mathit{FirstChar}(Z) \cdot \Sigma^* \cap L(X) \neq \phi \Rightarrow X \succ Y \cdot Z \tag{4.2}$$

where L(X) denotes the language generated by X, Σ^* is any sequence of characters, and the symbol $\succ$ indicates "is in conflict with". It was suggested that these could be computed by testing all pairs/triplets of terminal symbols for the above conditions by eg "construction of the finite automaton for $L(X) \bigcap L(Y)$ and then testing if the automaton accepts a sentence of the length less than the number of states" (ibid).

While this should indeed give the sets of lexical conflicts in the language, it would seem possible do so in a significantly more efficient manner. In particular, by simply computing the combined DFA of all lexical items (as a lex-style program would anyway), much of the necessary information falls straight out.

A moderately efficient algorithm, therefore, is as follows. First, compute a full DFA of all lexical items. The set of identity conflicts is trivially obtained from each state which has multiple accepting items.

To locate longest match conflicts, retain the previous DFA, and for each accepting state:

- If it has no out-transitions, then the state is clearly free of LM conflicts.

- Else for each transition, if there also exists a transition on the same symbol from the start state, then there is a longest match conflict on that edge.

Following Nawrocki's analysis, if the transition symbol is not also valid from the start state, it does not represent a "real" conflict, as no token begins with that symbol. Therefore accepting now would be guaranteed to raise an error at the next attempt to read a token.

It should be noted that ignoring such "imaginary" conflicts is primarily an optimization - these would be eliminated at the next phase in any case, since if a symbol does not start any token, it certainly does not start any token which can follow the current one. However it may be desirable to produce a list of all identifiable lexical conflicts for a given language, which presumably should include only "real" conflicts. [1]

4.2 Conflict Resolution

The conflict resolution algorithms outlined here are based on the principle that the parser/scanner should process input as far as possible (using a single symbol of lookahead) before being forced to report an error, ie, if there exists two or more choices or actions, the one which does not result in an immediate error should be chosen. If more than one action is viable, then the language is not disambiguable by the techniques used, and the generator can either fall back on lex/yacc rules, or emit an error. If all available actions result in an error, the selection of action is based on the error recovery rules (above).

Conflict resolution is based on adjusting the behavior of the DFA according to the current parser (DPDA) state. In order to make use of this information, first the parser automata must be generated. For identity conflicts, at least LALR(1) is required[2], but longest match conflicts require LALR(2) (see below) to resolve all cases.

Identity conflicts are, again, the simpler case, and the most likely to be resolvable by these methods. First the parser DPDA is generated, with at least one token of lookahead (ie LALR(1)). Then, for each parser state and accepting DFA state, the acceptable tokens are the (possibly empty) intersection of the parser state's accept set and the DFA state's accept set.

To resolve longest match conflicts, some auxiliary information is required. For each DFA state, it is necessary to know all tokens which can be accepted by that state *or any successor state*. In effect, these sets represent all the tokens for which the strings reaching each state form a valid prefix.

The problem turns out to be quite similar to that of computing LALR(1) lookahead. The algorithm given below is a trivial modification of DeRemer & Pennello's Digraph algorithm to operate on states rather than transitions, which is itself "an adaptation of one given by Eve and Kurki-Souonio"[3]. The idea is essentially to (logically) collapse strongly connected components (ie loops), and then perform a depth-first search over the resulting acyclic graph. D&P have more detail with respect to the application of this algorithm, as well as

[1] The desirability of this is actually made dubious, however, by the fact that for any non-trivial language the typical number of conflicts is extremely large

[2] More precisely, at least one token of lookahead is required - the same techniques could be applied with SLR(1) or NQLALR(1). However the resolving power using such parsers will be limited by the larger lookahead sets

showing that its time complexity is O($V + E$), for V = number of vertices and E = number of edges. The basic theories of its operation are given in [4].

```
procedure computePostAccepts
  initially ∀ x ∈ STATES • x.postAccepts = x.accepts ∧ x.N = 0
  var S : Stack
  foreach x ∈ STATES | x.N = 0 do traverse(x) od
  procedure traverse(x : STATES)
    S.push(x)
    con d := S.depth
    x.N := d
    foreach y ∈ STATES | (x, y) ∈ EDGES do
      if y.N = 0 then traverse(y) fi
      x.N := minimum(x.N, y.N)
      x.postAccepts = x.postAccepts ∪ y.postAccepts
    od
    if x.N = d then
      repeat
        S.top.N := ∞
        S.top.postAccepts = x.postAccepts
      until S.pop = x
    fi
  end traverse
end computePostAccepts
```

Armed with this information, for each identified conflicting transition three token sets are extracted - the set of tokens immediately acceptable (if the transition is not taken), the tokens which are eventually acceptable following this transition, and the tokens which are eventually acceptable following a transition on the same symbol from the start state. For the following discussion these three sets are labelled NOW, NEXT, and ALT (alternative) respectively. It should be clear that if the scanner accepts at a given state, than it must return a member of the NOW set, followed by a member of the ALT set. If the scanner continues to scan, it must return a member of the NEXT set. [3]

For simplicity it is desired that the NOW set contain a single element, although the algorithms here can readily be adapted to work with multiple accepting items. For this reason it is suggested that all identity conflicts be resolved before examining longest match conflicts.

To give an example, consider a language containing only two tokens, 'if' and "[a-z]+". Assuming for the sake of clarity that the identity conflict in state 3 has already been resolved in favour of the 'if', we would obtain this DFA shown in figure 4.2.

Notice that there are three distinct character equivalence classes - [a-eghj-z],[i],[f] - for the purposes of conflict resolution it is necessary to treat an edge

[3] The exception to this is that if the scanner later encounters an error, it may backtrack to this point as the last accepting state.

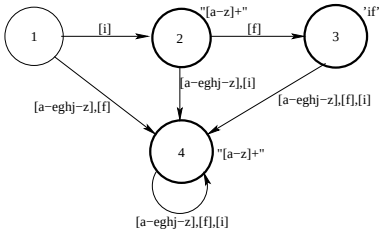

Figure 4.1: Combined DFA for 'if' and "[a-z]+"

which transitions on multiple symbols as separate edges (one per symbol) [4]. Using the definitions from the previous section, there are then 9 logical edges which are in LM conflict. Consider for the moment only the transition (3,[i]) (from state 3 on the character 'i'). The sets would then be NOW = {'if'} (the only acceptable token in state 3), NEXT = {"[a-z]+"} (the only token which the DFA could accept following a transition from state 3), and ALT = {"[a-z]+",'if'}

Then for each parser state, the set of tokens which may occur following whichever token(s) are acceptable in the current DFA state is needed. Given lalr(2) lookahead sets, this is the set

$$\{x, y : \mathit{Terminals} \mid \langle x, y\rangle \in \mathit{FOLLOWS}_2(\mathit{state}) \wedge x \in \mathit{NOW} \bullet y\} \tag{4.3}$$

That is, the second symbol in those lookahead strings which start with a token which is in the NOW set. The resultant set will be referred to just as FOLLOWS below.

From this information, it is then straightforward to produce a mapping, from (DFA edge, parser state) → SHIFT,ACCEPT,DONTCARE as follows:

For each (DFA edge, parser state),

- if NOW is disjoint from ACCEPTS, then SHIFT (accepting now is guaranteed to produce an immediate syntax error)
- else if NEXT is disjoint from ACCEPTS, then ACCEPT (any future acceptable symbol from here will produce an error)
- else if ALT is disjoint from FOLLOWS, then SHIFT (accepting results in the next token producing a guaranteed error)
- else DONTCARE (unresolvable conflict - either decision is locally viable)

The time complexity of this is clearly O(EL), where E is the number of DFA edges, and L is the number of LALR(1) states It is worth commenting on the use of LALR(2) here, particularly as Nawrocki attempted to avoid it by taking LALR(1) and following the reductions to determine the FOLLOWS sets. In fact, however, under certain conditions this technique results in lookahead sets being larger than those obtained from LALR(2), as pointed out in [2][5].

[4]Technically, the requirement is that for any edge leading from the current state, the set of transition symbols on that edge is a subset of the symbols on some edge leading from the start state. Practically speaking, the simplest way to ensure this is to make use of character equivalence classes.

[5]This was in relation to the use of reduce arcs as a way to produce arbitrary lookahead, but the problems are isomorphic

This was also noted in Nawrocki, and the solution given was to add a second pass over the lookahead sets to verify the configurations.

In any case, the net result is to compute LALR(2) lookahead, for which reason it is felt simpler both conceptually and in practice to simply perform such computations at the outset. The disadvantage of doing so, however, is that current LALR(k) algorithms appear to be somewhat less mature, and significantly slower than those for LALR(1). The algorithm used in the test system is due to Kristensen and Madsen[6] and the LALR(2) computations are unfortunately one of the slower parts of the system (although the implementation of that algorithm is not ideal either)

4.3 Runtime

There are two major approaches that are readily apparent when considering runtime implementations for lexical conflict resolution. The first is to maintain an auxiliary table which maps pairs of (parser state, DFA conflict point) to actions (SHIFT/ACCEPT). In this way, the DFA itself is as small as possible, although the conflict table may be quite large. However, the cost of an additional table lookup is incurred for every conflicting point in the DFA (the term "conflict point" is used to refer to either type of lexical conflict). In a DFA for a typical language, this results in at least one such lookup for almost every transition.

The second method is to produce a set of start states, similarly to how lex would operate, and select the start state based on the parser state at the beginning of each token. The result is that there is only one additional lookup per token, rather than per transition, but the DFA tables may be significantly larger. Lacking the conflict table, this scheme may in fact use less memory overall, but this is dependent on the exact language under consideration.

In this paper only the second will be examined, both as it represents the more runtime time-efficient option, and because it is somewhat more interesting in algorithmic terms.

A simple means of computing this is to copy the DFA wholly for each parser state, but making the necessary changes to each copy to reflect the conflict resolutions found in the previous step. (setting the accept symbol, deleting edges). Then, the well known DFA minimization algorithm[1] can be applied to all of the copies as if they were a single DFA, to produce a minimum sized DFA, along with the minimum number of distinct start states. It may or may not be immediately obvious that the minimization algorithm functions correctly on a disconnected graph of this nature - however it does indeed do so. At worst case, the time complexity of this step is O($(DL)^2$) where D is the number DFA states - the minimization itself is O(N^2), and there may be at most D x L states to minimize.

There is one problem with the foregoing, which is that for a language of moderate size, the number of DFA copies can become quite large, which results in the minimization step possibly taking a significantly long time. As a heuristic optimization, therefore, the number of initially copied states can be (often drastically) reduced by only copying those states which actually differ from the default DFA, or can precede (ie eventually transition to) a modified state - in all other cases the copy links backs to the default DFA. The worst case performance remains the same, but the best case is now O(D^2) (which represents a lexical

conflict free grammar).

Chapter 5

Syntactic and Semantic Issues

This chapter address various semantic and syntactic issues that arise in the conversion of independent parser/scanner modules to an integrated grammar. Inasmuch as possible the attempt is made to retain the most useful features, without greatly increasing source language complexity.

5.1 Access to lexical data

For all but the most trivial languages, there exist tokens for which the lexical string must be further processed by the language, for example identifiers, numbers, etc. In a standalone scanner this is normally done by means of actions attached to the tokens which perform any necessary processing before returning the correct token number.

In an integrated parser/scanner, with only grammar rules, this becomes more difficult to do, especially with efficiency. This is because at least one, and possibly an arbitrary number of tokens will be scanned before the rule which contains such a token is reduced and its action(s) executed. For example,

```
START: ‘‘[A-Za-z]+’’ ’=’ expr  { assign( $1, $3 ); } ;
expr : ‘‘[0-9]+’’ { $$ = strtol( $1 ); } ;
     | expr ’+’ expr { $$ = $1 + $3 };
```

By the time the call to assign is made, any number of tokens may have been scanned in. Simply changing it to something like

```
START: IDENT ’=’ expr { assign( $1, $3 ); } ;
IDENT: ‘‘[A-Za-z]+’’ { $$ = $1; }
```

removes the arbitrary aspect, but in most cases the ’=’ will still have been scanned before the IDENT action executes, due to the lookahead token [1].

[1] In this case, it most likely *won't* have been, as the reduce state for IDENT is LR(0), and so can be reduced without reading lookahead. Depending on this is not really advisable though.

There are a few possible solutions to this, the "simplest" of which is to copy the text of every lexical token into the parse stack when scanned. [2]. In languages such as C without garbage collection, this is perhaps not quite as simple as it might be, since the parser needs to ensure all copied strings are freed, but it is viable, if relatively inefficient. This is to say for example, it doesn't make much sense to copy a string, if it will only be converted to an integer anyway.

The next optimization of this is not to copy, but to in fact read the entire input into memory at once, and just pass start/length pairs down the parse stack. The rationale for this is that modern computers typically have far more memory available than the typical input file. Doing so also eliminates a lot of messy buffer management, and makes it possible to easily get the string for a complete production rather than just individual tokens. The disadvantage of this is that it doesn't work so well on stream input (continually resizing the input buffer - chained input buckets?), and of course it doesn't scale to ridiculously large file sizes

Another option, on systems which support it, is to use mmap [3] rather than explicity reading the input file. In this way, the file can more readily be paged in and out as needed by the system, and generally scales to the size of the process address space. However reports indicate that mmap is generally significantly less efficient than explicit reads for the types of predominantly sequential access done by the scanner.

The last option considered is that of specific lexical rules. These take a similar form to a normal grammar rule, but can only contain a single terminal on the right hand side:

```
IDENT = ``[A-Za-z]+'' { $$ = strdup( $1 ); }
```

This is then defined such that the action executes directly after scanning an IDENT, rather than when it would normally be reduced. The question perhaps arises as to why this is needed, as opposed to using a grammar rule as above, (and migrating the action to the scanner code) The reason is mostly due to the differing semantics:

- A grammar action is guaranteed not to be executed if it would result in an immediate error.[4] A lexical action however, will be executed whenever the token will be returned, even if the token is not actually acceptable by the parser (see error recovery for details on this).
- A lexical action has access to the lexical string, whereas a grammar action does not, unless one of the previous schemes is followed as well.
- Using a grammar production involves an (unnecessary) reduction at run-time, whereas the lexical rule does not.[5]
- Converting A : B into A = B automatically would tend to mask the differences, and most likely result in increased confusion in the user.

[2] An obvious optimization of this is to only copy those tokens which are actually referenced

[3] mmap(2) establishes a mapping between a file and a memory zone which can be demand paged by the operating system

[4] LALR(1) dilutes this slightly, in that it may reduce even though the lookahead symbol is erroneous

[5] Admittedly chain production elimination optimizations would probably eliminate these though

As a nice side-effect, it also allows IDENT to be treated as an alias of "[A-Za-z]+", which can make the parser debugging output slightly easier to read.

The current implementation of the generator as developed for this thesis uses the last option, as it seems to provide the most efficient, and clean, solution.

5.2 Whitespace

Another characteristic of virtually all non-trivial languages is the ability to use various forms of whitespace to separate tokens. That is, the whitespace does not in itself normally form a token. Comments are also typically treated in the same manner. It is generally desired that the grammar not contain explicit whitespace between practically every pair of tokens - this would be significantly harder to read.

How can an integrated parser/scanner handle whitespace? At the minimum it can be achieved by creating a "magic" token type, which defines what whitespace consists of, and simply skips it when recognized by the scanner. Due to time constraints, this is the only method currently implemented in the test system, using a lexical rule with the name WHITESPACE, eg

```
WHITESPACE = ``[ \t\n\r]'' ;
```

The previous covers the most common cases, but there are some additional possibilities. Firstly, it is possible to have comments which contain balanced text, and so cannot be recognized by an FSA alone. For example, there are some C compilers which are able to handle nested block comments - /* /* — */ */. [6] This should be able to be implemented as follows:

A grammar rule is defined to recognize whitespace. This rule is then implicitly added as an LR item (in initial position) to each parser state as it is calculated, unless the only other item in the state is also the whitespace rule. Finally, the reduction of the whitespace rule must be special, in that it should reduce without pushing a nonterminal onto the stack. In event of any conflicts between the rules of the grammar and the implicit whitespace, the grammar rules should take precedence. In general this should not result in any sizeable increase in the size of the grammar.

5.3 Semantically determined tokens

A hack that commonly occurs in processing some languages (notably C) is to use semantic information to determine the exact type of a token, which is needed to disambiguate an otherwise ambiguous language structure.

It would seem clear that the system thus far described is unable to handle this type of usage, but it may be possible to modify it to do so. One approach, as taken by the Anagram parser, is to permit a grammar rule to have more than one reduction, ie the actual reduction is chosen at runtime. A grammar using such a system may look something like this:

```
START: typeident varident ;
typeident, varident : ``[A-Za-z]+''
```

[6] The standards-compliance of such compilers is dubious, of course

```
{ $$ = lookup($1);
  if( istype($$) ) REDUCE(typeident);
  else REDUCE(varident);
} ;
```

Implementation of this is simply a matter of adding the rule with multiple reductions to each possible nonterminal, as if it was separately specified for each, and using the macro REDUCE to specify the correct one at runtime. This does however mean that a syntax error can occur (and must be able to be handled) following a sequence of reductions - it is no longer necessarily guaranteed that an error will be detected at the first erroneous token. This mechanism has not been implemented

5.4 Reserved Words

The final topic of note here is that of reserved words. Most, although not all, popular programming languages have a set of keywords that are not considered valid identifiers. This becomes an issue primarily because the standard way of representing an identifier is "[A-Za-z][A-Za-z0-9]*" or similar. In other words, every keyword is also a valid identifier. Simply using the techniques presented in this paper will actually allow keywords to be recognized as identifiers, under the condition that the keywords would not otherwise be valid continuations of the input.

Ideally, if reserved words are desired, one would use a representation of identifier which explicitly does not recognize the keywords of the language. This is complicated by the fact that this type of exclusion is cumbersome to represent in simple regular expressions. For example, with only the keyword 'if', an identifier would become "[A-Za-hj-z][A-Za-z0-9]*—i[A-Za-eg-z0-9]*—if[A-Za-eg-z0-9]+". A more useful mechanism would be to allow explicit exclusions, for example something like "[A-Za-z][A-Za-z0-9]!if".

A more general form of exclusion rule was proposed in [10], which allowed the statement that a given nonterminal does not produce another nonterminal. This level of generality was required for their scannerless parser, although it could potentially have uses in general parsing as well.

Chapter 6

Results

6.1 Implementation

As a major part of this project, a parser/scanner generator was implemented using the techniques described in this paper. This is currently in the vicinity of 6000 lines of C++ code, and produces C output (although it is readily extensible to other languages).

It does not currently include any automata optimizations, nor does it perform table compression, although these would presumably be required in a real system.

6.2 Resolution

Of primary interest, perhaps, is the effectiveness of these methods in resolving conflicts in actual grammars. Therefore the system was tested with two hopefully representative grammars. The first was a configuration file from a small system daemon called "keyd", which has a number of constructs of interest, such as lack of reserved words, and string matches to the end of line ("[^ n]*"). The second was ISO standard Pascal, which while not exercising any "difficult" constructs, makes a useful comparison with "real" languages.

Obviously these types of numbers are highly dependent on the exact grammar in use, but they hopefully demonstrate that left context alone can be sufficient to resolve a significant proportion of conflicts.

6.2.1 keyd

The keyd grammar is included in A.1, and has 46 terminals, 22 nonterminals, and 69 grammar productions. The generator produces 97 parser states and 156 DFA states for the default (combined) DFA.

Of these, 152 of the DFA states had identity conflicts, of which 118 were able to be wholly resolved by left context (77.6%). Additionally 5236 transitions were in longest match conflict, and 2707 of these (51.7%) were resolvable[1]. The final

[1] Earlier during a seminar it was stated that only 37% of edges in this grammar were resolved. In fact an incorrect simplifying assumption had been made, leading to edges being treated as equivalent when in fact they were not.

constructed (multiple start state) DFA has 369 states in total, and 24 unique start states.

6.2.2 Pascal

The Pascal grammar was adapted from a yacc/lex version available in the comp.sources.unix archives [2] and has 67 terminals, 135 nonterminals, and 254 productions. The generated automata have 410 parser states and 187 default DFA states.

Of these, 39 DFA states had identity conflicts, and only 11 (28%) were resolved by left context. 3663 transitions were in longest match conflict, with 1537 resolved (42%). The resulting DFA has 393 states total, and 61 unique start states. While these results are distinctly less impressive than the keyd ones, it is worth noting that all of the unresolved identity conflicts are of the form KEYWORD $\succ$ IDENTIFIER - and Pascal is designed to make use of reserved keywords. The current implementation does not have direct support for reserved words, but this is discussed further in 5.4.

6.3 Performance

In some senses it is almost meaningless to try to give quantifiable measures of performance, simply because there are so many variables involved. Runtime parser performance in particular is difficult to compare, due to the different levels of optimization performed by different generators - it is not a simple comparison of the basic techniques involved. Nevertheless, with these caveats, an attempt will be made to give some numbers. Note that these were produced with compiler optimizations off - in all cases compiling with -O2 resulted in roughly half the execution time.

6.3.1 keyd

On a PC, Pentium III 500MHz, the generator takes on average 2.4 seconds to produce code for the above grammar. Of this, almost half this time (1.2 sec) is devoted to minimizing the final DFA, with 0.43 sec for basic automata construction and 0.31 sec in constructing the LALR(2) lookahead. Actual conflict detection and resolution took only 0.3 seconds.

No runtime comparisons have been performed with this grammar, due to the difficulty involved in producing a meaningful, sufficiently large test case. However, possibly a more useful example is that used in the Pascal grammar, for which runtime tests have been done, below.

6.3.2 Pascal

The grammar for ISO Pascal provides perhaps a more useful comparison point, being fairly well known and widely implemented. The generation time was 15 seconds on average, with 5.5 sec in DFA minimization, 5.6 sec in LALR(2) lookahead, 1.6 sec in automata construction, and 2 sec in conflict analysis. While not quite infeasible at this point, this result is rather slower than might

[2] http://sources.isc.org/dirlist.perl?dir=devel/lang/&tarball=iso_pascal/iso_pascal/

be hoped, and is likely to be less than ideal for interactive development of large grammars.

For a 4000 line, 117Kb input file[3], on the same hardware, parsing was completed in 0.076 seconds. The equivalent parser built using bison/flex executes the same file in 0.048 seconds. Considering the current lack of optimization in the generated output, this is quite promising. However, perhaps even more interesting, is that with compiler optimizations on, the combined parser/scanner executes in 0.047 sec, but the bison/flex version only drops to 0.040 sec. With the same level of automata optimization as the existing tools, it seems quite likely that the performance of the test system could easily exceed theirs.

6.4 Conclusions

Automatic conflict resolution using left context is both able to resolve a large proportion of lexical conflicts in typical grammars, and can do so in a feasible and practical manner, at little to no additional runtime cost. As such, this represents a useful way to automatically and correctly handle a larger class of grammars than a yacc/lex parser, without losing generality wrt semantic actions.

6.5 Further Work

6.5.1 Additional rules

Exclusion and Adjacency restriction rules have been mainly discussed in the context of scannerless parsers[10], but should be able to be applied to conflict resolving scanners as well. It was hoped to be able to implement these in the system developed here, but this proved not to be feasible due to time constraints.

6.5.2 Right context

This paper has looked exclusively at the use of left context, in the form of parser state, to resolve lexical conflicts. While this has shown to be typically be effective for many grammars, there is nevertheless a substantial proportion of conflicts which cannot be resolved in this way. While some conflicts may be inherently ambiguous, there are also a number for which the single character of lookahead used by the scanner is insufficient to properly disambiguate the grammar. In lex-derived generators it is possible for a language author to add right context to the specification, and it seems likely that this right context could also be generated in an automatic fashion.

It is likely that the generation of such context would need to be optional, since a performance penalty does apply, but it has the potential to increase the range of grammars for which unambiguous scanners can be produced.

6.5.3 Performance

The experiments with Pascal have shown up some performance issues, in that it does not currently scale well to large grammars. The primary limiting factors

[3] A Pascal compiler, http://www.cwi.nl/ftp/pascal/pcom.p

appear to be the LALR(2) generation, and the DFA minimization - the remainder of the time taken by the generator is almost negligible by comparison. One possible partial solution is to skip the minimization step - this is primarily a runtime optimization after all, and use the unoptimized version for development purposes. A better solution would be to improve the efficiency of the minimization code, but it is not certain how great an improvement is likely to result from such efforts.

The LALR(2) lookahead, on the other hand, uses an LALR(k) algorithm[6] which predates DeRemer & Pennello's now-standard LALR(1) technique[3], and it seems possible that a somewhat more efficient LALR(k) algorithm could be developed from the more recent work on LALR(1).

Appendix A

Grammars

A.1 Keyd grammar

Following is the grammar for the keyd configuration files, referenced in 6.3.

```
block: statements
|
;

statements: statement
| statements statement
;

statement: modeblock
| keyline
| 'entry' '=' command
        | 'exit' '=' command
| 'start' '=' command
| error '\n'
;

modeblock: 'mode' IDENT '{' block '}'
;


keyline: modifiers key command
| key command
;

(keyent_t) key: 'keycode' NUMBER '='
;
(unsigned int) modifiers: modifier
| modifiers modifier
;

(unsigned int)
```

```
modifier: 'shift'
| 'alt'
| "control|ctrl"
| "lshift|shiftl"
| "rshift|shiftr"
| "lalt|altl"
| "ralt|altr"
| "lctrl|ctrll"
| "rctrl|ctrlr"
| 'up'
| 'down'
;

(action_t) command: modecmd | execcmd | suidcmd
        | mixercmd | cdcmd | consolecmd
|
;

(action_t) modecmd: 'mode' IDENT
;
(action_t) execcmd: 'exec' "[^\n]+" '\n'
;
(action_t) suidcmd: 'suid' NUMBER NUMBER 'exec' "[^\n]+" '\n'

;

(action_t) mixercmd: 'mixer' mixrec mixnam mixset
;
(int) mixrec:   '+'
| '-'
|
;
(char *) mixnam: IDENT
|
;
(struct { int op, l, r; }) mixset: NUMBER NUMBER
| SIGNNUMBER SIGNNUMBER
| NUMBER
| SIGNNUMBER
| 'mute'
|
;

(action_t) cdcmd: 'cd' device cdop
;

(char *) device: IDENT
|
;
```

```
(int) cdop: 'stop'
| 'pause'
| 'next'
| 'prev'
| 'eject'
| 'stopeject'
| 'lock'
| 'unlock'
;


(action_t) consolecmd: 'console' consoleop
;
(action_t) consoleop: 'new'
| 'next'
| 'prev'
| 'mark' NUMBER
| 'mark' NUMBER NUMBER
| 'restore' NUMBER
| NUMBER
;

(char *) IDENT = "[A-Za-z][A-Za-z0-9]*" ;
(int) NUMBER = "[0-9]+|0x[0-9a-fA-F]+" ;
(int) SIGNNUMBER = "[+-]([0-9]+|0x[0-9a-fA-F]+)" ;
WHITESPACE = "[ \t\r\n]";
```

A.2 Pascal

```
file : program
| module
;

program : program_heading semicolon block DOT
;

program_heading : PROGRAM identifier
| PROGRAM identifier LPAREN identifier_list RPAREN
;

identifier_list : identifier_list comma identifier
| identifier
;

block : label_declaration_part
constant_definition_part
type_definition_part
variable_declaration_part
procedure_and_function_declaration_part
```

```
statement_part
;

module : constant_definition_part
type_definition_part
variable_declaration_part
procedure_and_function_declaration_part
;

label_declaration_part : LABEL label_list semicolon
|
;

label_list : label_list comma label
| label
;

label : DIGSEQ
;

constant_definition_part : CONST constant_list
|
;

constant_list : constant_list constant_definition
| constant_definition
;

constant_definition : identifier EQUAL cexpression semicolon
;

cexpression : csimple_expression
| csimple_expression relop csimple_expression
;

csimple_expression : cterm
| csimple_expression addop cterm
;

cterm : cfactor
| cterm mulop cfactor
;

cfactor : sign cfactor
| cexponentiation
;

cexponentiation : cprimary
| cprimary STARSTAR cexponentiation
;
```

```
cprimary : identifier
| LPAREN cexpression RPAREN
| unsigned_constant
| NOT cprimary
;

constant : non_string
| sign non_string
| CHARACTER_STRING
;

sign : PLUS
| MINUS
;

non_string : DIGSEQ
| identifier
| REALNUMBER
;

type_definition_part : TYPE type_definition_list
|
;

type_definition_list : type_definition_list type_definition
| type_definition
;

type_definition : identifier EQUAL type_denoter semicolon
;

type_denoter : identifier
| new_type
;

new_type : new_ordinal_type
| new_structured_type
| new_pointer_type
;

new_ordinal_type : enumerated_type
| subrange_type
;

enumerated_type : LPAREN identifier_list RPAREN
;

subrange_type : constant DOTDOT constant
;
```

```
new_structured_type : structured_type
| PACKED structured_type
;

structured_type : array_type
| record_type
| set_type
| file_type
;

array_type : ARRAY LBRAC index_list RBRAC OF component_type
;

index_list : index_list comma index_type
| index_type
;

index_type : ordinal_type ;

ordinal_type : new_ordinal_type
| identifier
;

component_type : type_denoter ;

record_type : RECORD record_section_list END
| RECORD record_section_list semicolon variant_part END
| RECORD variant_part END
;

record_section_list : record_section_list semicolon record_section
| record_section
;

record_section : identifier_list COLON type_denoter
;

variant_part : CASE variant_selector OF variant_list semicolon
| CASE variant_selector OF variant_list
|
;

variant_selector : tag_field COLON tag_type
| tag_type
;

variant_list : variant_list semicolon variant
| variant
;
```

```
variant : case_constant_list COLON LPAREN record_section_list RPAREN
| case_constant_list COLON LPAREN record_section_list semicolon
variant_part RPAREN
| case_constant_list COLON LPAREN variant_part RPAREN
;

case_constant_list : case_constant_list comma case_constant
| case_constant
;

case_constant : constant
| constant DOTDOT constant
;

tag_field : identifier ;

tag_type : identifier ;

set_type : SET OF base_type
;

base_type : ordinal_type ;

file_type : PFILE OF component_type
;

new_pointer_type : UPARROW domain_type
;

domain_type : identifier ;

variable_declaration_part : VAR variable_declaration_list semicolon
|
;

variable_declaration_list :
  variable_declaration_list semicolon variable_declaration
| variable_declaration
;

variable_declaration : identifier_list COLON type_denoter
;

procedure_and_function_declaration_part :
proc_or_func_declaration_list semicolon
|
;

proc_or_func_declaration_list :
```

```
  proc_or_func_declaration_list semicolon proc_or_func_declaration
| proc_or_func_declaration
;

proc_or_func_declaration : procedure_declaration
| function_declaration
;

procedure_declaration : procedure_heading semicolon directive
| procedure_heading semicolon procedure_block
;

procedure_heading : procedure_identification
| procedure_identification formal_parameter_list
;

directive : FORWARD
| EXTERNAL
;

formal_parameter_list : LPAREN formal_parameter_section_list RPAREN ;

formal_parameter_section_list :
        formal_parameter_section_list semicolon formal_parameter_section
| formal_parameter_section
;

formal_parameter_section : value_parameter_specification
| variable_parameter_specification
| procedural_parameter_specification
| functional_parameter_specification
;

value_parameter_specification : identifier_list COLON identifier
;

variable_parameter_specification : VAR identifier_list COLON identifier
;

procedural_parameter_specification : procedure_heading ;

functional_parameter_specification : function_heading ;

procedure_identification : PROCEDURE identifier ;

procedure_block : block ;

function_declaration : function_heading semicolon directive
| function_identification semicolon function_block
| function_heading semicolon function_block
```

```
;

function_heading : FUNCTION identifier COLON result_type
| FUNCTION identifier formal_parameter_list COLON result_type
;

result_type : identifier ;

function_identification : FUNCTION identifier ;

function_block : block ;

statement_part : compound_statement ;

compound_statement : PBEGIN statement_sequence END ;

statement_sequence : statement_sequence semicolon statement
| statement
;

statement : open_statement
| closed_statement
;

open_statement : label COLON non_labeled_open_statement
| non_labeled_open_statement
;

closed_statement : label COLON non_labeled_closed_statement
| non_labeled_closed_statement
;

non_labeled_closed_statement : assignment_statement
| procedure_statement
| goto_statement
| compound_statement
| case_statement
| repeat_statement
| closed_with_statement
| closed_if_statement
| closed_while_statement
| closed_for_statement
|
;

non_labeled_open_statement : open_with_statement
| open_if_statement
| open_while_statement
| open_for_statement
;
```

```
repeat_statement : REPEAT statement_sequence UNTIL boolean_expression
;

open_while_statement : WHILE boolean_expression DO open_statement
;

closed_while_statement : WHILE boolean_expression DO closed_statement
;

open_for_statement : FOR control_variable ASSIGNMENT initial_value direction
final_value DO open_statement
;

closed_for_statement : FOR control_variable ASSIGNMENT initial_value direction
final_value DO closed_statement
;

open_with_statement : WITH record_variable_list DO open_statement
;

closed_with_statement : WITH record_variable_list DO closed_statement
;

open_if_statement : IF boolean_expression THEN statement
| IF boolean_expression THEN closed_statement ELSE open_statement
;

closed_if_statement : IF boolean_expression THEN closed_statement
ELSE closed_statement
;

assignment_statement : variable_access ASSIGNMENT expression
;

variable_access : identifier
| indexed_variable
| field_designator
| variable_access UPARROW
;

indexed_variable : variable_access LBRAC index_expression_list RBRAC
;

index_expression_list : index_expression_list comma index_expression
| index_expression
;

index_expression : expression ;
```

```
field_designator : variable_access DOT identifier
;

procedure_statement : identifier params
| identifier
;

params : LPAREN actual_parameter_list RPAREN ;

actual_parameter_list : actual_parameter_list comma actual_parameter
| actual_parameter
;

actual_parameter : expression
| expression COLON expression
| expression COLON expression COLON expression
;

goto_statement : GOTO label
;

case_statement : CASE case_index OF case_list_element_list END
| CASE case_index OF case_list_element_list SEMICOLON END
| CASE case_index OF case_list_element_list semicolon
otherwisepart statement END
| CASE case_index OF case_list_element_list semicolon
otherwisepart statement SEMICOLON END
;

case_index : expression ;

case_list_element_list : case_list_element_list semicolon case_list_element
| case_list_element
;

case_list_element : case_constant_list COLON statement
;

otherwisepart : OTHERWISE
| OTHERWISE COLON
;

control_variable : identifier ;

initial_value : expression ;

direction : TO
| DOWNTO
;
```

```
final_value : expression ;

record_variable_list : record_variable_list comma variable_access
| variable_access
;

boolean_expression : expression ;

expression : simple_expression
| simple_expression relop simple_expression
;

simple_expression : term
| simple_expression addop term
;

term : factor
| term mulop factor
;

factor : sign factor
| exponentiation
;

exponentiation : primary
| primary STARSTAR exponentiation
;

primary : variable_access
| unsigned_constant
| function_designator
| set_constructor
| LPAREN expression RPAREN
| NOT primary
;

unsigned_constant : unsigned_number
| CHARACTER_STRING
| NIL
;

unsigned_number : unsigned_integer | unsigned_real ;

unsigned_integer : DIGSEQ
;

unsigned_real : REALNUMBER
;

function_designator : identifier params
```

```
;

set_constructor : LBRAC member_designator_list RBRAC
| LBRAC RBRAC
;

member_designator_list : member_designator_list comma member_designator
| member_designator
;

member_designator : member_designator DOTDOT expression
| expression
;

addop: PLUS
| MINUS
| OR
;

mulop : STAR
| SLASH
| DIV
| MOD
| AND
;

relop : EQUAL
| NOTEQUAL
| LT
| GT
| LE
| GE
| IN
;

identifier : IDENTIFIER
;

semicolon : SEMICOLON
;

comma : COMMA
;


AND = "[Aa][Nn][Dd]" ;
ARRAY = "[Aa][Rr][Rr][Aa][Yy]" ;
CASE = "[Cc][Aa][Ss][Ee]" ;
CONST = "[Cc][Oo][Nn][Ss][Tt]" ;
DIV = "[Dd][Ii][Vv]" ;
```

```
DO = "[Dd][Oo]" ;
DOWNTO = "[Dd][Oo][Ww][Nn][Tt][Oo]" ;
ELSE = "[Ee][Ll][Ss][Ee]" ;
END = "[Ee][Nn][Dd]" ;
EXTERNAL = "[Ee][Xx][Tt][Ee][Rr][Nn]([Aa][Ll])?" ;
FOR = "[Ff][Oo][Rr]" ;
FORWARD = "[Ff][Oo][Rr][Ww][Aa][Rr][Dd]" ;
FUNCTION = "[Ff][Uu][Nn][Cc][Tt][Ii][Oo][Nn]" ;
GOTO = "[Gg][Oo][Tt][Oo]" ;
IF = "[Ii][Ff]" ;
IN = "[Ii][Nn]" ;
LABEL = "[Ll][Aa][Bb][Ee][Ll]" ;
MOD = "[Mm][Oo][Dd]" ;
NIL = "[Nn][Ii][Ll]" ;
NOT = "[Nn][Oo][Tt]" ;
OF = "[Oo][Ff]" ;
OR = "[Oo][Rr]" ;
OTHERWISE = "[Oo][Tt][Hh][Ee][Rr][Ww][Ii][Ss][Ee]" ;
PACKED = "[Pp][Aa][Cc][Kk][Ee][Dd]" ;
PBEGIN = "[Bb][Ee][Gg][Ii][Nn]" ;
PFILE = "[Ff][Ii][Ll][Ee]" ;
PROCEDURE = "[Pp][Rr][Oo][Cc][Ee][Dd][Uu][Rr][Ee]" ;
PROGRAM = "[Pp][Rr][Oo][Gg][Rr][Aa][Mm]" ;
RECORD = "[Rr][Ee][Cc][Oo][Rr][Dd]" ;
REPEAT = "[Rr][Ee][Pp][Ee][Aa][Tt]" ;
SET = "[Ss][Ee][Tt]" ;
THEN = "[Tt][Hh][Ee][Nn]" ;
TO = "[Tt][Oo]" ;
TYPE = "[Tt][Yy][Pp][Ee]" ;
UNTIL = "[Uu][Nn][Tt][Ii][Ll]" ;
VAR = "[Vv][Aa][Rr]" ;
WHILE = "[Ww][Hh][Ii][Ll][Ee]" ;
WITH = "[Ww][Ii][Tt][Hh]" ;
ASSIGNMENT = ':=' ;
CHARACTER_STRING = "'([^']|'')+'" ;
COLON = ':' ;
COMMA = ',' ;
DIGSEQ = "[0-9]+" ;
DOT = '.' ;
DOTDOT = '..' ;
EQUAL = '=' ;
GE = '>=' ;
GT = '>' ;
LBRAC = '[' ;
LE = '<=' ;
LPAREN = '(' ;
LT = '<' ;
MINUS = '-' ;
NOTEQUAL = '<>' ;
PLUS = '+' ;
```

```
RBRAC = ']' ;
REALNUMBER = "[0-9]+\.[0-9]+" ;
RPAREN = ')' ;
SEMICOLON = ';' ;
SLASH = '/' ;
STAR = '*' ;
STARSTAR = '**' ;
UPARROW = "->|^" ;

IDENTIFIER = "[a-zA-Z]([a-zA-Z0-9])*" ;
WHITESPACE = "[ \t\n]+|{[^}]*}|\(\*([^*]|\*+[^*)])*\*+\)" ;
```

Bibliography

[1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, Reading, Massachusettes, 1985.

[2] Theodore P. Baker. Extending lookahead for lr parsers. *Journal of Computer and System Sciences*, 22(2):243–259, 1981.

[3] Frank DeRemer and Thomas Pennello. Efficient computation of lalr(1) look-ahead sets. *ACM Transactions on Programming Languages and Systems*, 4(4):615–549, October 1982.

[4] J. Eve and R. Kurki-Suonio. On computing the transitive closure of a relation. *Acta Informatica*, 8:303–314, 1977.

[5] J. Grosch. Lalr - a generator for efficient parsers. *Software: Practice and Experience*, 20(11):1115–1135, November 1990.

[6] Bent B. Kristensen and Ole L. Madsen. Methods for computing lalr(k) lookahead. *ACM Transactions on Programming Languages and Systems*, 3(1):60–82, January 1981.

[7] Roman Krzemień and Andrzej Lukasiewicz. Automatic generation of lexical analyzers in a compiler-compiler. *Information Processing Letters*, 4(6):165–168, March 1976.

[8] Jerzy R. Nawrocki. Conflict detection and resolution in a lexical analyzer generator. *Information Processing Letters*, 38(6):323–329, June 1991.

[9] Jan Rekers. Parser generation for interactive environments. Technical report, University of Amsterdam, 1992.

[10] Daniel J. Salomon and Gordon V. Cormack. Scannerless nslr(1) parsing of programming languages. *SIGPLAN Notices*, 24(7):170–178, July 1989.

[11] David Spector. Efficient full lr(1) parser generation. *SIGPLAN Notices*, 23(12):143–150, December 1988.

[12] Kuo-Chung Tai. Noncanonical slr(1) grammars. *ACM Transactions on Programming Languages and Systems*, 1(2):295–320, October 1979.

[13] Masaru Tomita. *Efficient Parsing for Natural Language*. Kluwer Academic Publishers, Boston, 1985.

[14] Eelco Visser. Scannerless generalized-lr parsing. Technical Report P9707, University of Amsterdam Programming Research Group, August 1997.