LL(1) Conflict Resolution in a Recursive Descent Compiler Generator

Albrecht Wöß, Markus Löberbauer, Hanspeter Mössenböck

Johannes Kepler University Linz, Institute of Practical Computer Science,
Altenbergerstr. 69, 4040 Linz, Austria
{woess,loeberbauer,moessenboeck}@ssw.uni-linz.ac.at

Abstract. Recursive descent parsing is restricted to languages whose grammars are LL(1), i.e., which can be parsed top-down with a single lookahead symbol. Unfortunately, many languages such as Java, C++, or C# are not LL(1). Therefore recursive descent parsing cannot be used or the parser has to make its decisions based on semantic information or a multi-symbol lookahead.

In this paper we suggest a systematic technique for resolving LL(1) conflicts in recursive descent parsing and show how to integrate it into a compiler generator (Coco/R). The idea is to evaluate user-defined boolean expressions, in order to allow the parser to make its parsing decisions where a one symbol lookahead does not suffice.

Using our extended compiler generator we implemented a compiler front end for C# that can be used as a framework for implementing a variety of tools.

1 Introduction

Recursive descent parsing [16] is a popular top-down parsing technique that is simple, efficient, and convenient for integrating semantic processing. However, it requires the grammar of the parsed language to be LL(1), which means that the parser must always be able to select between alternatives with a single symbol lookahead. Unfortunately, many languages such as Java, C++ or C# are not LL(1) so that one either has to resort to bottom-up LALR(1) parsing [5, 1], which is more powerful but less convenient for semantic processing, or the parser has to resolve the LL(1) conflicts using semantic information or a multi-symbol lookahead.

In this paper we suggest a systematic technique for resolving LL(1) conflicts in recursive descent parsing and show how to integrate it into an existing compiler generator (Coco/R). Section 2 gives a brief overview of Coco/R. In Section 3 we explain the LL(1) property of grammars and show how to resolve certain LL(1) conflicts by grammar transformations. For those conflicts that cannot be eliminated, Section 4 introduces our conflict resolution technique, which is based on evaluation of user-defined boolean expressions. Section 5 describes a C# compiler front end built with our extended version of Coco/R, while Section 6 discusses related work. Finally, Section 7 summarizes the results and points out future plans.

2 The Compiler Generator Coco/R

Coco/R [8] is a compiler generator, which takes an attributed grammar and produces a scanner and a recursive descent parser from it. The user has to add classes (e.g. for symbol table handling, optimization and code generation), whose methods are called from the semantic actions of the attributed grammar.

Attributed grammars were introduced by Knuth [6] as a notation for describing the translation of languages. Originally they were declarative in nature. However, we use them as procedural descriptions. In this form, an attributed grammar consists of the following parts:

- A *context-free grammar* in EBNF as introduced by Wirth [15]. It describes the syntax of the language to be processed.
- *Attributes* that can be considered as parameters of the nonterminal symbols in the grammar. Input attributes provide information from the context of the nonterminal while output attributes deliver results computed during the processing of the non-terminal. In Coco/R attributes are enclosed in angle brackets (<...>).
- *Semantic actions*, which are statements in an imperative programming language (e.g. C#) that are executed during parsing. They compute attribute values and call methods, e.g. for symbol table handling or code generation. In Coco/R semantic actions are enclosed by (. and .).

Every production of an attributed grammar is processed from left to right. While the syntax is parsed the semantic actions are executed where they appear in the production. Here is an example of a production that processes variable declarations:

```
VarDeclaration            (. Structure type; string name; .)
= Type<out type>
  Ident<out name>         (. SymTab.Enter(name, type); .)
  { "," Ident<out name>   (. SymTab.Enter(name, type); .)
  } ";".
```

In order to increase the readability, the syntax parts are written on the left-hand side while semantic actions are written on the right-hand side. Curly braces ({...}) denote repetition, square brackets ([...]) denote optional parts, and vertical bars (|) separate alternatives. Coco/R translates the above production into the following recursive descent parsing routine where `sym` is the statically declared lookahead symbol:

```
static void VarDeclaration() {
  Structure type; string name;
  Type(out type);
  Ident(out name); SymTab.Enter(name, type);
  while (sym == Tokens.comma) {
     Scanner.Scan();
     Ident(out name); SymTab.Enter(name, type);
  }
  Expect(Tokens.semicolon);
}
```

Further information on how to use Coco as well as its source code and the executable can be obtained from [9, 14].

3 LL(1) Conflicts

A grammar is said to be LL(1) (i.e., parsable from **L**eft to right with **L**eft-canonical derivations and **1** symbol lookahead), if at any point, where the grammar allows a selection between two or more alternatives, the set of terminal start symbols of those alternatives are pairwise disjoint.

In EBNF grammars, there are the following three situations where LL(1) conflicts can occur (greek symbols denote arbitrary EBNF expressions such as `a[b]C`; *first*(α) denotes the set of terminal start symbols of the EBNF expression α; *follow*(A) denotes the set of terminal symbols that can follow the nonterminal A):

- **Explicit alternatives**
  e.g. $A = \alpha \mid \beta \mid \gamma$. check that *first*($\alpha$) $\cap$ *first*(β) = { } $\wedge$ *first*(α) $\cap$ *first*(γ) = { } $\wedge$ *first*(β) $\cap$ *first*(γ) = { }.
- **Options**
  e.g. $A = [\alpha]\ \beta$. check that *first*(α) $\cap$ *first*(β) = { }
  e.g. $A = [\alpha]$. check that *first*(α) $\cap$ *follow*(A) = { }
- **Iterations**
  e.g. $A = \{\alpha\}\ \beta$. check that *first*(α) $\cap$ *first*(β) = { }
  e.g. $A = \{\alpha\}$. check that *first*(α) $\cap$ *follow*(A) = { }

Resolving LL(1) Conflicts

Factorization. LL(1) conflicts can usually be resolved by factorization, i.e. by extracting the common parts of conflicting alternatives and moving them to the front. For example, the production

```
A = a b c | a b d.
```

can be transformed to

```
A = a b (c | d).
```

Left recursion. Left recursion always represents an LL(1) conflict. In the production

```
A = A b | c.
```

both alternatives start with `c` (because *first*(`A`) = {`c`}). However, left recursion can always be transformed into an iteration, e.g. the previous production becomes

```
A = c {b}.
```

Hard conflicts. Some LL(1) conflicts cannot be resolved by grammar transformations. Consider the following (simplified) production taken from the C# grammar:

```
IdentList = ident {"," ident} [","].
```

The conflict arises, because both the iteration and the option can start with a comma. There is no way to get rid of this conflict by transforming the grammar. The only way to resolve it, is to look ahead two symbols in order to see what follows after the comma. We will deal with this problem in Section 4.

Readability issues. Some grammar transformations can degrade the readability of the grammar. Consider the following example (again taken from a simplified form of the C# grammar):

```
UsingClause = "using" [ident "="] Qualident ";".
Qualident   = ident {"." ident}.
```

The conflict is in `UsingClause` where both the option and `Qualident` start with `ident`. Although this conflict could be eliminated by transforming the production to

```
UsingClause = "using" ident ( {"." ident}
                            | "=" Qualident
                            ) ";".
```

the readability would clearly deteriorate. It is better to resolve this conflict as shown in Section 4.

Semantic issues. Finally, factorization is sometimes inhibited by the fact that the semantic processing of conflicting alternatives differs, e.g.:

```
A = ident (. x = 1; .) {"," ident (. x++; .) } ":"
  | ident (. Foo(); .) {"," ident (. Bar(); .) } ";".
```

The common parts of these two alternatives cannot be factored out, because each alternative has its own way to be processed semantically. Again this problem can be solved with the technique explained in Section 4.

4 Conflict Resolvers

This section shows how to resolve LL(1) conflicts by so-called *conflict resolvers*, which are in a similar form also used in other compiler generators (e.g. JavaCC, ANTLR, see Section 6 for a description of their approaches).

A conflict resolver is a boolean expression that is inserted into the grammar at the beginning of the first of two conflicting alternatives and decides by a multi-symbol lookahead or by semantic checks, whether this alternative matches the actual input. If the resolver yields true, the alternative is selected, otherwise the next alternative will be checked. A conflict resolver is given in the form

```
Resolver = "IF" "(" {ANY} ")" .
```

where `{ANY}` means an arbitrary piece of code that represents a boolean expression. In most cases this will be a method call that returns true or false.

Thus we can resolve the LL(1) conflict from Section 3 in the following way:

```
UsingClause = "using" [IF(IsAlias()) ident "="]
              Qualident ";".
```

`IsAlias` is a user-defined method that reads two symbols ahead. It returns true, if ident is followed by "=", otherwise false.

4.1 Multi-symbol Lookahead

The Coco/R generated parser remembers the last recognized terminal symbol as well as the current lookahead symbol in the following two global variables:

```
Token t;     // last recognized terminal symbol
Token la;    // lookahead symbol
```

In case one wants to look ahead more than just one symbol, the Coco/R generated scanner now offers the following two methods to do this:

- *StartPeek()* initializes peeking in the scanner by synchronizing it with the position after the current lookahead symbol.
- *Peek()* returns the next symbol as a *Token* object. The symbols returned by *Peek* are not removed from the input stream, so that the scanner will deliver them again when "normal" parsing resumes.

Using these methods we can implement *IsAlias* in the following way:

```
static bool IsAlias () {
  Scanner.StartPeek();
  Token x = Scanner.Peek();
  return la.kind == Tokens.ident && x.kind == Tokens.eql;
}
```

The last conflict mentioned in Section 3 can be resolved by the grammar rule

```
A = IF(FollowedByColon())
      ident (. x = 1; .) {"," ident (. x++; .) } ":"
    | ident (. Foo(); .) {"," ident (. Bar(); .) } ";".
```

and the following implementation of the function *FollowedByColon*:

```
static bool FollowedByColon () {
  Scanner.StartPeek();
  Token x = la;
  while (x.kind == Tokens.comma || x.kind == Tokens.ident)
    x = Scanner.Peek();
  return x.kind == Tokens.colon;
}
```

4.2 Conflict Resolution With Semantic Information

A conflict resolver can base its decision not only on the result of checking arbitrarily many lookahead symbols but any other kind of information as well. For example it could access a symbol table or other semantic information.

The following LL(1) conflict between assignments and declarations can be found in many programming languages:

```
Statement = Type IdentList ";"
          | ident "=" Expression ";"
          | … .
Type      = ident | … .
```

Both of the shown alternatives of the *Statement* rule begin with *ident*. This conflict can be resolved by checking whether ident denotes a type or a variable:

```
Statment = IF(IsType()) Type IdentList ";"
         | ident "=" Expression ";"
         | … .
```

IsType looks up *ident* in the symbol table and returns true, if it is a type name:

```
static bool IsType () {
  if (la.kind == Tokens.ident) {
    object obj = SymTab.find(la.val);
    return obj != null && obj.kind == Type;
  } else return false;
}
```

4.3 Translation of Conflict Resolvers into the Parser

Coco/R treats conflict resolvers similarly to semantic actions and simply copies them into the generated parser at the same position where they appear in the grammar. The production for the *UsingClause* from above is translated into the following parser method:

```
static void UsingClause () {
  Expect(Tokens.using);
  if (IsAlias()) {
    Expect(Tokens.ident); Expect(Tokens.eql);
  }
  Qualident();
  Expect(Tokens.semicolon);
}
```

4.4 A Different Approach: Resolver Symbols

In a previous version of Coco/R, we have examined a different approach to resolve LL(1) conflicts: artificial tokens—so-called *resolver symbols*—were inserted into the input stream on-the-fly (at parse time) in order to guide the parser along the right way. The symbols had to be placed properly in the grammar rules. They were defined in a separate section of the grammar (keyword: `RESOLVERS`) and combined with a resolution routine that determined, whether the resolver symbol was needed. If so, the parser inserted it into the input stream. Coco/R put the invocations of the resolution routines automatically at the right places in the generated parser.

Comparing the two approaches, we find that the current conflict resolver version (`IF(…)`) requires less implementation changes compared to the original Coco/R (without conflict resolution), is less complex and appears to be easier to understand from the users perspective than the insertion of artificial tokens at parse time.

5 A Compiler Front End for C#

In a project supported by Microsoft under the Rotor initiative [11] we used Coco/R and its conflict resolution technique for building a compiler framework for C# that can be used to build compilers, source code analyzers and other tools. Our framework consists of an attributed C# grammar to which the user can attach semantic actions as required for the tool that he is going to implement. From this description Coco/R generates a scanner and a parser that also contains the attached semantic actions.

We started out by downloading the C# grammar from the ECMA standardization page [12]. Since this grammar was designed for being processed by a bottom-up parser, it is full of left recursions and other LL(1) conflicts. Some of these conflicts could be resolved by grammar transformations as discussed in Section 3, the others were eliminated using conflict resolvers. The resulting grammar is available at [17 or 3].

One of the first applications that have been developed with our framework is a white box testing tool [7] that instruments a C# program with counters in order to get path coverage information. Another application analyzes C# source code to provide helpful hints as to where performance optimizations could be applied or coding style rules have been violated [13]. Other applications that could be implemented with our framework are for example:

- Profilers that measure the execution frequencies or execution times of C# programs.
- Source code analyzers that compute complexity measures or data flow information from C# programs.
- Pretty printers that transform a C# program according to certain style guidelines.
- Proper compilers that translate C# programs to IL, machine code, Java bytecodes, or to XML schemata.

6 Related Work

Recursive descent parsers are usually written by hand, because this is straightforward to do. In manually written parsers it is common to resolve LL(1) conflicts by semantic information or by multi-symbol lookahead. However, these are only ad-hoc solutions and do not follow a systematic pattern as we described it in Section 4.

Compiler generators, on the other hand, usually generate bottom-up parsers that use the more powerful LALR(1) technique instead of LL(1) parsing. In bottom-up parsers, however, it is more difficult to integrate semantic processing, because semantic actions can only be performed at the end of productions.

Recently, there seems to be a growing interest in compiler generators that produce recursive descent parsers. In the following subsections we will describe two of these generators and their LL(1) conflict resolution strategies.

6.1 JavaCC

JavaCC [4] is a compiler generator that was jointly developed by Metamata and Sun Microsystems. It produces recursive descent parsers and offers various strategies for resolving LL(1) conflicts.

First, one can instruct JavaCC to produce an LL(k) parser with $k > 1$. Such a parser maintains k lookahead symbols and uses them to resolve conflicts. Many languages, however, not only fail to be LL(1), but they are not even LL(k) for an arbitrary, but fixed k. Therefore JavaCC offers local conflict resolution techniques. A local conflict resolver can be placed in front of alternatives in the same way as in Coco/R. The following syntax is available (greek letters denote arbitrary EBNF expressions):

- (LOOKAHEAD(k) α | β) tells the parser that it should look ahead k symbols in the attempt to recognize an α.
- (LOOKAHEAD(γ) α | β) allows the user to specify an EBNF expression γ. If the next input symbols match this expression, then α is selected.
- (LOOKAHEAD(*boolExpr*) α | β) allows the user to specify a boolean expression (which can also be the call to a function that returns true or false). If the expression evaluates to true, then α is selected. This is the only strategy currently supported by Coco/R.

So JavaCC offers more ways of how to resolve LL(1) conflicts than does Coco/R. Other differences are:

- JavaCC produces parsers in Java while Coco/R produces parsers in C#. Furthermore, the new conflict resolution feature will be incorporated in the Java version of Coco/R and a complete port to VB.NET is also on the agenda.
- Coco/R uses Wirth's EBNF [15], while JavaCC uses an EBNF notation that is similar to regular expressions.

6.2 ANTLR

ANother Tool for Language Recognition (ANTLR) [2, 10] is part of the Purdue Compiler Construction Tool Set (PCCTS), which includes also SORCERER (a tree parser generator) and DLG (a scanner generator). ANTLR generates recursive descent parsers from so-called "pred-LL(k)" grammars, where "pred" indicates that syntactic or semantic predicates are used to direct the parsing process. The current version of ANTLR can produce parsers in Java, C++, and C#.

Syntactic predicates allow the user to specify a syntax expression in front of an alternative. If this expression is recognized, the alternative is selected (this is similar to JavaCC's syntactic LOOKAHEAD feature (the second in the above list)):

```
stat: (list "=" )=> list "=" list
    |               list ;
```

Semantic predicates are used in two ways: Firstly, for checking context conditions, throwing an exception, if the conditions are not met:

```
decl: "var" ID ":" t:ID
      { isTypeName(t.getText()) }? ;
```

Secondly, they are used for selecting between alternatives:

```
stat: { isTypeName(LT(1)) }?
      ID ID ";"              // declaration (Type ident;)
    | ID "=" expr ";" ;      // assignment
```

Both cases have the same syntax. They are only distinguished by the position of the predicate: disambiguating predicates must appear at the beginning of an alternative.

ANTLR's approach is similar to that of JavaCC and Coco/R. All evaluate boolean expressions in order to guide the parsing process, i.e., the user is given control to define the conditions under which an alternative shall be selected.

7 Summary

In this paper we showed how to resolve LL(1) conflicts systematically by using boolean expressions—so-called *conflict resolvers*—that are evaluated at parse time in order to guide the parser along the right path. In this way the parser can exploit either semantic information or on-demand multi-symbol lookahead in its decision-making process. We also described the integration of our technique into the compiler generator Coco/R.

The work described in this paper is ongoing research. In the future we plan to add more resolution strategies to Coco/R in a similar way as it is done for JavaCC. For example, we plan to implement a feature that does an LL(k) lookahead without requiring the user to specify a resolver routine. We hope that it will even be possible to let Coco/R decide, at which points a k-symbol lookahead is necessary and to do it automatically without any intervention of the user. Finally, since the original version of Coco/R is available for a variety of languages such as C#, Java, C++ and others, we plan to port our conflict resolution technique from the C# version of Coco/R also to the other versions and offer a new VB.NET version. The current state of our project as well as downloads of the source code and the executables of Coco/R are available at [17, 3].

Acknowledgements

This work was supported by Microsoft under the Rotor initiative [11].

References

1. Aho A.V., Ullman J.D.: Principles of Compiler Design, Addison-Wesley, 1977
2. ANTLR (ANother Tool for Language Recognition). http://www.antlr.org
3. Coco Tools Project at the Rotor Community Site, http://cocotools.sscli.net

4. JavaCC™ (Java Compiler Compiler), The Java Parser Generator. http://www.experimentalstuff.com/Technologies/JavaCC
5. Knuth, D.E.: On the Translation of Languages from Left to Right. Information and Control, vol. 6, pp.607-639, 1965
6. Knuth, D.E.: Semantics of Context-free Languages. Mathematical Systems Theory, vol 2, pp.127-145, 1968
7. Löberbauer, M.: Ein Werkzeug für den White-Box-Test. Diploma thesis, Institute of Practical Computer Science, University of Linz, Austria, February 2003
8. Mössenböck, H.: A Generator for Production Quality Compilers. 3rd intl. workshop on compiler compilers (CC'90), Schwerin, Lecture Notes in Computer Science 477, Springer-Verlag, 1990, pp. 42-55.
9. Mössenböck, H.: Coco/R for various languages - Online Documentation. http://www.ssw.uni-linz.ac.at/Research/Projects/Coco/
10. Parr, T.: Language Translation Using PCCTS and C++. Automata Publishing Company, 1993.
11. Rotor: The Microsoft Rotor Project, http://research.microsoft.com/collaboration/university/europe/rotor/
12. Standard ECMA-334, C# Language Specification, December 2001, http://www.ecma-international.org/publications/standards/ecma-334.htm
13. Steineder, K.H.: Optimization Hint Tool. Diploma thesis, Institute for Practical Computer Science, University of Linz, Austria, August 2003.
14. Terry, P.: Compilers and Compiler Generators – An Introduction Using C++. International Thomson Computer Press, 1997.
15. Wirth, N.: What Can We Do about the Unnecessary Diversity of Notation for Syntactic Definitions? Communications of the ACM, November 1977
16. Wirth, N.: Compiler Construction. Addison-Wesley, 1996.
17. Wöß, A., Löberbauer, M.: SSCLI-Project: Compiler Generation Tools for C#. http://dotnet.jku.at/Projects/Rotor/