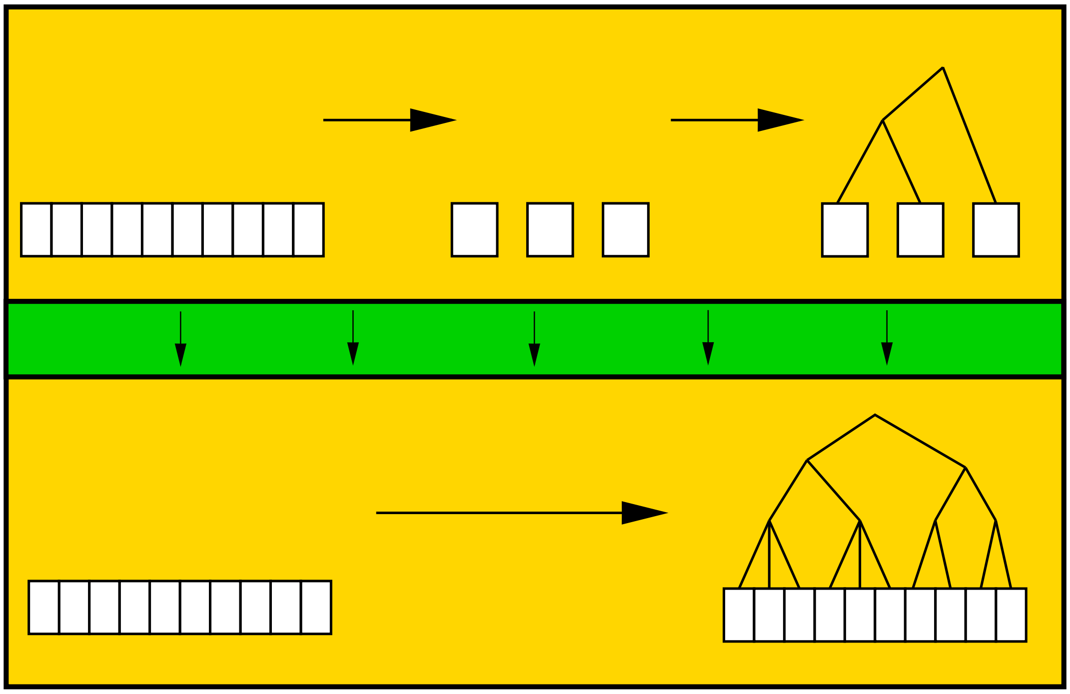
University of Amsterdam

Programming Research Group



Scannerless Generalized-LR Parsing

Eelco Visser

Report P9707 August 1997

University of Amsterdam

Department of Computer Science

Programming Research Group

Scannerless generalized-lr parsing

Eelco Visser

Report P9707 August 1997

E. Visser

Programming Research Group
Department of Computer Science
University of Amsterdam

Kruislaan 403
NL-1098 SJ Amsterdam
The Netherlands

tel. +31 20 525 7590
e-mail: visser@wins.uva.nl

Acknowledgments The author thanks Arie van Deursen, Jan van Eijck, Annius Groenink and Paul Klint, for useful suggestions and comments on previous versions of this paper.

This research was supported by the Netherlands Computer Science Research Foundation (SION) with financial support from the Netherlands Organisation for Scientific Research (NWO). Project 612-317-420: Incremental parser generation and context-sensitive disambiguation: a multi-disciplinary perspective.

Universiteit van Amsterdam, 1997

Contents

Scannerless Generalized-LR Parsing

Eelco Visser

Current deterministic parsing techniques have a number of problems. These include the limitations of parser generators for deterministic languages and the complex interface between scanner and parser. Scannerless parsing is a parsing technique in which lexical and context-free syntax are integrated into one grammar and are all handled by a single context-free analysis phase. This approach has a number of advantages including discarding of the scanner and lexical disambiguation by means of the context in which a lexical token occurs. Scannerless parsing generates a number of interesting problems as well. Integrated grammars do not fit the requirements of the conventional deterministic parsing techniques. A plain context-free grammar formalism leads to unwieldy grammars, if all lexical information is included. Lexical disambiguation needs to be reformulated for use in context-free parsing.

The scannerless generalized-LR parsing approach presented in this paper solves these problems. Grammar normalization is used to support an expressive grammar formalism without complicating the underlying machinery. Follow restrictions are used to express longest match lexical disambiguation. Reject productions are used to express the prefer keywords rule for lexical disambiguation. The SLR(1) parser generation algorithm is adapted to implement disambiguation by general priority and associativity declarations and to interpret follow restrictions. Generalized-LR parsing is used to provide dynamic lookahead and to support parsing of arbitrary context-free grammars including ambiguous ones. An adaptation of the GLR algorithm supports the interpretation of grammars with reject productions.

1 Introduction

Parsing is one of the areas of computer science where program generation is a routine technique that is successfully applied to generate parsers for programming languages given their formal definition by means of a context-free grammar. At least, in theory. In practice, most parser generators accept only a limited subset of the context-free grammars such as LL(1) or LALR(1) grammars. Since most natural grammars for languages do not respect these limitations, the language designer or compiler writer has to bend over backwards to fit the grammar into the restrictions posed by the grammar formalism by rewriting grammar rules, introducing ad-hoc solutions for parse table conflicts or resorting to side effects in the parser. Even if one succeeds in producing a grammar that respects the restrictions, a small extension or modification of the language might jeopardize the careful balance of tricks, which makes maintenance of tools for the language troublesome.

Another source of problems in generated parsers is the division between the lexical analysis phase and the context-free analysis phase and the corresponding division of the grammar into a regular grammar defining the lexical syntax and a context-free grammar defining the context-free syntax. A scanner divides the character string into tokens according to the lexical syntax. A parser structures the token string into a tree according to the context-free syntax.

At the interface between scanner and parser the lexical tokens are passed from the scanner to the parser. In the most straightforward scenario the scanner produces a stream of tokens without intervention from the parser. This entails that no knowledge of the parsing context is available in the scanner and thus no lexical analysis decisions can rely on such information. It is difficult to unambiguously define the lexical syntax of a language by means of only regular grammars. Therefore, lexical analysis and the interface with context-free analysis are usually extended. First lexical disambiguation heuristics such as 'prefer longest match' and 'prefer keyword' are applied to reduce the number of readings. If there remain ambiguities after application of these rules the scanner might produce multiple streams of tokens representing all possible partitionings of the string into tokens according to the regular grammar. The parser should then be able to cope with this non-linear input. It is also possible to supply feedback from the parser to the scanner to reduce the number of applicable grammar rules. For instance, by specifying the lexical categories that are expected for the next token.

In all such schemes lexical analysis becomes more complicated than the simple finite automaton model that motivated the use of regular grammars. Context-free parsing functionality starts to appear both inside the scanner and at the interface between scanner and parser and often operational elements corrupt the declarativity of the language definition. As a consequence, many grammars are ambiguous if only the pure regular and context-free grammar are considered as such and reasoning about the language being defined becomes difficult.

1.1 Scannerless Generalized-LR Parsing

In this paper we describe an approach to syntax definition and parser generation that overcomes many of these problems. The approach is based on the integration and improvement of scannerless parsing, generalized-LR parsing and grammar normalization. Because of the integration of the former two, the approach is called *scannerless generalized-LR parsing*.

Scannerless Parsing Scannerless parsing is a parsing technique that does not use a scanner to divide a string into lexical tokens. Instead lexical analysis is integrated in the context-free analysis of the entire string. It comes up naturally when considering grammars that completely describe the syntax of a language. The term scannerless parsing was coined by Salomon and Cormack (1989, 1995). They use 'complete character level grammars' describing the entire syntax of a language down to the character level. Since conventional LR parser generation yields tables with too many conflicts, they use an extension of SLR(1) parser generation called non-canonical SLR(1). However, even this extension makes it hard to define a grammar without conflicts.

Generalized-LR Parsing The conventional LR parsing techniques and especially scannerless LR parsing suffer from conflicts in the parse table. There are two causes for conflicts in LR parse tables: ambiguities and lack of lookahead. If a conflict is caused by an ambiguity, any of the possible actions will lead to a successful parse. If it was caused by a lookahead problem, one of the actions will lead to success and the others will fail. Which action will be successful cannot be decided statically. Since ambiguity of a context-free grammar is undecidable (Floyd, 1962), it is also undecidable whether a conflict is due to an ambiguity or to a lack of lookahead. Because complete character level grammars frequently need arbitrary length lookahead, methods to solve conflicts in the table will not always succeed.

Generalized-LR parsing is an extension of LR parsing that interprets the conflicts in the parse table by forking off a parser from the main parser for each possible action in case of a conflict. If such a conflict turns out to lead to an ambiguity the parser constructs a parse forest, a compact representation of all possible parse trees for a sentence. But if the conflict was caused by lack of lookahead, the forked parsers for the wrong track will fail. In this manner lookahead is handled dynamically. Therefore, generalized-LR parsing is an ideal technique to solve the lookahead problems of scannerless parsing. Generalized-LR parsing was introduced by Tomita (1985) building on the theoretical framework of Lang (1974). It was improved by Rekers (1992) to handle all context-free grammars. In this paper we extend Rekers' version of the algorithm with reject reductions, a facility needed for lexical disambiguation.

Grammar Normalization An aspect of the division between lexical and context-free syntax that affects the specification of syntax is the definition of layout, i.e., the whitespace and comments that can occur at arbitrary places between tokens. In the conventional setting layout is analyzed by the scanner and then thrown away. The parser never sees the layout tokens. Therefore, layout can also be ignored in the specification of context-free syntax. However, in a complete character level grammar all aspects of the syntax are completely defined, including the syntax and positions of layout. This can lead to rather unwieldy grammars that declare the occurrence of layout as separator between all grammar symbols in context-free productions.

Grammar normalization is a technique used to define an expressive grammar formalism in terms of simple context-free grammars. An example of a normalization procedure is the addition of layout symbols between the symbols in context-free productions. Other examples are the definition of regular expressions by means of productions and the flattening of modular grammars. An important aspect of the scannerless generalized-LR approach is the use of grammar normalization to keep grammars small and usable. The syntax definition formalism SDF2 used in the approach is a formalism for concise definition of complete character level grammars. SDF2 is a generalization of the syntax definition formalism SDF of Heering *et al.* (1989). The formalism and normalization procedure is defined in Visser (1997c).

1.2 Architecture

The typical architecture of an application of SDF2 is depicted in Figure 1. A program text[1] processor that transforms text into text is composed of (1) a parser front-end that analyzes the input text and produces a structured representation of the text in the form of a parse tree, (2) the actual processor that performs a transformation from a parse tree to another one and (3) a pretty-printer back-end that produces text corresponding to a transformed parse tree. Processors can be, for instance, interpreters, compilers, data flow analyzers or program transformation tools.

The input language of a processor is specified in the syntax definition formalism SDF2. Given a language definition in SDF2 and a tree to tree processor, the corresponding text to text processor is constructed using a grammar normalizer, a parser generator, a parser and a pretty-printer generator.

Grammar Normalizer A language definition in SDF2 is normalized to a plain context-free grammar extended with character classes, priority rules, follow restrictions and reject productions. Normalization is briefly discussed in §3. A full definition of SDF2 normalization can be found in Visser (1997c).

Parser Generator From a normalized syntax definition a parse table is generated using an extension of the standard SLR(1) algorithm with character classes, priorities, follow restrictions, and reject productions. The parser generator accepts arbitrary context-free grammars. The techniques used in the parser generator are discussed in §5.

Parser A parse table is interpreted by a generic, language independent SGLR parser, which reads a text and produces a parse tree. At the heart of the parser is an extension of the GLR algorithm of Rekers (1992) that reads characters directly, without using a scanner. The extension of the GLR algorithm with reject reductions is discussed in §8.

Pretty-Printer A pretty-printer is used to translate the output tree of the processor to text. The pretty printer itself can also be generated from the definition of the output language. This is described in Van den Brand and Visser (1996) and is not further discussed here.

1.3 Contributions

The scannerless generalized-LR parsing approach presented in this paper is a new powerful parsing method that supports concise specification of languages. The technical contributions (the details of which will be discussed later on) of the approach are:

- The normalization of grammars to eliminate features enhancing the expressivity of the formalism, in particular, the integration of lexical and context-free syntax by means of normalization into a single grammar.

- The use of GLR parsing for scannerless parsing to deal with unbounded lookahead.

[1] Here text denotes a linear representation of a program in some character code, e.g., ASCII or UniCode.

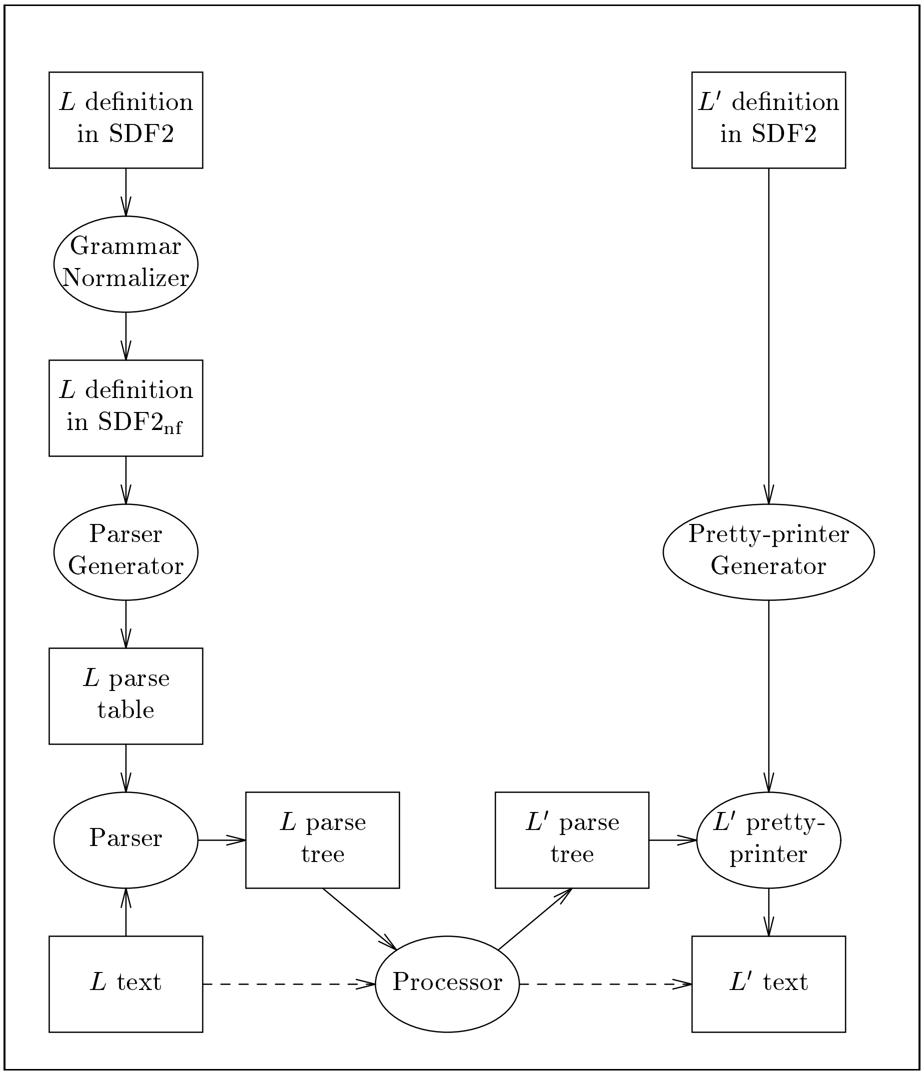


Figure 1: Architecture of an SDF2 application.

- Static disambiguation by means of priorities by interpreting priority declarations in the parser generator. Priorities are completely expressed in the parse table.
- The use of character classes in grammars to compact the parse table.
- The use of follow restrictions to define longest match disambiguation and the interpretation of follow restrictions in the parse table.
- Prefer literals disambiguation by means of reject productions. Several expressivity results about context-free grammars with reject productions. Implementation of parsers for such grammars in an extension of the GLR algorithm.

1.4 Overview

In the next section we will argue in more detail that scannerless parsing has a number of definite advantages over parsing with scanners, but that it has not been introduced before because of the limitations of conventional parsing techniques. In the rest of the paper we present several techniques that overcome these limitations and result in a combined approach encompassing grammar formalism and parsing techniques that does make scannerless parsing feasible.

2 Scannerless Parsing

The term *scannerless parsing* was coined by Salomon and Cormack (1989, 1995) to indicate parsing without a separate lexical analysis phase using a scanner based on a deterministic finite automaton. The parser directly reads the characters of a text. This entails the integration of the definition of lexical and context-free syntax in one grammar.

Consider the following SDF2 definition of a simple language of expressions consisting of identifiers, additions and multiplications.

```
sorts Id Exp
lexical syntax
  [a-z]+    -> Id
  [\ \t\n] -> LAYOUT
context-free syntax
  Id           -> Exp
  Exp "*" Exp -> Exp {left}
  Exp "+" Exp -> Exp {left}
context-free priorities
  Exp "*" Exp -> Exp >
  Exp "+" Exp -> Exp
```

The first line declares the sorts (say the non-terminals) of the grammar. The next three lines declare the lexical syntax of the language such that identifiers are lists of one or more lowercase letters and layout consists of spaces, tabs and newlines. The next four lines declare the context-free syntax of the language. An expression is either an identifier or an addition or multiplication of two expressions. Observe that the grammar is ambiguous and that in order to disambiguate it, priority and associativity declarations have been added. The last three lines declare that multiplication has higher priority than addition. The `left` attribute declares addition and multiplication to be left-associative.

The conventional way to interpret such a grammar to parse a string is as follows: (1) Divide the string into tokens according to the lexical syntax in all possible ways. (2) Apply lexical disambiguation rules to select the desired division. For instance, given the string $\texttt{ab}\textvisiblespace$, the rule 'prefer longest match' would prefer the division $\boxed{\texttt{ab}}\boxed{\textvisiblespace}$ over $\boxed{\texttt{a}}\boxed{\texttt{b}}\boxed{\textvisiblespace}$, i.e., the longest possible identifier is selected. (3) Throw away layout tokens. (4) Parse the resulting token string according to the context-free syntax. The result is a parse tree that contains as leafs the tokens yielded by lexical analysis.

In scannerless parsing we have the following sequence: (1) Combine the definition of lexical and context-free syntax into a single context-free grammar. All tokens on the left-hand side of productions in the context-free syntax are

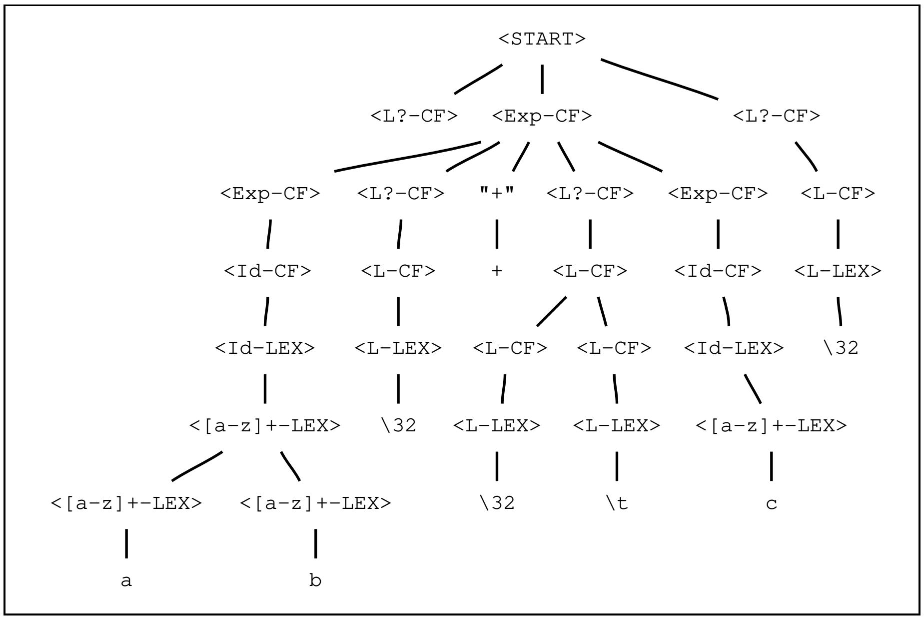


Figure 2: Parse tree for the string `ab␣+␣\tc␣`

explicitly separated by layout. All grammar symbols are renamed, such that the symbols occurring in the lexical syntax have the form ⟨`_-LEX`⟩ and those in the context-free syntax have the form ⟨`_-CF`⟩. This is done to keep the two levels separated. For instance, the addition production is transformed into

```
<Exp-CF> <LAYOUT?-CF> "+" <LAYOUT?-CF> <Exp-CF> -> <Exp-CF>
```

The symbol ⟨`LAYOUT?-CF`⟩ represents the syntax of layout that can appear between tokens. In §3 this will be explained in more detail. The complete integrated grammar corresponding to the definition above is presented in Figure 3 on page 14. (2) Parse the characters of the string according to the normalized grammar. The result is a parse tree that contains as leafs the characters of the string. The tokens are recognizable as subtrees. For example, consider the parse tree in Figure 2. Observe that the symbols ⟨`L-LEX`⟩ and ⟨`L-CF`⟩ are abbreviations for ⟨`LAYOUT-LEX`⟩ and ⟨`LAYOUT-CF`⟩ and denote layout nodes.

In a sense, nothing is new. In a conventional parser, if we would instruct the scanner to make each character into a corresponding token, the parser that reads these tokens would in effect be 'scannerless'. The reason that we distinguish scannerless parsing from parsing with a real scanner is that the former generates some special problems that are avoided by using a scanner.

2.1 Advantages

Now that we have an understanding of what scannerless parsing is, we might ask why it is any good. We will discuss the advantages one by one.

No Scanner The obvious advantage of scannerless parsing is that no implementation of a scanner and scanner generator is needed and that the complicated interface between scanner and parser can be eliminated.

Integrated Uniform Grammar Formalism A language is completely defined by means of one grammar. All grammar rules are explicit and formally specified. Lexical syntax and context-free syntax are specified with the same formalism. There is no longer a distinction between regular and context-free grammars. This makes the formalism more uniform and orthogonal. All features available for lexical syntax are available for context-free syntax and vice versa. This simplifies use and implementation of the formalism.

Disambiguation by Context Because of the integration of lexical and context-free syntax, lexical analysis is guided by context-free analysis. If a token does not make sense at some position, it will not even be considered. For instance, in the example above, the longest match rule does not have to be used to prefer $\boxed{\texttt{ab}}\boxed{\sqcup}$ over $\boxed{\texttt{a}}\boxed{\texttt{b}}\boxed{\sqcup}$ because the latter situation—two adjacent identifiers—is never syntactically correct.

The paradigmatic example of context-dependent lexical disambiguation is the interplay between subrange types and floating point number constants in Pascal. Subrange types have the form `k..l`, where `k` and `l` are constants. If floating point number constants could have the form `i.` and `.j` with `i` and `j` numbers, then `i..j` could be scanned either as $\boxed{\texttt{i}}\boxed{\texttt{..}}\boxed{\texttt{j}}$, i.e., the range from `i` to `j`, or as $\boxed{\texttt{i.}}\boxed{\texttt{.j}}$, i.e., two adjacent floating point numbers. In scannerless parsing, this ambiguity is solved automatically by context. A scanner that has no access to the context and applies the longest match rule, would always choose the second possibility (two adjacent numbers) and fail. Apparently for this reason the syntax of Pascal only allows real numbers of the form `i.j`, where `i` and `j` are non-empty lists of digits (Jensen and Wirth, 1975). Similar examples can be found in many languages.

Another example where the parsing context is relevant for making lexical decisions is the syntax of lists of statements that can be separated by semicolons or newlines. Consider the grammar

```
lexical syntax
  [\ \t\n] -> LAYOUT
context-free syntax
  "begin" {Stat ";"|"\n"}* "end" -> Block
```

The lexical syntax defines newlines (`\n`) to be layout. The context-free syntax defines blocks as lists of zero or more statements starting with the keyword `begin` and ending with the keyword `end`. The list is declared by the construct $\{A\ B\}*$, which declares a list of As separated by Bs, i.e., a list of the form $A\ B\ A\ \ldots\ B\ A$. In this case the separator is either a semicolon or a newline. This means that newlines are both layout and non-layout. If the disambiguation rule 'prefer non-layout' is applied to the tokens of this language, all newlines—even those not used as separator of statements—will be wrongly characterized as non-layout. A scannerless parser will recognize the newlines used as separator simply by considering the parsing context.

Conservation of Lexical Structure Scanners do usually not maintain the phrase structure of the tokens they produce. For example, the grammar

```
lexical syntax
  [a-z]+          -> Id
  "/"? {Id "/"}+ -> Path
```

defines the lexical syntax of path expressions as occur, for instance, in the naming conventions of tree-structured filesystems. This syntax has to be lexical since no layout should occur between the identifiers and separators of a path. A scanner would produce a string containing the characters of a path expression without the structure assigned to it by the grammar, i.e., the distinction between identifiers in the path is lost. This entails that the semantic processor must reparse such tokens to deal with their internal structure.

Conservation of Layout Scanners throw away the layout between tokens of a phrase. In this way the parser can ignore layout, which simplifies the parsing problem. However, there are examples of operations on programs that require the structure of the program, i.e., the parse tree, but also the layout in the source. Examples are source to source translations, transformations on the source text and program documentation tools. Although a conventional parser could be instructed to add the layout to the parse tree via some detour, this would usually require a non-standard extension of the method. If the layout would be explicitly specified in the grammar we would get an approach that is very similar to scannerless parsing.

Expressive Lexical Syntax Context-free grammars provide a more expressive grammar formalism for lexical syntax than regular grammars. This additional expressive power opens the way to concise definitions of nested comments and syntactically correct expressions in comments. For example, consider the following extension of the expression grammar above that defines C-like comments as a list of comment words between `/*` and `*/`.

```
sorts ComWord Comment
lexical syntax
  ~[\ \t\n\|\/]+ -> ComWord
context-free syntax
  "/*" ComWord* "*/" -> Comment
  Comment            -> LAYOUT
```

A comment word is a non-empty list of characters that are not whitespace, `|` or `/`. Since the definition of comments is part of the context-free syntax, comment words can be separated by layout. These comments are made into layout by the last line of the grammar, which is an injection of comment into layout. Because layout can occur between any two adjacent tokens, comment can as well.

According to this definition, comments can be nested, because comment words are separated by layout, which includes comments. For instance, the string

```
  h /* height */
/*
  * w /* width */
  * d /* depth */
*/
```

is a syntactically correct expression over the grammar above in which part of an expression including comments is commented out. This is a tedious job if nested comments are not supported by the language.

Moreover, the following extension of the grammar above defines that a comment word can also be an expression between two `|`s.

```
context-free syntax
  "|" Exp "|" -> ComWord
```

This entails that comments can contain quoted expressions that must be syntactically correct. For instance, the following sentence contains the expression `x + y` as part of a comment.

```
a + b /* an expression |x + y| denotes
         the addition of |x| and |y| */
+ c
```

This is useful for typesetting comments in literate programs and for generating cross-references.

2.2 Problems & Solutions

Now one might ask why scannerless parsing was not introduced earlier, if it has so many advantages. The answer is that there are several problems caused by the integration of lexical and context-free syntax as well. In this paper we discuss solutions to these problems that make scannerless parsing feasible.

Limitations of Parsing Techniques The main problem with scannerless parsing are the limitations of the conventional deterministic parsing techniques. Most complete character level grammars are *not* LR(1), LL(1), or even LR(k) due to lookahead needed for lexical elements. When parsing with a scanner a lookahead of 1 entails looking one token ahead. In scannerless parsing a lookahead of 1 entails only considering the next character. Furthermore, when layout is skipped by the scanner this need not be considered in the lookahead. The solution used in the SDF2 implementation is to use the generalized-LR parsing algorithm of Tomita (1985) and Rekers (1992) to get dynamic lookahead.

Grammar Size Another problem is the size of grammars. Complete character level grammars are large because all constructs have to be specified down to the character level. Furthermore, the placement of layout between tokens should be explicitly declared in productions. For maintenance and readability of grammars this is problematic. To support the development of complete character level grammars an expressive formalism is needed that hides the details of the interface between lexical and context-free syntax and of the placement of layout. In §3 we discuss the approach of grammar normalization in order to provide an expressive formalism with a minimal semantic basis. In §4 we discuss the extension of context-free grammars with various disambiguation constructs to keep grammars concise.

Lexical Disambiguation Although many lexical ambiguities are solved automatically through the integration of lexical and context-free syntax, there are still cases where disambiguation of lexical constructs needs to be expressed. Since lexical analysis is now based on context-free parsing, familiar lexical disambiguation rules such as 'prefer longest match' and 'prefer keyword' have to be redefined and their implementation reconsidered. In §4 we discuss two disambiguation constructs for lexical disambiguation: follow restriction and reject productions that suffice to express all common lexical disambiguation rules.

Interpretation of Disambiguation Rules There are a number of ways to interpret disambiguation constructs. One possibility is to implement them as a filter on parse forests as proposed in Klint and Visser (1994). However, for disambiguation of lexical constructs and context-free expressions with priorities this can lead to an exponential size of the parse forest before filtering, which makes the method too inefficient. In §5 we discuss the techniques used in parser generation to encode disambiguation rules in the parse tables such that decisions are taken early. In §8 an extension of the GLR parsing algorithm with reject reductions is presented.

Efficiency The first problem that comes to mind when considering scannerless parsing is efficiency. Since scanning with a finite automaton has a lower complexity than parsing with a stack, scannerless parsing, i.e., replacing the finite automaton part by a stack machine, should be less efficient. The following considerations led us to attempt scannerless parsing, nonetheless: (1) LR parsing is linear, in particular for regular grammars. Since lexical syntax is traditionally formulated by means of regular grammars, we should expect linear behaviour for the lexical part of scannerless parsers. (2) The complete complexity of the scanner/parser setup should be considered including lexical disambiguation. If lexical disambiguation rules cannot solve all ambiguities and disambiguation has to be deferred to the parser, a kind of graph structured stack has to be maintained to keep track of the possible segmentation of the string in tokens. (Such a setup is used in the ASF+SDF Meta-Environment (Klint, 1993) that forms the background for the development of SDF2.) It seems even more efficient to maintain a single graph structured stack, instead of two. (3) If more complex grammars for lexical syntax are used, we get into an area where scannerless parsing and parsing with scanners can no longer be properly compared because such syntax is not expressible in the scanner framework. Therefore, the worst case complexity of context-free parsing should not be taken as a reference point for considering the complexity of scannerless parsing.

Of course, these considerations should be verified by means of experiments. However, experiments with scannerless parsing can only be performed after solutions have been found for the other problems discussed above. It seems that these problems are the cause for the late introduction of scannerless parsing rather than bad efficiency of the method. In §9 we will discuss a few simple experiments that have been performed with the scannerless parsing method described in this paper and that seem to confirm our expectations.

3 Grammar Normalization

We need an expressive grammar formalism in which lexical syntax and context-free syntax are integrated and that supports concise syntax definitions. SDF2 is such an expressive formalism. It provides regular expressions, lexical and context-free syntax, character classes, literals, priorities, modules, renaming, and aliases. The first version of the formalism was described in Visser (1995b). The full definition is presented in Visser (1997c). Because it is expensive to extend tools to such an expressive formalism, all features that are expressible in more primitive features are eliminated by means of a normalization function on grammars.

3.1 Normal Form

The expressive power of the syntax definition formalism SDF2 can be characterized by the equation

$$\begin{aligned}\text{SDF2} = {} & \text{context-free grammars} + \text{character-classes} + \text{priorities} \\ & + \text{reject productions} + \text{follow restrictions}\end{aligned}$$

That is, any SDF2 definition is equivalent to a context-free grammar making use of character classes, priorities, reject productions and follow restrictions. All other features are expressed in terms of these features. The equivalence is such that a definition is equivalent to a definition of the form

$$\begin{array}{l}\text{sorts } s_1 \ldots s_j \\ \text{syntax } p_1 \ldots p_k \\ \text{priorities } pr_1, \ldots, pr_l \\ \text{restrictions } r_1 \ldots r_m\end{array}$$

where the s_i are sort symbols, the p_i are context-free productions of the form $\alpha \to A$, the pr_i priority declarations of the form p_j R $p_{j'}$ with R a priority relation, and the r_i follow restrictions of the form $\alpha \not\leftarrow cc$ with α a list of symbols and cc a character class.

A production can have a number of attributes that may include the attribute `reject`, which makes the production a reject production. A priority relation is one of `left`, `right`, `assoc`, `non-assoc` or `>`. A symbol can be a character class or some other symbol. Only character classes are interpreted during parser generation. Other symbols constructed using symbol operators are simply interpreted as a name. For instance, the symbol $A+$ used to indicate the iteration of symbol A has no special meaning after normalization.

Given a grammar $\mathcal{G}$ the following projection functions are defined:

$$\begin{aligned}\mathrm{S}(\mathcal{G}) &\mapsto \text{sorts of } \mathcal{G} \\ \mathrm{Syms}(\mathcal{G}) &\mapsto \text{symbols used in } \mathcal{G} \\ \mathrm{P}(\mathcal{G}) &\mapsto \text{productions of } \mathcal{G} \\ \mathrm{Pr}(\mathcal{G}) &\mapsto \text{priorities of } \mathcal{G} \\ \mathrm{R}(\mathcal{G}) &\mapsto \text{restrictions of } \mathcal{G}\end{aligned}$$

3.2 Normalization

As an example of the normal form, consider the grammar in Figure 3. It completely describes the lexical and context-free syntax of expressions with identifier, multiplication and addition—the same language described in the example in §2. In fact, this grammar is derived from that grammar by application of a normalization procedure. We briefly discuss the elements of this normalization that is formally specified in Visser (1997c). Refer to Figure 3 for examples of the normalization rules.

Lexical and Context-free Syntax The most important aspect of the normalization for this paper is the integration of lexical and context-free syntax. The productions of lexical and context-free syntax are merged. In order to avoid interference of lexical and context-free syntax the symbols in productions are renamed. The symbols in the lexical syntax—except for character classes and literals—are renamed using the symbol constructor $\langle\texttt{\_-LEX}\rangle$. For instance, `Id` becomes $\langle\texttt{Id-LEX}\rangle$ and `[\97-\122]+` becomes `<[\97-\122]+-LEX>`. Similarly, the symbols in the context-free syntax are renamed using $\langle\texttt{\_-CF}\rangle$. Furthermore, the symbols on the left-hand side of context-free productions are separated by $\langle\texttt{LAYOUT?-CF}\rangle$, which entails that layout can occur at that position. In this way two disjunct sets of symbols are created. The interface between lexical and context-free syntax is now expressed by an injection $\langle A\texttt{-LEX}\rangle \to \langle A\texttt{-CF}\rangle$ for each symbol A used both in the lexical and the context-free syntax.

Top Symbol A syntax definition defines a number of symbols. A text over such a definition can be one produced by any of its symbols. For context-free parsing we need a single start symbol from which all strings are generated. For this purpose for each sort A a production

$$\langle\texttt{LAYOUT?-CF}\rangle\ \langle A\texttt{-CF}\rangle\ \langle\texttt{LAYOUT?-CF}\rangle \to \langle\texttt{START}\rangle$$

is added to the grammar, defining the start symbol $\langle\texttt{START}\rangle$. The production also defines that a string can start and end with layout. Furthermore, to express the termination of a string the production

$$\langle\texttt{START}\rangle\ [\backslash\texttt{EOF}] \to \langle\texttt{Start}\rangle$$

defines that a string consists of a string generated by $\langle\texttt{START}\rangle$ followed by the end of file character.

Character Classes Character classes are expressions of the form $[cr_1 \ldots cr_n]$ where the cr_i are either characters or character ranges of the form $c{-}c'$. Character classes are normalized to a unique normal form by translating the characters to a numeric character code—the ASCII code—and by ordering and merging the ranges such that they are in increasing order and do not overlap. This normalization is formally specified and proven correct with respect to the set interpretation of character classes in Visser (1997b).

Literals Literals are abbreviations for fixed lists of characters. Literals are defined in terms of a production with the literal as result and singleton character classes corresponding to the characters as arguments. For example, the production

```
[\108] [\101] [\116] -> "let"
```

defines the literal `"let"` as the sequence of characters `l`, `e` and `t` in ASCII.

Regular Expressions An extensive set of regular expressions including optional, alternative, tupling, several kinds of iteration and permutation are expressed by means of defining productions. For instance, consider the definition of `<[\97-\122]+-LEX>` in Figure 3, which defines a list of one or more lowercase letters.

Priorities Priorities can be declared using chains of $>$ declarations and associativities of productions can be declared using groups and attributes. These are all defined in terms of binary priority and associativity declarations.

```
sorts Id Exp
syntax
  [\9-\10\32]                                -> <LAYOUT-LEX>
  <LAYOUT-LEX>                               -> <LAYOUT-CF>
  <LAYOUT-CF> <LAYOUT-CF>                    -> <LAYOUT-CF> {left}
                                             -> <LAYOUT?-CF>
  <LAYOUT-CF>                                -> <LAYOUT?-CF>

  [\42]                                      -> "*"
  [\43]                                      -> "+"

  [\97-\122]                                 -> <[\97-\122]+-LEX>
  <[\97-\122]+-LEX> <[\97-\122]+-LEX>        -> <[\97-\122]+-LEX>
                                                              {left}
  <[\97-\122]+-LEX>                          -> <Id-LEX>
  <Id-LEX>                                   -> <Id-CF>

  <Id-CF>                                      -> <Exp-CF>
  <Exp-CF> <LAYOUT?-CF> "*" <LAYOUT?-CF> <Exp-CF> -> <Exp-CF> {left}
  <Exp-CF> <LAYOUT?-CF> "+" <LAYOUT?-CF> <Exp-CF> -> <Exp-CF> {left}

  <LAYOUT?-CF> <Id-CF>  <LAYOUT?-CF>           -> <START>
  <LAYOUT?-CF> <Exp-CF> <LAYOUT?-CF>           -> <START>
  <START> [\EOF]                               -> <Start>
priorities
  <[\97-\122]+-LEX> <[\97-\122]+-LEX>        -> <[\97-\122]+-LEX>
  left
  <[\97-\122]+-LEX> <[\97-\122]+-LEX>        -> <[\97-\122]+-LEX>,
  <LAYOUT-CF> <LAYOUT-CF>                    -> <LAYOUT-CF> left
  <LAYOUT-CF> <LAYOUT-CF>                    -> <LAYOUT-CF>,
  <Exp-CF> <LAYOUT?-CF> "*" <LAYOUT?-CF> <Exp-CF> -> <Exp-CF> >
  <Exp-CF> <LAYOUT?-CF> "+" <LAYOUT?-CF> <Exp-CF> -> <Exp-CF>,
  <Exp-CF> <LAYOUT?-CF> "*" <LAYOUT?-CF> <Exp-CF> -> <Exp-CF> left
  <Exp-CF> <LAYOUT?-CF> "*" <LAYOUT?-CF> <Exp-CF> -> <Exp-CF>,
  <Exp-CF> <LAYOUT?-CF> "+" <LAYOUT?-CF> <Exp-CF> -> <Exp-CF> left
  <Exp-CF> <LAYOUT?-CF> "+" <LAYOUT?-CF> <Exp-CF> -> <Exp-CF>
```

Figure 3: Expression grammar in normal form. The grammar contains no restrictions or reject productions.

Modules An SDF2 definition can be divided over a number of modules. Modules can import other modules. This is used to share common syntax definitions in several language definitions. Renamings of symbols and productions can be used to adapt the definition in a module to some application. Furthermore, symbol aliases can be used to abbreviate long regular expressions. Modular syntax definitions are completely expanded by the normalization function.

3.3 Semantics

A syntax definition defines a language, i.e., a set of strings, and the structure that is assigned to those strings. The strings of the language are important to its users who write down programs. The structure of those strings is important for the definition of language processors such as compilers, interpreters and typecheckers. The productions of an SDF2 definition describe both the language and the structure assigned to strings in the language. The semantics of a syntax definition is a set of parse trees from which a set of strings can be derived. The mapping from trees to strings is achieved by taking the yield of a tree. The reverse mapping from strings to trees is called parsing. At this point, we formally define the semantics of context-free grammars without considering disambiguation rules such as priorities, reject productions and follow restrictions.

A context-free grammar $\mathcal{G}$ generates a family of sets of *parse trees* $\mathcal{T}(\mathcal{G}) = (\mathcal{T}(\mathcal{G})(X) \mid X \in \text{Syms}(\mathcal{G}))$, which contains the minimal sets $\mathcal{T}(\mathcal{G})(X)$ such that

$$\frac{c \in cc}{c \in \mathcal{T}(\mathcal{G})(cc)} \tag{Char}$$

$$\frac{A_1 \ldots A_n \to A \in \mathrm{P}(\mathcal{G}),\ t_1 \in \mathcal{T}(\mathcal{G})(A_1),\ \ldots\ ,\ t_n \in \mathcal{T}(\mathcal{G})(A_n)}{[t_1 \ldots t_n \to A] \in \mathcal{T}(\mathcal{G})(A)} \tag{Prod}$$

In rule (Char) c is a character and cc a character class. We will write t_α for a list $t_1 \ldots t_n$ of trees where α is the list of symbols $X_1 \ldots X_n$ and $t_i \in \mathcal{T}(\mathcal{G})(X_i)$ for $1 \leq i \leq n$. Correspondingly we will denote the set of all lists of trees of type α as $\mathcal{T}(\mathcal{G})(\alpha)$. Using this notation $[t_1 \ldots t_n \to A]$ can be written as $[t_\alpha \to A]$ and the concatenation of two lists of trees t_α and t_β is written as $t_\alpha t_\beta$ and yields a list of trees of type $\alpha\beta$.

The *yield* of a tree is the concatenation of its leafs. The *language* defined by a grammar $\mathcal{G}$ is the family $L(\mathcal{G}) = (L(\mathcal{G})(X) \mid X \in \text{Syms}(\mathcal{G}))$ of sets of strings that are yields of trees over the grammar, i.e., $L(\mathcal{G})(X) = \text{yield}(\mathcal{T}(\mathcal{G})(X))$. A *parser* is a function Π that maps a string of characters to a set of parse trees. A parser Π *accepts* a string w if $|\Pi(w)| > 0$. A parser for a context-free grammar $\mathcal{G}$ that accepts exactly the sentences in $L(\mathcal{G})$ is defined by

$$\Pi(\mathcal{G})(w) = \{t \in \mathcal{T}(\mathcal{G})(X) \mid X \in \text{Syms}(\mathcal{G}),\ \text{yield}(t) = w\}$$

A parser Π is *deterministic* if $|\Pi(w)| \leq 1$ for all strings w. A grammar is ambiguous if there are strings with more than one parse tree, i.e, $|\Pi(\mathcal{G})(w)| > 1$.

4 Disambiguation

Disambiguation methods are used to select the intendend tree from a set of possible parse trees for an ambiguous string. SDF2 provides three disambiguation methods. Priority and associativity declarations are used to disambiguate concise expression grammars. Follow restrictions and reject productions are used to express lexical disambiguation. In this section we discuss these methods.

4.1 Disambiguation by Priorities

By using priority and associativity declarations, fewer grammar symbols have to be introduced and a more compact abstract syntax can be achieved. Consider the following grammar of expressions in a functional programming language with binary function application and let binding.

```
sorts Var Term
lexical syntax
  [a-z]+   -> Var
  [\ \t\n] -> LAYOUT
context-free syntax
  Var                          -> Term
  Term Term                    -> Term {left}
  "let" Var "=" Term "in" Term -> Term
  Term "=" Term                -> Term {non-assoc}
  "(" Term ")"                 -> Term {bracket}
context-free priorities
  Term Term                    -> Term >
  Term "=" Term                -> Term >
  "let" Var "=" Term "in" Term -> Term
```

An example term over this grammar is

```
let sum = foldr plus zero in sum lst
```

The grammar is disambiguated by means of priorities. The binary application operator is declared as left-associative. This entails that $x\ y\ z$ should be read as $(x\ y)\ z$ and not as $x\ (y\ z)$. This is illustrated in Figure 4 that shows the right- and left-associative parse trees for three adjacent terms. The priority declaration defines applications to have higher priority than equalities. Consider the trees in Figure 5. According to the priority declaration, the first tree has a priority conflict and therefore only the second tree is a correct parse tree. The following definition formally defines the notion of priority conflicts.

Definition 4.1 Given some grammar $\mathcal{G}$ with priority declarations $\mathrm{Pr}(\mathcal{G})$, the set $\mathrm{conflicts}(\mathcal{G})$ of priority conflicts over grammar $\mathcal{G}$ is the smallest set of parse tree patterns of the form $[\alpha[\beta \to B]\gamma \to A]$ such that:

$$\frac{\alpha B\gamma \to A > \beta \to B \in \mathrm{Pr}(\mathcal{G})}{[\alpha[\beta \to B]\gamma \to A] \in \mathrm{conflicts}(\mathcal{G})} \tag{CF1}$$

$$\frac{\gamma \neq \epsilon,\ \beta \to B\ (\texttt{right} \cup \texttt{non-assoc})\ B\gamma \to A \in \mathrm{Pr}(\mathcal{G})}{[[\beta \to B]\gamma \to A] \in \mathrm{conflicts}(\mathcal{G})} \tag{CF2}$$

$$\frac{\alpha \neq \epsilon,\ \beta \to B\ (\texttt{left} \cup \texttt{assoc} \cup \texttt{non-assoc})\ \alpha B \to A \in \mathrm{Pr}(\mathcal{G})}{[\alpha[\beta \to B] \to A] \in \mathrm{conflicts}(\mathcal{G})} \tag{CF3}$$

A parse tree over a grammar $\mathcal{G}$ has a priority conflict if one of its nodes matches a pattern $[\alpha[\beta \to B]\gamma \to A] \in \mathrm{conflicts}(\mathcal{G})$. □

Using the notion of priority conflicts we can define a filter on sets of parse trees that selects the trees without a conflict. For example, according to rule (CF3) and because of the declaration of application as a left-associative operator, the pattern

$$[\langle\texttt{T-CF}\rangle\ \langle\texttt{L?-CF}\rangle\ [\langle\texttt{T-CF}\rangle\ \langle\texttt{L?-CF}\rangle\ \langle\texttt{T-CF}\rangle \to \langle\texttt{T-CF}\rangle] \to \langle\texttt{T-CF}\rangle]$$

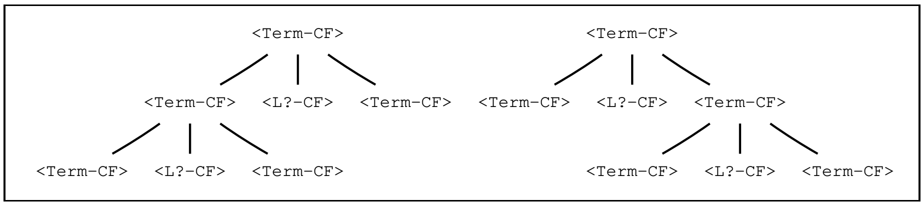


Figure 4: Left- and right-associative parse trees for binary term application.

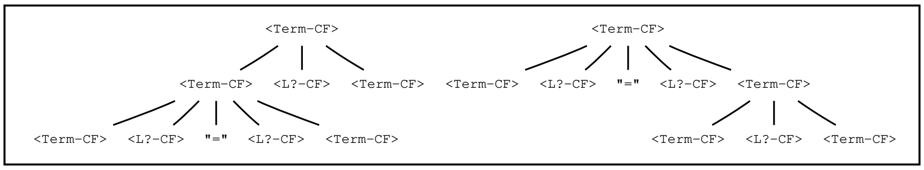


Figure 5: Two parse trees for application and equality.

describes a tree with a conflict. (`Term` and `LAYOUT` are abbreviated to `T` and `L`, respectively.) Therefore, the second tree in Figure 4 has a conflict and the first one is selected by the disambiguation method. According to rule (CF1) and because application has higher priority than equality, the pattern

$$[[\langle\texttt{T-CF}\rangle\ \langle\texttt{L?-CF}\rangle\ \texttt{"="}\ \langle\texttt{L?-CF}\rangle\ \langle\texttt{T-CF}\rangle \to \langle\texttt{T-CF}\rangle]\ \langle\texttt{L?-CF}\rangle\ \langle\texttt{T-CF}\rangle \to \langle\texttt{T-CF}\rangle]$$

is a member of the conflicts generated by the functional language grammar. This means that the first tree in Figure 5 has a priority conflict. The second tree has no conflict.

4.2 Lexical Disambiguation

If we consider the example of functional expressions again we see that it contains two occurrences of lexical ambiguities.

4.2.1 Longest Match In the first place there is a longest match problem caused by the syntax-less binary application operator. Two adjacent letters could be the concatenation of two letters forming a variable, or it could be the application of two single letter variables. Figure 6 shows two parse trees for the string `fa`. In the first tree the concatenation of letter lists is used to make them into a single variable. In the second tree each letter is interpreted as a variable on its own. We want to solve this ambiguity by means of the longest match rule that prefers the longest possible lexical token. In this case the string `fa` as a single variable. We define the longest match notion formally by comparing the lengths of tokens. For this definition we first need the notion of the token stream associated to a parse tree.

Definition 4.2 (Token Stream) The *token stream* associated with a parse tree is the list of subtrees that have as root either an injection $\langle A\texttt{-LEX}\rangle \to \langle A\texttt{-CF}\rangle$ or a literal defining production. The length $|t|$ of a token t is the number of characters in its yield. □

According to this definition the token streams for the trees in Figure 6 are the single token

[[[[f → <[a-z]+-LEX>][a → <[a-z]+-LEX>] → <[a-z]+-LEX>]
→ <Var-LEX>] → <Var-CF>]

and the tokens

[[[f → <[a-z]+-LEX>] → <Var-LEX>] → <Var-LEX>]
[[[a → <[a-z]+-LEX>] → <Var-LEX>] → <Var-LEX>]

The idea of longest match disambiguation is to compare two token streams from left to right. While the tokens have the same length the streams are similar. The first token that differs in length solves the ambiguity by taking the tree associated with the longer token. In the example above, the first token stream is larger because its first token has length 2 while the first token of the second stream has length 1.

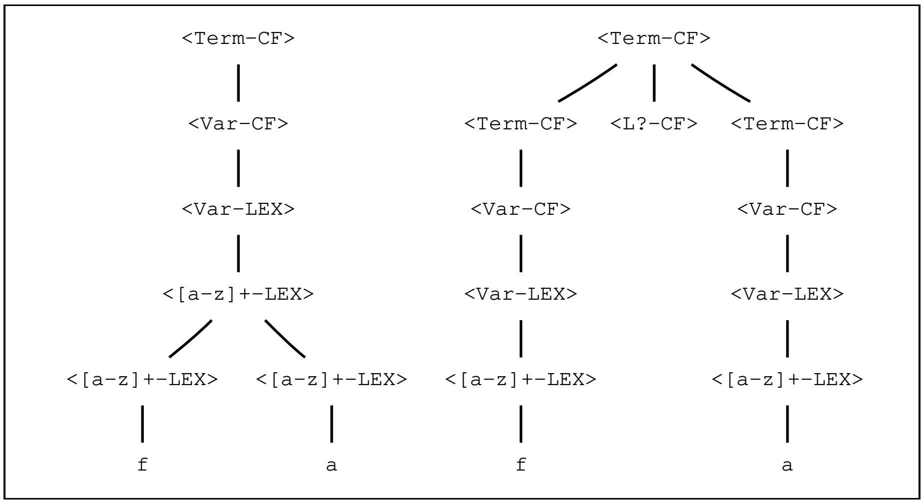


Figure 6: Two parse trees for `fa` over the functional expression grammar.

Formally we have the following definition of longest match disambiguation:

Definition 4.3 (Longest Match) Given the token streams $t_1 \ldots t_n$ associated with the tree t and $s_1 \ldots s_m$ associated with tree s, tree t is larger in the *longest match ordering* $>_{\mathrm{lm}}$ than s ($t >_{\mathrm{lm}} s$), if there is some $1 \leq i \leq \min(n, m)$ such that $|t_j| = |s_j|$ for $1 \leq j < i$ and $|t_i| > |s_i|$. □

This definition can be used as a method to filter parse forests by selecting the largest trees according to the longest match ordering. However, because a longest match ambiguity causes an exponential explosion of the parse forest this is not feasible. We need a method that can be applied during parsing, if possible as a filter on parse tables. A naive solution for the longest match problem in the example above is to require non-empty layout as a separator between the two terms of an application. In the example this would indeed solve the problem because the second tree would be forbidden. However, this solution is immediately refuted by considering the expression `f(a)`, where brackets are used around the argument.

A method that works in all cases we have encountered so far is that of follow restrictions. A follow restriction of the form $A_1 \ldots A_n \not\rightharpoondown cc$ declares that the symbols A_i should not be followed by any of the characters in the character class cc. In the example above the restriction

```
lexical restrictions
  Var -/- [a-z]
```

forbids a variable to be followed by a letter. This entails that the second tree in Figure 6 violates the follow restrictions and the desired first tree is selected.

4.2.2 Prefer Literals The second problem in the functional expression grammar is the overlap between the literals `"let"` and `"in"` and variables. This is particularly problematic in combination with the `=` operator on terms. A let binding `let x = t1 in t2` can be interpreted also as an equality `(let x) = (t1 in t2)`, where `let` and `in` are now read as variables. We clearly want to declare `let` and `in` as reserved words of the language that should not be used as variables. This lexical disambiguation rule is called 'prefer literals' and can be defined formally as follows.

Definition 4.4 (Prefer Literals) A tree violates the prefer literals rule if it contains a subtree with function $\langle A\texttt{-LEX}\rangle \rightarrow \langle A\texttt{-CF}\rangle$ and the yield of that tree is also used as literal in the grammar. □

This rule can be expressed by means of reject productions. A reject production is a production $\alpha \rightarrow A$ attributed with the attribute `reject`. It declares that a string is not of type A if it can also be derived from α. For example to disambiguate the grammar above we add the following productions.

```
lexical syntax
  "let" -> Var {reject}
  "in"  -> Var {reject}
```

This creates an ambiguity: `let` can be a variable in two ways, via the lexical definition or via the production above. Because this is a reject production both derivations are forbidden, i.e., `let` can only occur in the context of a `let` binding. We also need the restrictions

```
lexical restrictions
  "let" "in" -/- [a-z]
```

to prevent `letter` to be interpreted as the literal `let` and the variable `ter`. We will further discuss some properties of reject productions in §7.

4.2.3 Automatic Lexical Disambiguation We have defined two extensions of context-free grammars that enable us to express lexical disambiguation rules on grammars for integrated lexical and context-free syntax. However, it is desirable to derive the rules for lexical disambiguation automatically from the grammar. In §6 we will discuss this issue, after we have discussed parser generation.

5 Parser Generation

We have discussed a grammar formalism with disambiguation methods for concise definition of lexical and context-free syntax of languages. Now we turn our attention to deriving parsers from such syntax definitions. In this section we present the rules for the generation of parse tables for a shift-reduce parser. The rules constitute a modification of the well known SLR(1) algorithm. We first discuss shift-reduce parsing.

5.1 Shift-Reduce Parsing

A shift-reduce parser is a transition system that manipulates its state consisting of a stack and an input stream by repeatedly shifting a symbol from the input to the stack or reducing a number of elements on top of the stack to a single element until it enters an accepting state. The transitions between parse configurations are determined by the functions 'actions' and 'goto' as defined by the following transition rules:

$$\frac{\text{actions}(s_m, a_i) \ni \mathtt{shift}(s_{m+1})}{(s_0 t_1 s_1 \ldots t_m s_m \bullet a_i \ldots a_n) \Rightarrow (s_0 t_1 s_1 \ldots t_m s_m a_i s_{m+1} \bullet a_{i+1} \ldots a_n)} \quad \text{(Shi)}$$

$$\frac{\begin{array}{c}\text{actions}(s_{m+k}, a_i) \ni \mathtt{reduce}(p, k),\\ s = \text{goto}(s_m, p),\ t = \text{tree}(p, [t_{m+1}, \ldots, t_{m+k}])\end{array}}{\begin{array}{c}(s_0 t_1 \ldots t_m s_m t_{m+1} s_{m+1} \ldots t_{m+k} s_{m+k} \bullet a_i \ldots a_n)\\ \Rightarrow (s_0 t_1 \ldots t_m s_m t s \bullet a_i \ldots a_n)\end{array}} \quad \text{(Red)}$$

$$\frac{\text{actions}(s_1, \mathtt{\backslash EOF}) \ni \mathtt{accept}}{(s_0 t_1 s_1 \bullet \mathtt{\backslash EOF}) \Rightarrow \mathtt{accept}(t)} \quad \text{(Acc)}$$

Here a configuration $(s_0 t_1 s_1 \ldots t_m s_m \bullet a_i \ldots a_n)$ consists of a stack on the left side of the $\bullet$ and a list of input characters on the right side of the $\bullet$. The stack is filled alternatingly with states s and trees t. Parsing starts in the configuration $C_0 = (s_0 \bullet a_1 \ldots a_n)$, where s_0 is the initial state of the parser. Parsing succeeds if there is some sequence of steps $C_0 \Rightarrow C_1 \Rightarrow \ldots \Rightarrow \mathtt{accept}(t)$ that ends in the accepting configuration $\mathtt{accept}(t)$.

There are various ways to define the actions and goto functions that drive a shift-reduce parser. The SLR(1) algorithm of DeRemer (1971) and Anderson *et al.* (1973) is a simplification of the LR(k) parsing algorithm of Knuth (1965). It works by first constructing an LR(0) parse table. This involves no lookahead sets in the parse items. The lookahead of reductions is constrained to the follow set of the nonterminal defined by the production being reduced.

In the rest of this section we describe a modification of the SLR(1) algorithm that incorporates priorities and follow restrictions. This modification is based on the derivation in Visser (1997a), where starting with a schema for Earley's parsing algorithm, a parsing schema is derived such that the parser does not build trees with priority conflicts. Other changes are the use of character classes, the use of productions instead of symbols in follow and goto and the interpretation of follow restrictions to restrict the lookahead set of reductions.

5.2 First

The first set for a symbol contains those symbols with which a phrase for the symbol can start. Given some grammar $\mathcal{G}$, define for each list of symbols α and each character class cc the first characters in α followed by cc is the smallest character class $\mathrm{first}(\alpha, cc)$ such that:

$$\mathrm{first}(\epsilon, cc) = cc \tag{Fi1}$$

$$\mathrm{first}(cc'\ \alpha, cc) = cc' \tag{Fi2}$$

$$\frac{\alpha \to A \in \mathrm{P}(\mathcal{G})}{\mathrm{first}(A\beta, cc) \supseteq \mathrm{first}(\alpha\beta, cc)} \tag{Fi3}$$

The definition of the first set can be extended to the set of symbols that starts a sentence derived from a list of symbols.

$$\frac{\alpha \to A \in \mathrm{P}(\mathcal{G})}{\mathrm{first}_s(A\beta, \Phi) \supseteq \{A\} \cup \mathrm{first}(\alpha\beta, \Phi)} \tag{Fi4}$$

5.3 Follow

In the conventional SLR(1) algorithm the follow set is computed for each nonterminal of the grammar. It maps a nonterminal to the set of terminals that can follow that nonterminal in a sentence, i.e.,

$$\frac{\alpha A \beta \to^* \langle \mathtt{Start} \rangle}{\mathrm{follow}(A) \supseteq \mathrm{first}(\beta, \emptyset)}$$

This can be computed as the closure of

$$\frac{\beta A \gamma \to B \in \mathrm{P}(\mathcal{G})}{\mathrm{follow}(A) \supseteq \mathrm{first}(\gamma, \mathrm{follow}(B))} \tag{Fo1}$$

that adds the characters in the first set of γ to the follow set of A if γ follows A in some production. The follow of B is added in case γ can also produce the empty string.

This notion can be refined to the follow-set of productions. The rule

$$\frac{\beta A \gamma \to B \in \mathrm{P}(\mathcal{G})}{\mathrm{follow}(\alpha \to A) \supseteq \mathrm{first}(\gamma, \mathrm{follow}(\beta A \gamma \to B))} \tag{Fo2}$$

defines the follow-set of production $\alpha \to A$ as those characters that can follow A in some context. In case of plain context-free grammars, rule (Fo2) has the same effect as rule (Fo1). But if we consider priorities, the rule is extended to

$$\frac{\beta A \gamma \to B \in \mathrm{P}(\mathcal{G}), [\beta[\alpha \to A]\gamma \to B] \notin \mathrm{conflicts}(\mathcal{G})}{\mathrm{follow}(\alpha \to A) \supseteq \mathrm{first}(\gamma, \mathrm{follow}(\beta A \gamma \to B))} \tag{Fo3}$$

Here the follow-set of a production is restricted to those contexts where it can actually be used without causing a priority conflict. For instance, in the expression grammar of §2, the follow-set of the addition production does not contain the character $*$ because addition can not occur as a direct descendant of multiplication.

Finally, if the grammar also defines follow restriction rules $A \not\to cc$, the follow-set of a production for A can be further restricted as

$$\frac{\beta A \gamma \to B \in \mathrm{P}(\mathcal{G}), [\beta[\alpha \to A]\gamma \to B] \notin \mathrm{conflicts}(\mathcal{G}), A \not\to cc \in \mathrm{R}(\mathcal{G})}{\mathrm{follow}(\alpha \to A) \supseteq \mathrm{first}(\gamma, \mathrm{follow}(\beta A \gamma \to B)) \setminus cc} \tag{Fo4}$$

The production can be followed by the difference of the first set of the right context and the character class cc.

To see the effect of the last rule consider the follow-set of the production `[a-z]+ -> Var` in the functional expression grammar of the previous section. Because of the application production and the injection of variables into terms, the follow-set of `[a-z]+ -> Var` is `[\EOF\t\n\ \(\)\=a-z]`. The lexical restriction `Var -/- [a-z]` removes the character class `[a-z]` from this follow-set, resulting in `[\EOF\t\n\ \(\)\=]`. This entails that a variable cannot directly be followed by a letter.

5.4 Goto Table

The states of an LR parser are formed by item-sets. An item is an object of the form $[\alpha \bullet \beta \to A]$, i.e., a context-free production with a $\bullet$ somewhere in between the symbols on the left-hand side. Such an item indicates that a sentential form of type α has already been recognized.

The initial state of the parser for grammar $\mathcal{G}$ is the item-set $\mathrm{init}(\mathcal{G})$ defined as

$$\mathrm{init}(\mathcal{G}) = \mathrm{closure}([\bullet\langle\mathtt{START}\rangle\ [\backslash\mathtt{EOF}] \to \langle\mathtt{Start}\rangle])$$

This state expresses that a sentence can be recognized by recognizing a string of sort $\langle\mathtt{START}\rangle$ followed by the special end of file character that indicates the end of strings.

The closure of a set of items adds all initial items to an item set for which the result symbol is predicted by one of the items in the set.

$$\mathcal{I} \subseteq \mathrm{closure}(\mathcal{I}) \tag{Cl0}$$

$$\frac{[\alpha \bullet B\beta \to A] \in \mathrm{closure}(\mathcal{I}),\ \gamma \to B \in \mathrm{P}(\mathcal{G})}{[\bullet\gamma \to B] \in \mathrm{closure}(\mathcal{I})} \tag{Cl1}$$

In the presence of priorities the closure is restricted to those items that do not cause a priority conflict.

$$\frac{\begin{array}{c}[\alpha \bullet B\beta \to A] \in \mathrm{closure}(\mathcal{I}),\ \gamma \to B \in \mathrm{P}(\mathcal{G}),\\ [\alpha[\gamma \to B]\beta \to A] \notin \mathrm{conflicts}(\mathcal{G})\end{array}}{[\bullet\gamma \to B] \in \mathrm{closure}(\mathcal{I})} \tag{Cl2}$$

For example, the item $[\bullet E + E \to E]$ is not added to the closure as a result of the item $[E + \bullet E \to E]$ if this production is left-associative, because $[E + [E + E \to E] \to E]$ is a conflict pattern

The parsing of a string starts with the parser in the initial state. Upon recognition of a symbol, either by reading a character or by completing a production, the parser can enter other states as prescribed by the transitions of the goto graph. The goto function maps an item-set to another item-set, given the symbol that has been recognized. The function 'goto' is defined by

$$\mathrm{goto}(X, \mathcal{I}) = \mathrm{closure}(\mathrm{shift}(X, \mathcal{I}))$$

i.e., create a new item-set by shifting the $\bullet$ over the symbol X and produce the closure of the resulting item-set. In normal LR(0) parsing a shift with a symbol

B creates an item-set containing all items of the previous set that have the $\bullet$ before a B symbol.

$$\frac{[\alpha \bullet B\beta \to A] \in \mathcal{I}}{[\alpha B \bullet \beta \to A] \in \mathrm{shift}(B, \mathcal{I})} \tag{Sh1}$$

We refine the definition of shift to shifting with characters and shifting with productions. Shifting an item-set with a character or character-class is defined by the rule

$$\frac{[\alpha \bullet cc'\ \beta \to A] \in \mathcal{I},\ cc \subseteq cc'}{[\alpha\ cc' \bullet \beta \to A] \in \mathrm{shift}(cc, \mathcal{I})} \tag{Sh2}$$

The character-class cc induces a shift of each item that predicts a character-class cc' that is a superset of cc.

Shifting with nonterminals is refined to shifting with complete productions. A shift is only successful if the production do not cause a priority conflict as a direct descendant at the position of the predicted symbol.

$$\frac{[\alpha \bullet B\beta \to A] \in \mathcal{I},\ [\alpha[\gamma \to B]\beta \to A] \notin \mathrm{conflicts}(\mathcal{G})}{[\alpha B \bullet \beta \to A] \in \mathrm{shift}(\gamma \to B, \mathcal{I})} \tag{Sh3}$$

For example, the production $E + E \to E$ cannot be used to shift the item $[E + \bullet E \to E]$ if this production is left-associative, because $[E + [E + E \to E] \to E]$ is a conflict. This restriction of the closure and goto functions guarantees that we can never enter a state where we have built a parse tree with a priority conflict.

5.5 Action Table

The action table declares the actions to be taken in each state. Given an item-set, the function 'actions' maps a character to the set of actions that the parser can take. If the set of actions is empty the parser has reached an erroneous state. If the set contains more than one action there is more than one way to proceed.

$$\frac{[\alpha \bullet cc\ \beta \to A] \in \mathcal{I},\ c \in cc}{\mathrm{actions}(\mathcal{I}, c) \ni \mathtt{shift}(\mathrm{goto}([c], \mathcal{I}))} \tag{Shi}$$

$$\frac{[\alpha \bullet \to A] \in \mathcal{I},\ c \in \mathrm{follow}(\alpha \to A)}{\mathrm{actions}(\mathcal{I}, c) \ni \mathtt{reduce}(\alpha \to A, |\alpha|)} \tag{Red}$$

$$\frac{[\langle \mathtt{START} \rangle \bullet [\backslash \mathtt{EOF}] \to A] \in \mathcal{I}}{\mathrm{actions}(\mathcal{I}, \backslash \mathtt{EOF}) \ni \mathtt{accept}} \tag{Acc}$$

Note that $\mathtt{shift}(\mathcal{I})$ denotes the shift *action* to state $\mathcal{I}$, whereas $\mathrm{shift}(X, \mathcal{I})$ is the application of the shift *function* defined above.

The following proposition states that the actions and goto functions defined above constitute a correct shift-reduce parser.

Proposition 5.1 (Correctness) *Given the actions and goto functions for a grammar* $\mathcal{G}$, *we have that* $(\mathrm{init}(\mathcal{G}) \bullet w) \Rightarrow^* \mathtt{accept}(t)$ *iff* $t \in \Pi(\mathcal{G})(w)$ *and* t *contains no priority conflicts according to* $\mathrm{Pr}(\mathcal{G})$ *and violates no follow restrictions in* $\mathrm{R}(\mathcal{G})$.

5.6 Remarks

The transition rules for shift-reduce parsing are non-deterministic. If more than one action is possible in some configuration more than one transition is possible. If the actions function is deterministic, at most one transition path is possible for a string. Traditional parsing techniques only accept grammars that have a deterministic action function. In §8 we will discuss an efficient implementation for non-deterministic actions functions.

The rules for parser generation above ignore reject productions, i.e., they are treated just like other productions. In §7 we will discuss how reject productions can be interpreted by means of a filter on parse forests. In §8 we will discuss how reject productions can be interpreted during parsing by means of an adaptation of the GLR algorithm. For this purpose, an item-set $\mathcal{I}$ is marked as *rejectable* if it can be reached using a reject production, i.e., $\mathcal{I}$ is rejectable, if there is an $\mathcal{I}'$ such that $\text{goto}(\alpha \to A, \mathcal{I}') = \mathcal{I}$ and $\alpha \to A$ is a reject production.

6 Automatic Lexical Disambiguation

In §4 we discussed the specification of lexical disambiguation by means of follow restrictions and reject productions. Although this is an effective way to express lexical disambiguation, it is rather tedious to write down the rules. Therefore, it would be desirable to derive lexical disambiguation rules automatically from the other grammar rules such that the grammar is disambiguated according to the longest match and prefer literals criteria. Here we discuss some possibilities. The perfect rules for longest match disambiguation have not been found yet. It is a question whether this is possible at all, since it is undecidable whether a context-free grammar is ambiguous.

6.1 Prefer Literals

The prefer literals disambiguation rule can be expressed by generating reject productions according to the following rule:

$$\frac{\texttt{"}c_1 \ldots c_n\texttt{"} \in L(\langle A\texttt{-LEX}\rangle)}{\texttt{"}c_1 \ldots c_n\texttt{"} \to \langle A\texttt{-LEX}\rangle\ \{\texttt{reject}\} \in \mathrm{P}(\mathcal{G})}$$

i.e., if the literal is a lexical phrase of sort $\langle A\texttt{-LEX}\rangle$—there is an overlap—the reject rule is added to the grammar. This implements the reserved keywords rule. The only (implementation) problem is that a parser is needed to recognize the literals as lexicals. This can be solved by first generating a parser for the grammar without reject rules and using that parser to determine overlap between literals in the grammar and lexical categories. Reject rules can then be added to the grammar accordingly and a new parser can be generated for the extended grammar.

State Explosion A problem with reject productions to exclude keywords as lexicals is that it can add many items to item-sets. For instance, if a language contains 200 keywords that match with the identifiers of the language, each item-set containing an item $[\alpha \bullet \langle\texttt{Id-LEX}\rangle\beta \to A]$ would be expanded with 200 items $[\bullet\texttt{"}c_1 \ldots c_n\texttt{"} \to \langle\texttt{Id-LEX}\rangle\ \{\texttt{reject}\}]$ and 200 items $[\bullet[c_1] \ldots [c_n] \to \texttt{"}c_1 \ldots c_n\texttt{"}]$

along with many extra transitions. To prevent this expansion, we define the rejection of literals in an indirect way, as follows:

$$\frac{\texttt{"}c_1 \ldots c_n\texttt{"} \in L(\langle A\texttt{-LEX}\rangle)}{\begin{array}{c}() \ \langle A\texttt{-LIT}\rangle \to \langle A\texttt{-LEX}\rangle \ \{\texttt{reject}\} \in \mathrm{P}(\mathcal{G}) \\ \texttt{"}c_1 \ldots c_n\texttt{"} \to \langle A\texttt{-LIT}\rangle \in \mathrm{P}(\mathcal{G})\end{array}}$$

where the symbol () denotes the empty phrase, i.e., there is a production $\to ()$. The sort $\langle A\texttt{-LIT}\rangle$ is used to collect all literals to be rejected from $\langle A\texttt{-LEX}\rangle$. The production $() \ \langle A\texttt{-LIT}\rangle \to \langle A\texttt{-CF}\rangle \ \{\texttt{reject}\}$ defines the rejection for all literals at once. The effect of the empty symbol () in the second production is that only the item $[\bullet() \ \langle A\texttt{-LIT}\rangle \to \langle A\texttt{-LEX}\rangle]$ is added when $\langle A\texttt{-LEX}\rangle$ is predicted. This will cause a reduction with the production $\to ()$ to an item-set where $\langle A\texttt{-LIT}\rangle$ is predicted. This item-set is only computed once and is reused for all other item-sets that predict $\langle A\texttt{-LEX}\rangle$. It is the initial state of a finite automaton for the matching of literals.

As an example, consider how the prefer literals rule for our functional language example is expressed using this modified rule:

```
syntax
  () <Var-LIT> -> <Var-LEX> {reject}
  "let"        -> <Var-LIT>
  "in"         -> <Var-LIT>
```

Local Exclusion An alternative for the expression of the prefer literals rule is the rule

$$\frac{\{[\bullet\langle A\texttt{-LEX}\rangle \to \langle A\texttt{-CF}\rangle],\ [\bullet[c_1]\ldots[c_n] \to \texttt{"}c_1 \ldots c_n\texttt{"}]\} \subseteq \mathrm{closure}(\mathcal{I})}{[\bullet\texttt{"}c_1 \ldots c_n\texttt{"} \to \langle A\texttt{-CF}\rangle \ \{\texttt{reject}\}] \in \mathrm{closure}(\mathcal{I})}$$

that locally forbids predicted literals as lexicals by extending the parser generator. This does not implement the reserved keywords rule in the sense of forbidding the use of a keyword as a lexical in all positions. Only when a lexical and a literal can appear in the same place, the literal is preferred. Therefore, it might still lead to ambiguities.

6.2 Longest Match

It is less straightforward to find a general rule to express 'longest match' using follow restrictions. An attempt is the rule

$$\frac{\langle B\texttt{-LEX}\rangle \in \mathrm{follow_s}(\langle A\texttt{-LEX}\rangle \to \langle A\texttt{-CF}\rangle)}{\langle A\texttt{-LEX}\rangle \not\to \mathrm{first}(\langle B\texttt{-LEX}\rangle) \cap \mathrm{last}(\langle A\texttt{-LEX}\rangle) \in \mathrm{R}(\mathcal{G})} \qquad \text{(FR)}$$

This restricts the follow set of $\langle A\texttt{-LEX}\rangle$ by excluding the elements of the first set of $\langle B\texttt{-LEX}\rangle$ that can also be used at the end of $\langle A\texttt{-LEX}\rangle$ for those $\langle B\texttt{-LEX}\rangle$s that can follow the injection $\langle A\texttt{-LEX}\rangle \to \langle A\texttt{-CF}\rangle$. Here $\mathrm{follow_s}$ is the extension of the follow function to produce all symbols that can follow a production.

This rule is adequate in many cases. Consider for instance the functional expression grammar. The follow restriction for `Var` in §4.2.2 is derived exactly using this rule. However, the rule is not general enough. One counter example is the following grammar of expressions with single character variables and implicit multiplication operator. This describes mathematical expressions such as xy that denotes the multiplication of x and y.

```
lexical syntax
  [a-z]     -> Var
  [\ \t\n] -> LAYOUT
context-free syntax
  Var       -> Exp
  Exp Exp  -> Exp {left}
```

Rule (FR) would forbid `xy` as an expression forcing the use of whitespace, i.e., `x y`. Although this example shows that rule (FR) is unsound if considered as an analytic rule, one could also consider it as a normative rule forcing a clearer style of language definition.

Rule (FR) generates follow restrictions for lexicals. We also need restrictions for literals overlapping with lexicals. For instance, the restrictions

```
lexical restrictions
  "let" "in" -/- [a-z]
```

forbids the interpretation of `letter` as the literal `let` and the variable `ter`. The following rule adds restrictions to prevent this overlap.

$$\frac{\langle A\text{-}\mathtt{LEX}\rangle \in \mathrm{follow}(\texttt{"}c_1 \ldots c_n\texttt{"}), \\ c \in \mathrm{first}(\langle A\text{-}\mathtt{LEX}\rangle),\ \texttt{"}c_1 \ldots c_n c\texttt{"} \in L(\langle B\text{-}\mathtt{LEX}\rangle)}{\texttt{"}c_1 \ldots c_n\texttt{"} \not\to [c] \in \mathrm{R}(\mathcal{G})}$$

If the literal $L = \texttt{"}c_1 \ldots c_n\texttt{"}$ followed by some character c from the first set of a lexical $\langle A\text{-}\mathtt{LEX}\rangle$ that is a member of the follow set of L can form a lexical $\langle B\text{-}\mathtt{LEX}\rangle$, there is a longer match than the literal L. Therefore, c is restricted from the follow set of the literal.

This rule is stronger than the longest match filter we formulated before. It can forbid sentences that have a single unambiguous interpretation. For instance, consider the string `let x = 1 int`. Here `int` is forced to be read as a variable and not as the juxtaposition of the literal `in` and the variable `t`.

It is clear that these rules are not the final word about fully automatic lexical disambiguation. Further research is needed to decide what is sufficient.

7 Reject Productions

In §4 we introduced reject productions to express 'prefer literals' lexical disambiguation. The parser generator discussed in §5 treats reject productions as normal productions. This will cause ambiguous parses for those cases where a normal production and a reject production overlap. In this section we first define the semantics of context-free grammars with reject productions, then we investigate several properties of such grammars including an interpretation of rejects to solve such ambiguities. [The author thanks Jan van Eijck and Annius Groenink for the email discussion that led to the results in this section.]

7.1 Semantics

The semantics of reject productions is obtained by refining the inductive definition of parse trees from §4. The inductive rule (Prod) is restricted to exclude the construction of parse trees that have a yield that could be obtained via a reject production.

A context-free grammar $\mathcal{G}$ with reject productions generates a family of sets of *parse trees* $\mathcal{T}_{\mathrm{r}}(\mathcal{G}) = (\mathcal{T}_{\mathrm{r}}(\mathcal{G})(X) \mid X \in \mathrm{Syms}(\mathcal{G}))$, which contains the minimal sets $\mathcal{T}_{\mathrm{r}}(\mathcal{G})(X)$ such that

$$\frac{c \in cc}{c \in \mathcal{T}_{\mathrm{r}}(\mathcal{G})(cc)} \tag{CharR}$$

$$\frac{\begin{array}{c}A_1 \ldots A_n \to A \in \mathrm{P}(\mathcal{G}),\ t_1 \in \mathcal{T}_{\mathrm{r}}(\mathcal{G})(A_1),\ \ldots\ ,\ t_n \in \mathcal{T}_{\mathrm{r}}(\mathcal{G})(A_n),\\ \neg\exists\beta \to A\ \{\texttt{reject}\} \in \mathrm{P}(\mathcal{G}), t_\beta \in \mathcal{T}_{\mathrm{r}}(\mathcal{G})(\beta) : \mathrm{yield}(t_\beta) = \mathrm{yield}(t_1 \ldots t_n)\end{array}}{[t_1 \ldots t_n \to A] \in \mathcal{T}_{\mathrm{r}}(\mathcal{G})(A)} \tag{ProdR}$$

The second condition of (ProdR) excludes from $\mathcal{T}_{\mathrm{r}}(\mathcal{G})(A)$ those trees for which an A tree with the same yield could be built using a reject production at its root. This second condition is the only difference with the definition of $\mathcal{T}(\mathcal{G})$, i.e., we have $\mathcal{T}_{\mathrm{r}}(\mathcal{G}) \subseteq \mathcal{T}(\mathcal{G})$. Note that only trees in $\mathcal{T}_{\mathrm{r}}(\mathcal{G})(\beta)$ are excluded. That is, if there are nested reject productions such that some tree in $\mathcal{T}(\mathcal{G})(\beta)$ is rejected and thus not part of $\mathcal{T}_{\mathrm{r}}(\mathcal{G})(\beta)$, then it is not used to exclude trees using $\beta \to A\ \{\texttt{reject}\}$.

Unfortunately, this definition is inconsistent for grammars with a cycle containing a reject production. For instance, consider the grammar

```
syntax
  [a] -> A
  A   -> B
  B   -> A {reject}
```

and consider whether the string a is a member of the language of this grammar: if $[a \to A] \in \mathcal{T}_{\mathrm{r}}(A)$, then $[[a \to A] \to B] \in \mathcal{T}_{\mathrm{r}}(B)$ and hence $[a \to A] \notin \mathcal{T}_{\mathrm{r}}(A)$. Conversely, if $[a \to A] \notin \mathcal{T}_{\mathrm{r}}(A)$, then $[[a \to A] \to B] \notin \mathcal{T}_{\mathrm{r}}(B)$ and hence $[a \to A] \in \mathcal{T}_{\mathrm{r}}(A)$. For this reason, we restrict the class of grammars that we want to consider to grammars that do not contain a cycle (disregarding the rejects) for which one of the transitions is via a reject production.

7.2 Expressive Power

In the rest of this section we explore some of the properties of reject productions.

$a^n b^n c^n$ First of all context-free grammars with reject productions can be used to describe some non-context-free languages. Consider for example, the language $a^n b^n c^n$ with $n \geq 0$, which is a standard example of a non-context-free language. The following grammar, due to Van Eijck (1997), defines this language using reject productions. The first four productions define the language $a^* b^* c^*$. The next four productions define the sorts D and E denoting, respectively, $a^n b^n$ and $b^n c^n$. The last four productions exclude from sort S all strings for which one of the pairs $x^n y^m$ have unequal numbers of xs and ys.

```
A* B* C* -> S             -> D        D  B+ C* -> S {reject}
"a"      -> A       A D B -> D        A+ D  C* -> S {reject}
"b"      -> B             -> E        A* B+ E  -> S {reject}
"c"      -> C       B E C -> E        A* E  C+ -> S {reject}
```

Difference Given a context-free grammar defining sorts `A` and `B` we can define the difference of the languages of these sorts by adding the following productions.

```
A  -> AminB
B  -> AminB {reject}
```

The first adds all `A` trees to `AminB`, the second excludes from this all `A` trees that match with a `B` tree.

Intersection Extending this result, we can express the intersection between sorts `A` and `B` by adding two new sorts `AminB` and `AandB` and by adding the following productions:

```
A -> AminB                  A     -> AandB
B -> AminB {reject}         AminB -> AandB {reject}
```

This defines `AminB` as the difference `A`−`B`, and `AandB` as the difference `A`−(`A`−`B`), i.e., the intersection of `A` and `B`.

We can generalize the results above. Given two context-free languages, we can express the difference and intersection of those languages using context-free grammars with reject productions. Take the union of the context-free grammars for the two languages, after renaming symbols to prevent interference. Then add productions for the sorts to be intersected as explained above.

Weak Complement If we are only interested in the strings that can be generated from a grammar (and not in their structure), the complement of a the language generated by sort `A` is defined by extending a grammar with the following rules:

```
~[]* -> NotA
A    -> NotA {reject}
```

The first production defines the complement of `A` as a string of arbitrary characters. The complement `~[]` of the empty character class is the character class with all characters. The second production excludes from this language all strings in the language of `A`. Using this complement we can of course also express the weak intersection of two sorts.

Decidable We have seen that context-free grammars with reject productions are very expressive. It is now appropriate to ask whether it is even decidable whether a string is in the language of such a grammar. The following theorem states that this is indeed the case. The proof uses the notion of a parse forest that will be discussed in the next section. For the proof of the theorem we need the following proposition about generalized-LR parsers.

Proposition 7.1 *Let $\mathcal{G}$ be a context-free grammar. If $t_1, t_2 \in \mathcal{T}(\mathcal{G})(A)$ and* $\mathrm{yield}(t_1) = \mathrm{yield}(t_2) = w$*, then a GLR parse of w will result in an ambiguity node with t_1 and t_2 as possibilities.*

Theorem 7.2 *The parsing problem for context-free grammars with reject productions (without rejects in cycles) is decidable.*

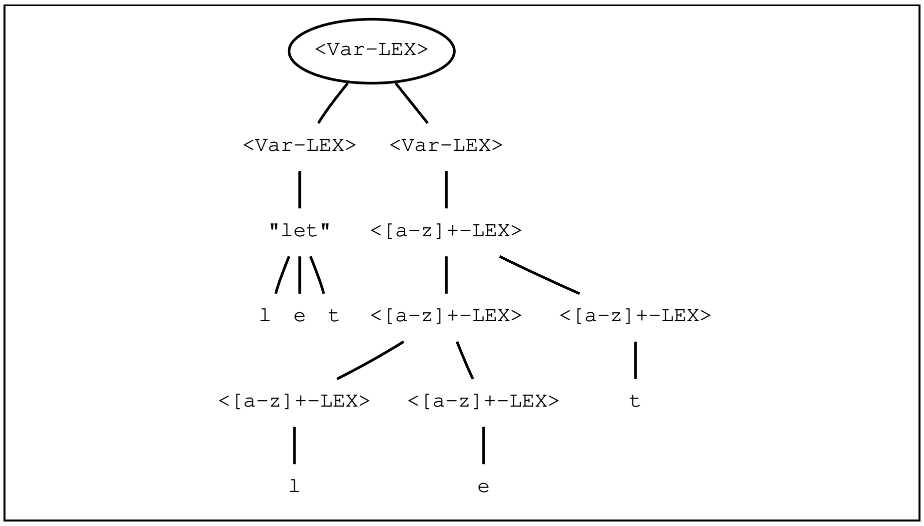


Figure 7: Ambiguity node caused by overlap between syntax for `<Var-LEX>` and reject production `"let" -> <Var-LEX>`

Proof. Given a context-free grammar $\mathcal{G}$ with reject productions (without rejects in cycles), construct a generalized-LR parser for $\mathcal{G}$ ignoring the reject annotations. The result is a parser for a possibly ambiguous context-free grammar. Now, given a string, parse it with this parser. If parsing fails, the string is also not in the language of the grammar with reject productions. Otherwise, the result of parsing is a parse forest. Since cycles do not contain rejects, these can be removed from the forest.

Now, if a tree $t = [t_1 \ldots t_n \to A]$ should be rejected according to the second condition of rule (ProdR), there is a reject production $\beta \to A$ and trees t_β such that $\mathrm{yield}(t_\beta) = \mathrm{yield}(t_1 \ldots t_n)$. But then, $\mathrm{yield}([t_\beta \to A]) = \mathrm{yield}([t_1 \ldots t_n \to A])$ and hence, according to the proposition above, the parse forest contains an ambiguity node on top of t also containing $[t_\beta \to A]$ as possibility.

Reject productions are now interpreted by traversing the forest, in a bottom-up manner, marking tree nodes according to the following rules: (1) Leafs are not marked. (2) A reduction node is marked if any of its direct descendants is marked. (3) An ambiguity node is marked if either all its direct descendants are marked, or if it contains an unmarked tree with as root label a reject production. Since the parse forest is finite, this procedure terminates.

If the root of the parse forest is marked after this procedure, the string is not accepted by the grammar, otherwise it is accepted and the forest without marked nodes represents all parse trees for the string. □

The tree in Figure 7 illustrates the proof. The overlap between the literal `"let"` and the syntax for variables causes an ambiguity. The ambiguity node is marked because `"let" -> <Var-LEX>` is a reject production. Therefore, the interpretation of `let` as a variable is dismissed.

This shows that we can construct a complete implementation of parsers for grammars with reject productions. In the next section we will discuss how reject productions can be interpreted *during* parsing to influence parse decisions to prevent trees with rejected subtrees from being built at all.

Expressive Power From the above we can conclude that context-free grammars with reject productions are stronger than pure context-free grammars, but have a decidable parsing problem. This gives a lower bound and upper bound for the expressive power of the formalism, but it is an open question what class of languages is described by context-free grammars with reject productions.

Regular Rejects We introduced reject productions in order to express the prefer literals rule. This means that only a regular language is excluded from a context-free one. This gives us the guarantee that the resulting language is still context-free. We could exploit this property and restrict the formalism to such regular reject productions and implement these by means of a grammar transformation. However, such a grammar transformation would probably yield large grammars. Furthermore, our implementation gives a general way to express the prefer literals rule and it allows the expression of other interesting grammars that have not been in the reach of declarative specification. This feature can give rise to as yet unforeseen applications.

8 Generalized-LR Parsing

In §5 we have defined the generation of shift-reduce parsers from context-free grammars with priority declarations and follow restrictions. If the actions function derived from a grammar is deterministic then the shift-reduce parser is also deterministic and can be implemented in a standard way.

However, since we do not restrict the class of grammars, it is not guaranteed that the actions function is deterministic. This can have two causes: (1) The lookahead needed for the grammar is more than provided by the parser generator. (2) The grammar is ambiguous. In the case of scannerless parsing we will frequently see grammars for which unbounded lookahead is needed. This entails that no variant of the LR parser generation algorithms will produce a deterministic actions function. Therefore, we need a non-deterministic implementation of the shift-reduce parsing algorithm. When a configuration is reached where more than one action is possible, all possibilities should be tried. In case of unbounded lookahead only one of the possible transitions leads to an accepting configuration. In case of an ambiguous string, multiple accepting configurations will be reached giving all possible parse trees for the string.

The advantage of such a non-deterministic approach is, first of all, the unbounded lookahead that it provides. Furthermore, a parser producing all parse trees for an ambiguous string can be used as a front-end for a disambiguation filter that selects the correct tree according to some disambiguation method. Finally, it is undecidable whether a grammar is ambiguous or has lookahead problems. Having a parser that yields all possible parses can help in detecting the ambiguities and resolve them in a much easier way than by inspecting conflicts in a parse table.

A naive way to implement such a non-deterministic parsing algorithm is to copy the entire configuration at each point where two or more actions are possible and to continue parsing with each those configurations. This will not be very efficient because of the memory requirements and because it will not reuse parses for substrings that are the same in two forked off configurations. Generalized-LR parsing is an efficient implementation of non-deterministic shift-reduce parsing. A GLR parser deals with conflicts in the parse table by splitting the parser into as many parsers as there are conflicts. If the conflict was due to a lack of lookahead, some of the parsers will not succeed in parsing the sentence and will die. If several parsers succeed in parsing, the grammar was ambiguous. In that case parse trees for all possible parses are built.

Generalized-LR parsing was developed for natural language processing by Tomita (1985). It is a specialization of the more general framework of Lang (1974) (later also described in Billot and Lang (1989)) for creating generalized parsers. The algorithm was improved by Rekers (1992) and applied to parsing of programming languages. The feasibility of GLR parsing for parsing of programming languages has been shown by the experience with GLR in the ASF+SDF Meta-Environment (Klint, 1993). More experience with GLR parsing of programming languages using an adaptation of Reker's algorithm is reported by Wagner and Graham (1997).

Besides the non-determinism in the parse table, we also need to interpret the reject productions in the grammar. In the previous section we showed how reject productions can be interpreted as a disambiguation filter after parsing. But we would rather interpret them earlier. In this section we explain GLR parsing and present an adaptation of the algorithm to interpret reject productions during parsing.

8.1 Parse Forest

A generalized parser deals with ambiguous grammars by producing all possible parse trees for an ambiguous string. In GLR parsing the possible parse trees are represented by means of a parse forest. This is a compact representation of a set of parse trees. A parse tree is constructed using application and ambiguity nodes. An application node represents the application of a production to a list of subtrees. An ambiguity node represents a set of possible parse trees for a (sub)string. By packing all trees for a substring into an ambiguity node, these parses can be shared in all trees for strings containing the substring.

For example, consider the following grammar of simple expressions with ambiguous addition and multiplication operator.

```
sorts Exp
syntax
  [a-z]         -> Exp
  Exp "+" Exp -> Exp
  Exp "*" Exp -> Exp
  Exp           -> <START>
```

To keep the example small, layout is not allowed between the tokens. The parse forest for the ambiguous string `a+b*c` is shown in Figure 8. The ellipse represents an ambiguity node. Observe that various subtrees are shared in the forest.

8.2 Graph Structured Stack

A GLR parser deals with conflicts in the parse table by maintaining a number of stacks in parallel. Each time a parse stack leads to n conflicting actions, n new stacks are created that continue the parse with those actions. These stacks are not copies of the old stack. The new top nodes have pointers to the old stack. If

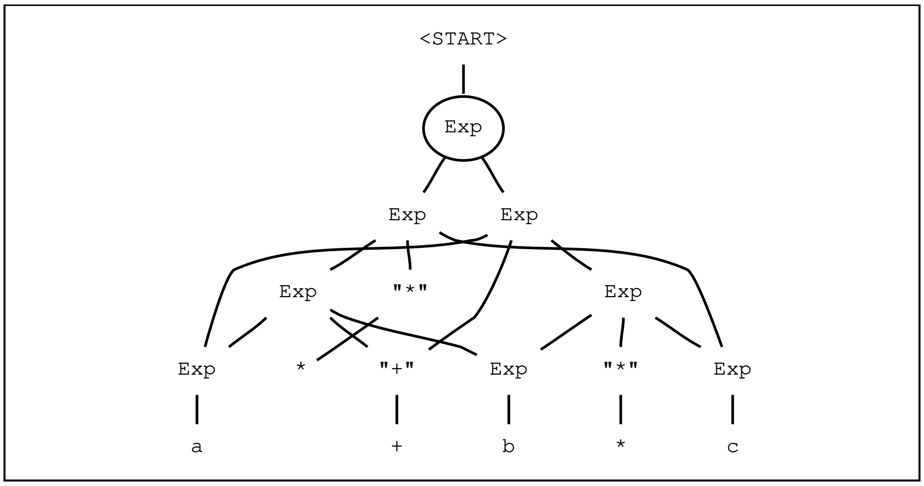


Figure 8: Parse forest with sharing for ambiguous string `a+b*c`.

in a later stage two stacks get into the same state, the stacks are merged again. In this manner a graph structured stack is built in which parses for ambiguous substrings are shared.

A graph structured stack node consists of a state number and a list of links. Each link contains a reference to a node in the parse forest and a reference to the previous stack.

As an example of the working of a GLR parser, consider the sequence of stack configurations during parsing the string `a+b*c` in Figure 9. This is the parse that created the parse forest in Figure 8. The figure shows the stacks during each cycle of the parsing algorithm. After shifting a character, all possible reductions are performed and then the next character is shifted. The trees pointed to by the stack links are abbreviated by their yield using square brackets to show the structure. The symbol after the colon denotes the main type of the tree at the link. We consider the configurations one by one.

(a) The initial stack with state 0 is created. The character `a` is shifted.

(+) The character `a` reduces to an expression using the production `[a-z] -> Exp`. The symbol `+` is shifted.

(b) The character `+` reduces to the literal `"+"`. The character `b` is shifted.

(*) The character `b` reduces to an expression. The sequence `[a+b]` reduces to an expression, resulting in a link from state 1 to state 0. From states 1 and 12 a shift can be performed with the next character `*`. Because both shifts lead to a stack with state 9, a single stack is created that has links to the two stacks.

(c) The character `*` reduces to the literal `"*"`. The character `c` is shifted.

(\EOF) The character `c` reduces to an expression. Now there are two possible reductions from the stack with state 13. First reduce `[b*c]` and then reduce `[a+[b*c]]`, or reduce `[[a+b]*c]`. Both reductions result in the creation of a stack with state 1 with a link to the initial stack. These

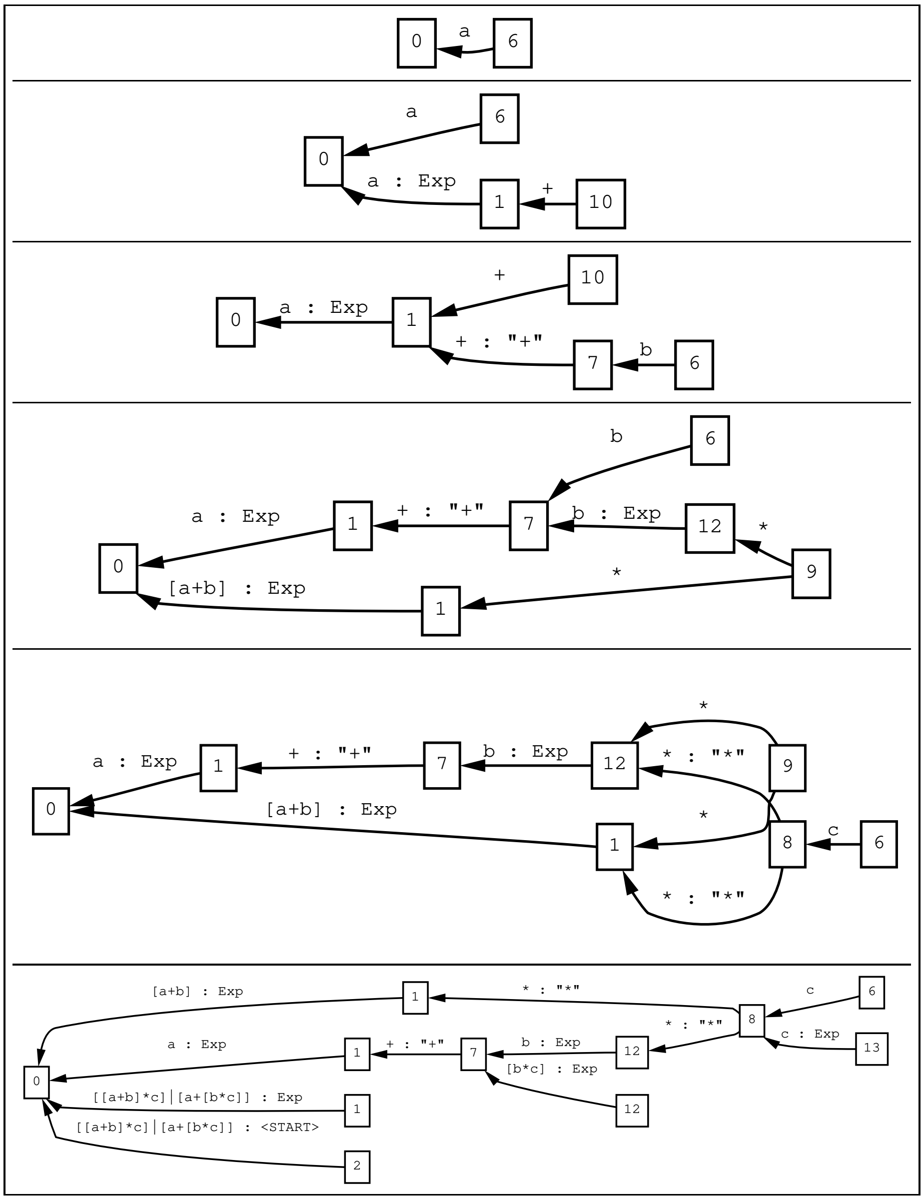


Figure 9: Parse configurations for the parse of string a+b*c.

stacks are shared and an ambiguity node is created that represents the two possible parse trees. At this point the entire string has been read and the next symbol is `\EOF`. Therefore, the expression is reduced to ⟨`START`⟩ and the string is accepted. The stack with state 2 is the accepting stack and the tree pointed to by its link is the parse tree for the entire string.

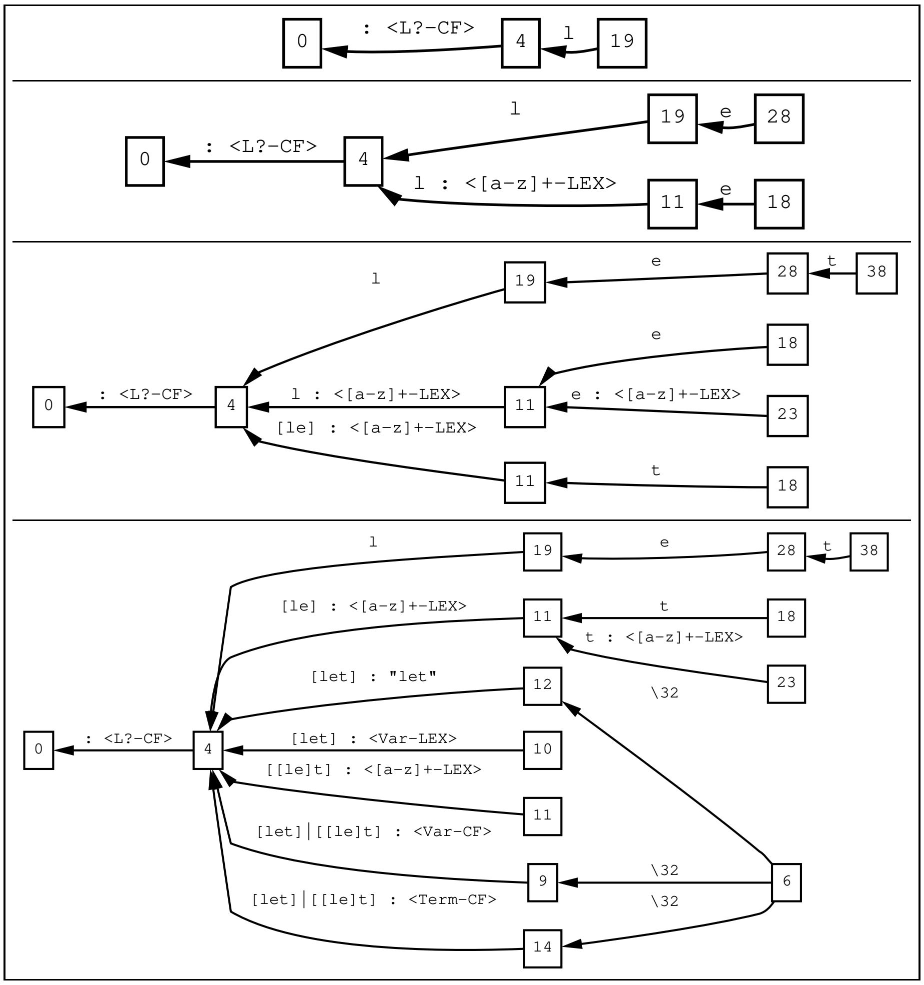


Figure 10: Parse of `let␣x...` when reject is ignored.

8.3 Reject Reductions

In §4 disambiguation with reject productions was introduced in order to express the prefer literals rule. In §7 we outlined a procedure for interpreting reject productions after parsing by pruning the parse forest. We would rather interpret reject productions during parsing to prevent trees containing reject productions from being built.

To understand how this can be achieved, recall the parse forest in Figure 7 that shows the ambiguity that is created when parsing the substring `let` in the functional expression grammar defined in §4.1. It can be interpreted using the lexical productions for variables or using the reject production `"let" -> <Var-LEX>` {`reject`}. Pruning this forest causes the ambiguity node to be eliminated from the parse forest, thereby rejecting the reading of `let` as a variable.

The parse configurations for this parse in Figure 10 show how this ambiguity is created. In the first three configurations the letters `l`, `e` and `t` are read.

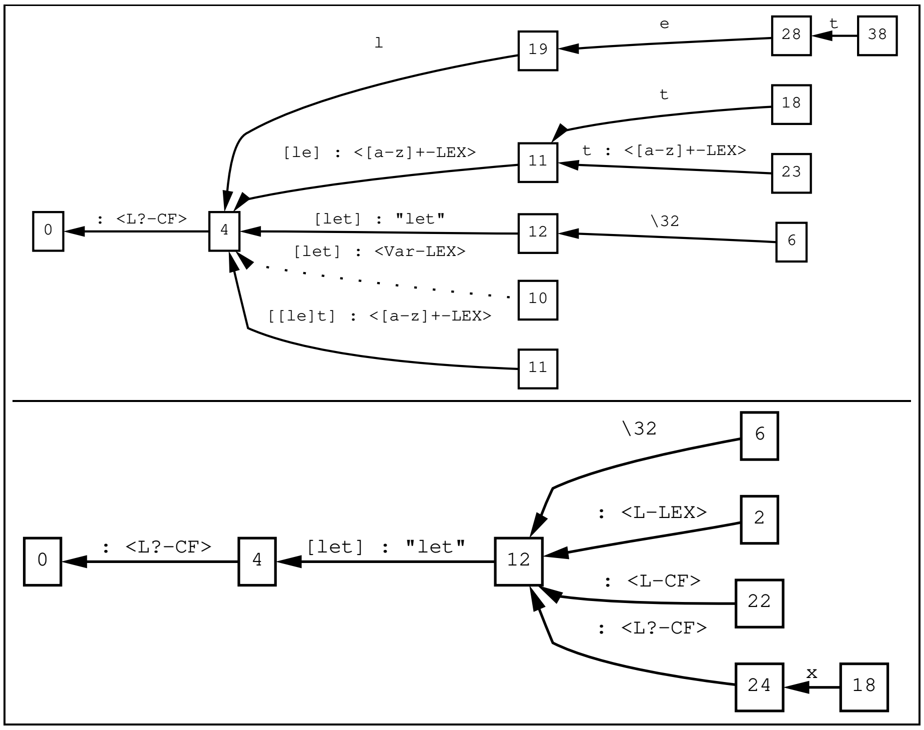


Figure 11: Parse of `let␣x...` with reject production that forbids `let` as a variable.

The fourth configuration is the interesting one. There are three parses for the substring `let`: as a variable constructed with `<[a-z]+-LEX>`, as the literal `"let"` of the reject production `"let" -> <Var-LEX>`, and as the literal `"let"` as part of the let construct. The reduction of the literal results in a stack with state 12. The reduction of the lexical and the reject rule lead to a merged stack with state 10 from where another reduction first leads to a stack with state 9 and then leads to a term and a stack with state 14. From the states 9, 12 and 14 parsing continues with a shift of the space character ␣ (32) to state 6.

The idea for the implementation of the reject rule is to forbid further actions with a state that has been reached using a reject reduction. The link that is created when reducing with a reject production is marked as rejected. If all links of a stack are marked as rejected all shifts and reductions from that state are forbidden.

In the last configuration of Figure 10 this would entail that the link from the stack with state 10 to the stack with state 4 is rejected. Therefore, the reduction to state 14 and the shift to state 6 would be forbidden. This is exactly what happens in the parse shown in Figure 11. The dotted link is rejected and no actions are performed from its stack. The parse of `let` as variable is preempted. In the next configuration parsing continues only with the stack with state 12, corresponding to a parse of `let` as a literal.

8.4 The Algorithm

Below the complete SGLR algorithm is presented. The differences with the GLR algorithm of Rekers (1992) are the use of productions in the goto function and the handling of reject reductions. Furthermore, the parser does not make use of a scanner, but reads characters from a file or string. This could of course be a stream of token codes and does not make a difference to the algorithm. As we discussed in the §5 character classes are handled in the parse table and are thus transparent to the parser.

Algorithm 8.1 (SGLR) Given the parse table *table* for some grammar, parse the string of characters in *file*. If the string is a sentence in the language described by the grammar, return the parse forest for the string, and an error message otherwise.

Parse The function parse reads the characters from a file and returns a parse tree if the text is syntactically correct, an error message otherwise. The list of active stacks is initialized to contain a single stack with the initial state of the parse table as its state. For each character in the input, the parser handles all actions for each active stack. The shifts for each stack are stored and performed by the shifter after all possible reductions have been performed. When all characters have been read or when no more stacks are alive, parsing terminates. If parsing succeeded, the accepting stack has a direct link to the initial state. This link has a reference to the parse forest with all possible parse trees for the entire string. If parsing failed an error term is returned.

```
PARSE(table, file)
  global accepting-stack := ∅
  global active-stacks := {new stack with state init(table)}
  do
    global current-token := get-next-char(file)
    PARSE-CHARACTER()
    SHIFTER()
  while current-token ≠ \EOF ∧ active-stacks ≠ ∅
  if accepting-stack contains a link to the initial stack with tree t then
    return t
  else
    return parse-error
```

Parse Character The list of active stacks is moved to the list of stacks of the actor that performs the actions for a stack unless the stack is rejected. The list of stacks for the actor is extended when reductions are performed. If actions for newly added stacks are performed before all links to it have been created, a stack that becomes rejected might escape. Therefore, new stacks are added to *for-actor-delayed* if they are rejectable and are only considered when all stacks on *for-actor* are exhausted. Then stacks are taken from the delayed list in order of priority. The operation 'pop' removes the stack with the highest priority from a list of stacks.

```
PARSE-CHARACTER()
```

global *for-actor* := *active-stacks*
global *for-actor-delayed* := $\emptyset$
global *for-shifter* := $\emptyset$
while *for-actor* $\neq \emptyset \wedge$ *for-actor-delayed* $\neq \emptyset$ **do**
  if *for-actor* $= \emptyset$ **then**
    for-actor := {pop(*for-actor-delayed*)}
  for each stack $st \in$ *for-actor* **do**
    if $\neg$ all links of stack st rejected **then**
      ACTOR(st)

Actor Handle the actions for stack st and the current token. A reduce action is immediately handled. Shift actions are saved on *for-shifter* for handling if after all reductions have been performed. An accept action results in saving the current stack as the accepting stack. An error action is ignored because the current stack can be a wrong attempt while other stacks are still alive. The entire parse fails if all stacks lead to error actions. This will become apparent after shifting because no more active stacks will be alive.

ACTOR(st)
  for each action $a \in$ actions(s, *current-token*) **do**
    case a **of**
      `shift`(s) $\Rightarrow$ *for-shifter* := $\{\langle st, s\rangle\} \cup$ *for-shifter*
      `reduce`($\alpha \to A$) $\Rightarrow$ DO-REDUCTIONS(st, $\alpha \to A$)
      `accept` $\Rightarrow$ *accepting-stack* := st

Reductions Function *do-reductions* performs a reduction for stack st with production $\alpha \to A$. For each path of length $|\alpha|$ following the links from st to some stack st_0 the trees along the path are collected and the reducer is called to handle the reduction.

DO-REDUCTIONS(st, $\alpha \to A$)
  for each path from stack st to stack st_0 of length $|\alpha|$ **do**
    kids := the trees of the links which form the path from st to st_0
    REDUCER(st_0, goto(state(st_0), $\alpha \to A$), $\alpha \to A$, *kids*)

Reducer Given a stack st, a state s, a production $\alpha \to A$ and a list of trees *kids*, the reducer creates the application node for the production and the list of direct descendants *kids* and creates a new stack with state s and a link to stack st. However, because there might already exist as stack with state s, the list of active stacks is searched. If there is no such stack a new stack is created (else branch) and added to the list of active stacks and the list of stacks for the actor. The new stack has state s and a link with a pointer to the newly created tree. If a stack with state s already exists and there is a direct link nl from st_1 to st_0, an ambiguity has been found. The tree t is added to the ambiguity node of the link. If there is no direct link, a new link is created from st_1 to st_0 with t as parse tree. Because this new link entails that new reductions from already inspected stacks might be possible, all active stacks are reconsidered. In all cases, the link that is created or extended is marked as rejected if the production is a reject production.

REDUCER(st, s, $\alpha \to A$, *kids*)

t := application of $\alpha \to A$ to *kids*
if $\exists st_1 \in$ *active-stacks* : state(st_1) = s
  if $\exists$ a direct link *nl* from st_1 to st_0 **then**
    add t to the possibilities of the ambiguity node at tree(*nl*)
    if $\alpha \to A$ is a reject production **then** mark link *nl* as rejected
  else
    add a link *nl* from st_1 to st_0 with tree t
    if $\alpha \to A$ is a reject production **then** mark link *nl* as rejected
    for each $st_2 \in$ *active-stacks*
      such that $\neg$ all links of st_2 rejected
        $\wedge$ $st_2 \notin$ *for-actor* $\wedge$ $st_2 \notin$ *for-actor-delayed* **do**
      for each reduce($\alpha \to A$) $\in$ actions(state(st_2), *current-token*)
      do
        DO-LIMITED-REDUCTIONS(st_2, $\alpha \to A$, *nl*)
else
  st_1 := new stack with state s
  add a link *nl* from st_1 to st_0 with tree t
  active-stacks := $\{st_1\} \cup$ *active-stacks*
  if rejectable(state(st_1)) **then**
    for-actor-delayed := push(st_1, *for-actor-delayed*)
  else
    for-actor := $\{st_1\} \cup$ *for-actor-delayed*
  if $\alpha \to A$ is a reject production **then** mark link *nl* as rejected

Limited Reductions The function *do-reductions* is used to do all reductions for some state and production that involve a path going trough link *nl*.

DO-LIMITED-REDUCTIONS(st, $\alpha \to A$, l)
  for each path from stack st to stack st_0 of length $|\alpha|$ going through link l
  do
    kids := the trees of the links that form the path from st to st_0
    REDUCER(st_0, goto(state(st_0), $\alpha \to A$), $\alpha \to A$, *kids*)

Shifter After all possible reductions have been performed, *for-shifter* contains a list of stacks that can do a shift. Only these stack make it into the next cycle of the parse. The list of active stacks is reinitialized to the empty list. For each stack st_0 in *for-shifter* a new stack is created with a link to st_0 with as tree the current token. That is, if a stack with state s was already created only a link from that stack to st_0 is created.

SHIFTER()
  active-stacks := $\emptyset$
  t := *current-token*
  for each $\langle s, st_0 \rangle \in$ *for-shifter* **do**
    if $\exists st_1 \in$ *active-stacks*: state(st_1) = s **then**
      add a link from st_1 to st_0 with tree t
    else
      st_1 := new stack with state s
      add a link from st_1 to st_0 with tree t
      active-stacks := $\{st_1\} \cup$ *active-stacks*
end

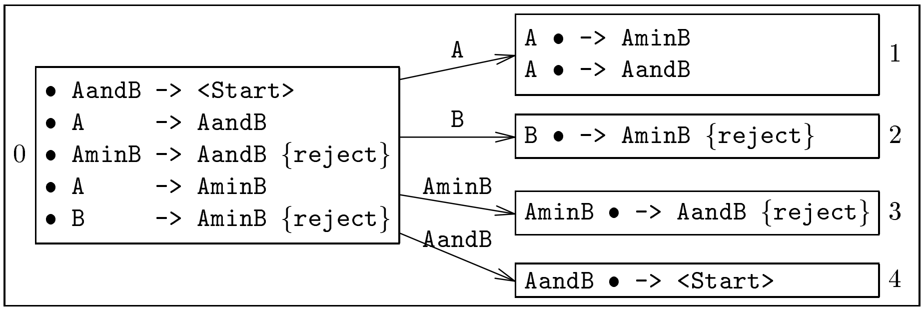


Figure 12: Goto graph for grammar with nested reject productions.

8.5 Remarks

The algorithm above does not actually mark stacks as rejected, but the link from a stack that is created with a reject production. Further action on a stack is forbidden if all links from that stacks are rejected. This is done because, in principle, there could be situations where two links are created from the same stack that are not merged (as is the case when the links are to the same stack) and only one is rejected. It is not clear whether such a situation can occur. But there is no proof of the contrary either.

The ordering on states that is assumed in the priority pop operation used in procedure PARSE-CHARACTER() is needed to ensure that nested reject productions are treated properly. For example, consider again a grammar extended with productions expressing the intersection of sorts `A` and `B`.

```
A -> AminB           A     -> AandB
B -> AminB {reject}  AminB -> AandB {reject}  AandB -> <Start>
```

This gives rise to the goto graph in Figure 12. States 3 and 4 are rejectable because they can be reached with a reject production. When parsing a string that is in `A` and in `B`, state 2 is reached using the reduction for `B`. The next reduce action with the reject production `B -> AminB {reject}` leads a stack with state 3, which is rejected. No further action is taken from that stack. The reduction of `A -> AandB` leads to a stack with state 4 and then, correctly, to acceptance of the string.

Now consider the case where a string is in `A`, but not in `B`. Then there is no reduction to state 2 and hence state 3 is not rejected, but there is a reduction to state 3 using `A -> AminB` and a reduction to state 4 using `A -> AandB`. Now there are two rejectable stacks on the *for-actor-delayed* list. If the stack with state 3 is released first a reduction with `AminB -> AandB {reject}` occurs and the stack with state 4, which is still on *for-actor-delayed*, is rejected; and parsing fails as it should. However, if state 4 is released first, parsing succeeds before the stack with state 4 is rejected. It is clear that in this case state 3 has higher priority than state 4.

It is not clear how the ordering on states should be determined in general. It would seem that a state s_1 with productions that are reachable from the productions in a state s_2 has higher priority. This is only a guess, however, and should be worked out more carefully. For single, i.e., non-nested reject productions the ordering plays no role. Therefore, the implementation of exclusion by means of reject productions, of which prefer literals is a special case, is not dependent on finding an ordering on states.

9 Implementation

In the previous sections we have presented an approach to scannerless parsing. These techniques are implemented as part of the SDF2 tools. The tools have been used to construct parsers for a number of languages including SDF2 itself. Although no detailed data on the performance of the implementation are available at the time of this writing, a couple of preliminary observations can be made nonetheless.

Grammar Normalizer The syntax definition formalism SDF2 is completely specified in ASF+SDF. Part of the definition is the grammar normalizer discussed in §3. This specification has been compiled to an executable term rewriter, which has a reasonable performance. The literate specification of SDF2 and the normalization of syntax definitions is presented in Visser (1997c). The specification also defines the format of parse trees encoded in the ATerm format of Van den Brand *et al.* (1997).

Parser Generator The parser generator described in §5 has been completely specified in ASF+SDF. The compiled specification of the parser generator is too inefficient. It is probably necessary to implement this component in an imperative language that allows direct access instead of lookup in lists.

There are several factors that make parser generation more difficult compared to normal SLR(1) parser generation for context-free grammars. There are more item-sets because of the productions for the lexical syntax. Extra productions are added because of the reject productions expressing the prefer literals rule, this increases the number of items in item-sets. The goto table contains a transition for each production instead of a transition for each non-terminal. The last factor can be reduced by sharing transitions to the same state.

Productions and item-sets are encoded by numbers. Character classes are important for reducing the size of the parse table. A set of actions that is shared by several characters is stored efficiently by means of a character class, i.e., 'actions' is a mapping from item-sets and *character classes* to sets of actions.

Parser The SGLR parsing algoritm has been implemented in C. The implementation makes use of the C implementation of ATerms (Van den Brand *et al.*, 1997) to represent stacks and trees.

The parser includes visualization tools for parse forests and graph structured parse stacks that were used to produce the pictures in this paper. The forest vizualization might be used as basis for an interactive disambiguation tool.

The C implementation of the SGLR parsing algorithm seems reasonably efficient, although sharing of trees can be improved. Output of parse trees is not optimal because sharing of subtrees is completely lost when writing out a parse forest in a linear term format. This can solved by using a linear encoding of graphs such as the graph exchange language GEL of Kamperman (1994). Furthermore, a mark-scan garbage collector for stack and tree nodes is used. This entails that all stack and tree nodes are visited on a collect, which is too expensive, since a large amount of the heap will not change status. A reference count garbage collector should make a difference.

Complexity of Lexical Analysis We have performed a few experiments to get an idea of the complexity of lexical analysis with scannerless generalized-LR parsers. The experiments were based on the simple expression grammar in §2. The experiments that were performed were of the form: (a) Parsing a single identifier of increasing length (up to 425KB). (b) Parsing an expression consisting of ten additions with identifier arguments of increasing length (up to 325KB). (c) Parsing an expression consisting of an increasing number of additions (up to 16K arguments with length 490KB).

For all these experiments we saw an almost linear behaviour for small files deteriorating to square behaviour for the large files. However, when garbage collection was turned off, this behaviour changed into linear for all experiments. This confirms the observation about the inappropriateness of the garbage collection algorithm. It also confirms the idea that lexical analysis will behave linearly for simple, i.e., regular lexical syntax. The prototype implementation should be further optimized before its performance can meaningfully be compared to scanner/parser combinations such as LEX/YACC. Nonetheless, these experiments show the feasibility of the scannerless generlized-LR parsing approach.

10 Related Work

The syntax definition formalism SDF2 is formally specified in Visser (1997c). The specification in ASF+SDF comprises the syntax of the formalism, the normalization procedure and the parse tree format defined by a grammar.

The syntax definition formalism SDF of Heering *et al.* (1989) was the starting point for the work discussed in this paper. The definition of SDF2 grew out of the specification of SDF in ASF+SDF. A number of generalizations where applied to make the formalism more orthogonal and uniform and a number of improvements and new features were added based on the experience with SDF in the ASF+SDF Meta-Environment (Klint, 1993). SDF introduced the integration of lexical syntax and context-free syntax, but only at the formalism level. In the implementation, an SDF definition is mapped to a regular grammar defining the lexical syntax and a context-free grammar definining the context-free syntax. The scanners produced for the lexical syntax yield a graph structured stream of all possible tokenizations of the input filtered by a set of lexical disambiguation rules. Although this is a fairly advanced setup, the interface suffers from several of the problems that we discussed in §1 and §2.

The generalized-LR parsing algorithm was first developed by Tomita (1985) for application in natural language processing. It was later improved by Rekers (1992) and applied in the ASF+SDF Meta-Environment for parsing of programming languages. The algorithm presented in §8 is based on Rekers' version. Wagner and Graham (1997) describe the use of GLR parsing in incremental parsing of programming languages. Earley (1970) described the first generalized parsing algorithm that is closely related to the LR algorithm of Knuth (1965). A more recent approach to parsing with dynamic lookahead is the extension of top-down parsing with syntactic predicates of Parr and Quong (1994).

Scannerless parsing was introduced by Salomon and Cormack (1989, 1995). They define an extension of SLR(1) parsing in which the lack of lookahead is repaired by extending item-sets if conflicts are found. This non-canonical SLR(1) parser generation works only for a limited set of grammars, making grammar development difficult. The follow restrictions presented in this paper are a simplification of the adjacency restriction rule of the NSLR(1) approach in which arbitrary grammar symbols can be forbidden to be adjacent. Our reject productions are called exclusion rules by Salomon and Cormack (1989, 1995). We have presented a complete implementation for follow restrictions and reject productions, whereas the adjacency restrictions and exclusion rules are only partially implemented in NSLR(1) parsing.

A similar approach using GLR parsing is tried in the area of natural language processing. Tanaka *et al.* (1996) discuss the integration of morphological and syntactic analysis of Japanese sentences in a single GLR parser. The morphological rules describe how words can be formed from characters. Segmentation of a string of characters into a string of words is guided by a connection matrix restricting the categories that can be adjacent in a sentence. These rules do usually not suffice to find an unambiguous segmentation. by integrating morphological composition into the context-free grammar of the syntactic phase, such 'contextual' ambiguities can be avoided. This creates the problem of disambiguating the combined context-free grammar usingthe morphological connection matrix. This is partly done as a filter on the generated LR table and partly dynamically during parsing.

Disambiguation by means of priority and associativity declarations was introduced simultaneously by Aho *et al.* (1975) and Earley (1975). The former describe the solution of conflicts in LR parse tables by means of a restricted form of priorities. Aasa (1991, 1992) describes the solution of LR table conflicts by means of precedence declarations. Thorup (1992, 1994a, 1994b) describes the solution of parse table conflicts by means of a collection of excluded subtrees. The method is more expressive than the priorities of SDF, but only succeeds if all conflicts are solved, which is not guaranteed.

In Klint and Visser (1994) logical disambiguation methods are formalized as *disambiguation filters* on sets of parse trees. Based on this approach an efficient implementation of disambiguation by priorities is derived in Visser (1995a) from the disambiguation filter for priorities. This derivation forms the foundation for the parser generator algorithm presented in this paper.

11 Conclusions

In this paper we have presented a new approach to parsing that has several advantages over conventional techniques. It overcomes the drawbacks of the traditional scanner/parser interface by abolishing the scanner completely (hence the name scannerless parsing). The lexical and context-free syntax of a language are described in a single integrated uniform grammar formalism. Lexical ambiguities can frequently be solved by means of the parsing context. Lexical structure and layout are preserved in the parse tree and thus accessible in semantic tools. A more expressive formalism for lexical syntax is obtained, such that for example nested comments can be expressed.

The approach encompasses an expressive syntax definition formalism. A grammar normalizer to reduce the complexity of the formalism by simplifying syntax definitions to context-free grammars with a few extensions. An SLR(1) parser generator that deals with character-classes, follow restrictions and priority and associativity rules. A generalized-LR parser that can be used for arbitrary context-free grammars with reject productions, at least if they are not nested. In parsing unambiguous languages, the GLR parser is used to dynamically handle lookahead problems by forking off parsers in parallel.

Reject productions turn out to be a very expressive device that brings us out of the domain of context-free languages. It is as yet unclear how expressive this formalism is exactly, but we have a lower bound—stronger than context-free because it describes $a^n b^n c^n$—and an upper bound because the parsing problem is decidable.

Priorities are compiled into the parse table such that no parse trees with priority conflicts can be produced by the parser. This reduces the size of the parse forest (in case of ambiguous binary expressions the parse forest grows exponentially) and decreases the number of paths in the graph structured stack. The technique is more general than conventional techniques for this kind of disambiguation and works even if there remain conflicts in the parse table due to other causes. For instance, if the grammar requires more lookahead than the parser generator provides.

An open issue is the fully automatic derivation of lexical disambiguation rules from the grammar that would make the method still easier to use. Apart from this minor point, scannerless generalized-LR parsing is a feasible parsing method that makes syntax definition more expressive and solves a number of problems with conventional parsing approaches.

Acknowledgments The author thanks Arie van Deursen, Jan van Eijck, Annius Groenink and Paul Klint, for useful suggestions and comments on previous versions of this paper.

This research was supported by the Netherlands Computer Science Research Foundation (SION) with financial support from the Netherlands Organisation for Scientific Research (NWO). Project 612-317-420: Incremental parser generation and context-sensitive disambiguation: a multi-disciplinary perspective.

References

Aasa, A. (1991). Precedences in specifications and implementations of programming languages. In J. Maluzynski and M. Wirsing, editors, *Programming Language Implementation and Logic Programming*, volume 528 of *Lecture Notes in Computer Science*, pages 183–194. Springer-Verlag.

Aasa, A. (1992). *User Defined Syntax*. Ph.D. thesis, Department of Computer Sciences, Chalmers University of Technology and University of Göteborg, S-412 96 Göteborg, Sweden.

Aho, A. V., Johnson, S. C., and Ullman, J. D. (1975). Deterministic parsing of ambiguous grammars. *Communications of the ACM*, **18**(8), 441–452.

Anderson, T., Eve, J., and Horning, J. (1973). Efficient LR(1) parsers. *Acta Informatica*, **2**(1), 12–39.

Billot, S. and Lang, B. (1989). The structure of shared forests in ambiguous parsing. In *Proceedings of the Twenty-Seventh Annual Meeting of the Association for Computational Linguistics*. Association for Computational Linguistics.

Van den Brand, M. G. J. and Visser, E. (1996). Generation of formatters for context-free languages. *ACM Transactions on Software Engineering and Methodology*, **5**(1), 1–41.

Van den Brand, M. G. J., Klint, P., Olivier, P., and Visser, E. (1997). ATerms: Representing structured data for exchange between heterogeneous tools. Technical report, Programming Research Group, University of Amsterdam.

DeRemer, F. L. (1971). Simple LR(k) grammars. *Communications of the ACM*, **14**, 453–460.

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

Earley, J. (1975). Ambiguity and precedence in syntax description. *Acta Informatica*, **4**(1), 183–192.

Van Eijck, J. (1997). Email, july 9.

Floyd, R. W. (1962). Syntactic analysis and operator precedence. *Communications of the ACM*, **5**(10), 316–333.

Heering, J., Hendriks, P. R. H., Klint, P., and Rekers, J. (1989). The syntax definition formalism SDF – reference manual. *SIGPLAN Notices*, **24**(11), 43–75.

Jensen, K. and Wirth, N. (1975). *PASCAL User Manual and Report*, volume 18 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, second edition edition.

Kamperman, J. F. T. (1994). GEL, a graph exchange language. Technical Report CS-R9440, CWI, Amsterdam.

Klint, P. (1993). A meta-environment for generating programming environments. *ACM Transactions on Software Engineering and Methodology*, **2**(2), 176–201.

Klint, P. and Visser, E. (1994). Using filters for the disambiguation of context-free grammars. In G. Pighizzini and P. San Pietro, editors, *Proc. ASMICS Workshop on Parsing Theory*, pages 1–20, Milano, Italy. Tech. Rep. 126–1994, Dipartimento di Scienze dell'Informazione, Università di Milano.

Knuth, D. E. (1965). On the translation of languages from left to right. *Information and Control*, **8**, 607–639.

Lang, B. (1974). Deterministic techniques for efficient non-deterministic parsers. In J. Loeckx, editor, *Proceedings of the Second Colloquium on Automata, Languages and Programming*, volume 14 of *Lecture Notes in Computer Science*, pages 255–269. Springer-Verlag.

Parr, T. J. and Quong, R. W. (1994). Adding semantic and syntactic predicates to LL(k): pred-LL(k). In P. A. Fritzson, editor, *Compiler Construction, 5th International Conference, CC'94*, volume 786 of *LNCS*, pages 263–277, Edinburgh, U.K. Springer-Verlag.

Rekers, J. (1992). *Parser Generation for Interactive Environments*. Ph.D. thesis, University of Amsterdam. ftp://ftp.cwi.nl/pub/gipe/reports/Rek92.ps.Z.

Salomon, D. J. and Cormack, G. V. (1989). Scannerless NSLR(1) parsing of programming languages. *SIGPLAN Notices*, **24**(7), 170–178.

Salomon, D. J. and Cormack, G. V. (1995). The disambiguation and scannerless parsing of complete character-level grammars for programming languages. Technical Report 95/06, Department of Computer Science, University of Manitoba, Winnipeg, Canada.

Tanaka, H., Tokunga, T., and Aizawa, M. (1996). Integration of morphological and syntactical analysis based on GLR parsing. In H. C. Bunt and M. Tomita, editors, *Recent Advances in Parsing Technology*, pages 325–342. Kluwer Academic Publishers, Dordrecht, The Netherlands.

Thorup, M. (1992). Ambiguity for incremental parsing and evaluation. Technical Report PRG-TR-24-92, Program Research Group, Oxford University, Oxford, U.K.

Thorup, M. (1994a). Controlled grammatic ambiguity. *ACM Transactions on Programming Languages and Systems*, **16**(3), 1024–1050.

Thorup, M. (1994b). Disambiguating grammars by exclusion of sub-parse trees. Technical Report 94/11, Dept. of Computer Science, University of Copenhagen, Denmark.

Tomita, M. (1985). *Efficient Parsing for Natural Languages. A Fast Algorithm for Practical Systems*. Kluwer Academic Publishers.

Visser, E. (1995a). A case study in optimizing parsing schemata by disambiguation filters. In S. Fischer and M. Trautwein, editors, *Proceedings Accolade95*, pages 153–167, Amsterdam. The Dutch Graduate School in Logic.

Visser, E. (1995b). A family of syntax definition formalisms. In M. G. J. van den Brand *et al.*, editors, *ASF+SDF'95. A Workshop on Generating Tools from Algebraic Specifications*, pages 89–126. Technical Report P9504, Programming Research Group, University of Amsterdam.

Visser, E. (1997a). A case study in optimizing parsing schemata by disambiguation filters. In A. Nijholt, editor, *International Workshop on Parsing Technology IWPT'97*, Boston, USA. (To appear).

Visser, E. (1997b). Character classes. Technical Report P9708, Programming Research Group, University of Amsterdam.

Visser, E. (1997c). A family of syntax definition formalisms. Technical Report P9706, Programming Research Group, University of Amsterdam.

Wagner, T. A. and Graham, S. L. (1997). Incremental analysis for real programming languages. *SIGPLAN Notices*, **32**(5), 31–43. Proc. of the 1997 ACM SIGPLAN Conferene on Programming Language Design and Implementation (PLDI).

Technical Reports of the Programming Research Group

Note: These reports can be obtained using the technical reports overview on our WWW site (URL `http://www.wins.uva.nl/research/prog/reports/`) or using anonymous ftp to `ftp.wins.uva.nl`, directory `pub/programming-research/reports/`.

[P9711] L. Moonen. *A Generic Architecture for Data Flow Analysis to Support Reverse Engineering.*

[P9710] B. Luttik and E. Visser. *Specification of Rewriting Strategies.*

[P9709] J.A. Bergstra and M.P.A. Sellink. *An Arithmetical Module for Rationals and Reals.*

[P9708] E. Visser. *Character Classes.*

[P9707] E. Visser. *Scannerless generalized-LR parsing.*

[P9706] E. Visser. *A family of syntax definition formalisms.*

[P9705] M.G.J. van den Brand, M.P.A. Sellink, and C. Verhoef. *Generation of Components for Software Renovation Factories from Context-free Grammars.*

[P9704] P.A. Olivier. *Debugging Distributed Applications Using a Coordination Architecture.*

[P9703] H.P. Korver and M.P.A. Sellink. *A Formal Axiomatization for Alphabet Reasoning with Parametrized Processes.*

[P9702] M.G.J. van den Brand, M.P.A. Sellink, and C. Verhoef. *Reengineering COBOL Software Implies Specification of the Underlying Dialects.*

[P9701] E. Visser. *Polymorphic Syntax Definition.*

[P9618] M.G.J. van den Brand, P. Klint, and C. verhoef. *Re-engineering needs Generic Programming Language Technology.*

[P9617] P.I. Manuel. *ANSI Cobol III in SDF + an ASF Definition of a Y2K Tool.*

[P9616] P.H. Rodenburg. *A Complete System of Four-valued Logic.*

[P9615] S.P. Luttik and P.H. Rodenburg. *Transformations of Reduction Systems.*

[P9614] M.G.J. van den Brand, P. Klint, and C. Verhoef. *Core Technologies for System Renovation.*

[P9613] L. Moonen. *Data Flow Analysis for Reverse Engineering.*

[P9612] J.A. Hillebrand. *Transforming an ASF+SDF Specification into a ToolBus Application.*

[P9611] M.P.A. Sellink. *On the conservativity of Leibniz Equality.*

[P9610] T.B. Dinesh and S.M. Üsküdarlı. *Specifying input and output of visual languages.*

[P9609] T.B. Dinesh and S.M. Üsküdarlı. *The VAS formalism in VASE.*

[P9608] J.A. Hillebrand. *A small language for the specification of Grid Protocols.*

[P9607] J.J. Brunekreef. *A transformation tool for pure Prolog programs: the algebraic specification.*

[P9606] E. Visser. *Solving type equations in multi-level specifications (preliminary version).*

[P9605] P.R. D'Argenio and C. Verhoef. *A general conservative extension theorem in process algebras with inequalities.*

[P9602b] J.A. Bergstra and M.P.A. Sellink. *Sequential data algebra primitives (revised version of P9602).*

[P9604] E. Visser. *Multi-level specifications.*

[P9603] M.G.J. van den Brand, P. Klint, and C. Verhoef. *Reverse engineering and system renovation: an annotated bibliography.*

[P9602] J.A. Bergstra and M.P.A. Sellink. *Sequential data algebra primitives.*

[P9601] P.A. Olivier. *Embedded system simulation: testdriving the ToolBus.*

[P9512] J.J. Brunekreef. *TransLog, an interactive tool for transformation of logic programs.*

[P9511] J.A. Bergstra, J.A. Hillebrand, and A. Ponse. *Grid protocols based on synchronous communication: specification and correctness.*

[P9510] P.H. Rodenburg. *Termination and confluence in infinitary term rewriting.*

[P9509] J.A. Bergstra and Gh. Stefanescu. *Network algebra with demonic relation operators.*

[P9508] J.A. Bergstra, C.A. Middelburg, and Gh. Stefanescu. *Network algebra for synchronous and asynchronous dataflow.*

[P9507] E. Visser. *A case study in optimizing parsing schemata by disambiguation filters.*

[P9506] M.G.J. van den Brand and E. Visser. *Generation of formatters for context-free languages.*

[P9505] J.M.T. Romijn. *Automatic analysis of term rewriting systems: proving properties of term rewriting systems derived from* ASF+SDF *specifications.*