Disambiguation Filters for Scannerless Generalized LR Parsers

M.G.J. van den Brand[1,4], J. Scheerder[2], J.J. Vinju[1], and E. Visser[3]

[1] Centrum voor Wiskunde en Informatica (CWI), Kruislaan 413, 1098 SJ Amsterdam, The Netherlands, {Mark.van.den.Brand,Jurgen.Vinju}@cwi.nl
[2] Department of Philosophy, Utrecht University, Heidelberglaan 8, 3584 CS Utrecht, The Netherlands, js@phil.uu.nl
[3] Institute of Information and Computing Sciences, Utrecht University, P.O. Box 80089, 3508TB Utrecht, The Netherlands, visser@acm.org
[4] LORIA-INRIA, 615 rue du Jardin Botanique, BP 101, F-54602 Villers-lès-Nancy Cedex, France

Abstract In this paper we present the fusion of generalized LR parsing and scannerless parsing. This combination supports syntax definitions in which all aspects (lexical and context-free) of the syntax of a language are defined explicitly in one formalism. Furthermore, there are no restrictions on the class of grammars, thus allowing a natural syntax tree structure. Ambiguities that arise through the use of unrestricted grammars are handled by explicit disambiguation constructs, instead of implicit defaults that are taken by traditional scanner and parser generators. Hence, a syntax definition becomes a full declarative description of a language. Scannerless generalized LR parsing is a viable technique that has been applied in various industrial and academic projects.

1 Introduction

Since the introduction of efficient deterministic parsing techniques, parsing is considered a closed topic for research, both by computer scientists and by practicioners in compiler construction. Tools based on deterministic parsing algorithms such as LEX & YACC [15,11] (LALR) and JavaCC (recursive descent), are considered adequate for dealing with almost all modern (programming) languages. However, the development of more powerful parsing techniques is prompted by domains such as reverse engineering and domain-specific languages.

The field of reverse engineering is concerned with automatically analyzing legacy software and producing specifications, documentation, or reimplementations. This area provides numerous examples of parsing problems that can only be tackled by using powerful parsing techniques.

Grammars of languages such as Cobol, PL1, Fortran, etc. are not naturally LALR. Much massaging and default resolution of conflicts are needed to implement a parser for these languages in YACC. Maintenance of such massaged grammars is a pain since changing or adding a few productions can lead to new conflicts. This problem is aggravated when different dialects need to be supported—many vendors implement their own Cobol dialect. Since grammar formalisms are not modular this usually leads to forking of grammars. Further trouble is caused by the embedding of 'foreign' language fragments, e.g., assembler code, SQL, CICS, or C, which is common practice in Cobol programs. Merging of grammars for several languages leads to conflicts at the context-free grammar level and at the lexical analysis level. These are just a few examples of problems encountered with deterministic parsing techniques.

The need to tackle such problems in the area of reverse engineering has led to a revival of generalized parsing algorithms such as Earley's algorithm, (variants of) Tomita's algorithm (GLR) [14,21,17,2,20], and even recursive descent backtrack parsing [6]. Although generalized parsing solves several problems in this area, generalized parsing alone is not enough.

In this paper we describe the benefits and the practical applicability of *scannerless generalized LR parsing*. In Section 2 we discuss the merits of scannerless parsing and generalized parsing and argue that their combination provides a solution for problems like the ones described above. In Section 3 we describe how disambiguation can be separated from grammar structure, thus allowing a natural grammar structure and declarative and selective specification of disambiguation. In Section 4 we discuss issues in the implementation of disambiguation. In Section 5 practical experience with the parsing technique is discussed. In Section 6 we present figures on the performance of our implementation of a scannerless generalized parser. Related work is discussed where needed throughout the paper. Finally, we conclude in Section 7.

2 Scannerless Generalized Parsing

2.1 Generalized Parsing

Generalized parsers are a class of parsing algorithms that are not constrained by restrictions on the class of grammars that they can handle, contrary to restricted parsing algorithms such as the various derivatives of the LL and LR algorithms. Whereas these algorithms only deal with context-free grammars in LL(k) or LR(k) form, generalized algorithms such as Earley's or Tomita's algorithms can deal with arbitrary context-free grammars. There are two major advantages to the use of arbitrary context-free grammars.

Firstly, the class of context-free grammars is closed under union, in contrast with all proper subclasses of context-free grammars. For example, the composition of two LALR grammars is very often not an LALR grammar. The compositionality of context-free grammars opens up the possibility of developing modular syntax definition formalisms. Modularity in programming languages and other formalisms is one of the key beneficial software engineering concepts. A striking example in which modularity of a grammar is obviously practical is the definition of hybrid languages such as Cobol with CICS, or C with assembly. SDF [10,23] is an example of a modular syntax definition formalism.

Secondly, an arbitrary context-free grammar allows the definition of declarative grammars. There is no need to massage the grammar into LL, LR, LALR, or any other form. Rather the grammar can reflect the intended structure of the language, resulting in a concise and readable syntax definition. Thus, the same grammar can be used for documentation as well as implementation of a language without any changes.

Since generalized parsers can deal with arbitrary grammars, they can also deal with *ambiguous* grammars. While a deterministic parser produces a single parse tree, a non-deterministic parser produces a collection (forest) of trees compactly representing all possible derivations according to the grammar. This can be helpful when developing a grammar for a language. The parse forest can be used to visualize the ambiguites in the grammar, thus aiding in the improvement of the grammar. Contrast this with solving conflicts in an LALR table. Disambiguation filters can be used to reduce a forest to the intended parse tree. Filters can be based on disambiguation rules such as priority and associativity declarations. Such filters solve the most frequent ambiguities in a natural and intuitive way without hampering the clear structure of the grammar.

In short, generalized parsing opens up the possibility for developing clear and concise language definitions, separating the language design problem from the disambiguation problem.

2.2 Scannerless Parsing

Traditionally, syntax analysis is divided into a lexical scanner and a (context-free) parser. A scanner divides an input string consisting of characters into a string of tokens. This tokenization is usually based on regular expression matching. To choose between overlapping matches a number of standard lexical disambiguation rules are used. Typical examples are prefer keywords, prefer longest match, and prefer non-layout. After tokenization, the tokens are typically interpreted by the parser as the terminal symbols of an LR(1) grammar.

Although this architecture proves to be practical in many cases and is globally accepted as the standard solution for parser generation, it has some problematic limitations. Only few *existing* programming languages are designed to fit this architecture, since these languages generally have an ambiguous lexical syntax. The following examples illustrate this misfit for Cobol, PL1 and Pascal.

In an embedded language, such as SQL in Cobol, identifiers that are reserved keywords in Cobol might be allowed inside SQL statements. However, the implicit "prefer keywords" rule of lexical scanners will automatically prohibit them in SQL too.

Another Cobol example; a particular "picture clause" might look like `"PIC 99"`, where `"99"` should be recognized as a list of `picchars`. In some other part of a Cobol program, the number `"99"` should be recognized as `numeric`. Both lexical categories obviously overlap, but on the context-free level there is no ambiguity because picture clauses do not appear where numerics do. See [13] for a Cobol syntax definition.

Another example of scanner and parser interference stems from Pascal. Consider the input sentence `"array [1..10] of integer"`, the range `"1..10"` can be tokenized in two different manners, either as the real `"1."` followed by the real ".10", or as the integer "1" followed by the range operator ".." followed by the integer "10". In order to come up with the correct tokenization the scanner must "know" it is processing an array declaration.

The problem is even more imminent when a language does not have reserved keywords at all. PL1 is such a language. This means that a straightforward tokenization is not possible when scanning a valid PL1 sentence such as "IF THEN THEN = ELSE; ELSE ELSE = THEN;".

Similar examples can be found for almost any existing programming language. A number of techniques for tackling this problem is discussed in [3]. Some parser generators provide a complex interface between scanner and parser in order to profit from the speed of lexical analysis while using the power of a parser. Some lexical scanners have more expressive means than regular expressions to be able to make more detailed decisions. Some parser implementations allow arbitrary computations to be expressed in a programming language such as C to guide the scanner and the parser. All in all it is rather cumbersome to develop and to maintain grammars which have to solve such simple lexical disambiguations, because none of these approaches result in declarative syntax specifications.

Scannerless parsing is an alternative parsing technique that does not suffer these problems. The term scannerless parsing was introduced in [18,19] to indicate parsing without a separate lexical analysis phase. In scannerless parsing, a syntax definition is a context-free grammar with characters as terminals. Such an integrated syntax definition defines all syntactic aspects of a language, including the full details of the lexical syntax. The parser derived from this grammar directly reads the characters of the input string and finds its phrase structure.

Scannerless parsing does not suffer the problems of implicit lexical disambiguation. Very often the problematic lexical ambiguities do not even exist at the context-free level, as is the case in our Cobol, Pascal and PL1 examples. On the other hand, the lack of implicit rules such as "prefer keywords" and "longest match" might give rise to new ambiguities at the context-free level. These ambiguities can be solved by providing *explicit* declarative rules in a syntax definition language. Making such disambiguation decisions explicit makes it possible to apply them selectively. For instance, we could specify longest match for a single specific sort, instead of for the entire grammar, as we shall see in Section 3.

In short, scannerless parsing does not need to make any assumptions about the lexical syntax of a language and is therefore more generically applicable for language engineering.

2.3 Combining Scannerless Parsing and Generalized Parsing

Syntax definitions in which lexical and context-free syntax are fully integrated do not usually fit in any restricted class of grammars required by deterministic parsing techniques because lexical syntax often requires arbitrary length lookahead. Therefore, scannerless parsing does not go well with deterministic parsing. For this reason the adjacency restrictions and exclusion rules of [18,19] could only be partly implemented in an extension of a SLR(1) parser generator and led to complicated grammars.

Generalized parsing techniques, on the other hand, can deal with arbitrary length lookahead. Using a generalized parsing technique solves the problem of lexical lookahead in scannerless parsing. However, it requires a solution for disambiguation of lexical ambiguities that are not resolved by the parsing context.

In the rest of this paper we describe how syntax definitions can be disambiguated by means of declarative disambiguation rules for several classes of ambiguities, in particular lexical ambiguities. Furthermore, we discuss how these disambiguation rules can be implemented efficiently.

3 Disambiguation Rules

There are many ways for disambiguation of ambiguous grammars, ranging from simple syntactic criteria to semantic criteria [12]. Here we concentrate on ambiguities caused by integrating lexical and context-free syntax. Four classes of disambiguation rules turn out to be adequate.

Follow restrictions are a simplification of the adjacency restriction rules of [18,19] and are used to achieve longest match disambiguation. Reject productions, called exclusion rules in [18,19], are designed to implement reserved keywords disambiguation. Priority and associativity rules are used to disambiguate expression syntax. Preference attributes are used for selecting a default among several alternative derivations.

3.1 Follow Restrictions

Suppose we have the simple context-free grammar for terms as presented in Figure 1. An `Id` is defined to be one ore more characters from the class `[a-z]+` and two terms are separated by whitespace consisting of zero or more spaces or newlines.

```
Term ::= Id | Nat | Term Ws Term
Id   ::= [a-z]+
Nat  ::= [0-9]+
Ws   ::= [\ \n]*
%restrictions
Id  -/- [a-z]
Nat -/- [0-9]
Ws  -/- [\ \n]
```

Figure 1. Term language with follow restrictions.

Without any lexical disambiguation, this grammar is ambiguous. For example, the sentence `"hi"` can be parsed as `Term(Id("hi"))` or as `Term(Id("h")), Ws(""), Term(Id("i"))`. Assuming the first is the intended derivation, we add a follow restriction, `Id -/- [a-z]`, indicating that an `Id` may not directly be followed by a character in the range `[a-z]`. This entails that such a character should be part of the identifier. Similarly, follow restrictions are added for

Nat and Ws. We have now specified a *longest match* for each of these lexical constructs.

In some languages it is necessary to have more than one character lookahead to decide the follow restriction. In Figure 2 we extend the layout definition of Figure 1 with comments. The expression ~[*] indicates any character except the asterisk. The expression [\(].[*] defines a restriction on two consecutive characters. The result is a *longest match* for the Ws nonterminal, including comments. The follow restriction on Star prohibits the recognition of the string "*)" within Comment. Note that it is straightforward to extend this definition to deal with *nested* comments.

```
Star        ::= [\*]
CommentChar ::= ~[\*] | Star
Comment     ::= "(*" CommentChar* "*)"
Ws          ::= ([\ \n] | Comment)*
%restrictions
Star -/- [\)]
Ws   -/- [\ \n] | [\(].[\*]
```

Figure 2. Extended layout definition with follow restrictions.

3.2 Reject Productions

Reject productions are used to implement keyword reservation. We extend the grammar definition of Figure 1 with the begin and end construction in Figure 3. The sentence "begin hi end" is either interpreted as three consecutive Id terms separated by Ws, or as a Program with a single term hi. By *rejecting* the strings begin and end from Id, the first interpretation can be filtered out.

The reject mechanism can be used to reject not only strings, but entire context-free languages from a nonterminal. We focus on its use for keyword reservation in this paper and refer to [23] for more discussion.

```
Program ::= "begin" Ws Term Ws "end"
Id      ::= "begin" | "end" {reject}
```

Figure 3. Prefer keywords using reject productions

3.3 Priority and Associativity

For completeness we show an example of the use of priority and associativity in an expression language. Note that we have left out the Ws nonterminal for brevity[1]. In Figure 4 we see that the binary operators + and * are both defined as left associative and the * operator has a higher priority than the + operator. Consequently the sentence "1 + 2 + 3 * 4" is interpreted as "(1 + 2) + (3 * 4)".

[1] By doing grammar normalization a parse table generator can automatically insert layout between the members in the right-hand side. See also Section 5.

```
Exp ::= [0-9]+
Exp ::= Exp "+" Exp {left}
Exp ::= Exp "*" Exp {left}
%priorities
Exp ::= Exp "*" Exp > Exp ::= Exp "+" Exp
```

Figure 4. Associativity and priority rules.

3.4 Preference Attributes

A preference rule is a generally applicable rule to choose a default among ambiguous parse trees. For example, it can be used to disambiguate the notorious dangling else construction. Again we have left out the `Ws` nonterminal for brevity. In Figure 5 we extend our term language with this construct.

The input sentence `"if 0 then if 1 then hi else ho"` can be parsed in two ways: `if 0 then (if 1 then hi) else ho` and `if 0 then (if 1 then hi else ho)`. We can select the latter derivation by adding the `prefer` attribute to the production without the `else` part. The parser will still construct an ambiguity node containing both deriviations, namely, `if 0 then (if 1 then hi {prefer}) else ho` and `if 0 then (if 1 then hi else ho) {prefer}`. But given the fact that the *top* node of the latter derivation tree has the prefer attribute this derivation is selected and the other tree is removed from the ambiguity node.

The dual of `{prefer}` is the `{avoid}` attribute. Any other tree is preferred over a tree with an avoided top production. One of its uses is to prefer keywords rather than reserving them entirely. For example, we can add an `{avoid}` to the `Id ::= [a-z]+` production in Figure 1 and not add the reject productions of Figure 3. The sentence `"begin begin end"` is now a valid `Program` with the single derivation of a `Program` containing the single `Id "begin"`.

```
Term ::= "if" Nat "then" Term {prefer}
Term ::= "if" Nat "then" Term "else" Term
Id   ::= "if" | "then" | "else" {reject}
```

Figure 5. Dangling else construction disambiguated

4 Implementation Issues

Our implementation of scannerless generalized parsing consists of the syntax definition formalism SDF that supports concise specification of integrated syntax definitions, a grammar normalizer that injects layout and desugars regular expressions, a parse table generator and a parser that interprets parse tables.

The parser is based on the GLR algorithm. For the basic GLR algorithms we refer to the first publication on generalized LR parsing by Lang [14], the work by Tomita [21], and the various improvements and implementations [17,2,20]. We will not present the complete SGLR algorithm, because it is essentially the standard GLR algorithm where each character[2] is a separate token. For a detailed description of the implementation of GLR and SGLR we refer to [17] and [22] respectively.

[2] The current implementation of SGLR supports the Latin-1 character set.

The algorithmic differences between standard GLR and scannerless GLR parsing are centered around the disambiguation constructs. From a declarative point of view each disambiguation rule corresponds to a filter that prunes parse forests. In this view, parse table generation and the GLR algorithm remain unchanged and the parser returns a forest containing all derivations. After parsing a number of filters is executed and a single tree or at least a smaller forest is obtained.

Although this view is conceptually attractive, it does not fully exploit the possibilities for pruning the parse forest *before* it is even created. A filter might be implemented statically, during parse table generation, dynamically, during parsing, or after parsing. The sooner a filter is applied, the faster a parser will return the filtered derivation tree. In which phase they are applicable depends on the particulars of specific disambiguation rules. In this section we discuss the implementation of the four classes of disambiguation rules.

4.1 Follow Restrictions

Our parser generator generates a simple SLR(1) parse table, however we deviate at a number of places from the standard algorithm [1]. One modification is the calculation of the follow set. The follow set is calculated for each individual production rule instead of for each nonterminal. Using priority and associativity relations may lead to different follow sets for productions with the same non-terminal in the left-hand side. Another modification is that the transitions between states (item-sets) in the LR-automaton are not labeled with a nonterminal, but with a production rule. These more fine-grained transitions increase the size of the LR-automaton, but it allows us to generate parse tables with fewer conflicts.

Follow restriction declarations with a single lookahead can be used during parse table generation to remove reductions from the parse table. This is done by intersecting the follow set of each production rule with the set of characters in the follow restrictions for the produced nonterminal. The effect of this filter is that the reduction in question cannot be performed for characters in the follow restriction set.

Restrictions with more than one lookahead must be dealt with dynamically by the parser. The parse table generator marks the reductions that produce a nonterminal that has restrictions with more than one character. Then, while parsing, before such a reduction is done the parser must retrieve the required number of characters from the string and check them with the restrictions. If the next characters in the input match these restrictions the reduction is not allowed, otherwise it can be performed. This parse-time implementation prohibits shift/reduce conflicts that would normally occur and therefore saves the parser from performing unnecessary work.

4.2 Reject Productions

Disambiguation by means of reject productions cannot be implemented statically, since this would require computing the intersection of two syntactic categories, which is not possible in general. Even computing such intersections for regular grammars would lead to very large automata. When using a generalized parser, filtering with reject productions can be implemented effectively during parsing.

Consider the reject production `Id ::= "begin" {reject}`, which declares that `"begin"` is not a valid `Id` in *any* way (Figure 3). Thus, each and every derivation of the subsentence `"begin"` that produces an `Id` is illegal. During parsing, without the reject production the substring `"begin"` will be recognized both as an `Id` and as a keyword in a `Program`. By adding the reject production to the grammar another derivation is created for `"begin"` as an `Id`, resulting in an ambiguity of two deriviations. If one derivation in an ambiguity node is rejected, the entire parse stack for that node is deleted. Hence, `"begin"` is not recognized as an identifier in any way. Note that the parser must wait until each ambiguous derivation has returned before it can delete a stack[3]. The stack on which this substring was recognized as an `Id` will not survive, thus no more actions are performed on this stack. The only derivation that remains is where `"begin"` is a keyword in a `Program`.

Reject productions could also be implemented as a backend filter. However, by terminating stacks on which reject productions occur as soon as possible a dramatic reduction in the number of ambiguities can be obtained.

4.3 Priority and Associativity

Associativity of productions and priority relations can be processed during the construction of the parse table. We present an informal description here and refer to [23] for details.

There are two phases in the parse table generation process in which associativity and priority information is used. The first place is during the construction of the LR-automaton. Item-sets in the LR-automaton contain dotted productions. Prediction of new items for an item-set takes the associativity and priority relations into consideration. If a predicted production is in conflict with the production of the current item, then the latter production is *not* added to the item-set. The second place is when shifting a dot over a nonterminal in an item. In case of an associativity or priority conflict between a production rule in the item and a production rule on a transition, the transition will not be added to the LR-automaton.

We will illustrate the approach described above by discussing the construction of a part of the LR-automaton for the grammar presented in Figure 4. We create the transitions in the LR-automaton for state s_i which contains the items

```
[Exp ::= . Exp "+" Exp] [Exp ::= . Exp "*" Exp] [Exp ::= . [0-9]+]
```

[3] Our parser synchronizes parallel stacks on shifts, so we can wait for a shift before we delete an ambiguity node.

In order to shift the dot over the nonterminal `Exp` via the production rule `Exp ::= Exp "+" Exp` every item in s_i is checked for a conflict. The new state s_j has the item-set

```
[Exp ::= Exp . "+" Exp]
```

Note that s_j does not contain the item `[Exp ::= Exp . "*" Exp]`, since that would cause a conflict with the given priority relation `"*" > "+"`.

By pruning the transitions in a parse table in the above manner, conflicts at parse time pertaining to associativity and priority can be ruled out. However, if we want priority declarations to ignore injections (or chain rules) this implementation does not suffice. Yet it is natural to ignore injections when applying disambiguation rules, since they do not have any visible syntax. Priorities module chain rules require an extension of this method or a parse-time filter.

4.4 Preference Attributes

The preference filter is a typical example of an after parsing filter. In principle it could be applied while parsing, however this will complicate the implementation of the parser tremendously without gaining efficiency. This filter operates on an ambiguity node, which is a set of ambiguous subtrees, and selects the subtrees with the highest preference.

The simplest preference filter compares the trees of each ambiguity node by comparing the `avoid` or `prefer` attributes of the top productions. Each preferred tree remains in the set, while all others are removed. If there is no preferred tree, all avoided trees are removed, while all others remain. Ignoring injections at the top is a straightforward extension to this filter.

By implementing this filter in the backend of the parser we can exploit the redundancy in parse trees by caching filtered subtrees and reusing the result when filtering other identical subtrees. We use the ATerm library [5] for representing a parse forest. It has *maximal sharing* of subterms, limiting the amount of memory used and making subtree identification a trivial matter of pointer equality.

For a number of grammars this simple preference filter is not powerful enough, because the production rules with the `avoid` or `prefer` are not at the root (modulo injectons) of the subtrees, but deeper in the subtree. In order to disambiguate these ambiguous subtrees, more subtle preference filters are needed. However, these filters will always be based on some heuristic, e.g., counting the number of "preferred" and "avoided" productions and applying some selection on the basis of these numbers, or by looking a the depth at which a "preferred" or "avoided" production occurs. In principle, for any chosen heuristic counter examples can be constructed for which the heuristic fails to achieve its intended goal, yielding undesired results.

5 Applications

5.1 Asf+Sdf Meta-Environment

In the introduction of this paper we claimed that generalized parsing techniques are applicable in the fields of reverse engineering and language prototyping, i.e., the development of new (domain-specific) languages. The Asf+Sdf Meta-Environment [4] is used in both these fields. This environment is an interactive development environment for the automatic generation of interactive systems for manipulating programs, specifications, or other texts written in a formal language. The parser in this environment and in the generated environments is an SGLR parser. The language definitions are written in the Asf+Sdf formalism [8] which allows the definition of syntax via Sdf (Syntax Definition Formalism) [10] as well as semantics via Asf (Algebraic Specification Formalism). Figure 6 shows an Sdf specification of the previous examples.

Asf+Sdf has been used in a number of industrial and scientific projects. Amongst others it was used for parsing and compiling Asf+Sdf specifications, automatically renovating Cobol code, program analysis of legacy code via so-called island grammars [16], and development of new Action Notation syntax [9].

5.2 XT

XT [7] is a collection of basic tools for building program transformation systems including the Stratego transformation language [24], and the syntax definition formalism Sdf supported by SGLR. Tools standardize on ATerms [5] as common exchange format. Several meta-tools are provided for generating transformation components from syntax definitions, including a data type declaration generator that generates the data type corresponding to the abstract syntax of an Sdf syntax definition, and a pretty-printer generator that generates default pretty-print tables.

To promote reuse and standardization of syntax definitions, the XT project has initiated the creation of the Online Grammar Base[4] currently with some 25 syntax definitions for various general purpose and domain-specific languages, including Cobol, Java, SDL, Stratego, YACC, and XML. Many syntax definitions were semi-automatically reengineered from LEX/YACC definitions using grammar manipulation tools from XT, producing more compact syntax definitions. Sdf/SGLR based parsers have been used in numerous projects built with XT in areas ranging from software renovation and grammar recovery to program optimization and compiler construction.

6 Benchmarks

We have benchmarked our implementation of SGLR by parsing a number of large files and measuring the `user time`. Table 1 shows the results with and

[4] `http://www.program-transformation.org/gb`

[5] All benchmarks were performed on a 1200 Mhz AMD Athlon(tm) with 512Mb memory running Linux.

```
module Program
imports If
exports
  sorts Program
  context-free syntax
    "begin" Term "end" -> Program
    "begin" | "end"    -> Id      {reject}

module If
imports Terms
exports
  context-free syntax
    "if" Nat "then" Term             -> Term {prefer}
    "if" Nat "then" Term "else" Term -> Term
    "if" | "then" | "else"           -> Id   {reject}

module Terms
imports Comment
exports
  sorts Term
  lexical syntax
    [0-9]+    -> Nat
    [a-z]+    -> Id
  lexical restrictions
    Id      -/- [a-z]
    Nat     -/- [0-9]
  context-free syntax
    Term Term -> Term {left}
    Id | Nat  -> Term

module Comment
exports
  lexical syntax
    [\*]                    -> Star
    ~[\*] | Star            -> CommentChar
    "(*" CommentChar* "*)" -> Comment
    [\ \n] | Comment        -> LAYOUT
  lexical restrictions
    Star -/- [\)]
  context-free restrictions
    LAYOUT? -/- [\ \n] | [\(].[\*]
```

Figure 6. A modular SDF definition combining some of the previous examples. This example also shows the use of a special "LAYOUT" nonterminal, the use of regular expressions (e.g. "|" for alternative and "*" for repetition) and the use of multiple start nonterminals.

Grammar	Average file size	Characters/second with filter & tree[5]	Characters/second w/o filter & tree[5]
ATerms	106,000 chars	108,000	340,000
BibTEX	455,000 chars	85,000	405,000
Box	80,000 chars	34,000	368,000
Cobol	170,000 chars	58,000	146,000
Java	105,000 chars	37,000	210,000
Java (LR1)	105,000 chars	53,000	242,000

Table 1. Some figures on SGLR performance.

Grammar	Productions	States	Actions	Actions with conflicts	Gotos
ATerms	104	128	8531	75	46569
BibTEX	150	242	40508	3129	98901
Box	202	385	19249	1312	177174
Cobol	1906	5520	170375	32634	11941923
Java	726	1561	148359	5303	1535446
Java (LR1)	765	1597	88561	3354	1633156

Table 2. Some figures on the grammars and the generated parse tables.

without parse tree construction and backend filtering. All filters implemented in the parse table or during parsing are active in both measurements. The table shows that the parser is fast enough for industrial use. An interesting observation is that the construction of the parse tree slows down the entire process quite a bit. Further speedup can be achieved by optimizing parse tree construction.

Table 2 shows some details of the SLR(1) parse tables for the grammars we used. We downloaded all but the last grammar from the Online Grammar Base. ATerms is a grammar for prefix terms with annotations, BibTEX is a bibliography file format, Box is a mark-up language used in pretty-print tools. Cobol and Java are grammars for the well-known programming languages. We have benchmarked two different Java grammars. The first is written from scratch in SDF, the second was obtained by transforming a Yacc grammar into SDF. So, the first is a more natural definition of Java syntax, while the second is in LR(1) form.

The number of productions is measured after SDF grammar normalization[6]. We mention the number of states, gotos and actions in the parse table. Remember that the parse table is specified down to the character level, so we have more states than usual. Also, actions and gotos are based on productions, not nonterminals, resulting in a bigger parse table. The number of actions with more than one reduce or shift (a conflict) gives an indication of the amount of "ambiguity" in a grammar. The two Java results in Table 1 show that ambiguity of a grammar has a limited effect on performance. Note that after filtering, every parse in our testset resulted in a single derivation.

[6] So this number does not reflect the size of the grammar definition.

7 Conclusions

In this paper we discussed the combination of generalized LR parsing with scannerless parsing. The first parsing technique allows for the development of modular definition of grammars whereas the second one relieves the grammar writer from interface problems between scanner and parser. The combination supports the development of declarative and maintainable syntax definitions that are not forced into the harness of a restricted grammar class such as LL(k) or LR(k). This proves to be very beneficial when developing grammars for legacy languages such as Cobol and PL/I, but it also provides greater flexibility in the development of new (domain-specific) languages.

One of the assets of the SGLR approach is the separation of disambiguation from grammar structure. Thus, it is not necessary to encode disambiguation decisions using extra productions and non-terminals. Instead a number of disambiguation filters, driven by disambiguation declarations solve ambiguities by pruning the parse forest. Lexical ambiguities, which are traditionally handled by adhoc default decisions in the scanner, are also handled by such filters. Filters can be implemented at several points in time, i.e., at parser generation time, parse time, or after parsing.

SGLR is usable in practice. It has been used as the implementation of the expressive syntax definition formalism SDF. SGLR is not only fast enough to be used in interactive tools, like the ASF+SDF Meta-Environment, but also to parse huge amounts of Cobol code in an industrial environment.

SGLR and the SDF based parse table generator are open-source and can be downloaded from `http://www.cwi.nl/projects/MetaEnv/`.

Acknowledgements

User feedback has been indispensable while developing SGLR. Hayco de Jong and Pieter Olivier dedicated considerable time on improving SGLR efficiency. Merijn de Jonge and Joost Visser were instrumental in the development of the Online Grammar Base that serves as a testbed for SGLR. Jan Heering and Paul Klint provided valuable input when discussing design and implementation of SGLR.

References

1. A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers. Principles, Techniques and Tools.* Addison-Wesley, 1986.
2. J. Aycock and R.N. Horspool. Faster generalized LR parsing. In S. Jähnichen, editor, *CC'99*, volume 1575 of *LNCS*, pages 32–46. Springer-Verlag, 1999.
3. J. Aycock and R.N. Horspool. Schrödinger's token. *Software, Practice & Experience*, 31:803–814, 2001.
4. M.G.J. van den Brand, A. van Deursen, J. Heering, H.A. de Jong, M. de Jonge, T. Kuipers, P. Klint, L. Moonen, P. A. Olivier, J. Scheerder, J.J. Vinju, E. Visser, and J. Visser. The ASF+SDF Meta-Environment: a Component-Based Language Development Environment. In R. Wilhelm, editor, *CC'01*, volume 2027 of *LNCS*, pages 365–370. Springer-Verlag, 2001.

5. M.G.J. van den Brand, H.A. de Jong, P. Klint, and P.A. Olivier. Efficient Annotated Terms. *Software, Practice & Experience*, 30(3):259–291, 2000.
6. J.R. Cordy, C.D. Halpern-Hamu, and E. Promislow. TXL: A rapid prototyping system for programming language dialects. *Computer Languages*, 16(1):97–107, 1991.
7. M. de Jonge, E. Visser, and J. Visser. XT: A bundle of program transformation tools. In M. G. J. van den Brand and D. Parigot, editors, *Workshop on Language Descriptions, Tools and Applications (LDTA'01)*, volume 44 of *Electronic Notes in Theoretical Computer Science*. Elsevier Science Publishers, 2001.
8. A. van Deursen, J. Heering, and P. Klint, editors. *Language Prototyping*, volume 5 of *AMAST Series in Computing*. World Scientific, 1996.
9. K.-G. Doh and P.D. Mosses. Composing programming languages by combining action-semantics modules. In M.G.J. van den Brand and D. Parigot, editors, *Electronic Notes in Theoretical Computer Science*, volume 44, 2001.
10. J. Heering, P.R.H. Hendriks, P. Klint, and J. Rekers. The syntax definition formalism SDF – reference manual. *SIGPLAN Notices*, 24(11):43–75, 1989.
11. S. C. Johnson. YACC—yet another compiler-compiler. Technical Report CS-32, AT & T Bell Laboratories, Murray Hill, N.J., 1975.
12. P. Klint and E. Visser. 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, 1994. Tech. Rep. 126–1994, Dipartimento di Scienze dell'Informazione, Università di Milano.
13. R. Lämmel and C. Verhoef. VS COBOL II grammar[7], 2001.
14. B. Lang. Deterministic techniques for efficient non-deterministic parsers. In J. Loeckx, editor, *Proceedings of the Second Colloquium on Automata, Languages and Programming*, volume 14 of *LNCS*, pages 255–269. Springer-Verlag, 1974.
15. M. E. Lesk and E. Schmidt. *LEX — A lexical analyzer generator*. Bell Laboratories, 1986. UNIX Programmer's Supplementary Documents, Volume 1 (PS1).
16. L. Moonen. Generating robust parsers using island grammars. In *Proceedings of the 8th Working Conference on Reverse Engineering*, pages 13–22. IEEE Computer Society Press, 2001.
17. J. Rekers. *Parser Generation for Interactive Environments*. PhD thesis, University of Amsterdam, 1992. ftp://ftp.cwi.nl/pub/gipe/reports/Rek92.ps.Z.
18. D.J. Salomon and G.V. Cormack. Scannerless NSLR(1) parsing of programming languages. *SIGPLAN Notices*, 24(7):170–178, 1989.
19. D.J. Salomon and G.V. Cormack. The disambiguation and scannerless parsing of complete character-level grammars for programming languages. Technical Report 95/06, Dept. of Computer Science, University of Manitoba, 1995.
20. E. Scott, A. Johnstone, and S.S. Hussain. Technical Report TR-00-12, Royal Holloway, University of London, Computer Science Dept., 2000.
21. M. Tomita. *Efficient Parsing for Natural Languages. A Fast Algorithm for Practical Systems*. Kluwer Academic Publishers, 1985.
22. E. Visser. Scannerless generalized-LR parsing. Technical Report P9707, Programming Research Group, University of Amsterdam, 1997.
23. E. Visser. *Syntax Definition for Language Prototyping*. PhD thesis, University of Amsterdam, 1997.
24. E. Visser. Stratego: A language for program transformation based on rewriting strategies. System description of Stratego 0.5. In A. Middeldorp, editor, *RTA'01*, volume 2051 of *LNCS*, pages 357–361. Springer-Verlag, 2001.

[7] `http://www.cs.vu.nl/grammars/browsable/vs-cobol-ii/`