Rafael Román Otero

Intro to Compiler Design

Compiler Report

Introduction

Together with this report is submitted a file called “roman-otero-compiler.tar.gz” wich contains all the files neccesary for building and testing the project, and some design specification files as well.

Project Status

It includes the lexical analyzer, the syntactic analyzer, the semantic analizer, and part of the code generator which is not finished.

Architecture and Design

Its entirely written in C. Although it was build and tested under a linux platform, the code was written under the C99 standarization; therefore, any compiler compatible with C99 standarization can build the project. In fact, the only feature used which is included in the C99 is the *variable declaration* inside a *for-loop;* so, changing this, it should compile with any ANSI C compiler.

The structure of the compiler is the one proposed in the compiler guidelines; in this particular case, it was matched a file per module. The detailed file structure can be found in *files_diagram.pdf*. Likewise, the transition diagrams used for token recognition by the lexical analyzer (*scanner.c)* can be found with the name *transitions_diagrams.pdf*.

The approach used for implementating the transition diagrams in code was to fuse all of them into one big transition diagram; thus, there is only one big function containing one big switch implementing all the transition diagrams.

The error recognition is made, in part, via the transitions diagrams as some errors --such as a numeral “4.” or a “.7.”-- are recognized as “bad tokens”, and some others characters, such as *&* or *$*, are matched to *unknow* tokens. Adittionaly, in the case of the right closing comment “*/” missing, an error token is matched.

Another particularity is the conversion of non-extended aschii character into the character with value 0x96. The reason is that EOF character is mapped as “-1” and therefore does not allow to use unsigned char type. So, all characters with value > 127 are changed to 0x96. This also means that when an error token is returned, and the token lexeme is the character with this value, it's impossible to know if it's the 0x96 value, or an extended-ascii character value.

Implementation

They can be changed easily by modifying the constants in c11.h. The size limitations fixed to the compiler are: maximum number of tokens, which is established by the constant MAX_NUM_OF_TOKENS; maximum lenght in characters of any given line, which is defined by the constant MAX_LINLEN; maximum number of characters in a string literal, which is found in the constant MAXLEN_STRLIT; and number of reserved words in the constant NUM_OF_RESERVED_WORDS.

The maximum number of tokens is a concern of the symbol table, because the symbol table uses that number to declare the size of the table.

The maximum number of characters a line can have is arbitrarily 499.

The maximum number of characters in a string literal (e.g. “very very large string literal...”) has effects on the scanner module, as it uses this constant to declare the *lexeme* variable. In other words, a string literal is considered to be the largest possible lexeme.

Regarding the implementation and construction of the abstract syntax tree see the files *syntax_tree_implementation.pdf* and *syntax_directed_definition_plus_typing_rules.pdf*

Building and Use

After extracting *roman-otero-compiler.tar.gz* a folder with the same name will be created.

Whitin the *sources* folder are all the *.c,* and *.h* files and one *makefile* to build the project. Adittionaly, two source files for testing purpouses are included:

hola.cs11
non_visible.cs11

The file also contains an executable file that accepts any .cs11 file as parameter. So, to run the examples:

```
~/sources$ ./c11 hola.cs11
~/sources$ ./c11 non_visible.cs11
```

However, if it it desired to compile and build the project, one just need to use the makefile:

```
~/sources$ make
```

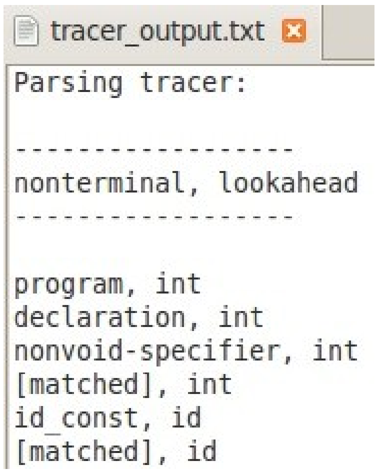
After this, an executable file called *c11* will be created

Tracer

The printed information and the output for the tracer can be both modified. The function called *tracer_init()* (in line 107 within the parse() function in the parser module -parser.c)accepts two parameters. The first one can be STDOUT_FLAG, whichs shows the tracer output in the std output, or FILE_FLAG, which shows the tracer output on the file called tracer_output.txt. The second parameter can be MATCH_FLAG, which prints out only the matched lexemes, or COMPLETE_FLAG, which prints the complete route including non-terminals. If the tracer_init() function is commented, the tracer is turned off.

The file output of the tracer is specified whithin *tracer.c* by the variable name *tracer_ofile[]*. If nothing is changed it is *tracer_output.txt*.

It looks like this:



```
Parsing tracer:

-----------------
nonterminal, lookahead
-----------------

program, int
declaration, int
nonvoid-specifier, int
[matched], int
id_const, id
[matched], id
```

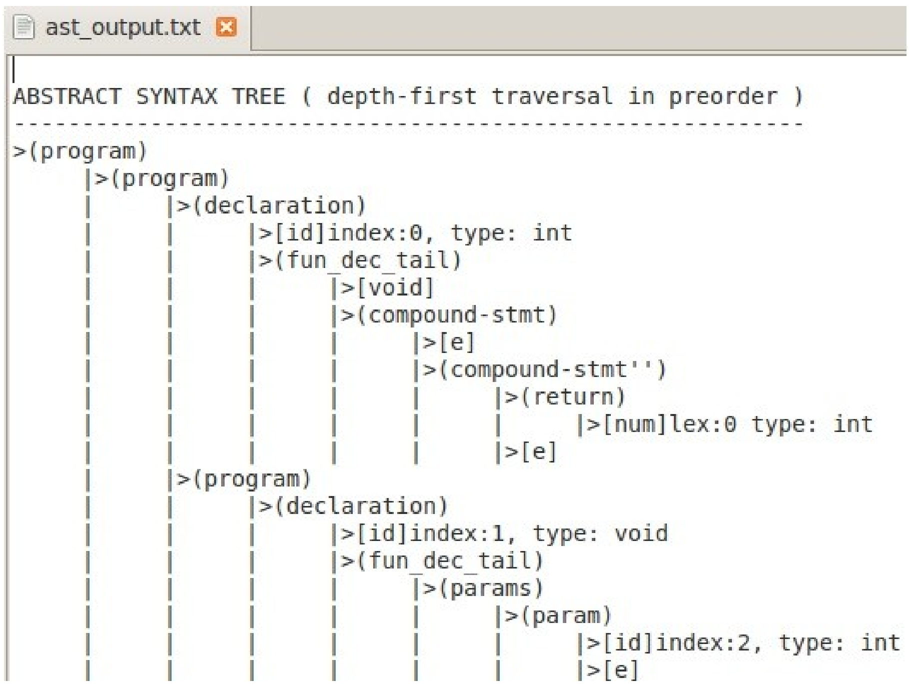
Abstract syntax tree

The original EBNF grammar from the guidelines was modified to be BNF. See *bnf_grammar.pdf*

The function that prints the AST, printtree(), is called by the parser module, *parser.c,* in line 120. If commented, the ast is not printed, otherwise it prints the ast in the file specified in the ast module.

The output of the *printree()* function is specified within *ast.c* by the variable name *ast_ofile[]*. If nothing changed it should be *ast_output.txt*.

It looks like this:



```
ABSTRACT SYNTAX TREE ( depth-first traversal in preorder )
-----------------------------------------------------------
>(program)
     |>(program)
     |     |>(declaration)
     |     |     |>[id]index:0, type: int
     |     |     |>(fun_dec_tail)
     |     |     |     |>[void]
     |     |     |     |>(compound-stmt)
     |     |     |     |     |>[e]
     |     |     |     |     |>(compound-stmt'')
     |     |     |     |     |     |>(return)
     |     |     |     |     |     |     |>[num]lex:0 type: int
     |     |     |     |     |     |>[e]
     |     |>(program)
     |     |     |>(declaration)
     |     |     |     |>[id]index:1, type: void
     |     |     |     |>(fun_dec_tail)
     |     |     |     |     |>(params)
     |     |     |     |     |     |>(param)
     |     |     |     |     |     |     |>[id]index:2, type: int
     |     |     |     |     |     |     |>[e]
```

Some terminals are printed within brackets (e.g. [id], [num]), whereas nonterminals within parenthesis (e.g. (program), (declaration), etc). Some other such as (return) are tokens, but are printed within parenthesis because they have children nodes. Epsilon is denoted by [e].

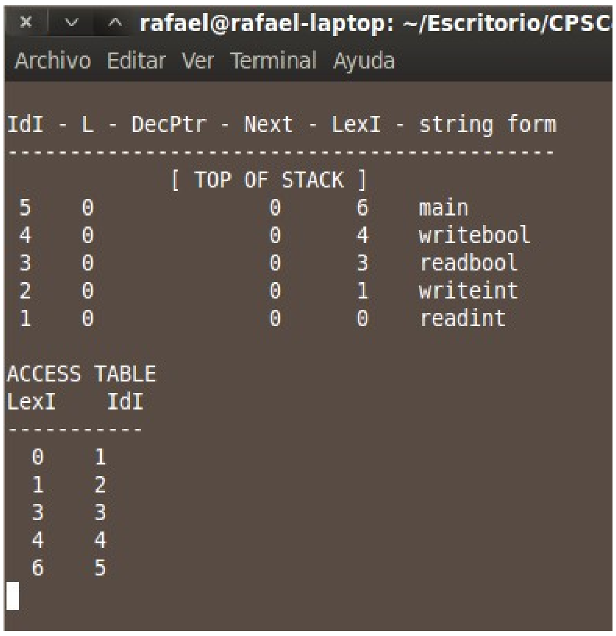
For information on how it's constructed see *syntax_tree_implementation.pd*f and *syntax_directed_definition_plus_typing_rules.pdf*

Semantic Analyzer

The semantic analyzer allows to visualize the construction of the *identification table* during the scope analysis and part of what the function *param_consistency_checking()* does when called.

The control for activating/desactivating is in line 37 of the file *semanalyzer.c.* It consists of two definitions SEE_ITABLE_CONST and PARAM_CONSIST_DEBUG. If anyone is uncommented, then the output will appear in console as the semantic analyzer runs. If commented nothing is printed.

For instance, if the definition SEE_ITABLE_CONST is uncommented, then the semantic analysis for a a source file with nothing more than a main definition -and the dummy definitions- , at some point, shows the following in console:



rafael@rafael-laptop: ~/Escritorio/CPSC
Archivo Editar Ver Terminal Ayuda

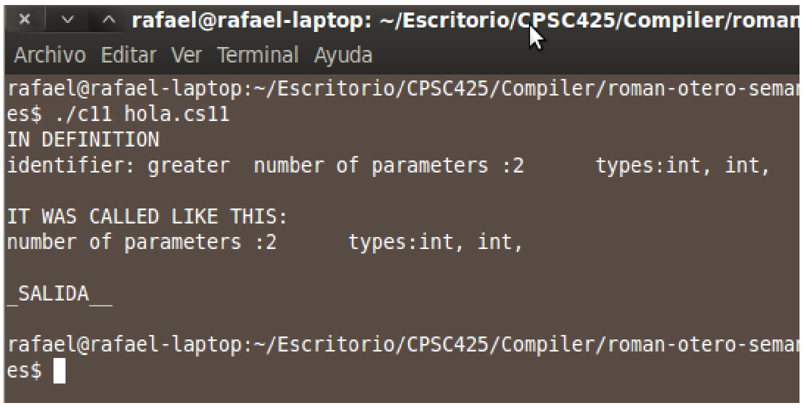
```
IdI - L - DecPtr - Next - LexI - string form
--------------------------------------------
             [ TOP OF STACK ]
 5    0             0      6    main
 4    0             0      4    writebool
 3    0             0      3    readbool
 2    0             0      1    writeint
 1    0             0      0    readint

ACCESS TABLE
LexI    IdI
-----------
  0    1
  1    2
  3    3
  4    4
  6    5
```

Both tables change their content as the scope analysis is performed.

Similarly, if the definition PARAM_CONSIST_DEBUG is uncommented, then the semantic analysis for the following source file:

```
int greater( int a, int b ){
        if( a > b )
                return a;
        else
                return b;
}
int main( void ){
        int grt;

        grt := greater( 5, 10 );

        return 0;
}
```

Shows the following in console:



rafael@rafael-laptop: ~/Escritorio/CPSC425/Compiler/roman
Archivo Editar Ver Terminal Ayuda

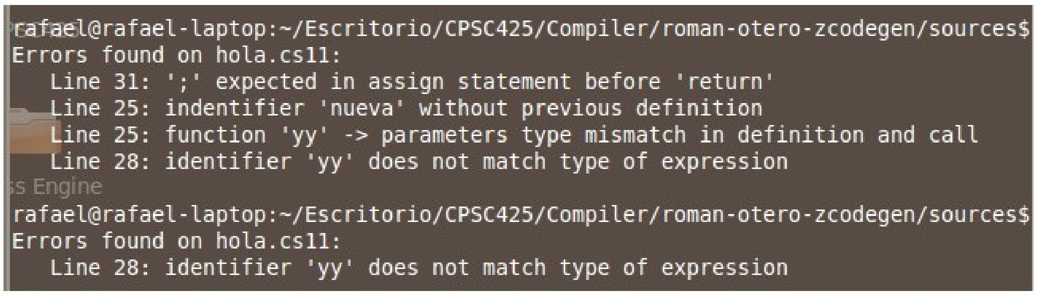
```
rafael@rafael-laptop:~/Escritorio/CPSC425/Compiler/roman-otero-sema
es$ ./c11 hola.cs11
IN DEFINITION
identifier: greater  number of parameters :2      types:int, int,

IT WAS CALLED LIKE THIS:
number of parameters :2      types:int, int,

_SALIDA__

rafael@rafael-laptop:~/Escritorio/CPSC425/Compiler/roman-otero-sema
es$
```

This just shows the process of checking parameter consistency within the function *check_param_consistency(...)* in *semanalizer.c*.

In this case, it is shown that the definition uses 2 parameter of type int, and when called it used two arguments of type int. Therefore the call matches the definition. _SALIDA_ just tells that the funtions is leaved.

Error reporting

Error reporting is limited to a maximum of 10 error messages. It looks like this:



```
rafael@rafael-laptop:~/Escritorio/CPSC425/Compiler/roman-otero-zcodegen/sources$
Errors found on hola.cs11:
   Line 31: ';' expected in assign statement before 'return'
   Line 25: indentifier 'nueva' without previous definition
   Line 25: function 'yy' -> parameters type mismatch in definition and call
   Line 28: identifier 'yy' does not match type of expression
rafael@rafael-laptop:~/Escritorio/CPSC425/Compiler/roman-otero-zcodegen/sources$
Errors found on hola.cs11:
   Line 28: identifier 'yy' does not match type of expression
```

NOTE: Given the lack of use of a proper AST interface. Some type of syntactical errors may cause a segment violation. Although they are only a few, must of them are correctly notified as errors. This is so because, even though semantic analysis does not continue when a syntactic error is found, AST pointer handling in the semantic analisys assumes a correct format on the AST and some times it may try to manipulate a NULL pointer.

Code

c11.h

contains the definition of the token data structure, an enumeration with all the token names, and some constants used by almost all the modules.

input.c

Reads directly the source file indicated by admin.c, and provides a pointer to a buffer containing a line readed from the source file.

admin.c

It is the one the human interfaces with, and it serves as interface between the scanner module and the input module. It converts the line-buffer provided by input.c into a stream of characters useful for the scanner module.

scanner.c

Performs token recognition. Provides a stream of tokens when asked.

symbol_table.c

Contains the declaration of the symbol table data structure and all the functions used to insert, lookup and hash the key.

word_table.c

contains the declaration of the word table data structure and all the functions used to insert, lookup and hash the key. In contrary with the symbol table, it contains an adittional function wich fills the word table with all the reserved words in c11 specification.

p*arser.c*

Recognizes the syntactic structure of a syntactically correct program. It uses ad hoc implementation of FIRST sets. Builds the ast by calling the function *build_tree* and traces himself by calling the function *parser_trace*

tracer.c

It traces the parser. The path is builded each time the function parse_trace is called by each nonterminal function in the parser module.

ast.c

Contains the root node of the AST, therefore contains the AST. It provides interface functions to build it, modify it, and print it.

access_table.c

Contains the access table defined in the guidelines, and provides an interface for adding and retrieving elements.

ident_table.c

The identification table described in the guidelines. It's implemented as a stack

semanalyzer.c

Checks the semantic consistency of a syntactically correct intermediate representation of the input program. It creates an annotated AST from the received AST from the syntactic analyzer. Annotations are created by filling fields of the data estructure that up to this points used to have no significant value.

error_reporting.c

Keeps a list of error strings, limited to 10 strings. Uses one function to add an entry, and three more functions for printing the added strings. It does not perform any kind of error recovery. It just notifies errors.

interm_codegen.c

It's not finished

Test and Observations

1. uminus derivation is not supported.

2. Dynamic memory used by the AST, identifier table, and symbol table are not deallocated when the compiling process is finished.

3. Given the lack of use of a proper AST interface. Some type of syntactic errors may cause a segment violation.

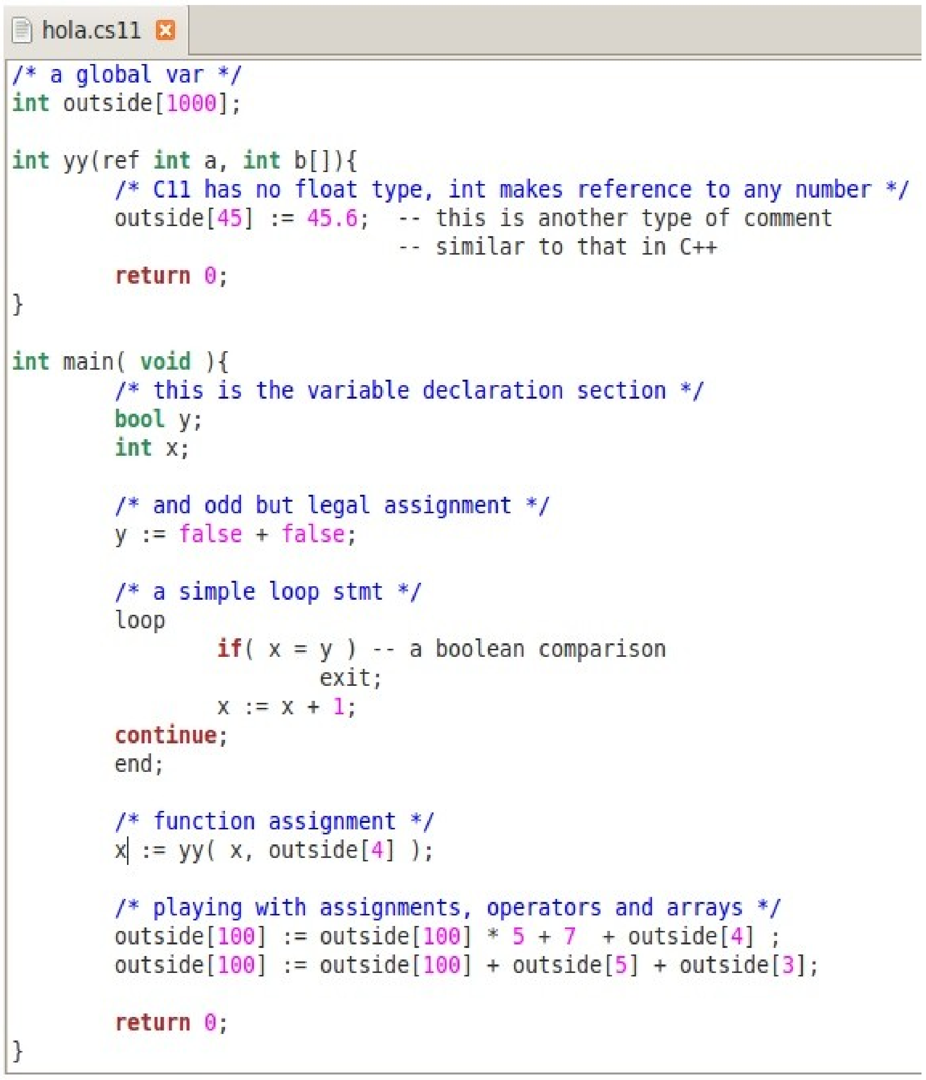
4. Nested comments of the type /* /* */ */ are not supported

5. A double error is generated in case of a extended-aschii character; instead of just one.

6. No error recovery is performed, and a maximum of ten erros messages are reported

7. Global variables, local variables and parameters variables names, none of them are allowed to have the same name. It is reported as a double definition error.

This is an exmaple of a source file with no errors:



hola.cs11

```
/* a global var */
int outside[1000];

int yy(ref int a, int b[]){
        /* C11 has no float type, int makes reference to any number */
        outside[45] := 45.6;  -- this is another type of comment
                              -- similar to that in C++
        return 0;
}

int main( void ){
        /* this is the variable declaration section */
        bool y;
        int x;

        /* and odd but legal assignment */
        y := false + false;

        /* a simple loop stmt */
        loop
                if( x = y ) -- a boolean comparison
                        exit;
                x := x + 1;
        continue;
        end;

        /* function assignment */
        x := yy( x, outside[4] );

        /* playing with assignments, operators and arrays */
        outside[100] := outside[100] * 5 + 7  + outside[4] ;
        outside[100] := outside[100] + outside[5] + outside[3];

        return 0;
}
```