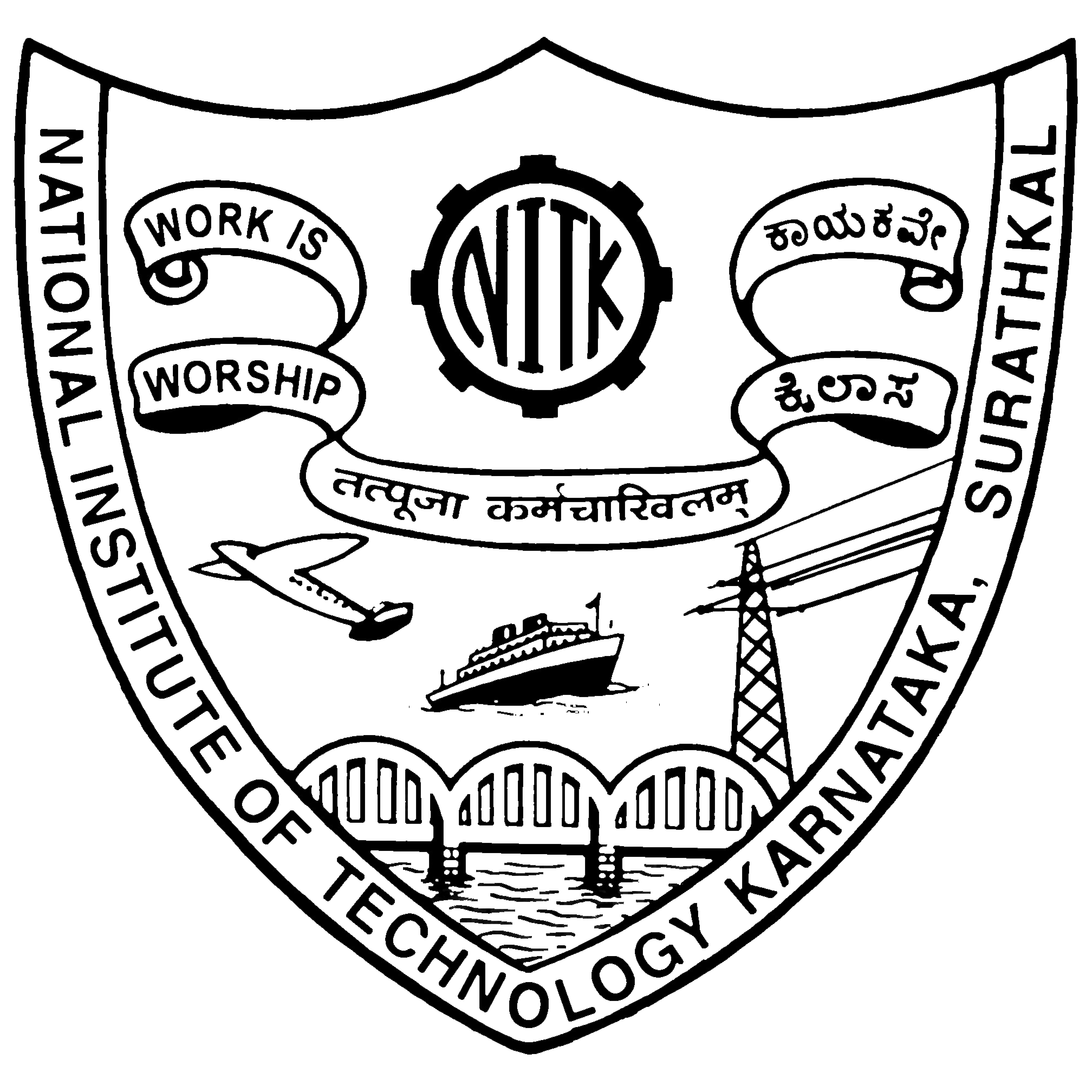
Syntax Analyser for the C Language



National Institute of Technology Karnataka Surathkal

**Date: 25 – 01 – 2017**

**Submitted To: Ms. Uma Priya**

**Group Members:**

Ananda Rao H – 14CO103

Sandeep N Menon - 14CO140

**Abstract:**

A compiler translates the code written in one language to some other language without changing the meaning of the program. It is also expected that a compiler should make the target code efficient and optimized in terms of time and space. Compiler design principles provide an in-depth view of translation and optimization process. Compiler design covers basic translation mechanism and error detection & recovery. It includes lexical, syntax, and semantic analysis as front end, and code generation and optimization as back-end. Compiler designers use a scanner generator framework like Lex to generate tokens from the input source code and a parser generator framework like YACC to construct abstract syntax tree. We use FLEX as the scanner generator tool and YACC as the parser generator tool to design a simple compiler. Three-ad three-address code is used as an intermediate language within compilers dress code is used as an intermediate language within compilers.

Table of Contents

[Introduction 4](#_Toc474276134)

[Lexical Analysis 4](#_Toc474276135)

[Yacc Script 4](#_Toc474276136)

[C Program 5](#_Toc474276137)

[Design of Programs 5](#_Toc474276138)

[Flow 5](#_Toc474276139)

[Code 5](#_Toc474276140)

[Parser.y 5](#_Toc474276141)

[Test Case 9](#_Toc474276142)

[Source Code 1 9](#_Toc474276143)

[Output 1 9](#_Toc474276144)

[Source Code 2 9](#_Toc474276145)

[Output 2 10](#_Toc474276146)

[Explanation 10](#_Toc474276147)

[Test Cases 11](#_Toc474276148)

[Without Errors 11](#_Toc474276149)

[With Errors 12](#_Toc474276150)

[Implementation 12](#_Toc474276151)

[Future work 15](#_Toc474276152)

# Introduction

## Lexical Analysis

Syntax analysis or parsing is the second phase of a compiler. In this chapter, we shall learn the basic concepts used in the construction of a parser.

We have seen that a lexical analyser can identify tokens with the help of regular expressions and pattern rules. But a lexical analyser cannot check the syntax of a given sentence due to the limitations of the regular expressions. Regular expressions cannot check balancing tokens, such as parenthesis. Therefore, this phase uses context-free grammar (CFG), which is recognized by push-down automata.

## Yacc Script

Computer program input generally has some structure; in fact, every computer program that does input can be thought of as defining an ``input language'' which it accepts. An input language may be as complex as a programming language, or as simple as a sequence of numbers. Unfortunately, usual input facilities are limited, difficult to use, and often are lax about checking their inputs for validity.

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 han- dles 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.

The yacc file is divided into three sections, separated by lines that contain only two percent signs, as follows:

*Definition section*

*%%*

*Rules section*

*%%*

*Subroutines section*

# C Program

This section describes the input C program which is fed to the yacc script in order to generate the yacc file after taking all the rules mentioned in account. Finally, a file called y.tab.c is generated, which when executed checks the syntax of the code.

# Design of Programs

## Flow

## Code

We have implemented the syntax analysis phase using 2 codes.

1. Scanner.l: This lex code takes the C program as the input and extracts the tokens from it. These tokens are classified based on their type and stored in a text file.
2. Parser.y : This yacc code takes the token stream as the input and validates using grammar specified in the code.

### Parser.y

%{

#include <math.h>

#include <stdio.h>

#include <stdlib.h>

%}

%token ID NUM SIZEOF REAL

%token PTR DOT

%token TYPEDEF INT FLOAT VOID STRUCT

%token IF ELSE WHILE RETURN FOR DO SWITCH CASE BREAK DEFAULT CONTINUE

%token PRINTF SCANF

%token STRING

%token PREPROC

%token MUL\_ASSIGN SUB\_ASSIGN DIV\_ASSIGN ADD\_ASSIGN

%left GT LT LE GE NE EQ

%left AND OR

%right '='

%left '+' '-'

%left '\*' '/'

%%

start: Function start

| Declaration

| PREPROC start

;

/\* Declaration block \*/

Declaration: Type Assignment ';'

| Assignment ';'

| FunctionCall ';'

| ArrayUsage ';'

| Type ArrayUsage ';'

| StructStmt ';'

| error

;

/\* Assignment block \*/

Assignment: ID assign\_operator Assignment

| ID assign\_operator FunctionCall

| ID assign\_operator ArrayUsage

| ArrayUsage assign\_operator Assignment

| ID ',' Assignment

| NUM ',' Assignment

| ID '+' Assignment

| ID '-' Assignment

| ID '\*' Assignment

| ID '/' Assignment

| NUM '+' Assignment

| NUM '-' Assignment

| NUM '\*' Assignment

| NUM '/' Assignment

| REAL '+' Assignment

| REAL '-' Assignment

| REAL '\*' Assignment

| REAL '/' Assignment

| '\'' Assignment '\''

| '(' Assignment ')'

| '-' '(' Assignment ')'

| '-' NUM

| '-' REAL

| '-' ID

| NUM

| REAL

| ID

;

assign\_operator: '='

| MUL\_ASSIGN

| SUB\_ASSIGN

| DIV\_ASSIGN

| ADD\_ASSIGN

;

/\* Function Call Block \*/

FunctionCall : ID'('')'

| ID'('Assignment')'

;

/\* Array Usage \*/

ArrayUsage : ID'['Assignment']'

;

/\* Function block \*/

Function: Type ID '(' ArgListOpt ')' CompoundStmt

;

ArgListOpt: ArgList

|

;

ArgList: ArgList ',' Arg

| Arg

;

Arg: Type ID

;

CompoundStmt: '{' StmtList '}'

;

StmtList: StmtList Stmt

|

;

Stmt: WhileStmt

| Declaration

| ForStmt

| IfStmt

| PrintFunc

| ';'

| RETURN ';'

| RETURN Assignment ';'

| BREAK ';'

| CONTINUE ';'

;

/\* Type Identifier block \*/

Type: INT

| FLOAT

| VOID

;

/\* Loop Blocks \*/

WhileStmt: WHILE '(' Expr ')' Stmt

| WHILE '(' Expr ')' CompoundStmt

;

/\* For Block \*/

ForStmt: FOR '(' Expr ';' Expr ';' Expr ')' Stmt

| FOR '(' Expr ';' Expr ';' Expr ')' CompoundStmt

| FOR '(' Expr ')' Stmt

| FOR '(' Expr ')' CompoundStmt

;

/\* IfStmt Block \*/

IfStmt : IF '(' Expr ')' Stmt

| IF '(' Expr ')' CompoundStmt

| IF '(' Expr ')' CompoundStmt ELSE CompoundStmt

;

/\* Struct Statement \*/

StructStmt : STRUCT ID '{' Type Assignment '}'

;

/\* Print Function \*/

PrintFunc : PRINTF '(' Expr ')' ';'

;

/\*Expression Block\*/

Expr:

| Expr LE Expr

| Expr GE Expr

| Expr NE Expr

| Expr EQ Expr

| Expr GT Expr

| Expr LT Expr

| Assignment

| ArrayUsage

;

%%

#include "lex.yy.c"

#include "ctype.h"

int count=0;

int main(int argc,char \*argv[])

{

FILE \*file;

file = fopen(argv[1], "r");

if (!file)

{

fprintf(stderr, "Could not open %s\n", argv[1]);

exit(1);

}

yyin = file;

if(!yyparse())

printf("\nParsing done\n");

else

printf("\nParsing failed\n");

fclose(yyin);

return 0;

}

void yyerror(char \*s)

{

printf("%d : %s %s \n",yylineno,s,yytext);

}

## Test Case

### Source Code 1

#include <stdio.h>

int main**()**

**{**

float n **=** 2.2**;**

**if(**n**>**10**)**

**{**

n **\*=** 2**;**

**}**

**else**

**{**

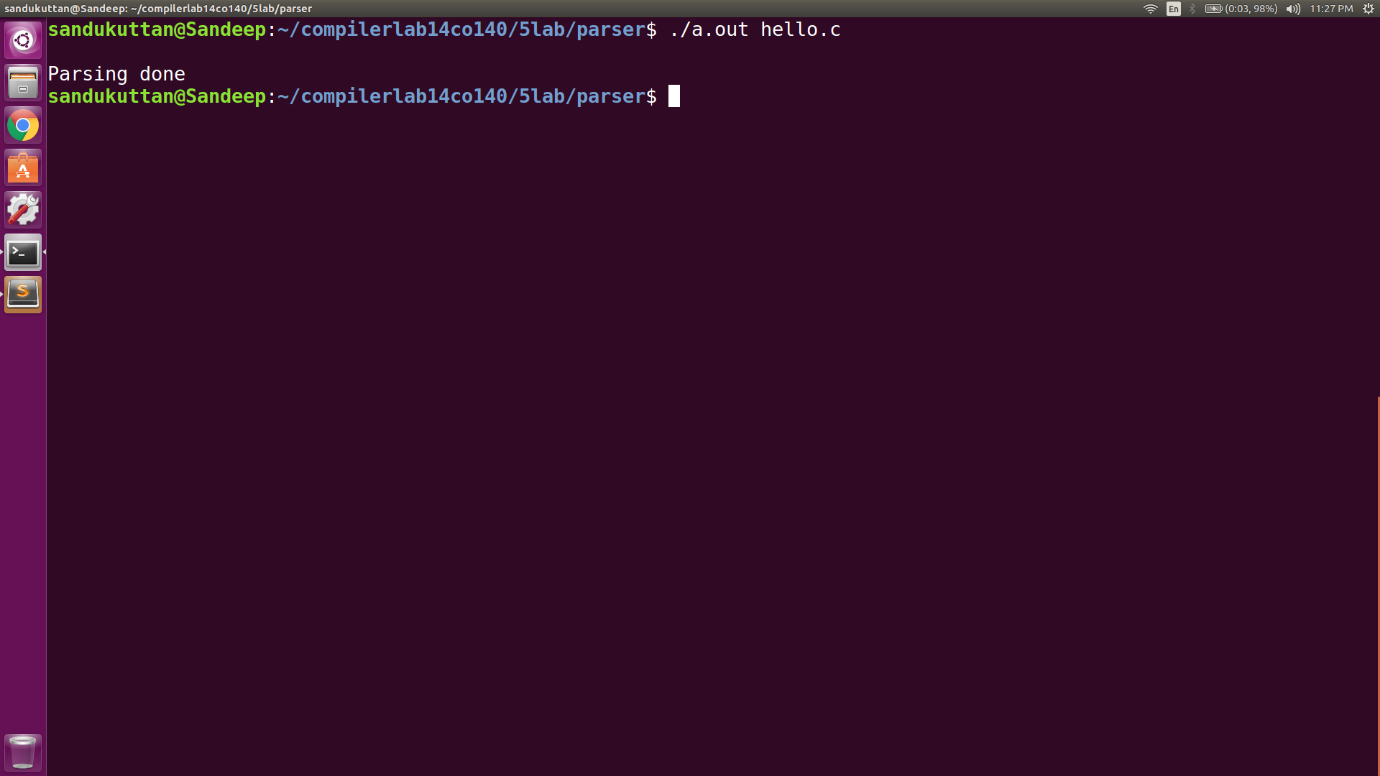
n**+=**10**;**

**}**

**return** 0 **;**

**}**

### Output 1



### Source Code 2

#include <stdio.h>

int main**()**

**{**

float n **=** 2.2**;**

**if(**n**>**10**)**

**{**

n **\*=** 2**;**

**else**

**{**

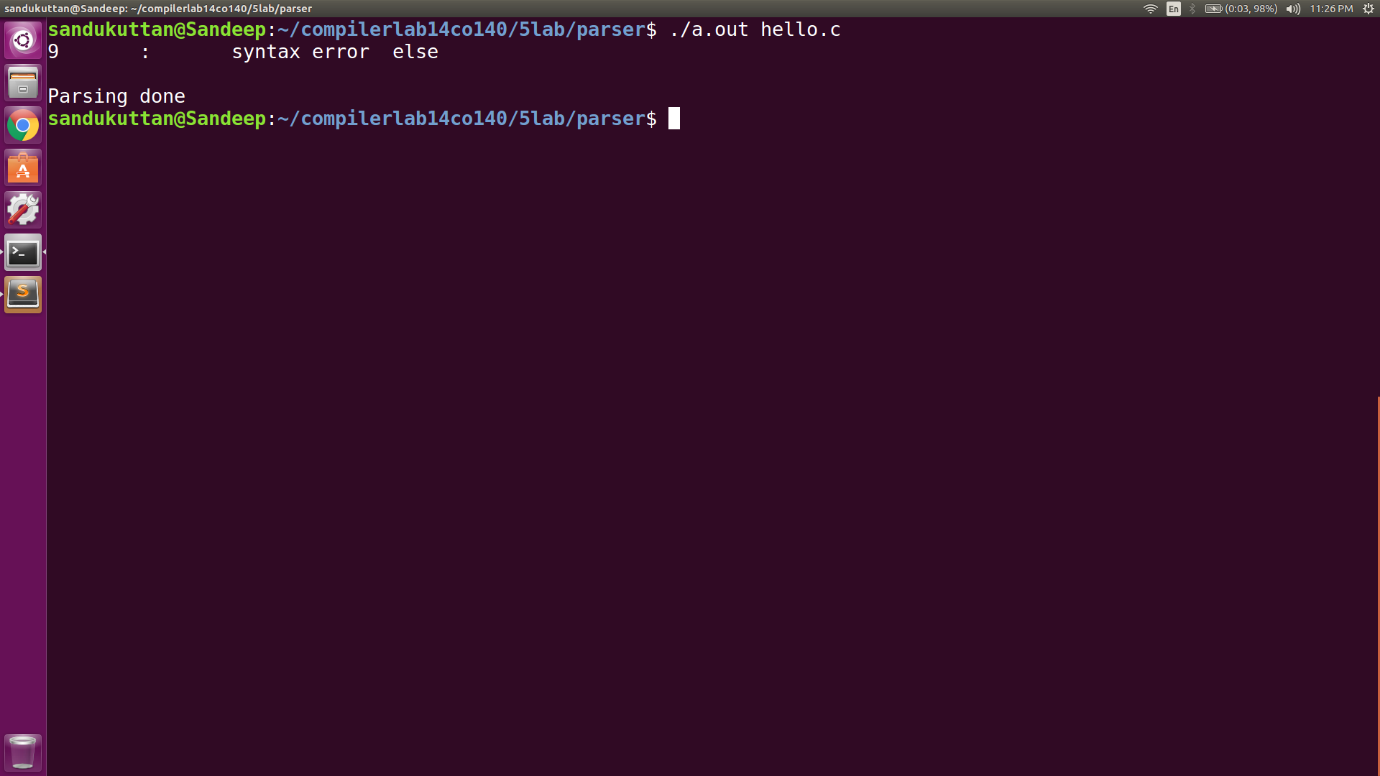
n**+=**10**;**

**}**

**return** 0 **;**

**}**

### Output 2

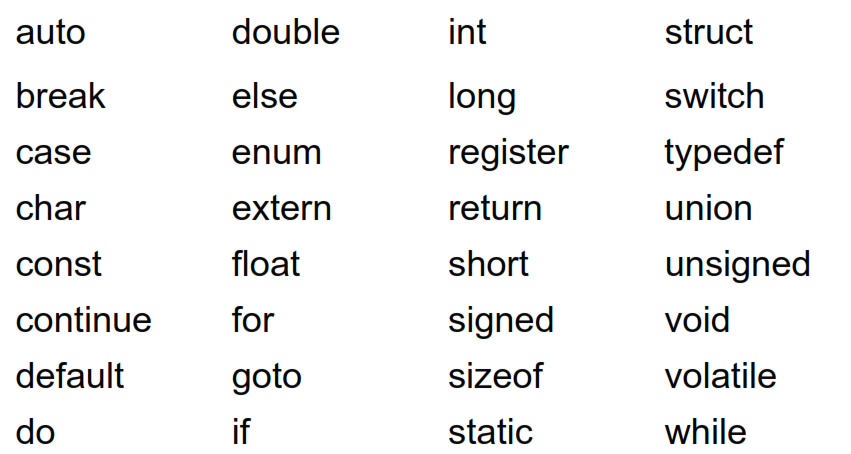


## Explanation

The yacc script recognises the following syntaxes in the C code using context free grammars:

* Pre-processor instructions
* Single declaration
* Multiple declaration
* Structure declaration
* Function declaration
* For statement
* While statement
* If statement
* If else statement
* Nested For
* Nested While
* Nested If Else
* Arithmetic expressions
* Relational expressions
* Assignment (Simple and compound)
* Array declaration and usage
* Input output statement

Keywords accounted for:



# Test Cases

## Without Errors

|  |  |  |  |
| --- | --- | --- | --- |
| Serial No | Test Case | Expected Output | Status |
| 1 | int a=25; | KEYWORD: int  IDENTIFIER: a  OPERATOR: =  INTEGER: 25  SEMICOLON | PASS |
| 2 | if ( a < b )  {  a += 1;  }  else  {  b += 1;  } | IF '(' Expr ')' CompoundStmt  ELSE  CompoundStmt | PASS |
| 3 | float b = 1.2; | ({digit}\*\.{digit}+)|({digit}+\.{digit}\*) | PASS |

## With Errors

|  |  |  |  |
| --- | --- | --- | --- |
| Serial No | Test Case | Expected Output | Status |
| 1 | int 3a; | Invalid Identifier (Lexical Error) | FAIL |
| 2 | /\* dead meat | Lexical Error | FAIL |
| 3 | while ( a < b; )  {  a += 1;  b -= 1;  } | Syntax Error. Did not expect ‘;’ | FAIL |
| 4 | for(i=1;i <  563  652222222233333333333332 |  |  |

# Implementation

The CFG for most of the features of C are fairly straightforward. However, a few features require a significant amount of thought, such as:

* **The CFG for nested if:** The nested if statements are recognised using the following CFG rules.

IfStmt : IF '(' Expr ')' Stmt

| IF '(' Expr ')' CompoundStmt

| IF '(' Expr ')' CompoundStmt ELSE CompoundStmt

;

CompoundStmt: '{' StmtList '}'

;

StmtList: StmtList Stmt

|

;

Stmt: WhileStmt

| Declaration

| ForStmt

| IfStmt

| PrintFunc

| ';'

| RETURN ';'

| RETURN Assignment ';'

| BREAK ';'

| CONTINUE ';'

;

* **Arrays:** Array is a collection of homogenous data stored in continuous memory.

RE - {alpha}({alpha}|{digit})\*\[{digit}\*\]

CFG - ArrayUsage : ID'['Assignment']'

;

* **Assignment:** Assignment operators with various operands are implemented.

Assignment: ID assign\_operator Assignment

| ID assign\_operator FunctionCall

| ID assign\_operator ArrayUsage

| ArrayUsage assign\_operator Assignment

| ID ',' Assignment

| NUM ',' Assignment

| ID '+' Assignment

| ID '-' Assignment

| ID '\*' Assignment

| ID '/' Assignment

| NUM '+' Assignment

| NUM '-' Assignment

| NUM '\*' Assignment

| NUM '/' Assignment

| REAL '+' Assignment

| REAL '-' Assignment

| REAL '\*' Assignment

| REAL '/' Assignment

| '\'' Assignment '\''

| '(' Assignment ')'

| '-' '(' Assignment ')'

| '-' NUM

| '-' REAL

| '-' ID

| NUM

| REAL

| ID

;

* **Functions:** Function calls and functions are implemented using the following rules.

Function: Type ID '(' ArgListOpt ')' CompoundStmt

;

FunctionCall : ID'('')'

| ID'('Assignment')'

;

* **Expressions:** Arithmetic and Logical operators are implemented using the following rules.

Expr:

| Expr LE Expr

| Expr GE Expr

| Expr NE Expr

| Expr EQ Expr

| Expr GT Expr

| Expr LT Expr

| Assignment

| ArrayUsage

;

At the end of the this phase, the parser updates the tokens into a text file along with its recognised attributes i.e line number and token type.

The text file is given as input to the symbol table generator. This generator populates the symbol table.

We use the following technique to implement this:

* We maintain a map of structures with the token name as the key.
* The structure contains the token attributes ( name, id, list of types )
* Two functions have been implemented, namely insert() and lookup() which is used for adding a new identifier/constant to the map and for checking if the identifier/constant is already present in the map, respectively.
* Whenever we encounter an identifier/constant, we call the insert() function which in turns call lookup() and adds it to the map.
* All the identifiers/constants identified are printed on the screen.

# Future work

The yacc script presented in this report takes care of all the rules of C language, but is not fully exhaustive in nature. Our future work would include making the script even more robust in order to handle all aspects of C language and making it more efficient.

Some features to be added:

* enums
* unions
* conditional statements ( ternary operators )