RV COLLEGE OF ENGINEERING ®
BENGALURU-560059
(Autonomous Institution Affiliated to VTU, Belagavi)

"Intermediate Code Generation"

Report

Compiler Design

(18IS54)

Submitted By

Ankit Kumar Singh (1RV18IS007)

Under the Guidance of

B. K. Srinivas

Asst. Professor

in partial fulfillment for the award of degree of

Bachelor of Engineering

in

INFORMATION SCIENCE AND ENGINEERING

2020-21

Abstract

After the lexical phase, the compiler enters the syntax analysis phase. This analysis is done by a parser. The parser uses the stream of tokens from the scanner and assigns them datatype if they are identifiers.

The parser code has a functionality of taking input through a file or through standard input. This makes it more user-friendly and efficient at the same time.

Contents:

Introduction:

Syntactic Analysis and Parser:

After the lexical analysis stage, we get the stream of tokens from source C code which is given as input to the parser. Parser verifies that a string of token names can be generated by the grammar of the source language. We expect the parser to report any syntax errors in an intelligible manner and to recover from the commonly occurring errors to continue processing the remainder of the program. Parser detects the following types of errors:

1. Errors in structure
2. Missing operator
3. Misspelt keywords
4. Unbalanced parenthesis

Conceptually, for well-formed programs, the parser constructs a parse tree and passes it to the rest of the compiler for further processing.
There are generally 3 types of parsers for grammars:

1. Universal
2. Top-down
3. Bottom-up

The methods commonly used in compilers can be classified as being either top-down (parse from root to leaves) or bottom-up (parse from leaves to root).

Yacc Script

Yacc provides a general tool for describing the input to a computer program. The Yacc user specifies the structures of his input, together with code to be invoked as each such structure is recognized. Yacc turns such a specification into a subroutine that handles the input process frequently, it is convenient and appropriate to have most of the flow of control in the user's application handled by this subroutine. Lexer can be used to make a simple parser. But it needs making extensive use of the user-defined states.

The input subroutine produced by Yacc calls a user-supplied routine to return the next basic input item. Thus, the user can specify his input in terms of individual input

characters, or in terms of higher-level constructs such as names and numbers. The user-supplied routine may also handle idiomatic features such as comment and continuation conventions, which typically
defy easy grammatical specification.

Yacc is written in portable C. The class of specifications accepted is a very general one: LALR(1) grammars with disambiguating rules.
The structure of our Yacc script is given below; files are divided into three sections, separated by lines that contain only two percent signs, as follows:

```
Definition section
%%
Rules section
%%
C code section
```

The definition section defines macros and imports header files written in C. It is also possible to write any C code here, which will be copied verbatim into the generated source file.
In the rules section, each grammar rule defines a symbol in terms of:

1. Other symbols
2. Tokens (or terminal symbols) which come from the lexer.

Each rule can have an associated action, which is executed *after* all the component symbols of the rule have been parsed. Actions are basically C-program statements surrounded by curly braces.
The C code section contains C statements and functions that are copied verbatim to the generated source file. These statements presumably contain code called by the rules in the rules section. In large programs, it is more convenient to place this code in a separate file linked in at compile time.

C Program

This section describes the input C program which is fed to the yacc script for parsing.
The workflow is explained as under:

1. Compile the script using Yacc tool

   ```
   $ yacc -d c_parser.y
   ```

2. Compile the flex script using Flex tool

   ```
   $ flex c_lexer.l
   ```

3. After compiling the lex file, a lex.yy.c file is generated. Also, y.tab.c and y.tab.h files are generated after compiling the yacc script.
4. The three files, lex.yy.c, y.tab.c and y.tab.h are compiled together with the options –ll and –ly

```
$ gcc -o compiler lex.yy.c y.tab.h y.tab.c -ll -ly
```

5. The executable file is generated, which on running parses the C file given as a command line input

```
$ ./compiler test.c
```

The script also has an option to take standard input instead of taking input from a file.

Design of Program

```
%{
      void yyerror(char* s);
      int yylex();
      #include "stdio.h"
      #include "stdlib.h"
      #include "ctype.h"
      #include "string.h"
      void ins();
      void insV();
      int flag=0;

      #define ANSI_COLOR_RED           "\x1b[31m"
      #define ANSI_COLOR_GREEN    "\x1b[32m"
      #define ANSI_COLOR_CYAN          "\x1b[36m"
      #define ANSI_COLOR_RESET    "\x1b[0m"

      extern char curid[20];
      extern char curtype[20];
      extern char curval[20];

%}

%nonassoc IF
%token INT CHAR FLOAT DOUBLE LONG SHORT SIGNED UNSIGNED STRUCT
%token RETURN MAIN
%token VOID
%token WHILE FOR DO
%token BREAK
%token ENDIF
```

```
%expect 2

%token identifier
%token integer_constant string_constant float_constant
character_constant

%nonassoc ELSE

%right leftshift_assignment_operator rightshift_assignment_operator
%right XOR_assignment_operator OR_assignment_operator
%right AND_assignment_operator modulo_assignment_operator
%right multiplication_assignment_operator
division_assignment_operator
%right addition_assignment_operator subtraction_assignment_operator
%right assignment_operator

%left OR_operator
%left AND_operator
%left pipe_operator
%left caret_operator
%left amp_operator
%left equality_operator inequality_operator
%left lessthan_assignment_operator lessthan_operator
greaterthan_assignment_operator greaterthan_operator
%left leftshift_operator rightshift_operator
%left add_operator subtract_operator
%left multiplication_operator division_operator modulo_operator

%right SIZEOF
%right tilde_operator exclamation_operator
%left increment_operator decrement_operator


%start program

%%
program
                : declaration_list;
```

```
declaration_list
                : declaration D

D
                : declaration_list
                | ;

declaration
                : variable_declaration
                | function_declaration
                | structure_definition;

variable_declaration
                : type_specifier variable_declaration_list ';'
                | structure_declaration;

variable_declaration_list
                : variable_declaration_identifier V;

V
                : ',' variable_declaration_list
                | ;

variable_declaration_identifier
                : identifier { ins(); } vdi;

vdi : identifier_array_type | assignment_operator expression ;

identifier_array_type
                : '[' initilization_params
                | ;

initilization_params
                : integer_constant ']' initilization
                | ']' string_initilization;

initilization
                : string_initilization
                | array_initialization
```

```
                | ;

type_specifier
                : INT | CHAR | FLOAT | DOUBLE
                | LONG long_grammar
                | SHORT short_grammar
                | UNSIGNED unsigned_grammar
                | SIGNED signed_grammar
                | VOID ;

unsigned_grammar
                : INT | LONG long_grammar | SHORT short_grammar | ;

signed_grammar
                : INT | LONG long_grammar | SHORT short_grammar | ;

long_grammar
                : INT | ;

short_grammar
                : INT | ;

structure_definition
                : STRUCT identifier { ins(); } '{' V1  '}' ';';

V1 : variable_declaration V1 | ;

structure_declaration
                : STRUCT identifier variable_declaration_list;


function_declaration
                : function_declaration_type
function_declaration_param_statement;

function_declaration_type
                : type_specifier identifier '('  { ins();};

function_declaration_param_statement
```

```
                : params ')' statement;

params
                : parameters_list | ;

parameters_list
                : type_specifier parameters_identifier_list;

parameters_identifier_list
                : param_identifier
parameters_identifier_list_breakup;

parameters_identifier_list_breakup
                : ',' parameters_list
                | ;

param_identifier
                : identifier { ins(); } param_identifier_breakup;

param_identifier_breakup
                : '[' ']'
                | ;

statement
                : expression_statment | compound_statement
                | conditional_statements | iterative_statements
                | return_statement | break_statement
                | variable_declaration;

compound_statement
                : '{' statment_list '}' ;

statment_list
                : statement statment_list
                | ;

expression_statment
                : expression ';'
                | ';' ;
```

```
conditional_statements
                : IF '(' simple_expression ')' statement
conditional_statements_breakup;

conditional_statements_breakup
                : ELSE statement
                | ;

iterative_statements
                : WHILE '(' simple_expression ')' statement
                | FOR '(' expression ';' simple_expression ';'
expression ')'
                | DO statement WHILE '(' simple_expression ')' ';';

return_statement
                : RETURN return_statement_breakup;

return_statement_breakup
                : ';'
                | expression ';' ;

break_statement
                : BREAK ';' ;

string_initilization
                : assignment_operator string_constant { insV(); };

array_initialization
                : assignment_operator '{' array_int_declarations
'}';

array_int_declarations
                : integer_constant array_int_declarations_breakup;

array_int_declarations_breakup
                : ',' array_int_declarations
                | ;
```

```
expression
                : mutable expression_breakup
                | simple_expression ;

expression_breakup
                : assignment_operator expression
                | addition_assignment_operator expression
                | subtraction_assignment_operator expression
                | multiplication_assignment_operator expression
                | division_assignment_operator expression
                | modulo_assignment_operator expression
                | increment_operator
                | decrement_operator ;

simple_expression
                : and_expression simple_expression_breakup;

simple_expression_breakup
                : OR_operator and_expression
simple_expression_breakup | ;

and_expression
                : unary_relation_expression and_expression_breakup;

and_expression_breakup
                : AND_operator unary_relation_expression
and_expression_breakup
                | ;

unary_relation_expression
                : exclamation_operator unary_relation_expression
                | regular_expression ;

regular_expression
                : sum_expression regular_expression_breakup;

regular_expression_breakup
                : relational_operators sum_expression
                | ;
```

```
relational_operators
                : greaterthan_assignment_operator |
lessthan_assignment_operator | greaterthan_operator
                | lessthan_operator | equality_operator |
inequality_operator ;

sum_expression
                : sum_expression sum_operators term
                | term ;

sum_operators
                : add_operator
                | subtract_operator ;

term
                : term MULOP factor
                | factor ;

MULOP
                : multiplication_operator | division_operator |
modulo_operator ;

factor
                : immutable | mutable ;

mutable
                : identifier
                | mutable mutable_breakup;

mutable_breakup
                : '[' expression ']'
                | '.' identifier;

immutable
                : '(' expression ')'
                | call | constant;

call
```

```
                : identifier '(' arguments ')';

arguments
                : arguments_list | ;

arguments_list
                : expression A;

A
                : ',' expression A
                | ;

constant
                : integer_constant    { insV(); }
                | string_constant     { insV(); }
                | float_constant{ insV(); }
                | character_constant{ insV(); };

%%

extern FILE *yyin;
extern int yylineno;
extern char *yytext;
void insertSTtype(char *,char *);
void insertSTvalue(char *, char *);
void incertCT(char *, char *);
void printST();
void printCT();

int main(int argc , char **argv)
{
      yyin = fopen(argv[1], "r");
      yyparse();

      if(flag == 0)
      {
            printf(ANSI_COLOR_GREEN "Status: Parsing Complete - Valid"
ANSI_COLOR_RESET "\n");
            printf("%30s" ANSI_COLOR_CYAN "SYMBOL TABLE"
```

```
ANSI_COLOR_RESET "\n", " ");
            printf("%30s %s\n", " ", "------------");
            printST();

            printf("\n\n%30s" ANSI_COLOR_CYAN "CONSTANT TABLE"
ANSI_COLOR_RESET "\n", " ");
            printf("%30s %s\n", " ", "--------------");
            printCT();
      }
}

void yyerror(char *s)
{
      printf("%d %s %s\n", yylineno, s, yytext);
      flag=1;
      printf(ANSI_COLOR_RED "Status: Parsing Failed - Invalid\n"
ANSI_COLOR_RESET);
}

void ins()
{
      insertSTtype(curid,curtype);
}

void insV()
{
      insertSTvalue(curid,curval);
}

int yywrap()
{
      return 1;
}
```

Explanation

The lex code is detecting the tokens from the source code and returning the corresponding token to the parser. In phase 1 we were just printing the token and now we are returning the token so that the parser uses it for further computation. We are using the symbol table and constant table of the previous phase only. We added functions like insertSTtype(), insertSTvalue() and insertSTline() to the existing functions. Lexical Analyser installs the token in the symbol table whereas parser calls these functions to add the value of attributes like data type, value assigned to identifier and where the identifier was declared i.e. updates the information in the symbol table.

Declaration Section

In this section we have included all the necessary header files,function declaration and flag that was needed in the code.

Between declaration and rules section we have listed all the tokens which are returned by the lexer according to the precedence order. We also declared the operators here according to their associativity and precedence.This ensures the grammar we are giving to the parser is unambiguous as LALR(1) parser cannot work with ambiguous grammar.

Rules Section

In this section production rules for the entire C language are written. The rules are written in such a way that there is no left recursion and the grammar is also deterministic.
Non-deterministic grammar was converted to deterministic by applying left factoring. This was done so that grammar is for LL(1) parser. This is so because all LL(1) grammar are LALR(1) according to the concepts.
The grammar production does the syntax analysis of the source code. When a complete statement with proper syntax is matched by the parser.

Test Cases

Valid Test Cases:

```
================================  Running TestCase 1  ================================
Status: Parsing Complete - Valid
                          SYMBOL TABLE
                           ------------
    SYMBOL |           CLASS |       TYPE |      VALUE |    LINE NO
-------------------------------------------------------------------------------
         i |      Identifier |        int |         10 |          4
         n |      Identifier |        int |            |          4
         x |      Identifier |        int |         10 |          9
       for |         Keyword |            |            |          7
      char |         Keyword |            |            |          5
        ch |      Identifier |       char |            |          5
        if |         Keyword |            |            |          8
       int |         Keyword |            |            |          3
      main |      Identifier |        int |            |          3
     while |         Keyword |            |            |         10


                          CONSTANT TABLE
                           --------------
      NAME |            TYPE
-------------------------------------------------------------------------------
        10 | Number Constant
         0 | Number Constant
```

```
================================  Running TestCase 2  ================================
Status: Parsing Complete - Valid
                          SYMBOL TABLE
                           ------------
    SYMBOL |           CLASS |       TYPE |      VALUE |    LINE NO
-------------------------------------------------------------------------------
         a |      Identifier |        int |          2 |          8
         b |      Identifier |        int |            |          8
         c |      Identifier |        int |            |          8
         d |      Identifier |        int |            |          8
         e |      Identifier |        int |            |          8
         f |      Identifier |        int |            |          8
         g |      Identifier |        int |            |          8
         h |      Identifier |        int |            |          8
         x |      Identifier |       char |            |          3
      char |         Keyword |            |            |          3
       fun |      Identifier |        int |            |          3
    return |         Keyword |            |            |          4
       int |         Keyword |            |            |          3
      main |      Identifier |       void |            |          7
      void |         Keyword |            |            |          7


                          CONSTANT TABLE
                           --------------
      NAME |            TYPE
-------------------------------------------------------------------------------
         2 | Number Constant
```

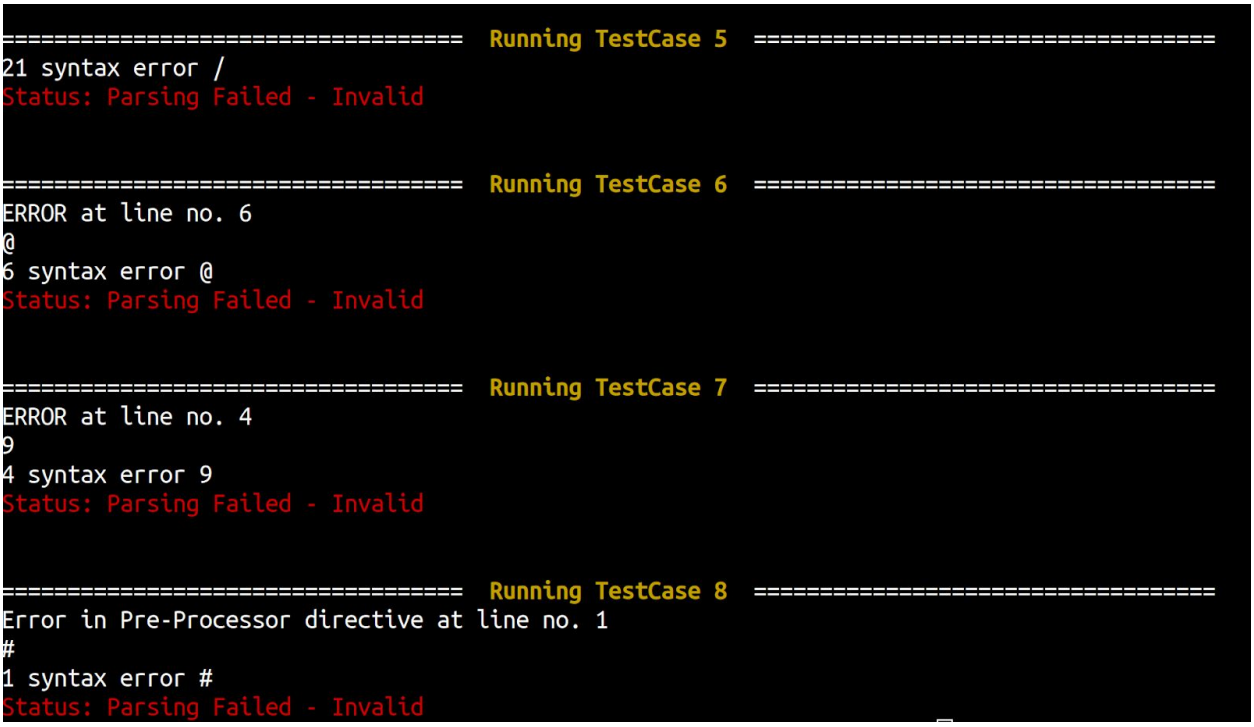
```
================================  Running TestCase 3  ================================
Status: Parsing Complete - Valid
                            SYMBOL TABLE
                             ------------
    SYMBOL |           CLASS |       TYPE |      VALUE |    LINE NO
-------------------------------------------------------------------------------
         a |      Identifier |        int |          0 |          5
         b |      Identifier |        int |          0 |         17
       int |         Keyword |            |            |          3
      main |      Identifier |        int |            |          3
    printf |      Identifier |            |       "%d" |          8
     while |         Keyword |            |            |          6


                            CONSTANT TABLE
                             --------------
      NAME |            TYPE
-------------------------------------------------------------------------------
"Hello world" | String Constant
      "%d" | String Constant
         0 | Number Constant
         4 | Number Constant
         5 | Number Constant
```

```
================================  Running TestCase 4  ================================
Status: Parsing Complete - Valid
                            SYMBOL TABLE
                             ------------
    SYMBOL |           CLASS |       TYPE |      VALUE |    LINE NO
-------------------------------------------------------------------------------
         A | Array Identifier |       char |          0 |          6
         B | Array Identifier |       char |    "Hello" |          7
  unsigned |         Keyword |            |            |          9
         a |      Identifier |       char |          1 |          9
      char |         Keyword |            |            |          6
        ch |      Identifier |       char |        'B' |          8
    return |         Keyword |            |            |         12
       int |         Keyword |            |            |          4
      main |      Identifier |        int |            |          4
    printf |      Identifier |            | "String = %s Value of Pi = %f" |         10


                            CONSTANT TABLE
                             --------------
      NAME |            TYPE
-------------------------------------------------------------------------------
       'B' | Character Constant
"#define MAX 10" | String Constant
   "Hello" | String Constant
"String = %s Value of Pi = %f" | String Constant
      3.14 | Floating Constant
         0 | Number Constant
         1 | Number Constant
```

Invalid Test Cases:



Implementation:

The lexer code submitted in the previous phase took care of most of the features of C using regular expressions. Some special corner cases were taken care of using custom regex.

These were:

A. The Regex for Identifiers
B. Multiline comments should be supported
C. Literals
D. Error Handling for Incomplete String
E. Error Handling for Nested Comments

The parser code requires exhaustive token recognition and because of this reason, we utilised the lexer code given under the C specifications with the parser. The parser implements C grammar using a number of production rules.

The parser takes tokens from the lexer output, one at a time and applies the corresponding production rules to append to the symbol table with type , value and line of declaration. If the parsing is not successful, the parser outputs the line number with the corresponding error.

The following functions were written in order to maintain symbol table:

1. `LookupST()` - This function checks whether the token is already present in the

symbol table or not. If yes it returns 1 else 0.(Called by Scanner)

2. `InsertST()` - This function installs the token in the symbol table if it is not already present along with the token class.(Called by Scanner)
3. `InsertSTtype()` - This function appends the datatype of the identifier in the symbol table. (Called by Parser).
4. `InsertSTvalue()`- This function appends the value of the identifier in the symbol table. (Called by Parser).
5. `InsertSTline()`- This function appends the line of declaration of the identifier in the symbol table. (Called by Parser).
6. `LookupCT()` - This function checks whether the token is already present in the constant table or not. If yes it returns 1 else 0.(Called by Scanner).
7. `InsertST()` - This function installs the token in the constant table if it is not already present along with the token class.(Called by Scanner)
8. `PrintST()` - This function displays the entire content of the symbol table.
9. `PrintCT()` - This function displays the entire content of the constant table.