**NATIONAL INSTITUTE OF TECHNOLOGY, DELHI**

Department of Computer Science and Engineering

**CSB353: Compiler Design**

**Project Work**

**C - Ladder**

Submitted to:

Dr. Shelly Sachdeva, Dept. of CSE

Submitted By:

Gourav Bansal (191220020)

Kishan Srivastava (191220030)

Vardan Agarwal (191220051)

CSE 3rd year

Contents

1 – Introduction

2 – Acknowledgement

3 – Problem Statement and Problem Domain

4 – Lexical Analysis Phase

5 – Syntax Analysis Phase

6 – Semantic Analysis Phase

7 – Intermediate Code Generation

8 – Code Optimization

9 – Target Code Generation

10 - References

Introduction

A compiler is a translating program that translates high-level language instructions to machine-level language. A program that is input to the compiler is called a **Source program**. This program is now converted to a machine-level language by a compiler known as the **Object code.**

Following students of CSE 3rd year are the part of a team working on this project:

1. **Gourav Bansal – 191220020**
2. **Kishan Srivastava – 191220030**
3. **Vardan Agarwal – 191220051**

Acknowledgement

We would like to express our special thanks of gratitude to Dr. Shelly Sachdeva as well as all the TAs who gave us the golden opportunity to do this wonderful project on the topic **C - Ladder**, which also helped us in doing a lot of Research and we came to know about so many new things, we are really thankful to them.

Secondly we would also like to thank our parents and mates who helped us a lot in finalizing this project within the limited time frame.

Problem statement

To demonstrate all stages involved in compilation of C files.

Problem domain

● Looping construct: while, for, do-while

● Data types: (signed/unsigned) int, float

● Arithmetic and Relational Operators

● Data structure: Arrays

● User defined functions

● Keywords of C language

● Single and Multi-line comments

● Identifiers and Constant errors

● Selection statement: (nested) if-else

LEXICAL ANALYSIS PHASE

The Lexical Analyzer is the first phase of a compiler's Analysis (front end) stage. In layman's terms, the Lexical Analyzer (or Scanner) scans through the input source program character by character, identifies 'Lexemes', and categorizes them into 'Tokens'.

'Tokens' are represented as a symbol table and are given as input to the Parser (second phase of the front end of a compiler).

* **TOKENS:**

Tokens are essentially just a group of characters with some meaning or relation. The Lexical Analyzer detects these tokens with the help of 'Regular Expressions'. While writing the Lexical Analyzer, we have to specify rules for each Token type using Regular Expression. These rules are used to check whether a specific group of characters fall under a given token category or not.

* **LEXEMES:**

Lexemes are instances of Tokens.

E.g., 'for' is an instance of the 'Keyword' Token

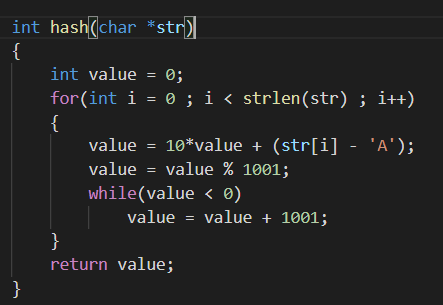
* **SYMBOL TABLE:**

A symbol table is generated in the Lexical Analyzer stage, which is a table with the columns' Symbol', 'Type', and 'Token ID'. The symbol is the Lexeme itself, the 'Type' is the token category, and the 'Token ID' is a unique ID given to a token used in the parser stage. There are no duplicate entries in a symbol table. Each symbol is recorded only once, even if there are multiple instances. A Lexical Analyzer is internally implemented based on the concept of FSMs (Finite State Machines). A DFA (Deterministic Finite State Automata) is internally built for each Token based on the Regular Expression provided. This is used to identify Lexemes and categorize them into Tokens.

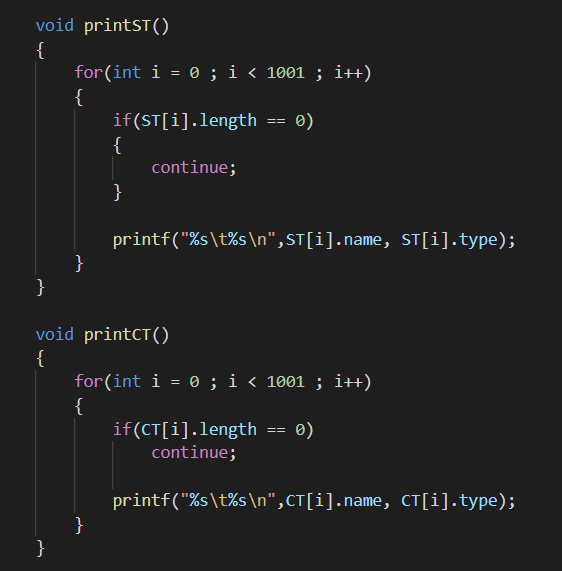
**EXPLANATION:**

Definition Section:

In the definition section of the program, all necessary header files were included. Apart from that structure declaration for both the symbol table and constant table were made. In order to convert a string of the source program into a particular integer value a hash function was written that takes a string as input and converts it into a particular integer value. Standard table operations like look-up and insert were also written. Linear Probing hashing technique was used to implement the symbol table i.e. if there is a collision, then after the point of collision, the table is searched linearly in order to find an empty slot. Functions to print the symbol table and constant table was also written.



Hash function used



Print symbol and constant table

Rules section:

In this section rules related to the specification of C language were written in the form Page 12 of valid regular expressions.

E.g. for a valid C identifier the regex written was [A-Za-z\_][A-Za-z\_0-9]\* which means that a valid identifier need to start with an alphabet or underscore followed by 0 or more occurrence of alphabets, numbers or underscore. In order to resolve conflicts we used lookahead method of scanner by which a scanner decides whether a expression is valid token or not by looking at its adjacent character.

E.g. in order to differentiate between comments and division operator lookahead characters of a valid operator were also given in the regular expression to resolve a conflict.

If none of the patterns matched with the input, we said it is a lexical error as it does not match with any valid pattern of the source language. Each character/pattern along with its token class was also printed.

DE "define"

IN "include"

operator [[<][=]|[>][=]|[=][=]|[!][=]|[>]|[<]|[\|][\|]|[&][&]|[\!]|[=]|[\^]|[\+][=]|[\-][=]|[\\*][=]|[\/][=]|[\%][=]|[\+][\+]|[\-][\-]|[\+]|[\-]|[\\*]|[\/]|[\%]|[&]|[\|]|[~]|[<][<]|[>][>]]

**%%**

\n   {yylineno++;}

([#][" "]\*({IN})[ ]\*([<]?)([A-Za-z]+)[.]?([A-Za-z]\*)([>]?))/["\n"|\/|" "|"\t"] {printf("%s \t-Pre Processor directive\n",yytext);}  //Matches #include<stdio.h>

([#][" "]\*({DE})[" "]\*([A-Za-z]+)(" ")\*[0-9]+)/["\n"|\/|" "|"\t"] {printf("%s \t-Macro\n",yytext);} //Matches macro

\/\/(.\*) {printf("%s \t- SINGLE LINE COMMENT\n", yytext);}

\/\\*([^\*]|[\r\n]|(\\*+([^\*/]|[\r\n])))\*\\*+\/  {printf("%s \t- MULTI LINE COMMENT\n", yytext);}

[ \n\t] ;

; {printf("%s \t- SEMICOLON DELIMITER\n", yytext);}

, {printf("%s \t- COMMA DELIMITER\n", yytext);}

\{ {printf("%s \t- OPENING BRACES\n", yytext);}

\} {printf("%s \t- CLOSING BRACES\n", yytext);}

\( {printf("%s \t- OPENING BRACKETS\n", yytext);}

\) {printf("%s \t- CLOSING BRACKETS\n", yytext);}

\[ {printf("%s \t- SQUARE OPENING BRACKETS\n", yytext);}

\] {printf("%s \t- SQUARE CLOSING BRACKETS\n", yytext);}

\: {printf("%s \t- COLON DELIMITER\n", yytext);}

\\ {printf("%s \t- FSLASH\n", yytext);}

\. {printf("%s \t- DOT DELIMITER\n", yytext);}

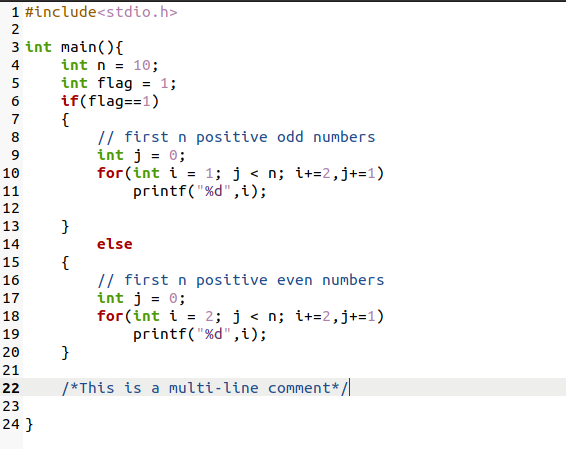
Some of the grammar rules

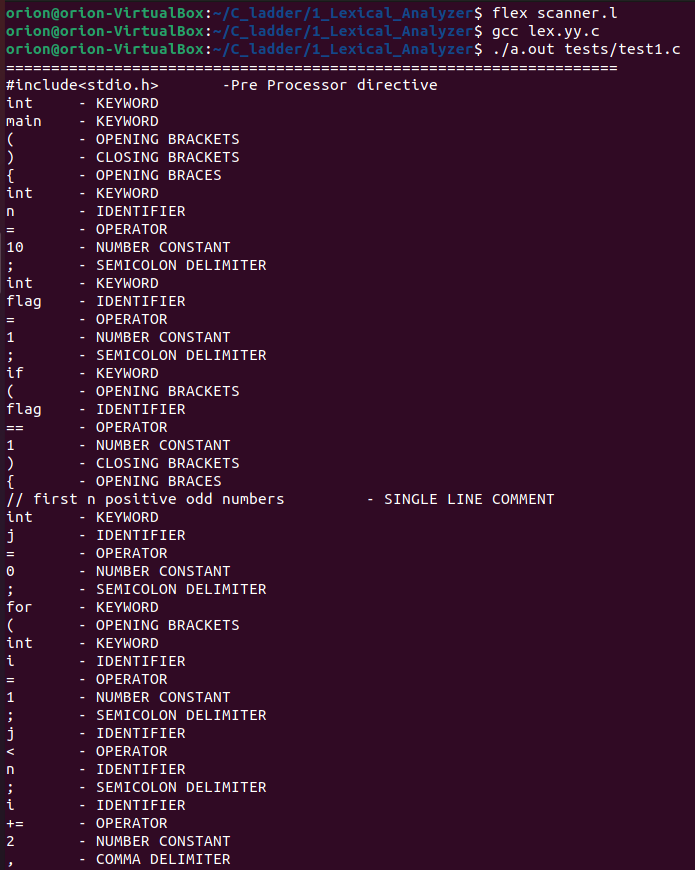
C code section:

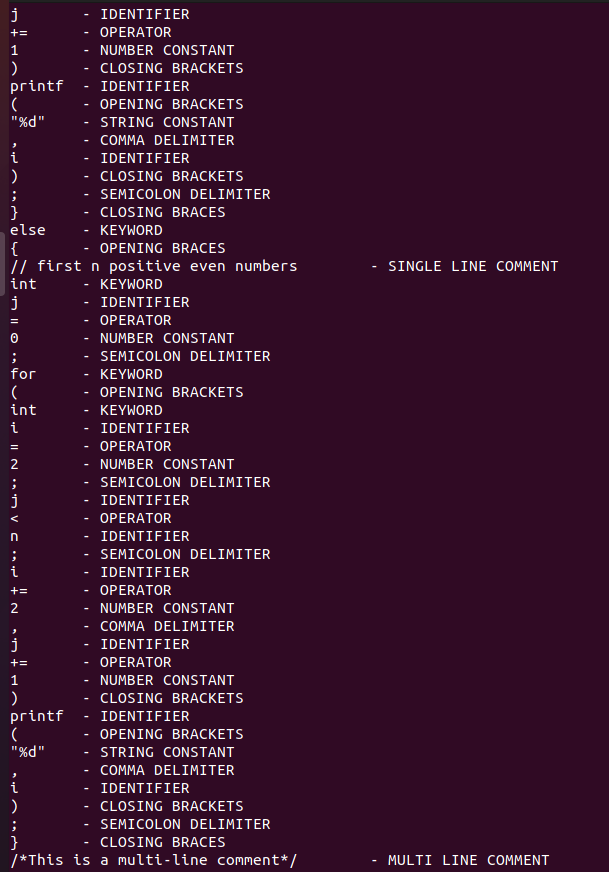
In this section both the tables (symbol and constant) were initialised to 0 and yylex( ) function was called to run the program on the given input file. After that, both the symbol table and constant table were printed in order to show the result.

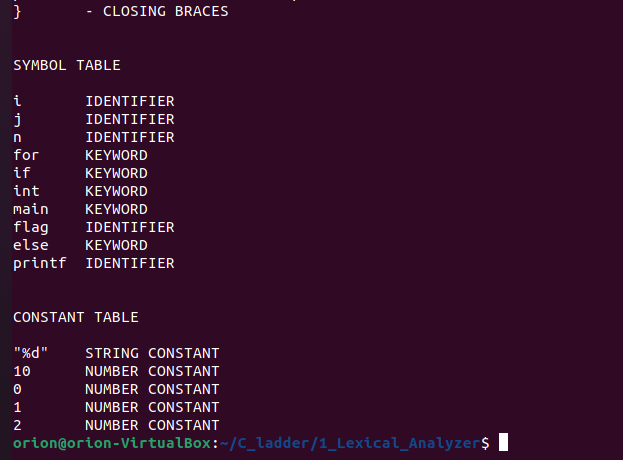
**TEST CASES:**

1. **test1.c**

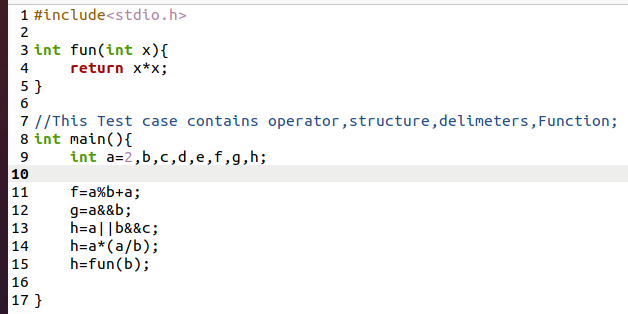
****

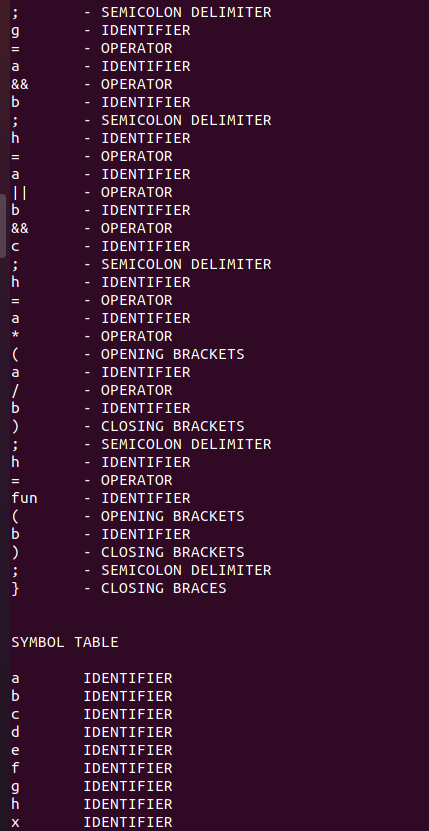
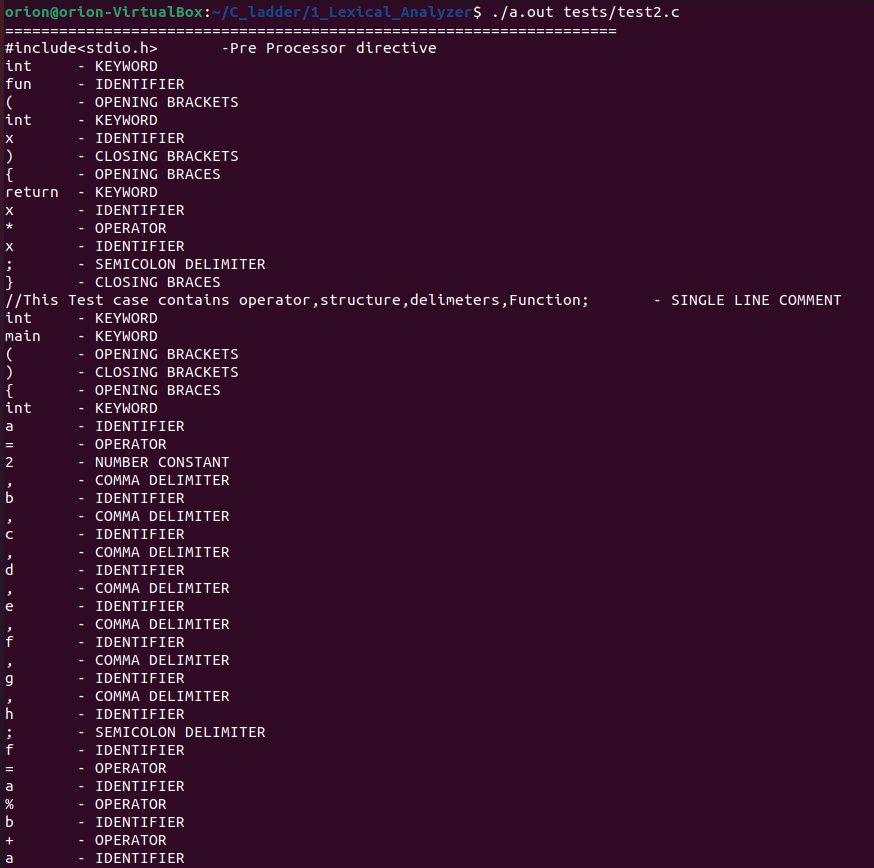
****

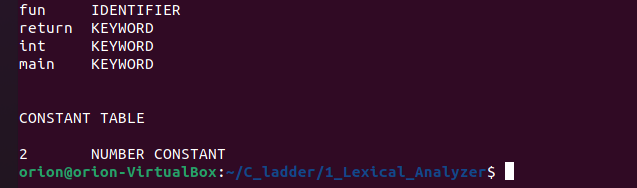
****

****

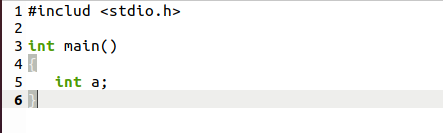
1. **test2.c**

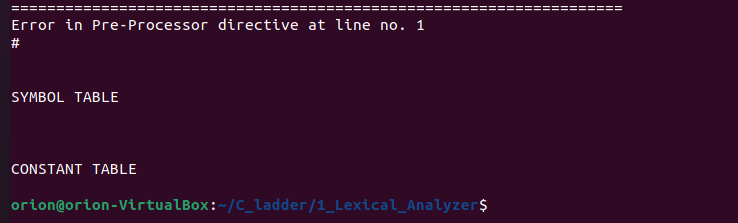


****

****

1. **test3.c**

****

****

SYNTAX ANALYSIS PHASE

After the lexical analysis stage, we get the stream of tokens from the source C code, given as input to the Parser. Parser verifies that the grammar of the source language can generate a string of token names. We expect the Parser to report any syntax errors intelligibly and 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. Misspelled 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 three types of parsers for grammar:

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

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

* **YACC**

Yacc provides a general tool for describing the input to a computer program. The Yacc user specifies the structures of his information and code to be invoked as each such structure is recognized. Yacc turns such a specification into a subroutine that handles the input process frequently, and 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 to make 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 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 accepted specifications are very general:

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.

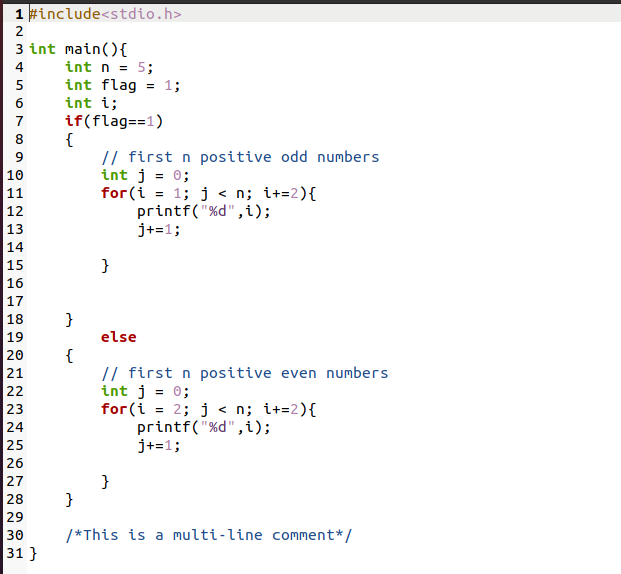
Each grammar rule defines a symbol in the rules section in terms of

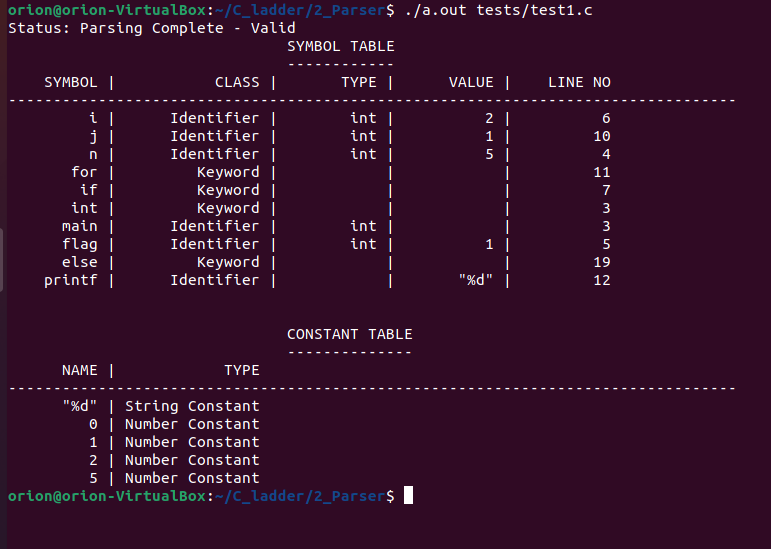
1. other symbols
2. Tokens (or terminal symbols) come from the lexer.

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

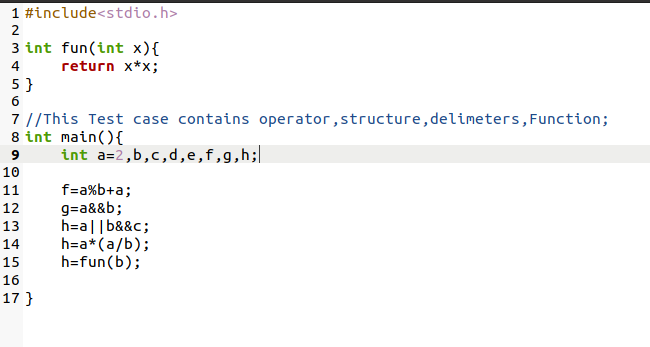
**TEST CASES:**

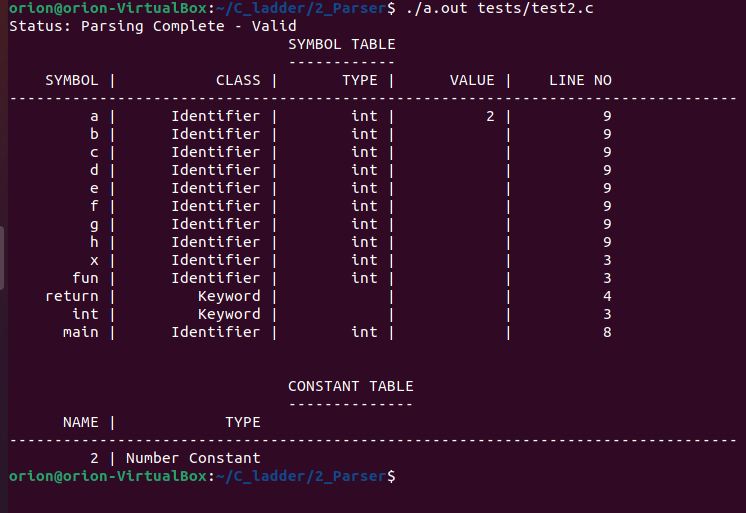
1. **test1.c**

****

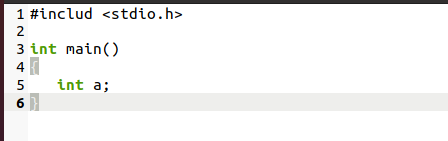
****

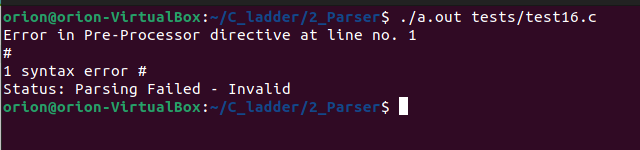
1. **test2.c**

****

****

1. **test3.c**



****

SEMANTIC ANALYSIS

**Introduction:**

This report contains the details of the tasks finished in the Semantic Phase of the C compiler. We have developed a Parser for C language that uses the C lexer to parse the given C input file. In the previous phase, we checked if the given input code matches the language defined in the parser. We used a lexer to convert the input code into a stream of tokens provided to the parser. Parser matches the stream with the defined productions of the language. We used look-ahead for checking errors in comments and some other lexical errors. But lexical analyzer cannot detect errors in the structure of a language (syntax), unbalanced parenthesis, etc. These errors were handled by a parser. But in the syntax analysis phase, we don’t check if the input is semantically correct. After the parser checks if the code is structured correctly, the semantic analysis phase checks if that syntax structure constructed in the source program derives any meaning or not. The output of the syntax analysis phase is a parse tree, whereas that of the semantic phase is annotated parse tree. Semantic analysis is done by modifications in the parser code only. Tasks performed in Semantic Analysis are:

1. Type Checking – Data types must be used in a manner that is consistent with their definition
2. Label Checking – Labels referenced in a program must exist.
3. Array Bounds Checking – The subscript should be adequately defined when declaring an array.

We have mentioned some of the semantic errors that the semantic analyzer is expected to recognize:

1. Type mismatch a. Return type mismatch. b. Operations on mismatching variable types.
2. Undeclared variable
3. Check if a variable is undeclared globally.
4. Check if a variable is visible in the current scope.
5. Reserved identifier misuse.
6. Function name and variable name cannot be the same.
7. Declaration of a keyword as a variable name.
8. Multiple declarations of a variable in scope.
9. Accessing an out-of-scope variable.
10. Actual and formal parameter mismatches

**Explanation:**   
The lex code detects the tokens from the source code and returns the   
corresponding token to the parser. In phase 1, we were printing the token and now we are returning the token so that parser uses it for further computation. We are only using the symbol table and constant table of the previous phase. We added functions like insertSTnest(), insertSTparamscount(), checkscope(), deletedata(), duplicate()​etc., in order to check the semantics​.​In the grammar production rules, semantic actions are written and performed by the functions listed above.

**Example Code Snippet:**

*void* insertSTnest(*char* \**s*, *int* *nest*)

    {

        if(lookupST(s) && ST[lookupST(s)].nestval != 9999)

        {

*int* pos = 0;

*int* value = hash(s);

            for (*int* i = value + 1 ; i!=value ; i = (i+1)%1001)

            {

                if(ST[i].length == 0)

                {

                    pos = i;

                    break;

                }

            }

            strcpy(ST[pos].name,s);

            strcpy(ST[pos].class,"Identifier");

            ST[pos].length = strlen(s);

            ST[pos].nestval = nest;

            ST[pos].params\_count = -1;

            ST[pos].lineno = yylineno;

        }

        else

        {

            for(*int* i = 0 ; i < 1001 ; i++)

            {

                if(strcmp(ST[i].name,s)==0 )

                {

                    ST[i].nestval = nest;

                }

            }

        }

    }

*void* insertSTparamscount(*char* \**s*, *int* *count*)

    {

        for(*int* i = 0 ; i < 1001 ; i++)

        {

            if(strcmp(ST[i].name,s)==0 )

            {

                ST[i].params\_count = count;

            }

        }

    }

**Declaration Section**:   
This section has included all the necessary header files, function declarations, and flags needed in the code. Between the declaration and rules section, we have listed all the tokens 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.

**Example Code Snippet:**

%{

*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;

    extern *char* curid[20];

    extern *char* curtype[20];

    extern *char* curval[20];

    extern *int* currnest;

*void* deletedata (*int* );

*int* checkscope(*char*\*);

*int* check\_id\_is\_func(*char* \*);

*void* insertST(*char*\*, *char*\*);

*void* insertSTnest(*char*\*, *int*);

*void* insertSTparamscount(*char*\*, *int*);

*int* getSTparamscount(*char*\*);

*int* check\_duplicate(*char*\*);

*int* check\_declaration(*char*\*, *char* \*);

*int* check\_params(*char*\*);

*int* duplicate(*char* \**s*);

*int* checkarray(*char*\*);

*char* currfunctype[100];

*char* currfunc[100];

*char* currfunccall[100];

*void* insertSTF(*char*\*);

*char* gettype(*char*\*,*int*);

*char* getfirst(*char*\*);

    extern *int* params\_count;

*int* call\_params\_count;

%}

**Rules Section**:  
In this section, production rules for the entire C language are written. The grammar productions do the syntax analysis of the source code. Along with rules, semantic actions associated with the rules are also written and corresponding functions are called to do the necessary actions.

**Example Code Snippet:**

*simple\_expression*

            : simple\_expression *OR\_operator* and\_expression {if($1 == 1 && $3==1) $$=1; else $$=-1;}

            | and\_expression {if($1 == 1) $$=1; else $$=-1;};

*and\_expression*

            : and\_expression *AND\_operator* unary\_relation\_expression {if($1 == 1 && $3==1) $$=1; else $$=-1;}

              |unary\_relation\_expression {if($1 == 1) $$=1; else $$=-1;} ;

*unary\_relation\_expression*

            : *exclamation\_operator* unary\_relation\_expression {if($2==1) $$=1; else $$=-1;}

            | regular\_expression {if($1 == 1) $$=1; else $$=-1;} ;

*regular\_expression*

            : regular\_expression relational\_operators sum\_expression {if($1 == 1 && $3==1) $$=1; else $$=-1;}

              | sum\_expression {if($1 == 1) $$=1; else $$=-1;} ;

**C-Program Section:**  
In this section, the parser links the extern functions, variables (declared in the lexer), external files generated by the lexer etc. The ‘main’ function takes the input source code file and prints the final symbol table.

**Example Code Snippet**

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

{

    yyin = fopen(argv[1], "r");

    yyparse();

    if(flag == 0)

    {

        printf("Status: Parsing Complete - Valid\n");

        printf("%30s SYMBOL TABLE\n", " ");

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

        printST();

        printf("\n\n%30s CONSTANT TABLE\n", " ");

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

        printCT();

    }

}

**Test Cases**

**test1.c** (without error)

#include<stdio.h>

int myfunc(int b){

    int x;

    return x;

}

void main(){

    int n,i;

    char ch;//Character Datatype

    int x;

    int a[10];

    for (i=0;i<10;i++){

        if(i<10){

            int x;

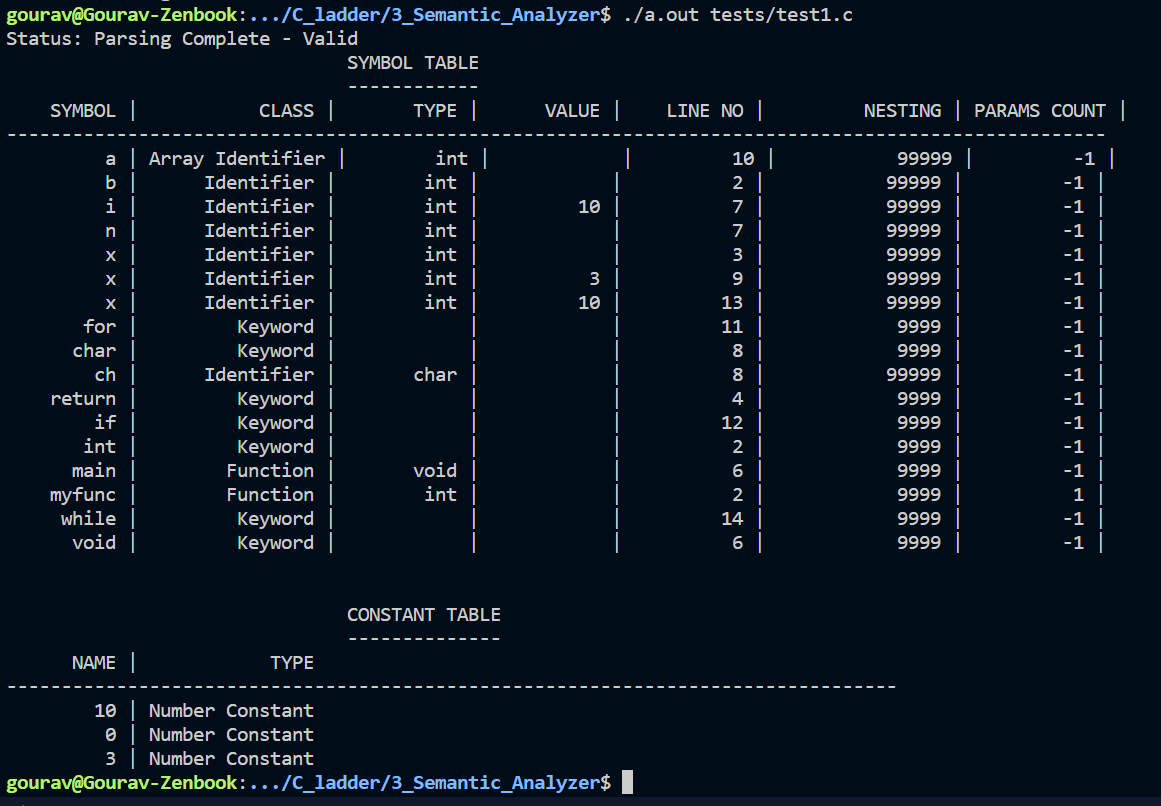
            while(x<10){

                x++;

            }}}

   x=3;

}



**test2.c** (without error)

#include<stdio.h>

int main(){

    int a = 5;

    while(a>0){

        printf("Hello world");

        a--;

    }

    a=4;

    while(a>0){

        printf("%d",a);

        a--;

        int b;

        b= 4;

        while(b>0){

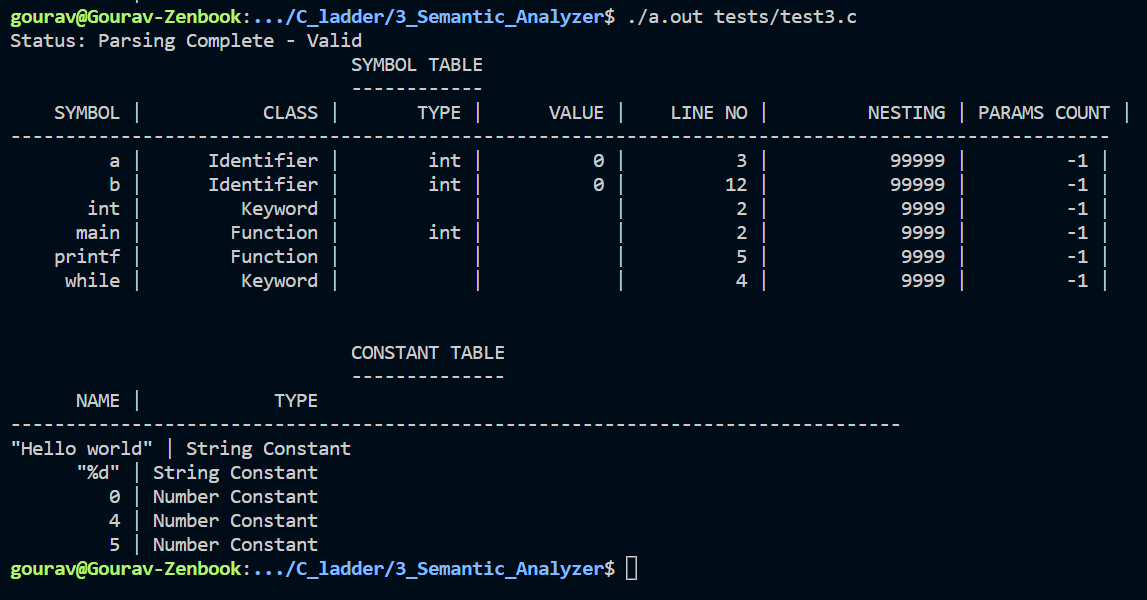
            printf("%d", a\*b);

            b--;

        }

    }

}



**test12.c** (with error)

 #include <stdio.h>

void func(int a, int b){

    return a;

}

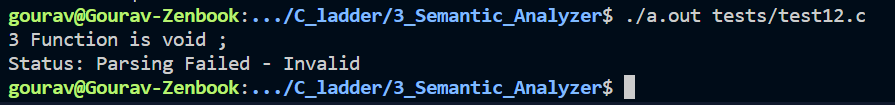
int main(){

    int z = 5;

    func(5,z,z);

    printf("wow\n");

}



­INTERMEDIATE CODE GENERATION

Intermediate code is used to translate the source code into the machine code. Intermediate code lies between the high-level language and the machine language. The given program in a source language is converted into an equivalent program in an intermediate language by the intermediate code generator. Intermediate codes are machine independent codes.

Roles of Intermediate code generation phase are :

* It acts as a glue between front-end and backend (or source and machine codes).
* If the compiler directly translates source code into the machine code without generating intermediate code then a full native compiler is required for each new machine.
* The intermediate code keeps the analysis portion same for all the compilers that's why it doesn't need a full compiler for every unique machine.
* Intermediate code generator receives input from its predecessor phase and semantic analyzer phase. It takes input in the form of an annotated syntax tree.
* Using the intermediate code, the second phase of the compiler synthesis phase is changed according to the target machine.
* Intermediate code generator lowers abstraction from source level.

The intermediate code representation are

* Graphical representation e.g. Abstract Syntax Tree(AST) , DAGS
* Postfix notations
* Three Address codes

**Three Address Code**

**Three address code** is a type of intermediate code which is easy to generate and can be easily converted to machine code.It makes use of at most three addresses and one operator to represent an expression and the value computed at each instruction is stored in temporary variable generated by compiler. The compiler decides the order of operation given by three address code.

**General representation –**

**a = b op c**

Where a, b or c represents operands like names, constants or compiler generated temporaries and op represents the operator

* Assignments x=y op z ;x = op y.
* Copy x = y.
* Unconditional jump goto L.
* Conditional jumps if x relop y goto L.
* Parameters param x.
* Function call y = call p

**Implementation of Three Address Code –**  
There are 3 representations of three address code namely

1. Quadruple
2. Triples
3. Indirect Triples

**Code snippets with explanation**

Snippets from **scanner.l**

void insertSTnest(char \*s, int nest)

{

if(lookupST(s) && ST[lookupST(s)].nestval != 9999)

{

int pos = 0;

int value = hash(s);

for (int i = value + 1 ; i!=value ; i = (i+1)%1001)

{

if(ST[i].length == 0)

{

pos = i;

break;

}

}

strcpy(ST[pos].name,s);

strcpy(ST[pos].class,"Identifier");

ST[pos].length = strlen(s);

ST[pos].nestval = nest;

ST[pos].params\_count = -1;

ST[pos].lineno = yylineno;

}

else

{

for(int i = 0 ; i < 1001 ; i++)

{

if(strcmp(ST[i].name,s)==0 )

{

ST[i].nestval = nest;

}

}

}

}

void insertSTparamscount(char \*s, int count1)

{

for(int i = 0 ; i < 1001 ; i++)

{

if(strcmp(ST[i].name,s)==0 )

{

ST[i].params\_count = count1;

}

}

}

int checkscope(char \*s)

{

int flag = 0;

for(int i = 0 ; i < 1000 ; i++)

{

if(strcmp(ST[i].name,s)==0)

{

if(ST[i].nestval > currnest)

{

flag = 1;

}

else

{

flag = 0;

break;

}

}

}

if(!flag)

{

return 1;

}

else

{

return 0;

}

}

void deletedata (int nesting)

{

for(int i = 0 ; i < 1001 ; i++)

{

if(ST[i].nestval == nesting)

{

ST[i].nestval = 99999;

}

}

}

int duplicate(char \*s)

{

for(int i = 0 ; i < 1000 ; i++)

{

if(strcmp(ST[i].name,s)==0)

{

if(ST[i].nestval == currnest)

{

return 1;

}

}

}

return 0;

}

Snippets from **parser.y**

variable\_declaration\_identifier

: identifier {if(duplicate(curid)){printf("Duplicate\n");exit(0);}insertSTnest(curid,currnest); ins(); } vdi

| array\_identifier {if(duplicate(curid)){printf("Duplicate\n");exit(0);}insertSTnest(curid,currnest); ins(); } vdi;

call

: identifier '('{

if(!check\_declaration(curid, "Function"))

{ printf("Function not declared"); exit(0);}

insertSTF(curid);

strcpy(currfunccall,curid);

if(gettype(curid,0)=='i' || gettype(curid,1)== 'c')

{

$$ = 1;

}

else

$$ = -1;

call\_params\_count=0;

}

arguments ')'

{ if(strcmp(currfunccall,"printf"))

{

if(getSTparamscount(currfunccall)!=call\_params\_count)

{

yyerror("Number of arguments in function call doesn't match number of parameters");

exit(8);

}

}

callgen();

};

void codegen()

{

strcpy(temp,"t");

char buffer[100];

itoa(count,buffer,10);

strcat(temp,buffer);

printf("%s = %s %s %s\n",temp,s[top-2].value,s[top-1].value,s[top].value);

top = top - 2;

strcpy(s[top].value,temp);

count++;

}

void codegencon()

{

strcpy(temp,"t");

char buffer[100];

itoa(count,buffer,10);

strcat(temp,buffer);

printf("%s = %s\n",temp,curval);

push(temp);

count++;

}

int isunary(char \*s)

{

if(strcmp(s, "--")==0 || strcmp(s, "++")==0)

{

return 1;

}

return 0;

}

void genunary()

{

char temp1[100], temp2[100], temp3[100];

strcpy(temp1, s[top].value);

strcpy(temp2, s[top-1].value);

if(isunary(temp1))

{

strcpy(temp3, temp1);

strcpy(temp1, temp2);

strcpy(temp2, temp3);

}

strcpy(temp, "t");

char buffer[100];

itoa(count, buffer, 10);

strcat(temp, buffer);

count++;

if(strcmp(temp2,"--")==0)

{

printf("%s = %s - 1\n", temp, temp1);

printf("%s = %s\n", temp1, temp);

}

if(strcmp(temp2,"++")==0)

{

printf("%s = %s + 1\n", temp, temp1);

printf("%s = %s\n", temp1, temp);

}

top = top -2;

}

**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 parser uses it for further computation. We are using the symbol table and constant table of the previous phase only. We added functions like insertSTnest(), insertSTparamscount(), checkscope(), deletedata(), duplicate() ​etc., in order to check the semantics​.​In the production rules of the grammar semantic actions are written and these are performed by the functions listed above. Along with semantic actions SDT also included function to generate the 3 address code.

**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. Along with this semantic actions were also added to each production rule to check if the structure created has some meaning or not. Then we added the function to generate the 3 address code with production so that we can generate the desired intermediate code. In order to generate 3 address code we made use of explicit stack. Whenever we came across an operator, operand or constant we pushed it to stack. Whenever reduction occurred (Since LALR(1) parser is bottom up parser it evaluates SDT when reduction occurs) codegen( ) function generated the 3 address code by creating a new temporary variable and by making use of the entries in the stack, after that it popped those entries from the stack and pushed the temporary variable to the stack so that it gets used in further computation. Similarly functions like labels were used to assign appropriate labels while using conditional statements or iterative statements. All the functions used are described below :

1. codegen( ) : This function is called whenever a reduction of an expression takes place. It creates the temporary variable and displays the desired 3 address code i.e x = y op z.
2. codegencon():This function is especially written for reductions of expression involving constants since its 3 address code is x op z.
3. isunary( ) : This function checks if the operator is an unary operator like ‘++’. If so it returns true else false.
4. genunary( ) : This function is specifically designed to generate 3 address code for unary operations. It makes use of isuanary function mentioned above. E.g. if a = i++ then it converts into t0 = i + 1, a = t0.

**Testcases with outputs**

**Testcase 1 - Valid Testcase: O**​**perator, Delimiters, Assignments, Nested Conditional Statements, for and while loops**

#include <stdio.h>

int myfunc(int a,int b)

{

return a+b;

}

void main()

{

int a,b,i;

while(a<3)

{

a = a+b;

for(i=0;i<b;i++)

{

b++;

myfunc(a,b);

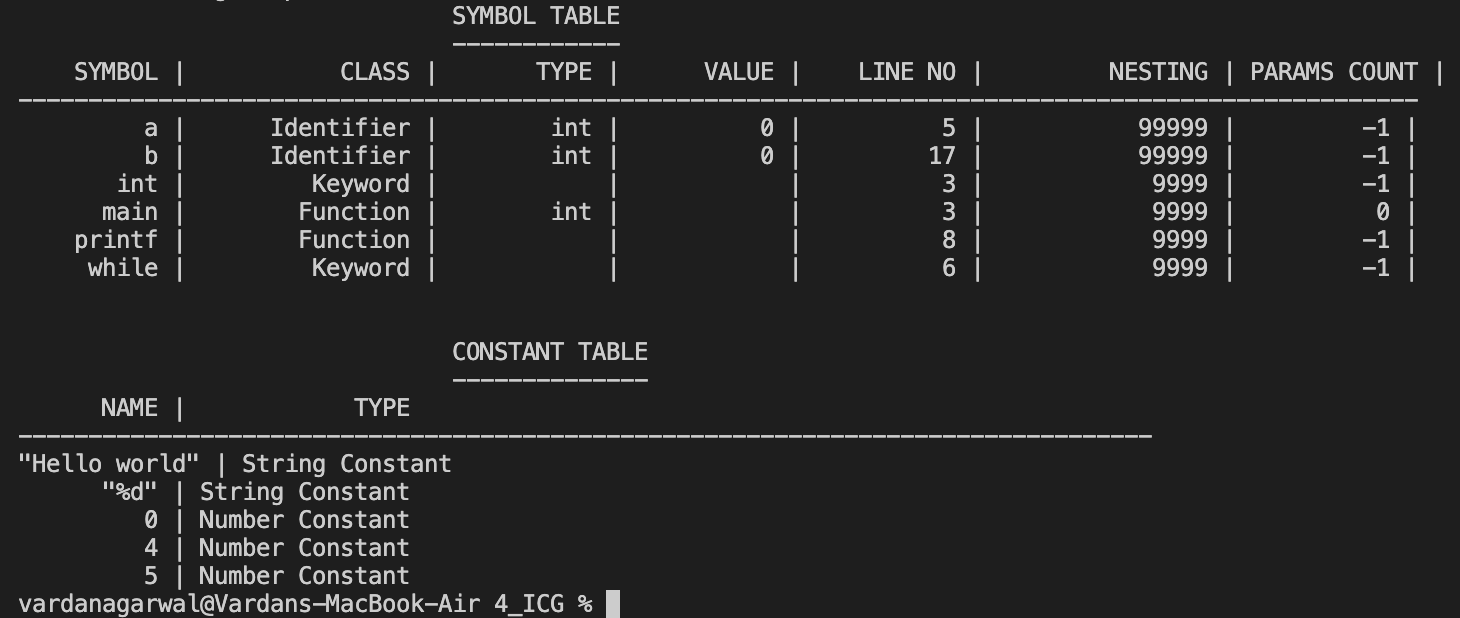
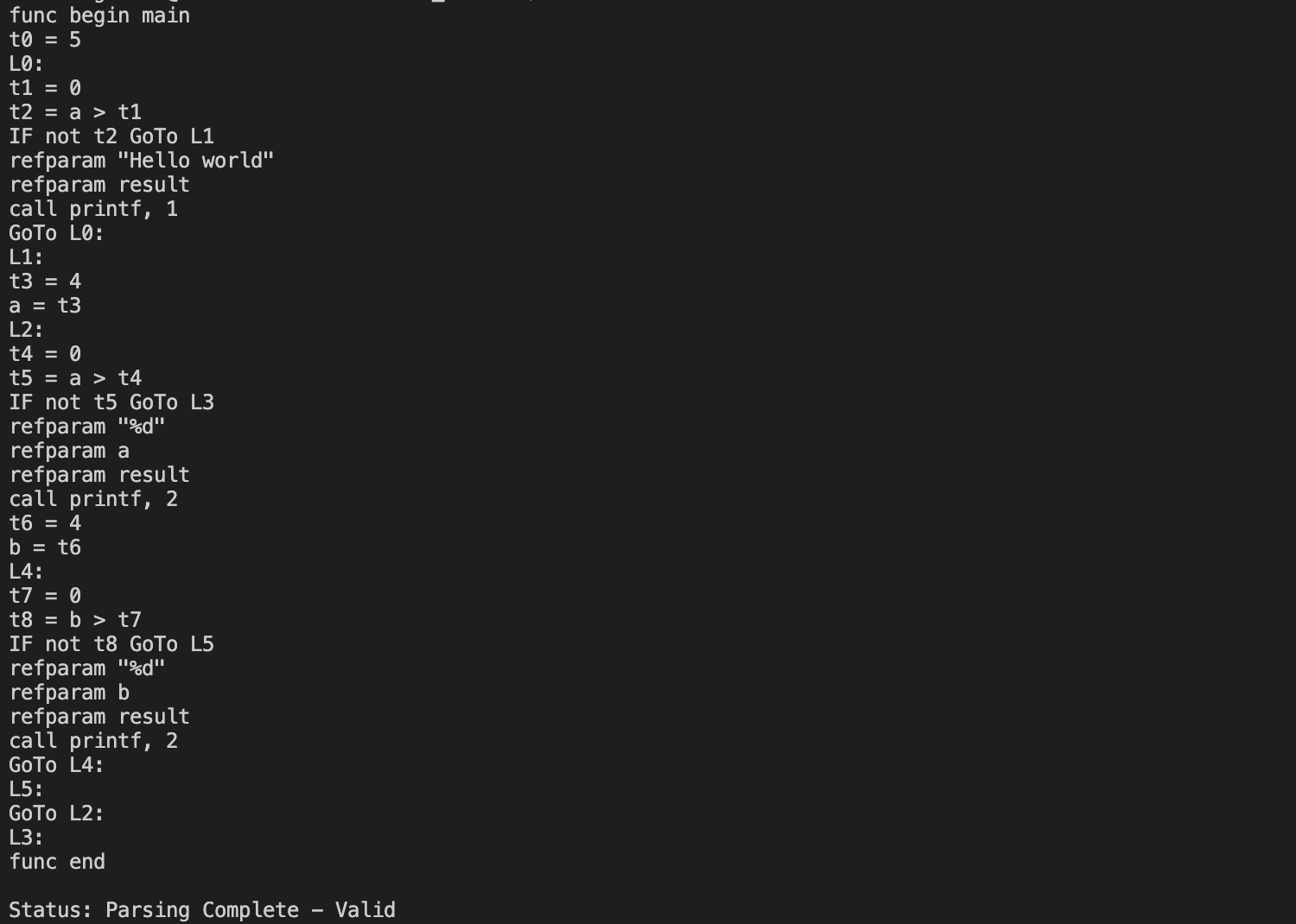
}

a++;

}

}

**Output: -**



**Testcase 2 - Inalid Testcase:**

// Implicit Error that our Language doesn't support

#include<stdio.h>

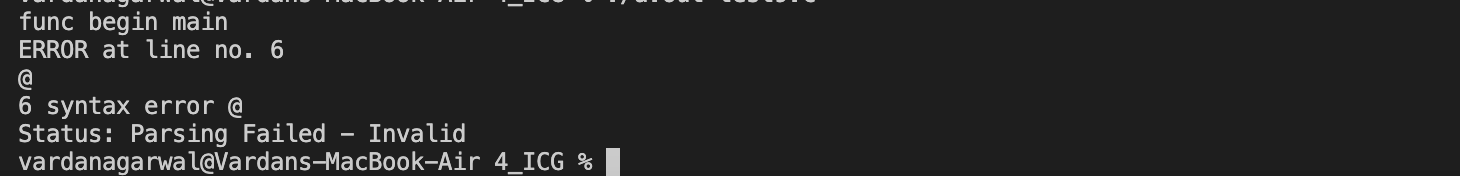
int main() {

char @hello;

@hello = 'c';

}

**Output: -**



**Testcase 3 - Inalid Testcase:**

#include<stdio.h>

int square(int a, int b)

{

int b = 2;

return b;

}

int main()

{

int num = 2;

int num2;

square(num,num);

//printf("Square of %d is %d", num, square2(5));

return 0;

}

**Output: -**



CODE OPTIMIZATION

The code optimization in the synthesis phase is a program transformation technique, which tries to improve the intermediate code by making it consume fewer resources (i.e., CPU, Memory) so that faster-running machine code will result. Compiler optimizing process should meet the following objectives:

* The optimization must be correct, it must not, in any way, change the meaning of the program.
* Optimization should increase the speed and performance of the program.
* The compilation time must be kept reasonable.
* The optimization process should not delay the overall compiling process.

**Types of Code Optimization –**The optimization process can be broadly classified into two types:

1. **Machine Independent Optimization –** This code optimization phase attempts to improve the **intermediate code** to get a better target code as the output. The part of the intermediate code which is transformed here does not involve any CPU registers or absolute memory locations.
2. **Machine Dependent Optimization –** Machine-dependent optimization is done after the **target code** has been generated and when the code is transformed according to the target machine architecture. It involves CPU registers and may have absolute memory references rather than relative references. Machine-dependent optimizers put efforts to take maximum **advantage** of the memory hierarchy.

**Phases of Optimization**  
There are generally two phases of optimization:

* **Global Optimization:**  
  Transformations are applied to large program segments that includes functions,procedures and loops.
* **Local Optimization:**  
  Transformations are applied to small blocks of statements.The local optimization is done prior to global optimization.

TARGET CODE GENERATION

Input: intermediate language (IL)

Output: target language program Target languages:

– absolute binary (machine) code

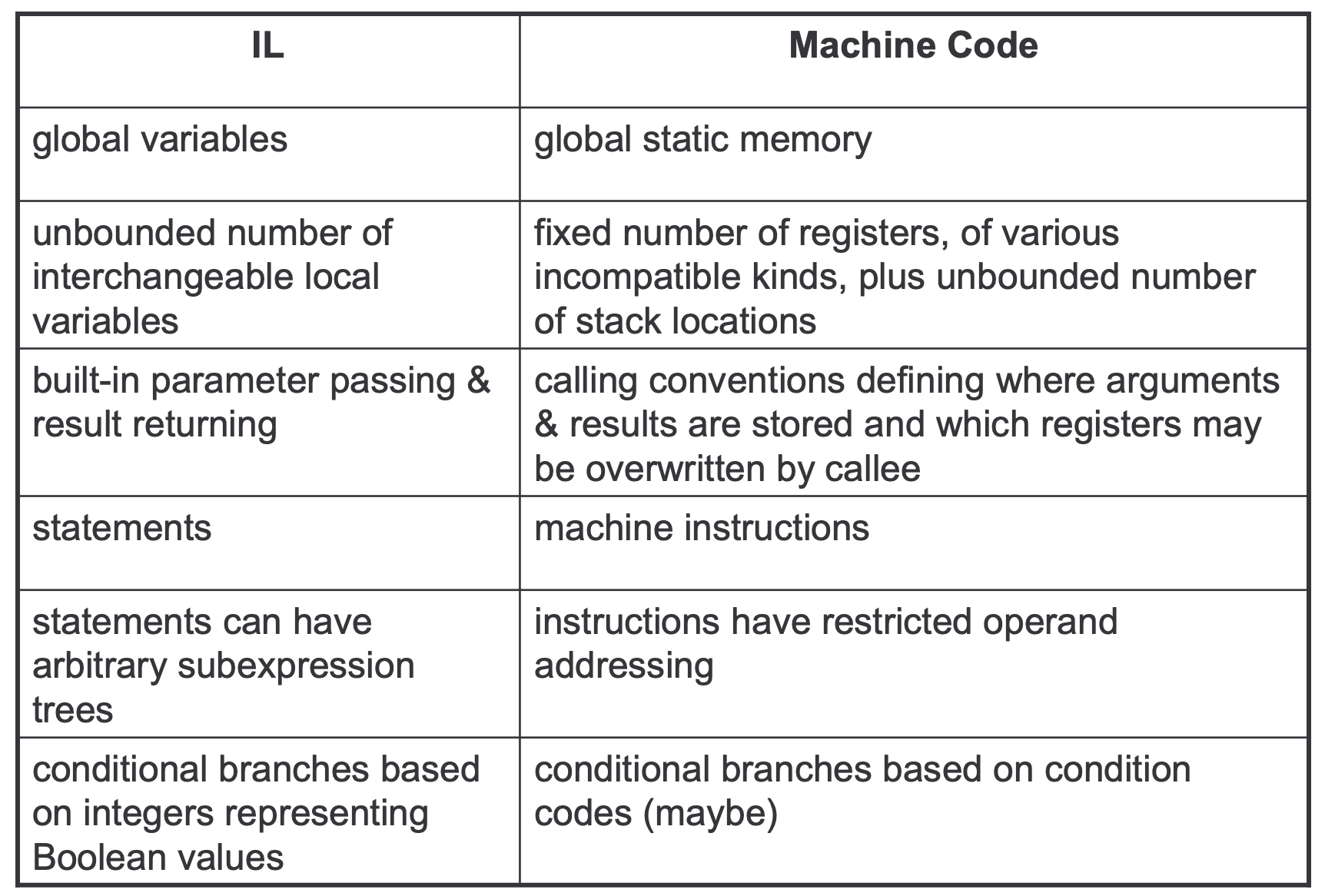
– relocatable binary code

– assembly code

– C

Target code generation must bridge the gap.

The gap, if target is machine code.



**Tasks of Code Generator**

**Register allocation**

– for each IL variable, select register/stack location/global memory location(s) to hold it.

• can depend on type of data, which operations manipulate it

• **Stack frame layout**

– compute layout of each function’s stack frame

• **Instruction selection**

– for each IL instruction (sequence), select target language instruction (sequence).

• includes operand addressing mode selection

• **Can have complex interactions**

– instruction selection depends on where operands are allocated

– some IL variables may not need a register, depending on the instructions & addressing modes that are selected.

**Register Allocation**

Intermediate language uses unlimited temporary variables

• makes ICG easy

Target machine has fixed resources for representing “locals” plus other internal things such as stack pointer

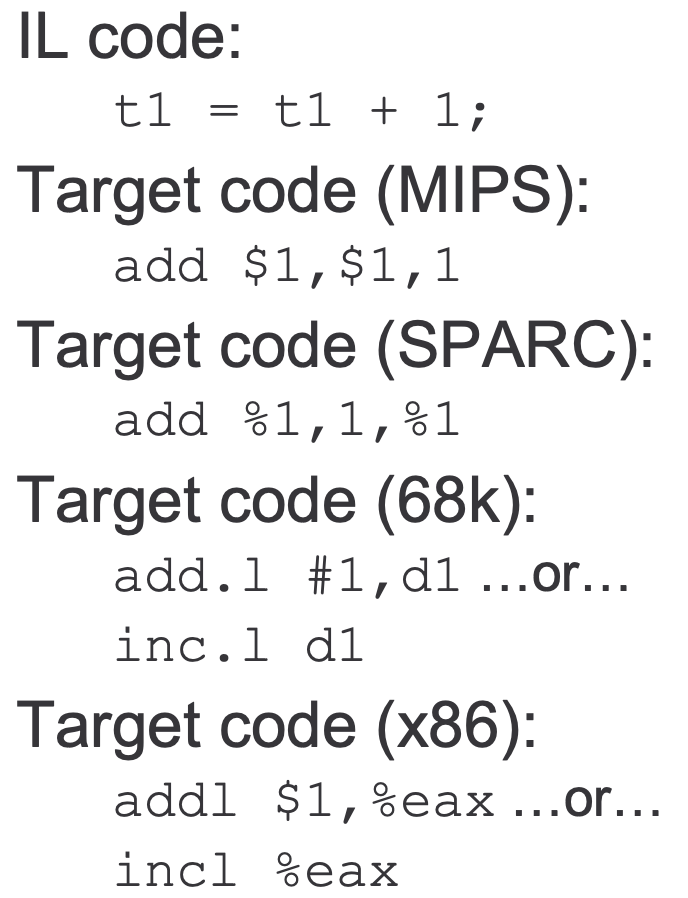
• MIPS, SPARC: 31 registers + 1 always-zero register

• 68k: 16 registers, divided into data and address regs

• x86: 8 word-sized integer registers (with a number of instruction specific restrictions on use) plus a stack of floating-point data manipulated only indirectly.

Registers are much faster than memory.

**Example:**



References

* 1. <https://www.collegenote.net/pastpapers/4254/question/#gsc.tab=0>
  2. <https://www.geeksforgeeks.org/intermediate-code-generation-in-compiler-design/>
  3. https://courses.cs.washington.edu/courses/cse401/06sp/codegen.pdf
  4. Compilers: Principles, Techniques, and Tools: Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman