AnLex and AnSin: A Compiler Generator System for Beginners

Marta Patiño-Martínez, Ricardo Jiménez-Peris, J. Ignacio Castelló-Gómez
Universidad Politécnica de Madrid
Depto. de Lenguajes y Sistemas Informáticos, Facultad de Informática
Campus de Montegancedo s/n, 28660 Boadilla del Monte, Madrid, Spain
{mpatino, rjimenez}@fi.upm.es, nacho@camel.tid.es

Abstract

The study of compiler generators is an integral part of compiler construction, and for this reason it is customary to have a programming project entirely devoted to it in compiler courses. There are many compilers generators, but their use in a compiler course presents several problems (e.g. the parsers generated are difficult to understand and to debug). In this paper, we describe such problems and present a compiler generator system, AnLex-AnSin, that solves these problems, and can thus be used in compiler programming projects.

1. Introduction

Programming a compiler is not an easy task. The different parts of a compiler are heavily coupled. The size of a compiler is usually too big for student to handle, and it takes a long time to program. These problems are bigger when students face tools that are not designed for them.

To acomplish these problems, some authors [8] propose the use of compiler writing tools such as Lex [5] and Yacc [4] in compiler programming projects. Although the use of these tools or similar ones can be adequate for large projects, it may not be the best choice in an educational environment. In particular, there is significant overhead when learning Lex and Yacc effectively; and the implementation environment is tied to C and (most likely) Unix [2]. Due to the difficulties in debugging and understanding Yacc generated parsers, some authors [6] have developed tools to visualize them in order to ease their understanding. Instead of C, a high-level modular language is desirable for writing a compiler. Modula-2 has been proposed as an adequate implementation language [3].

SIGCSE '97 CA, USA
© 1997 ACM 0-89791-889-4/97/0002...$3.50

Debugging can be a long and time consuming task, especially if there is no correspondence between what the programmer writes (a grammar with annotated semantic actions) and the generated compiler [2]. This is a very common problem found in most of the parser generators. This problem is even worse with tabular parsers, where the syntax analysis and thus, the flow of the parser is codified in a data structure traversed by an algorithm generated by the parser generator. For instance, the GNAT Ada-95 compiler uses a recursive descent parser to increase the legibility and understability of the compiler.

A different approach [8] consist in providing students with supplementary material in order to allow them to write a compiler in a single-term course, freeing them of routinary programming tasks. In particular, the material provided in [8] consists of a unique label generator, a symbol-table management module and a string concatenation function.

We have developed AnLex-AnSin, a compiler generator system designed to be used in a compiler course, that generates compilers easy to debug and understand. This is achieved by using a high-level language, Modula-2, as the implementation language, generating recursive descent parsers and using a parser description language that is powerful, easy to understand and use. Furthermore, a set of libraries is provided to ease the compiler programming task.

In section 2, the problems found in other scanner and parser generators are discussed. AnLex, the scanner generator and AnSin, the parser generator, are presented in sections 3 and 4, respectively. Our conclusive remarks are found in section 5.

2. Difficulties with Scanner and Parser Generators

Although Lex and Yacc are very powerful tools to generate compilers, they are not adequate for compiler courses. In this section, we point out some of the problems these tools have when used by students in compiler-writing projects.

- One of the problems is that the implementation language is C, and thus the generated code inherits the difficulties of debugging C programs. This problem can be fixed by using a tool that generates code in a high-level language with modular facilities.
- Another difficulty is the fact that the code corresponding to semantic actions cannot be easily separated from the syntax rules in the parser. It would be interesting to write the semantic actions without including code of the implementation language. In this way the parser description (syntax definition and semantic actions) would result in a more legible and thus easier to debug code.
- Yacc is based on LALR grammars. Although LALR grammars are more powerful than LL grammars, they are more difficult to understand (and debug). In particular, the cases that cannot be parsed with LL parsers usually correspond to errors due to misunderstandings of the grammar.
- Another problem with LALR grammars is that it is difficult to propagate inherited attributes through different rules.
- Tabular parser generators, like Yacc, are more efficient that recursive descent ones, but are more difficult to debug, because the structure of the compiler is codified in a data structure. Recursive descent parsers are easier to debug as the syntax analysis is translated into implementation language code; debuggers are more suited to debug this kind of parsers. This approach has been proposed by some authors [11].
- To take full advantage of Yacc, one has to deal with very low level features of C [10], with the associated dangers. As compilers are difficult to write, it is better to avoid the use of low-level programming as it complicates debugging a lot.
- Lex and Yacc do not incorporate auxiliary libraries to help in the compiler construction task, and thus, the student has to write a lot of code. A system providing a set of auxiliary libraries would allow the student to focus on the problems of the compiler [8].
- Finally, the lack of correspondence between the code written by the programmer and the behaviour of the generated parser complicates the debugging process even more.

Our approach is to provide a tool that solves these problems, generating descent recursive parsers in a modular high-level language, in particular, Modula-2; providing the student with a set of auxiliary libraries, allowing him or her to concentrate on the essence of the compiler. Both, the choice of the implementation language, and the kind of parser, are aimed to facilitate the debugging of the generated parser which is one of the main difficulties encountered by students. Another reason for choosing Modula-2 is that it is the main language used throughout the degree program in our university.

In our tool, most of the checkings are done by the parser generator, and not are delegated to the implementation language compiler. Thus, the student does not have to search for most of the errors in the generated parser, but in the source description. The relationship between the parser and its description is more direct than in tabular parsers, so whenever an error is found in the generated parser, it will be easier to find its corresponding location in the source parser description. What is more, because it is a high-level language, the compiler detects more errors than would have been picked up by the debugging process.

3. AnLex: A Scanner Generator

AnLex is a scanner generator designed to be used in conjuction with the parser generator AnSin. It generates Modula-2 code, although it can be configured to generate code in any other language. We have chosen for AnLex a syntax similar to Alex's [7], due to the latter's clarity. But otherwise, the two generators are very different.

A scanner description in AnLex is very simple. It consists of several sections, each one describing some aspect of the scanner. The structure of a scanner description in AnLex is as follows:

```
SCANNER scannerName
  [ CASE SENSITIVE (ON | OFF) ]
  CHARACTER SETS {SetDeclaration}
  KEYWORDS {KeywordDeclaraction}
  TOKEN CLASSES {TokenDeclaration}
  SINGLE TOKENS {SingleTokenDeclaration}
  BLANKS = SetExpression
END
```

The first section, CASE SENSITIVE, indicates whether capital letters in identifiers and keywords are significant or not.

The second section allows for the declaration of set constants to ease the token class declarations. It is also possible to define sets in terms of other sets using set operations. ANY is a predefined set containing all non control and non graphical characters. Some examples of character set declarations are:

```
letters = 'A'..'Z' + 'a'.. 'z'.
digits = '0123456789' .
hexDigits = digits + 'ABCDEF' .
quote = "'" .
noQuote = ANY - quote .
extended = CHR(128)..CHR(255) .
```

The first four examples are self-explanatory. The fifth one defines the set of non graphical printable characters

excluding the single quote. The last example defines the set of the extended characters. Notice that it is possible to represent a character by its ASCII code.

Keywords are defined in the KEYWORDS section. In order to be referenced in the parser, they must have an associated identifier. That identifier is generated, by default, by a naming scheme preventing to introduce each keyword twice. The associated identifier is equal to the string it represents, but this name can be changed if needed. For instance:

```
start = 'BEGIN'          -- The name will be start.
'END'                    -- The name will be end.
```

Tokens are generated as an enumerated type, including keywords. This prevents to misuse a token in an arithmetic expression, due to the strong typing of the implementation language

The TOKEN CLASSES section is useful to describe tokens that can take different lexemes (e.g. an integer literal, 23, -235, etc.). Each token class is defined by means of a regular expression using the EBNF notation [1]. Some examples are:

```
integer = [ '+' | '-'] digit {digit}.
ident = letter {letter | digit}.
realLit = digit {digit} '.' {digit}
                ['E'['+' | '-'] digit {digit}]
```

These examples define the lexical aspects of integer and real literals, as well as identifiers in Modula-2.

In some instances, some characters are needed to delimit a token, but they are not wanted in the lexeme. To exclude such characters from the lexeme they must appear enclosed between <, >. For instance:

```
stringLit = <quote> {noQuote} <quote>
```

In this example, the quotes delimiting the string literal will not be included in the lexeme.

In case of ambiguity, scanners usually follow the greedy criterion, which consists in taking the longest token. (e.g. 1..2 could be the real 1. or the integer 1 and the subrange token, ..). AnLex is not an exception, but provides a mechanism to change this criterion: the IF FOLLOWED BY clause. For instance:

```
natLit = digit {digit} IF FOLLOWED BY ("." ".").
realLit = digit {digit} "." {digit}.
```

Another ambiguity happens when keywords are included by another token class, usually the identifier class. To prevent this, AnLex provides the clause EXCEPT KEYWORDS, that gives preference to keywords in case of collision. For example:

```
ident = letter {digit | letter} EXCEPT KEYWORDS.
```

In the SINGLE TOKENS section, the tokens with only one possible lexeme are declarated. Examples of this are:

```
plus = '+'
assignment = ':='
ge = '>='
not = CHR(126)
```

Separator characters are defined in the BLANKS section. There are predefined character constants to help in this declaration, as can be seen in the following example:

```
BLANKS = ' ' + TAB + CR + LF.
```

Finally, comment delimiters are declared in the COMMENTS section. For instance:

```
COMMENTS    FROM '(' '*' TO '*' ')' NESTED
            FROM '-' '-' TO LF
```

In the first line, MODULA-2-like comments are defined, that is between the delimiters (* and *). Notice that the clause NESTED allows to indicate that those delimiters can be nested. In the second one, Ada-like ones are defined. These are line comments that start with the delimiter -- and finish at the end of the line.

As previously mentioned, AnLex can be configurated to generate scanners in any language. This is achieved with a language that describes how to traverse the data structures describing the automaton, and which code to generate during the traversal, but the description of this language is out of the scope of this paper. This feature can be of interest to practise in the writing of scanner generators.

The structure of AnLex has been presented. It is a concise, simple and fairly easy to use. Students just have to identify the lexical aspects of the source language and describe them. This makes AnLex a powerful tool, adequate to program compiler projects.

4. AnSin: A Parser Generator

AnSin is a parser generator that incorporates a language to describe parsers. The structure of a parser description in AnSin is as follows:

```
PARSER parserName
  SEMANTICS { SemanticDeclaration }
  TERMINALS TerminalsDeclaration
  NONTERMINALS NonTerminalsDeclaration
  RULES GrammarRules
END
```

First of all, semantic actions to be used must be declared in the SEMANTICS section. In other systems, semantic actions are usually defined as a literal string containing the code to be generated. There is no checking on this piece of code as it is treated as a string literal. In our system, calls to semantic actions are AnSin code, and thus, they are checked, that is, to use a semantic action, it must be previously declared. This helps to structure the parser description, not allowing to introduce an arbitrary piece of code, and what is more, it helps to keep a clean description, since only the grammar and calls to semantic actions appear. In this way, a grammar rule in the compiler description will contain just the rule and calls to semantic actions.

Each declaration indicates the module where the semantic actions are implemented, the types used, the action names, and the initialization and finalization actions (if any). The type declaration allows the use of a type in the attribute declaration of non terminals. For instance, a semantic declaration for a symbol table management could be:

```
MODULE "SymbolTable" INIT "InitSymbolTable".
 TYPE EntryType.
 ACTION NewEntry, LookupEntry, NewScope, EndScope.
```

The call to the initialization action is done automatically by the code generated by AnSin, and it is warranted that this action will take place before any other action of the module is called. This allows to keep a clear description of the parser and avoids to use tricks (more difficult to understand and debug) to execute such actions. The EntryType will be useful to declare attributes of that type.

In the TERMINALS section, the terminals to be used are declared. It is usually included in the terminal list generated by AnLex (with the INCLUDE command).

The NONTERMINALS section is where non terminals used in the grammar are declared. Not only the non terminal names are enumerated, but also their interface is specified, that is, name and type of inherited and synthesized attributes. As previously mentioned, action and attribute types must be declared in the SEMANTICS section. This section enhances the legibility of the parser description because the type and name of the attributes are documented and checked by the parser generator, AnSin.

Another benefit is that AnSin checks that actual attributes correspond to formal ones (as subprogram parameters). This kind of errors is not delayed until the generated parser is compiled and thus marked in its code. On the contrary, they are shown in the source description of the parser as the result of its compilation by AnSin. This is very helpful as it prevents, in most cases, to edit the generated parser to search the error there, and to look for it in the parser description. For instance, the non terminal corresponding to an expression could be declarared as follows:

```
Expression  WITH < IN ExpectedType : StringType,
                   OUT Expr : GenTreeType > .
```

where 'ExpectedType' is an inherited attribute and 'Expr' a synthetized one. 'StringType' and 'GenTreeType' are ADTs from the set of libraries of the system.

Finally, the grammar of the language to be recognized and the semantic actions to be taken are defined in the RULES section. Rules are written using the EBNF notation that helps to keep them simple and improves their legibility. In particular it is not necessary to decompose a rule when it specifies a list of items, as can be seen in the next example. For instance, the rule for a Pascal subprogram header would be:

```
<Header> ::=
  (function ident [<ArgumentList>] dots <SimpleType |
        procedure ident [<ArgumentList>]) semicolon .
```

Calls to semantic actions are inserted in the rule definition. There is a predefined semantic action, '<-' that assigns values to attributes.

The following example shows how semantics actions are inserted:

```
<Type> ::=
  <StandardType> !! type <- StandardType.stType !!
  | array opBr num subrng num clBr of <StandardType>
  !! type <- BuildArray(StandardType.stType,
        num[1].NUMERIC, num[2].NUMERIC);
  !!
```

In this example, semantic actions appear between !!. Nonterminals appear between < and >. Dot notation is used to access inherited and synthetized attributes, as well as terminal lexemes. In the examples 'type' and 'stType' are synthetized attributes of the nonterminals <Type> and <StandardType>, respectively. NUMERIC accesses a terminal lexeme (a string), but returns it converted to a cardinal type. In the case that a terminal or a nonterminal is used more than once in a rule, an index is used to identify it. For example, 'num[1]' and 'num[2]' access the lexeme of the first and the second instance of the terminal num in the rule. BuildArray is a user semantic action that must be declared prior to its use.

Semantic actions are designed and written by the compiler programmer, but most of them are very repetitive and are just routine for a compiler course student, who ends up spending a lot of time writing and debugging the code, which compels the student to wander away from the essence of the compiler. Our approach is to provide a set of libraries that help the student with routinary tasks, allowing him or her to concentrate on the main aspects of the compiler. The set of libraries provided by our system (designed to be used as semantic actions) are the following generic ADTs: dynamic strings, lists, general trees and hash tables. These ADTs are guarded against not initialization [9], which saves a lot of debugging time. In addition, there is an output management module, and a simple error management module. The source code is made available to students, so they can extend and modify it, if needed.

Dynamic strings can be of arbitrary size and the most frequent operations are provided. In fact, the code generated by AnSin uses this ADT (e.g. in the strings corresponding to lexemes).

The ouput module allows output redirection to a file or a dynamic string, as well as conversion from simple types to dynamic strings. The redirection to a dynamic string is useful when output has to be delayed because the previous output is not ready. In this case, the output can be redirected to a dynamic string and written later to the output file, when possible. It is also well suited for programming preprocessors.

The error management module generates error files according to the Borland format, so students can use a Borland environment to run their compiler and see errors generated by their compiler in an editor window, marked in the source code.

5. Conclusions

A compiler-writing system designed for students has been presented, AnSin-AnLex. It is easy to learn and use. It generates compilers easy to debug, allowing students to concentrate on the principles of compilers without being distracted by peripheral issues. The included set of libraries facilitates the whole process. AnLex-AnSin has been used effectively in our compiler course.

The features of our system can be summarized in this list:

- The implementation language is a high-level one, Modula-2, thus students do not spend time fixing errors produced y the use of low level features.
- The compiler description is independent from the implementation language. This helps to separate the parser description from the implementation of the semantic actions. This also means that most errors in the description will be detected by the compiler genarator instead of the implementation language compiler, which eases compiler writing and debugging.
- The use of LL grammars prevents the use of difficult to debug and understand grammars, as is the case with LR grammars..
- As recursive descent parsers are generated, they can be more easily debugged with usual debuggers than tabular ones.
- The set of libraries provided, helps to concentrate in compiler essentials and not in the building of data structures (that students already know).

The code of the generated compiler keeps the structure of the parser description, so it is very easy to relate one to another. This is very helpful especially when an error is searched in the code of the generated parser.

Acknowlegments

We want to thank to Sami Khuri and Ángel Velázquez for their comments and help.

References

1. A. V. Aho, R. Sethi and J. D. Ullman, *Compilers, Techniques, and Tools.* Addison-Wesley, 1985.
2. E. F. Elsworth, The MSL Compiler Writing Project. *SIGCSE Bulletin 24*, 2 (June 1992), 41-44.
3. E. F. Elsworth, Modula-2 in a Compiler Writing Course. *Proceedings of the 1st European Modula-2 Conference*, Polytechnic of Wales, Pontypridd, 1990.
4. S. C. Johnson, *Yacc - yet another compiler compiler.* Computing Science Technical Report 32, ATT Bell Laboratories, Murray Hill, 1975.
5. M. E. Lesk. *Lex - a lexical analyzer generator.* Computing Science Technical Report 39, ATT Bell Laboratories, Murray Hill, 1975.
6. M. E. Lovato and M. F. Kleyn. Parser Visualizations for Developing Grammars with Yacc. *Proceedings of the 25th SIGCSE Technical Symposium on Computer Science Education*, 1995, pp. 345-349.
7. H. Mossenbock. Alex - A Simple and Efficient Scanner Generator. *SIGPLAN Notices 21*, 12 (Dec. 1986), 139-148.
8. R. J. Reid. *A Toolkit for Individualized Compiler-Writing Projects.* Proceedings of the 20th SIGCSE Technical Symposium on Computer Science Education, 1990, pp. 81-85.
9. J. Savit. Uninitialized Modula-2 Abstract Objects, Revisited. *SIGPLAN Notices 22*, 2 (Feb. 1987), 78-84.
10. A. T. Shreiner and H. G. Friedman, Jr., *Introduction to Compiler Construction with Unix.* Prentice-Hall, 1985.
11. D. A. Watt, *Programming Language Processors.* Prentice-Hall, 1993.