Free University of Bozen - Bolzano

Formal Languages and Compiler

Compiler Project

HSC

Author:

Giorgio Musciagna gmusciagna@unibz.it
Ravinder Singh rsingh@unibz.it

January 21, 2018

1 Introduction

The aim of the project is the realization of a compiler for a particular language by means of lex and yacc. A compiler is a program which converts a program written in a language, usually denoted as source, into an equivalent program written in another language, usually denoted as target language. The process of compilation comprises two parts: analysis and syntesis.

Analysis: the source program is read and its structure is analysed.
Syntesis: an intermediate code is generated from the intermediate representation of the language returned at the end of the analysis part. An optimization process will occur and the target program will be eventually generated.

The project is focused on the first part of the compilation only i.e. the analysis part which consists of two subtasks:

1. Split the source program and categorize each sequence of characters into tokens by reading it left-to-right. Regular expression are used in order to tokenize the source program. (Lex)

2. Find the hierarchical structure of the source program expressed by recursive rules reflecting a context-free grammar. Moreover, additional information to give meaning to the program are computed involving adding information to the *symbol table* and type checking. (Yacc)

2 Symbol table

The symbol table is a data structure whose aim is to save additional information we might want to preserve in order to give meaning to the program. It is usually shared among lex and yacc. During the lexical analysis the code is tokenized and categorized. On the other hand, this process of categorization usually results in a loss of information related to the categorized token such as name, value. As a result, a symbol table could be used in order to

preseve this information we might lose. Additional attributes will be later on added during the parsing phase such as type, scope, etc. which could be only expressed by the hierchical structure of the program. In this project, the symbol table is a linked list where each node, usually a variable, is denoted by the following five attributes: name, type, value, scope level, initialized. Here an example follows.

Name	Type	Value	Scope	Initialized
price	real	140.05	1	1
done	boolean	1	0	1
price	real	0	0	0
rate	real	12.57	0	1

3 Features

The main features of the developed language are the following:

- Variable declaration (single or multiple)
- Variable assignment
- Printing (strings and expressions) also with multiple args
- Typing (real and boolean)
- Operations (arithmetic and boolean: $+ \; - \; * \; / \; ! \; \&\& \; || \; == \; <= \; > \;$ *etc.*)
- Scoping
- Comments
- Strings and meta chars

3.1 General information

Each program as usual consists of a list of statements. Moreover, reserved words and special chars are used in order to let the user express some particular commands. Here some general rules which globally apply to the language:

1. Statement delimiter: each statement is delimited by `newline`.
2. Stop execution: with the command `halt` the execution of the program will be interrupted.
3. Strings: literals containing meta chars are also captured, delimited by single quotes. To use them as part of the string type two consecutive single quotes `''`. Newlines and tabs can be used via `\n` and `\t`.

Interesting technical info: regarding multiple declaration (see * below) a right recursive rule has been used in the yacc source file. The main reason is because the type token is at the end of the statement, therefore we have to accumulate all the variable names on the stack. Then, once the end has been reached, reductions will occur and type will be set for each variable name.

3.2 Syntax and examples

Here some examples showing the syntax of the developed language

Variable declaration: single and multiple declaration. *

```
var pippo : real
var pluto : boolean
var x, y, z, w : boolean
```

Variable assignment: after or during declaration.

```
var pluto : boolean
pluto := 40 > 20
var paperino : real = 0.25
```

Comments.

```
x := 10 (* This is a
multiline comment *)
```

Printing. Two versions available: `write` and `writeln`. The latter puts a newline afterwards. It is possible to print boolean or arithmetic expressions and strings. Moreover you can print a list of expressions or strings using just one single command. Writeln could also just be used without args to just insert a newline.

```
writeln 'hello ''world'', how are you?\nFine... "Awwww"'
write -2 * -4
writeln
writeln
writeln 'Euler''s num is ', math.e, '\tPI is ', math.pi

(*
        hello 'world', how are you?
        Fine... "Awww"
        8.000000

        Euler's num is 2.718281      PI is 3.1415927
*)
```

Operations. Arithmetic and boolean operations. Some constants are also available: Pi and the Euler's number.

```
pippo := 2
pippo := -pippo + math.e * 40 / math.pi
writeln 'Is pippo less or equal to 2? ', pippo <= 2
pluto := true
pluto := !pluto && (!true || false) && true
```

Scoping. Scoping is also part of the language. Create a scope by enclosing some statements via reserved keywords `begin` and `end`. The scope of a variable and consequently its visibility starts from its declaration until the end of the block scope it belongs to. If a block contains a second block in which a variable is redeclared, then this variable hides the previous variable till the end of the second scope. As soon as the inner block ends, then the first declaration will be available.

```
x := 1
writeln x                    (* here x has value 1 *)
begin
    x := 12.5
    writeln x                (* here x has value 12.5 *)
    var x : real
    x := 2
    writeln x                (* here x has value 2 *)
    begin
        var x : boolean
        x := true
        writeln x            (* here x has value true *)
    end
    writeln x                (* here x has value 2 *)
end
writeln x                    (* here x has value 12.5 *)
```

4 References

Unibz course webpage:
http://www.inf.unibz.it/~artale/Compiler/compiler.htm

Helpful online resources:
http://www.cs.man.ac.uk/~pjj/cs212/ex2_str_comm.html
http://epaperpress.com/lexandyacc/str.html
https://www.programiz.com/c-programming/c-strings
http://www.learn-c.org/en/Linked_lists
https://stackoverflow.com/questions/1653958/why-are-ifndef-and-define-used-in-c-header-files
https://www.youtube.com/watch?v=EzBTm73_oU8
https://stackoverflow.com/questions/7751366/malloc-memory-for-c-string-inside-a-structure https://stackoverflow.com/questions/3219393/stdlib-and-colored-output-in-c