
Report for Compilers Lab

Prepared by:
Shyam Gopal – 11CO89
Jayendra Shyam Dewani – 11CO106

Computer and Science Department
National Institute of Technology Karnataka, Surathkal

24th April, 2014

- **Lexical Analyzer:**

Lexical analysis is the process of converting a sequence of characters into a sequence of tokens. A program or function that performs lexical analysis is called a **lexical analyzer, lexer, tokenizer** or **scanner**. A lexer is combined with a parser, which together analyze the syntax of computer languages, such as in compilers for programming languages, HTML parsers, etc. The (context-free) syntax of the language is divided into two pieces:

- the lexical syntax, which is processed by the scanner; and
- the phrase structure, which is processed by the parser.

The lexical syntax is usually a regular language, whose atoms are individual characters, while the phrase syntax is usually a context-free language, whose atoms are tokens produced by the lexer. A **token** is a string of one or more characters that is significant as a group. The process of forming tokens from an input stream of characters is called **tokenization**.

Tokens are identified based on the specific rules of the lexer. Some methods used to identify tokens include: regular expressions, flags, delimiters, and explicit definition by a dictionary. Special characters, including punctuation characters, are used by lexers to identify tokens because of their natural use in written and programming languages.

Tokens are often categorized by character content or by context within the data stream. Categories, defined by the lexer, often involve grammar elements of the language used in the data stream. Programming languages often categorize tokens as identifiers, operators, grouping symbols, or by data type. Categories are used for post-processing of the tokens either by the parser or by other functions in the program.

FLEX

Flex is a fast lexical analyser generator. It is a tool for generating scanners. A scanner, sometimes called a lexical analyzer, is a program which recognizes lexical patterns in text. The flex program reads user-specified input files, or its standard input if no file names are given, for a description of a scanner to generate. The description is in the form of pairs of regular expressions and C code, called rules. Flex generates a C source file named, "lex.yy.c", which defines the function yylex(). The file "lex.yy.c" can be compiled and linked to produce an executable. When the executable is run, it analyzes its input for occurrences of text matching the regular expressions for each rule. Whenever it finds a match, it executes the corresponding C code.

The flex input file consists of three sections, separated by a line containing only '%%'.

```
%%
    definitions
%%
    rules
%%
    user code
```

The flex file used for our compiler is as follows:

```
%{

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include "parser.tab.h"

%}

%x comment
%%

/* math operators */
"+" { return OP_ADD; }
"-" { return OP_SUB; }
"*" { return OP_MUL; }
"/" { return OP_DIV; }
```

```

/* comparison operators */
">"    { return OP_GT;    }
"<"    { return OP_LT;    }
"=="   { return OP_EQ;    }
"!="   { return OP_NE;    }
"<="   { return OP_LE;    }
">="   { return OP_GE;    }

/* assignment operator */
"="     { return OP_ASSIGN; }

/* statement terminator */
";"     { return SEMICOLON; }

/* grouping operators */
"{"     { return LBRACE;    }
"}"     { return RBRACE;    }
"("     { return LPAREN;    }
")"     { return RPAREN;    }

/* data type keywords */
"int"   { return KW_INT;    }
"float" { return KW_FLOAT;  }

/* control flow keywords */
"if"    { return KW_IF;     }
"else"  { return KW_ELSE;   }
"while" { return KW_WHILE;  }

/* regular expression for identifier names */
[_a-zA-Z][_a-zA-Z0-9]* {
    yylval.identifier = strdup(yytext);
    return IDENTIFIER;
}

/* regular expression for integer constants */
[0-9]+ {
    yylval.int_const = atoi(yytext);
    return INT_CONSTANT;
}
[0-9]+\.[0-9]+ {
    yylval.float_const = (float) atof(yytext);
    return FLOAT_CONSTANT;
}

/* ignore whitespace */
[ \r\t]+ { }

/* ignore newlines */
\n      { }

```

```

/* ignore unknown characters */
.      { }

\\[[0-9]*\\.\\.*\\]   {printf("Invalid Array index\\n");}

/* single line comments are ignored */
\\.*      { }

/* multi line comments are ignored */
\\.*      {BEGIN(comment);}
<comment>\\.*\\      {BEGIN(INITIAL);}
<comment>.

```

- **Parser:**

Parsing or **syntactic analysis** is the process of analysing a string of symbols, either in natural language or in computer languages, according to the rules of a formal grammar. It refers to the formal analysis by a computer of a sentence or other string of words into its constituents, resulting in a parse tree showing their syntactic relation to each other, which may also contain semantic and other information.

A **parser** is a software component that takes input data and builds a data structure that is some kind of parse tree, abstract syntax tree or other hierarchical structure – giving a structural representation of the input, checking for correct syntax in the process. The parser is often preceded by a separate lexical analyzer, which creates tokens from the sequence of input characters. Parsers may be programmed by hand or may be automatically or semi-automatically generated by a parser generator.

An abstract syntax tree (AST), or **syntax tree**, is a tree representation of the abstract syntactic structure of source code written in a programming language. Each node of the tree denotes a construct occurring in the source code.

YACC

Yacc is a computer program for the Unix operating system. The name is an acronym for "Yet Another Compiler Compiler". It is a LALR parser generator, based on an analytic grammar written in a notation similar to BNF.

Yacc produces only a parser; for full syntactic analysis this requires an external lexical analyzer to perform the first tokenization stage (word analysis), which is then followed by the parsing stage.

Yacc provides a general tool for imposing structure on the input to a computer program. The Yacc user prepares a specification of the input process; this includes rules describing the input structure, code to be invoked when these rules are recognized, and a low-level routine to do the basic input. Yacc then generates a function to control the input process. This function, called a parser, calls the user-supplied low-level input routine (the lexical analyzer) to pick up the basic items (called tokens) from the input stream. These tokens are organized according to the input structure rules, called grammar rules; when one of these rules has been recognized, then user code supplied for this rule, an action, is invoked; actions have the ability to return values and make use of the values of other actions.

The Yacc file for our compiler:

```
%{

#include <stdio.h>
#include <stdlib.h>
#include "ast.h"

void yyerror(char *s);

%}

%union {
    int int_const;
    float float_const;

    int type_specifier;
    char *identifier;

    struct ast_node *node;
    struct ast_declaration *declaration;
    struct ast_expression *expression;
    struct ast_statement_list *statement_list;
    struct ast_selection_statement *selection_statement;
    struct ast_while_statement *while_statement;
    struct ast_compound_statement *compound_statement;
    struct ast_translation_unit *translation_unit;
}

/* math operators */
%left  OP_ADD OP_SUB
%left  OP_MUL OP_DIV
```

```

/* comparison operators */
%left  OP_GT OP_LT OP_EQ OP_NE OP_LE OP_GE

/* assignment operator */
%token OP_ASSIGN

/* statement terminator */
%token SEMICOLON

/* grouping symbols */
%token LBRACE
%token RBRACE
%token LPAREN
%token RPAREN

/* keywords */
%token KW_INT
%token KW_FLOAT
%token KW_IF
%token KW_ELSE
%token KW_WHILE

/* integer and floating constants */
%token <int_const>  INT_CONSTANT
%token <float_const>  FLOAT_CONSTANT

/* identifier's name */
%token <identifier>  IDENTIFIER

/* rule types */
%type <type_specifier>  type_specifier
%type <expression>      primary_expression expression assignment
%type <declaration>     declaration
%type <node>            statement
%type <statement_list>  statement_list
%type <compound_statement>  compound_statement
%type <selection_statement>  selection_statement selection_rest_statement
%type <while_statement>     while_statement
%type <translation_unit>    translation_unit

%%

/* rule that matches a translation unit (aka. a source file) */

```

```

translation_unit: statement_list {
    $$ = create_translation_unit($1);
    print_node((struct ast_node *) $$);
}
;

/* rules for: 1) statements
              2) list of statements
              3) compound statements */
compound_statement: LBRACE statement_list RBRACE {
    $$ = create_compound_statement($2);
}
;

statement_list: statement {
    $$ = create_statement_list($1);
}
| statement_list statement {
    $$ = statement_list_add_statement($1, $2);
}
;

statement: declaration SEMICOLON { $$ = (struct ast_node *) $1; }
| expression SEMICOLON { $$ = (struct ast_node *) $1; }
| assignment SEMICOLON { $$ = (struct ast_node *) $1; }
| selection_statement { $$ = (struct ast_node *) $1; }
| while_statement { $$ = (struct ast_node *) $1; }
;

/* while rule */
while_statement: KW_WHILE LPAREN expression RPAREN
compound_statement {
    $$ = create_while_statement($3, $5);
}
;

/* if/then/else rule */
selection_statement: KW_IF LPAREN expression RPAREN
selection_rest_statement {
    $$ = $5;
    $$->condition = $3;
}
;

```



```

selection_rest_statement: compound_statement {
    $$ = create_selection_statement(NULL, $1, NULL);
}
| compound_statement KW_ELSE compound_statement {
    $$ = create_selection_statement(NULL, $1, $3);
}
;

/* rule to match an assignment statement */
assignment: IDENTIFIER OP_ASSIGN expression {
    $$ = create_expression(AST_ASSIGN, $3, NULL);
    $$->primary_expr.identifier = $1;
}
;

/* expression evaluation rules */
expression: expression OP_ADD expression {
    $$ = create_expression(AST_ADD, $1, $3);
}
| expression OP_SUB expression {
    $$ = create_expression(AST_SUB, $1, $3);
}
| expression OP_MUL expression {
    $$ = create_expression(AST_MUL, $1, $3);
}
| expression OP_DIV expression {
    $$ = create_expression(AST_DIV, $1, $3);
}
| expression OP_GT expression {
    $$ = create_expression(AST_GT, $1, $3);
}
| expression OP_LT expression {
    $$ = create_expression(AST_LT, $1, $3);
}
| expression OP_EQ expression {
    $$ = create_expression(AST_EQ, $1, $3);
}
| expression OP_NE expression {
    $$ = create_expression(AST_NE, $1, $3);
}
| LPAREN expression RPAREN {
    $$ = $2;
}
| primary_expression {

```

```

    $$ = $1;
}
;

primary_expression: IDENTIFIER {
    $$ = create_expression(AST_IDENTIFIER, NULL, NULL);
    $$->primary_expr.identifier = $1;
}
| INT_CONSTANT {
    $$ = create_expression(AST_INT_CONSTANT, NULL, NULL);
    $$->primary_expr.int_constant = $1;
}
| FLOAT_CONSTANT {
    $$ = create_expression(AST_FLOAT_CONSTANT, NULL, NULL);
    $$->primary_expr.float_constant = $1;
}
;

/* variable declaration rules */
declaration: type_specifier IDENTIFIER {
    $$ = create_declaration($1, $2);
}
;

type_specifier: KW_INT    { $$ = TYPE_INT;    }
               | KW_FLOAT { $$ = TYPE_FLOAT; }
               ;

%%

void
yyerror(char *s)
{
    printf("%s\n", s);
}

int main(int argc, char *argv[])
{
    yyparse();
    return 0;
}

```

- **Semantic Analysis**

Semantic analysis is the task of ensuring that the declarations and statements of a program are semantically correct, i.e. that their meaning is clear and consistent with the way in which control structures and data types are supposed to be used.

Semantic analysis typically involves:

- **Type checking** – Data types are used in a manner that is consistent with their definition (i.e., only with compatible data types, only with operations that are defined for them, etc.)
- **Label Checking** – Labels references in a program must exist.
- **Flow control checks** – control structures must be used in their proper fashion (no GOTOs into a FORTRAN DO statement, no breaks outside a loop or switch statement, etc.)

Semantic analysis is not a separate module within a compiler. It is usually a collection of procedures called at appropriate times by the parser as the grammar requires.

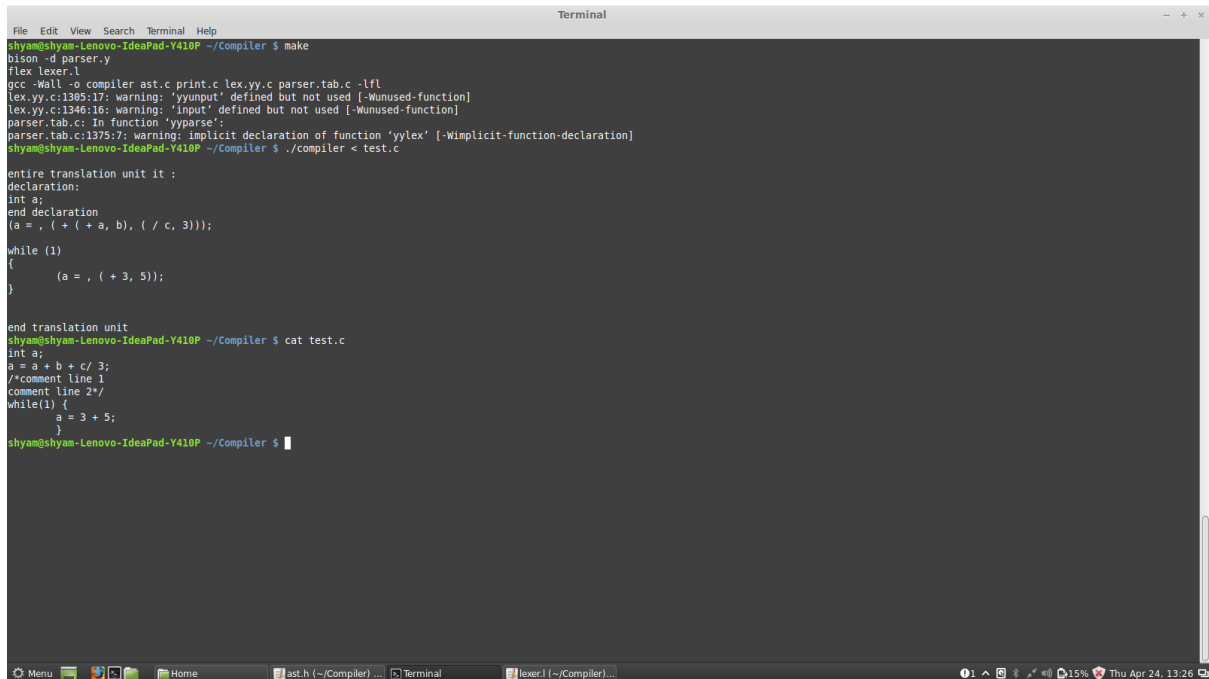
Implementing the semantic actions is conceptually simpler in recursive descent parsing because they are simply added to the recursive procedures.

Implementing the semantic actions in a table action driven LL(1) parser requires the addition of a third type of variable to the productions and the necessary software routines to process it.

Type Checking

Type checking is the process of verifying that each operation executed in a program respects the type system of the language. This generally means that all operands in any expression are of appropriate types and number. Much of what we do in the semantic analysis phase is type checking. Sometimes the rules regarding operations are defined by other parts of the code (as in function prototypes), and sometimes such rules are a part of the definition of the language itself (as in "both operands of a binary arithmetic operation must be of the same type"). If a problem is found, e.g., one tries to add a char pointer to a double in C, we encounter a type error. A language is considered strongly-typed if each and every type error is detected during compilation. Li Type checking can be done compilation, during execution, or divided across both.

Screenshots

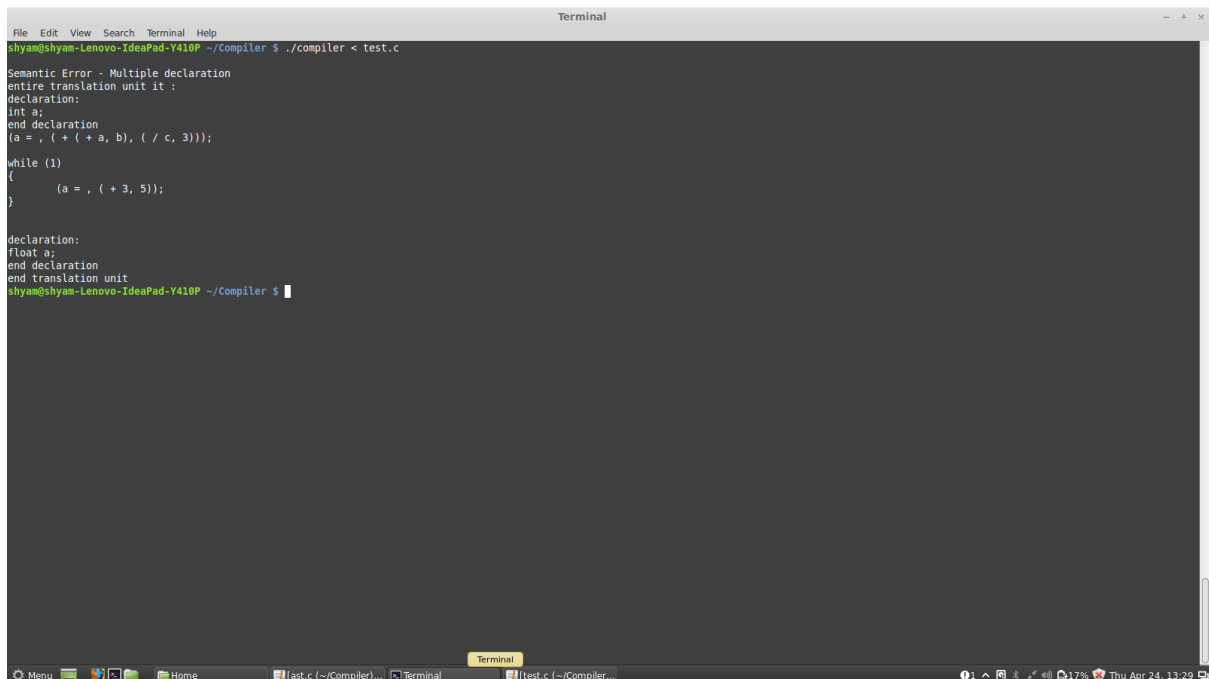


```
shyam@shyam-Lenovo-IdeaPad-Y410P ~/Compiler $ make
bison -d parser.y
flex lexer.l
gcc -Wall -o compiler ast.c print.c lex.yy.c parser.tab.c -lfl
lex.yy.c:1395:17: warning: 'yyunput' defined but not used [-Wunused-function]
lex.yy.c:1346:16: warning: 'input' defined but not used [-Wunused-function]
parser.tab.c: In function 'yyparse':
parser.tab.c:1375:7: warning: implicit declaration of function 'yylex' [-Wimplicit-function-declaration]
shyam@shyam-Lenovo-IdeaPad-Y410P ~/Compiler $ ./compiler < test.c

entire translation unit it :
declaration:
int a;
end declaration
(a = , ( + ( + a, b), ( / c, 3)));

while (1)
{
    (a = , ( + 3, 5));
}

end translation unit
shyam@shyam-Lenovo-IdeaPad-Y410P ~/Compiler $ cat test.c
int a;
a = a + b + c / 3;
/*comment line 1
comment line 2*/
while(1) {
    a = 3 + 5;
}
```



```
shyam@shyam-Lenovo-IdeaPad-Y410P ~/Compiler $ ./compiler < test.c

Semantic Error - Multiple declaration
entire translation unit it :
declaration:
int a;
end declaration
(a = , ( + ( + a, b), ( / c, 3)));

while (1)
{
    (a = , ( + 3, 5));
}

declaration:
float a;
end declaration
end translation unit
shyam@shyam-Lenovo-IdeaPad-Y410P ~/Compiler $
```