FACTA UNIVERSITATIS (NIŠ)
SER.: ELEC. ENERG. vol. 18, No. 2, August 2005, 299-307

Parsing in Different Languages

Miroslav D. Ćirić and Svetozar R. Rančić

Abstract: A compiler is a translator that accepts as input formatted source file or files, and produces as output a file that may be run directly on a computer. Given the same ANSI C++ compliant input file, two different ANSI C++ compliant compilers running on the same operating system produce two different executable programs that should execute in exactly the same way. To some degree, this is achieved by the standardization of the C++ language, but it is also possible because computer programming languages like C++ can be compiled using reliable technologies with long traditions and understood characteristics. LALR(k), as practical version of LR, is such reliable technology for parsing. Traditional LALR(1) tool YACC has proved his value during years of successful applications. Nowadays there are a few commercial and noncommercial alternatives that are very interesting and promising. This paper will examine some of the them with ability of parsing in different programming languages.

Keywords: Parsing, LALR(1), programming languages.

1 Introduction

Although compilers are not the only computer applications that make use of parsing technology, this section will briefly explain compiler architecture in order to make clear where lexical analysis, parsing, and semantic analysis fit into the bigger picture of compilation. After a source file has been preprocessed so that it contains only those symbols that are properly part of the programming language being compiled, it is broken into discrete tokens. This task has been accomplished by lexical analyzer or scanner. As a result we have a stream of tokens such as constants, variable names, keywords and operators. In essence, the lexical analyzer performs low-level syntax analysis. For efficiency reasons, each class of tokens is given a

Manuscript received

Miroslav D. Ćirić and Svetozar R. Rančić are with the Faculty of Sciences and Mathematics, University of Niš, Višegradska 33, 18000 Niš, Serbia and Montenegro (e-mail: `mciric@pmf.ni.ac.yu` and `rancic@pmf.ni.ac.yu`).

unique internal representation number. Also, some scanners place constants, labels and variable names in appropriate tables. The lexical analysis supplies tokens to the syntax analyzer. These tokens may take the form of pair of items. The first item may give the representation number of the token, while the second item is the address or the location of the token in some table. Both of them may also be wrapped in some structure. The syntax analyzer is much more complex than the lexical analyzer see [1] [2]. Its function is to take the source program in the form of tokens and determine whether or not a series of tokens satisfies the expressed syntactic rules of language. In syntax analysis we are concerned with grouping tokens into larger syntactic classes such as expression, statement or procedures. The syntax analyzer or parser outputs a syntax tree (or its equivalents) in which its leaves are the tokens and every nonleaf node represents a syntactic class type. The general description of the syntactic form of a language is called grammar. The syntax tree produced by syntax analyzer is used by the semantic analyzer. The function of semantic analyzer is to determine the meaning (or semantic) of the source program. The meaning is in term of operations on a computer. For a bit of code regardless of how a given computer achieves the desired results, has one semantic meaning. Although it is conceptually desirable to separate the syntax of the source program from its semantics, the syntax and semantic analyzers work in close cooperation.

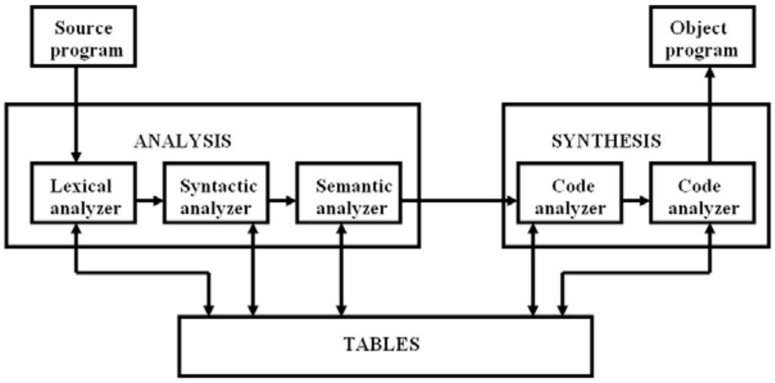


Fig. 1. Structure of a compiler

The semantics analyzer often interacts with the various tables of the compiler in performing its tasks. His actions may involve the generation of an intermediate form of the source code, as shown in Figure 1.

The output of the semantic analyzer is passed on to the code generator. Some high level clishés and constructs are very useful for making source code read clearly to other human beings, but to a computer, expressed inefficiently. Optimizing compilers take this into account and restructure some of these constructs in ways that allow the final program to be in optimal or near optimal form for the next stage of compilation, code generation. At this stage collection of many semantically correct intermediate representations of the source code is mapped into the target language of machine or some lower level intermediate abstract representation just above the level of the actual machine. The output of the code generator is passed to the code optimizer. This process is present in sophisticated compilers. Its purpose is to produce a more efficient object program, and to profile target code according to meet need of the code authors. Finally, output from code optimizer, in form of the target code is passed to the linker. Process of converting the generated target code into a format that is directly executable on a given machine, running a given operating system is known as linking.

2 Overview of parsing and tools

As already discussed lexical analysis, or tokenization, is the process of decomposing input text into discrete tokens. In general sense, a token is something that represents something else. The task of the tokenization is to scan through the characters of the source text and assemble tokens, either one at a time or all from file, and feed these to the parser. This is done by Finite State Automaton (FSA). Useful way to express tokens formally is by regular expression. Grammar is formal generative mechanism for describing language. Standardized and formal form for defining grammars of class 2 in the Chomsky Grammar Hierarchy is Backus-Naur form. Expressed purely in Backus-Naur form grammars do not specify how a given parser goes about determining if a given sequence of tokens satisfies the rules expressed by that grammar. In practise, there are two main approaches to doing this: from the top down, and from the bottom up.

Time-consuming work on language design and development until 1975 was greatly simplified after Lesk [3] and Johnson [4] published papers on Lex and Yacc. Implementation details for Lex and Yacc can be found in [1]. Nowadays there are tools for support and speedup process. Some of them are concerned only on tokenizing process and some on the tokenizing and parsing. They take token or grammar definition as input and produce scanners or parsers, or both of them. Also we noticed that although LR(1) parsing generator is the most general type of parse, LALR(1) parsing is widely used, because of the complexity of the construction process to obtain an LR parser. Traditional, widely used, and in UNIX de facto standard, Yacc is the most famed parser generator and is based on LALR(1). There are few variants of Yacc, some of them are free like GNU Bison, and some of them are commercial, like high quality Mortice Kern Systems Yacc. Also can be found more LALR(1) based parser generators like: ANTLR, Grammatica, Spirit, GOLD Parser Builder written by Devin Cook which are free. There is even a LALR(k) parser generator Visual Parse which is commercial. Those tools work on mainly three different manners:

1. Compiler-compiler: takes grammar as input, then produces parser tables and mixes them with source code of parser engines in target programming languages. This approach is basic and allows developers to add some of theirs code. Yacc is based on this idea.

2. Component: takes grammar as input, then produces ActiveX component with parser tables. Approach is used by commercial applications Visual Parse and Clear Parse.

3. Creates parser tables only: This approach is newest and produces all tables needed for parsing and stores them in a file. Later, user should read all tables from file, build them and build parser engine, and he is able to parse input. Approach is used by GOLD Parser Builder [5] and a file is called compiled grammar file.

Next table shaw different programming languages and parsing systems which support them:

Table 1. Parser comparison.

Language	GOLD	YACC/Bison	ANTLR	Grammatica	Spirit
ANSI C	√	√			
C++	√	√	√	√	√
C#	√		√	√	
Delphi 5& 6	√				
Java	√		√		
Python	√				
Visual Basic 6	√				
Visual Basic .NET	√				
All .NET Lang.	√				
All ActiveX Lang.	√				

3 Requests for parsing

A software practise and development applications in many areas, such as typesetting, communication protocols, XML database querying, math expression evaluation, graphics description languages - any application that must process text files fits into compiler architecture. As noticed in applications that work on other program code in software engineering in subfields like software metrics, software testing, code reformatting and beautifying, code documenting must behave like compilers and collect information. Also need for creating interpreters and scripting languages arises in different areas.

Requests appear from fields with great diversity and those fields have already established different software development environment and programming languages. In that case parsing should be done in them. To accomplish this task first opportunity is to port generated parsing tables and engines from one programming language to another. It is not easy and is error prone. Problems can arise later if the grammar will changed so the whole difficult and tedious process must be repeated. Second opportunity is to use ActiveX component and commercial solutions. Third opportunity is to use GOLD Parser Builder which is language independent. The sense of language independency presented here is in using file with parser tables generated by this tool.

4 GOLD Parser Builder

GOLD is an acronym for Generalized Object-oriented Language Developer. According to words of Devin Cook, the author of the tool, the GOLD Parser is a free, pseudo-open-source parser generator that you can use to develop your own programming languages, scripting languages and interpreters. The actual LALR and DFA algorithms are easy to implement since they rely on tables to determine actions and to move between states. Consequently, it is the computation of these tables that is both time-consuming and complex. Unlike usual practise established in common compiler-compilers, the GOLD Parser does not require developer to embed source code directly into the grammar and mix them. Instead, they stay separated, the application analyzes the grammar and then saves the parse tables to a separate file called compiled grammar file. This file can be subsequently loaded by the actual parser engine and used to parse input. Since the parse tables are programming language independent, the parser engine can be implemented in different programming languages. As a result, the GOLD Parser supports a myriad of programming languages and can be used on multiple platforms.

The tool has many advantages. We want to point out some of them:

- Windows standard feel and look.
- Support the full Basic Multi-lingual Plane of the Unicode characters and as a result parser is not limited to 256 characters in ASCII.
- Export complete tables and sets in HTML format.
- Test the grammar. GOLD Parser Builder offers interactively testing capabilities. You are aloud to entering an input string to test the grammar. Further-

more it builds and draws the syntax tree in case of success.

- Input grammar is written using regular expression and is expected in Backus-Naur form .
- It is free.

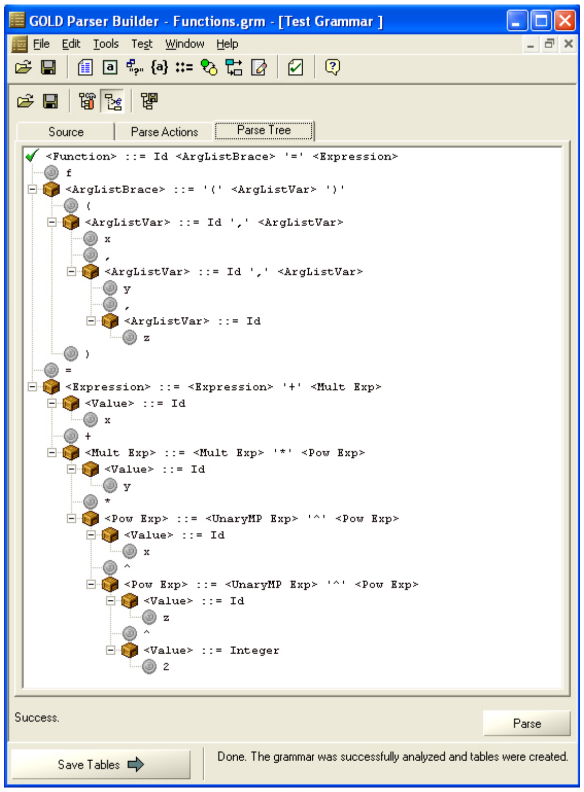
As disadvantages we can notice:

- A lack of %left %right %nonassoc notations for operator precedence and associativity that exist in Yacc. Operator precedence actually consists of a series of rules. In the case with Yacc, the extra rules needed to implement the proper logic are created ”behind the scene”. This makes sense for YACC since the additional rules can be hidden from the programmer and the special logic needed for the parser engine is already implemented.
- A lack of actions accompanying rule which consist of code executed each time an instance of rule is recognized as it exists in Yacc. GOLD Parser Builder does not support such code as it basic intention to be generator of tables, and not of code.

5 Experience

We have used in Compiler construction course classical tool Lex and Yacc, and recently GOLD Parser. We find that GOLD Parser is applicable and very grateful in teaching process and more. It supports interactively writing grammar rules, refining them and checking and reporting ambiguities. Also, students can see DFA and LALR states, transitions among them and shift and reduce actions during parsing inputs in table based report. In case of success in parsing an input, the tool offers graphical representation of a reduction tree with left to right direction transformed to top to bottom direction. Students is able to recognize reduced rules as well as tokens. The interface of GOLD Parser with generated reduction tree is shown on figure 2. Figure show the reduction tree of the input $f(x,y,z) = x + y * x^{z^2}$.

We develop our parsing engine based on GOLD Parser Builder. The engine is written in C++. We are planning to port the engine in other programming languages, Java as first. With this we are enabled to take over full control on parsing process and to add some of well known and very useful properties of Yacc like actions. It overthrows the lack of mixed code with grammar rules since the developer can call a function or do something else when a rule is reduced. We develop our grammar aimed for parsing mathematical expression, more precisely parsing explicitly defined functions. Our grammar is added in appendix and can be used as an example of achieving left to right associativity, right to left associativity and operator precedence using Backus-Naur form.


GOLD Parser Builder - Functions.grm - [Test Grammar]
File Edit Tools Test Window Help
Source
Parse Actions
Parse Tree
Parse
Save Tables

```
<Function> ::= Id <ArgListBrace> '=' <Expression>
    f
    <ArgListBrace> ::= '(' <ArgListVar> ')'
        (
        <ArgListVar> ::= Id ',' <ArgListVar>
            x
            ,
            <ArgListVar> ::= Id ',' <ArgListVar>
                y
                ,
                <ArgListVar> ::= Id
                    z
        )
    =
    <Expression> ::= <Expression> '+' <Mult Exp>
        <Value> ::= Id
            x
        +
        <Mult Exp> ::= <Mult Exp> '*' <Pow Exp>
            <Value> ::= Id
                y
            *
            <Pow Exp> ::= <UnaryMP Exp> '^' <Pow Exp>
                <Value> ::= Id
                    x
                ^
                <Pow Exp> ::= <UnaryMP Exp> '^' <Pow Exp>
                    <Value> ::= Id
                        z
                    ^
                    <Value> ::= Integer
                        2
```

Success.

Done. The grammar was successfully analyzed and tables were created.

Fig. 2. GOLD Parser Builder.

6 Appendix

Grammar for definition explicit function of *n* variables written in Backus-Naur form follows:

”Name”	=	Function Definition Example
”Version”	=	Example
”About”	=	This grammar demonstrates explicit function definitions
”Case Sensitive”	=	True
”Start Symbol”	=	$<$ Function $>$

```
E                  =    ’E’
Pi                 =    ’Pi’
I                  =    ’I’
SinL               =    ’Sin(’
CosL               =    ’Cos(’
TanL               =    ’Tan(’
CtgL               =    ’Ctg(’
LogL               =    ’Log(’
LnL                =    ’Ln(’
ArcsinL            =    ’Arcsin(’
ArccosL            =    ’Arccos(’
ArctanL            =    ’Arctan(’
ArcctgL            =    ’Arcctg(’
SinHL              =    ’Sh(’
CosHL              =    ’Ch(’
Id                 =    LetterAlphaNumeric* Integer = Digit+
Racional1          =    Digit+’.’Digit*
Racional2          =     Digit+’.’Digit*’E’[’-”+’]?Digit?Digit?
< Function >       ::=  Id < ArgListBrace >′=′< Expression >
< ArgListBrace >   ::=  ′(′ < ArgListVar > ′)′
                    |
< ArgListVar >     ::=  Id ′,′ < ArgListVar >
                    |   Id
< Expression >     ::=  < Expression > ′+′ < Mult Exp >
                    |   < Expression > ′−′ < Mult Exp >
                    |   < Mult Exp >
< Mult Exp >       ::=  < Mult Exp > ′∗′ < Pow Exp >
                    |   < Mult Exp > ′/′ < Pow Exp >
                    |   < Pow Exp >
< Pow Exp >        ::=  < UnaryMP Exp > ′^′ < Pow Exp >
                    |   < UnaryMP Exp >
< UnaryMP Exp >    ::=  < Negate Exp >
                    |   < Plus Exp >
< Plus Exp >       ::=  < UnaryPlus >< Value >
                    |   < Value >
< Negate Exp >     ::=  < UnaryMinus >< Value >
< Value >          ::=  Id
```

$$\begin{array}{lcl}
 & | & \text{Integer} \\
 & | & <\text{Racional}> \\
 & | & '(' <\text{Expression}> ')' \\
 & | & <\text{UniArgF}> \\
 & | & <\text{Constant}> \\
<\text{UnaryMinus}> & ::= & '-' \\
<\text{UnaryPlus}> & ::= & '+' \\
<\text{Racional}> & ::= & \text{Racional1} \\
 & | & \text{Racional2} \\
<\text{UniArgF}> & ::= & \text{SinL} <\text{Expression}> ')' \\
 & | & \text{CosL} <\text{Expression}> ')' \\
 & | & \text{TanL} <\text{Expression}> ')' \\
 & | & \text{CtgL} <\text{Expression}> ')' \\
 & | & \text{LogL} <\text{Expression}> ')' \\
 & | & \text{LnL} <\text{Expression}> ')' \\
 & | & \text{ArcsinL} <\text{Expression}> ')' \\
 & | & \text{ArccosL} <\text{Expression}> ')' \\
 & | & \text{ArctanL} <\text{Expression}> ')' \\
 & | & \text{ArcctgL} <\text{Expression}> ')' \\
 & | & \text{SinHL} <\text{Expression}> ')' \\
 & | & \text{CosHL} <\text{Expression}> ')'
\end{array}$$

$$\begin{array}{lcl}
<\text{Constant}> & ::= & \text{E} \\
 & | & \text{Pi} \\
 & | & \text{I}
\end{array}$$

References

[1] A. Alfred, R. Sethi, and U. Jeffrey, *Compilers, Principles, Techniques and Tools*. Reading, Massachusetts: Addison-Wesley, 1986.

[2] Jean-Paul Tremblay and P.G. Sorenson, *The Theory and Practice of Compiler Writing*. New York - St. Louis - San Francisco: McGraw-Hill Book Company, 1985.

[3] M. E. Lesk and E. Schmidt, “Lex - a lexical analyzer generator,” Computing Science Tehnical Report, No. 39, Bell Laboratories, New Jersey, 1975.

[4] J. Stephen, “Yacc: Yet another compiler compiler,” Computing Science Tehnical Report, No. 32, Bell Laboratories, New Jersey, 1975.

[5] D. Cook. Gold parser builder. [Online]. Available: www.devincook.com/goldparser