**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 AnotherCompiler 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 assigment 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.

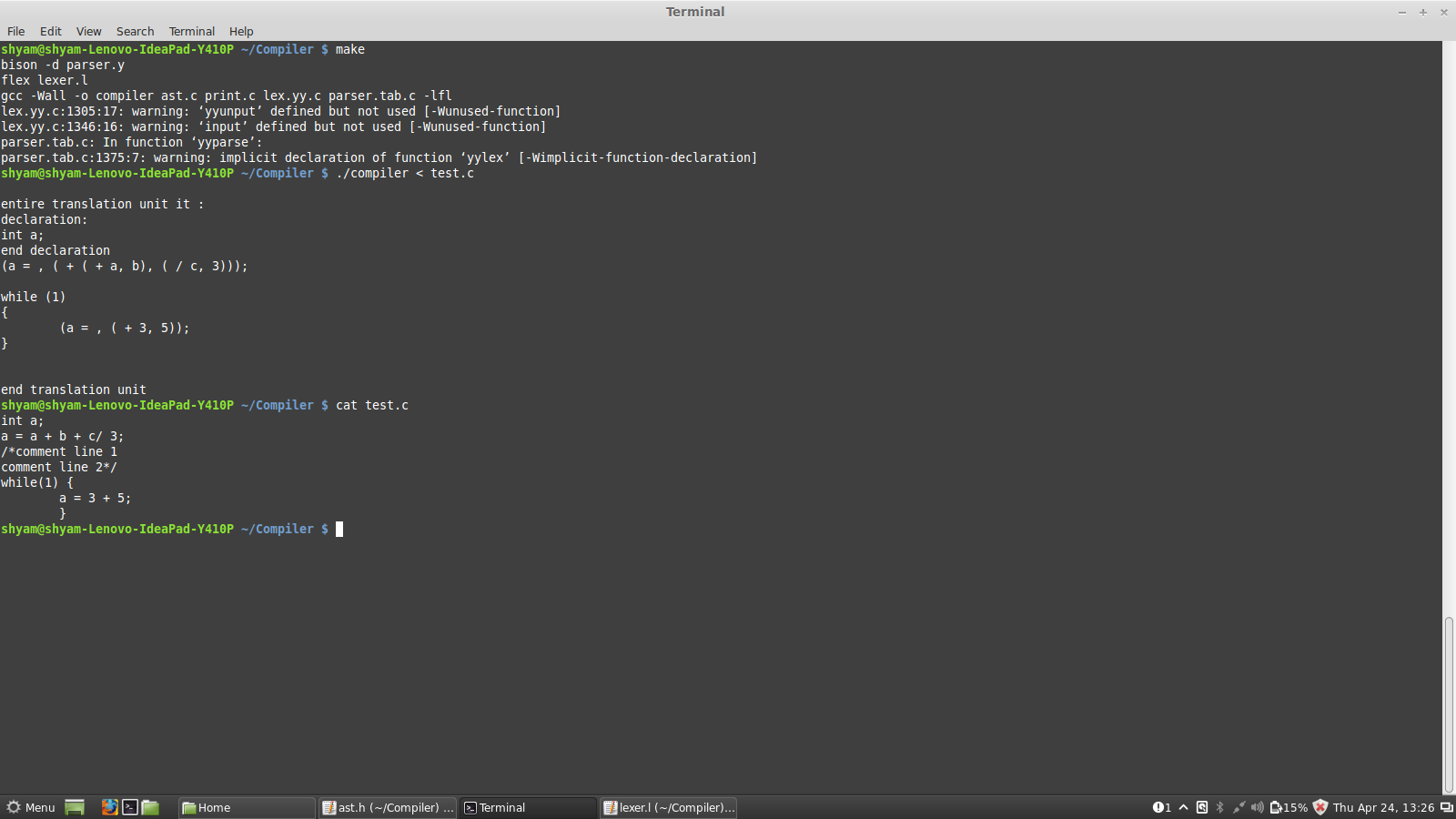
Implementing the semantic actions is conceptually simpler in recursive descent parsing because they are simply added to the recursive procedures.

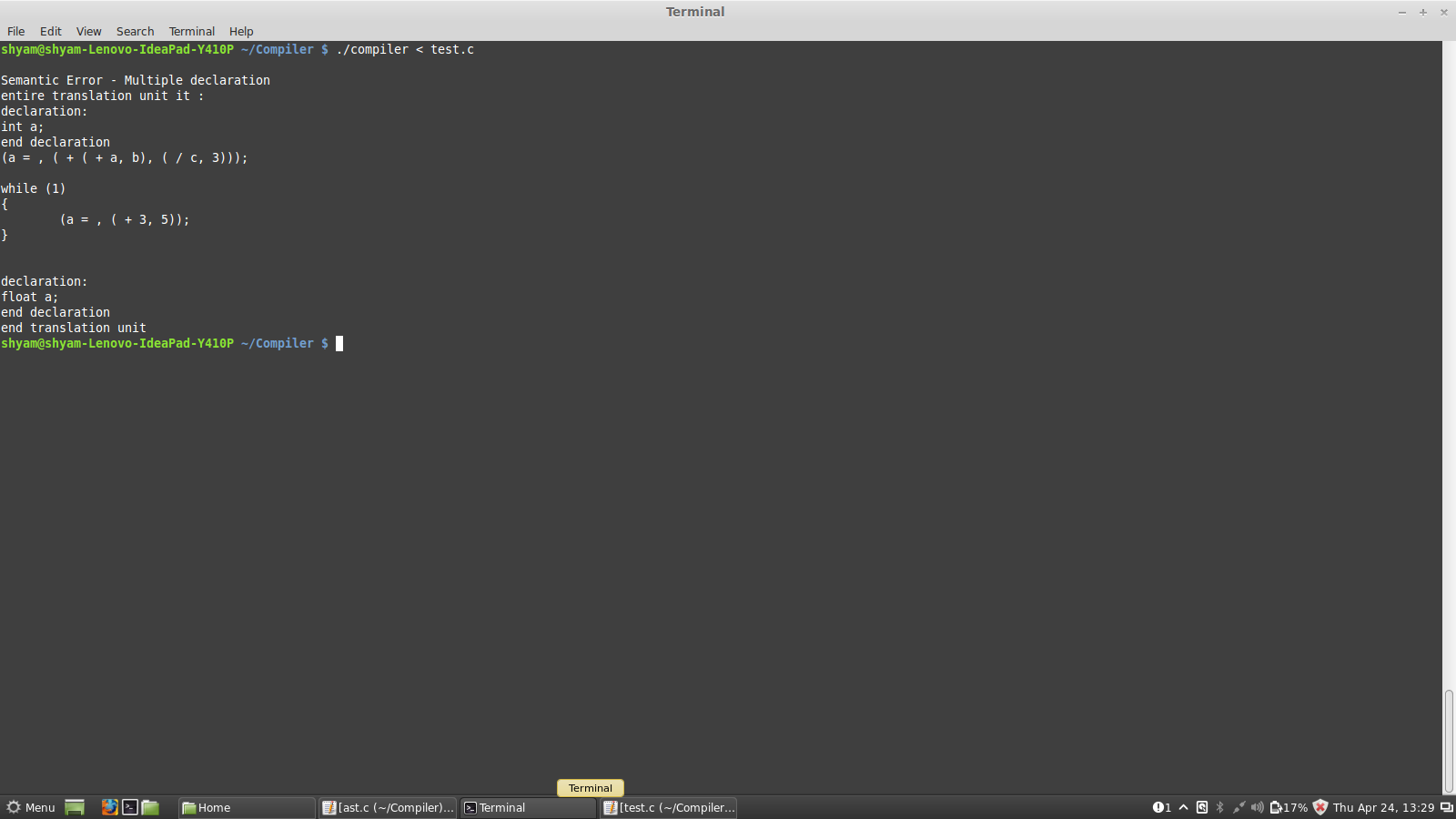
Implementing the semantic actions in a table action driven LL(1) parser requires the addition of a third type of variable to the productions and the necessary software routines to process it.

**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**

****

****