

COMPILER DESIGN

THE SIMPLE C PROGRAMMING LANGUAGE

SESSIONAL TEST TEST II

COMPILER DESIGN

NAME:- GAURAV KUMAR

ROLL-NO:-CSB17051

B-tech 6th Sem

The Simplified C programming Language

For making programming easier I will write a simplified version of C Language. There are a lot of things which are similar like C.

Changes in the simplified C :

- There are Less Keywords or reserved words as compared to C.
- We use only Basic data type (char,int,float,double).
- There is no include statement. So we can not include libraries.
- Simple regular expressions for the lexical units or tokens of lexical analyser
- We can define variables at the beginning of program only
- There is no need for the main function. We can directly start our program.
- There are not any dynamic semantic rules.
- We are not getting input as we did in C.
- Only static memory allocation.
- Only While loop available for iteration
- No function can be defined in the program

Tokens

Tokens can be any keywords, operators, identifiers, constants, strings etc.

Keywords:

- CHAR (char data type)
- INT (int data type)
- FLOAT (float data type)
- CONTINUE (to get next iteration of loop)
- BREAK(to break out loop)
- WHILE (for loop declaration)
- VOID (void data type)

Operators:

- ADD_OP (+ or -)
- MUL_OP (*)
- DIV_OP (/)
- INCR (++ or --)
- OR_OP (||)
- AND_OP (&&)
- EQU_OP (== or !=)
- REL_OP (> or < or >= or <=)

Identifiers,constants and strings:

- ID are used for the name of identifiers which start with a letter and contain character or number after that.
- INT_C is an integer constant that can start with 1-9 and follow by numbers 0-9.

-
- FLOAT_C is a floating point constant.
 - CHAR_C is any printable ASCII character.
 - STRING's are printable ASCII characters in between of two “.

Other tokens

- ‘(‘)’ parentheses
- ‘[‘]’ brackets
- ‘{ ‘}’ bracelets
- ‘;’ ‘,’ ‘.’ semi,comma,dot
- ‘=’ ‘&’ assign and reference

Basic Structure

Main function:

Main function have two parts

- Variable declaration (data type, initialization of variables)
- Body/Statements (body statements contain arithmetic expressions,for,print() etc.)

Example:

```
int j; //defining variable at initial
float value=3.0; //initializing variable
for(j=0;j<10;j++){
    j=j+10;
}
```

```
j++;
```

Lexical Analyzer

A lexical analyzer creates “tokens” by analysing character stream. A token can be anything from keyword, operators, identifiers or even constants/literals.

Symbol Table

Symbol table is a way to store the information of identifiers so that we can access the information for an identifier anytime when it needed.

An identifier can be :

- Program
- Variables
- Subroutine parameters
- Instruction labels
- Constants
- Data type

We have to store all the information for semantic analysis.

As example:

```
int m;
```

1. Identify the keyword int and return token to parser

-
- | |
|---|
| <ol style="list-style-type: none">2. Identify the identifier <code>m</code>3. Add it to the symbol table and return the pointer of the identifier to the parser.4. Recognize the semicolon <code>;</code> and return the token the parser |
|---|

When is the symbol table being constructed?

1. We first add the identifiers in lexical analysis with some initial information.
2. Most information added in the syntax analysis, cause there is more information available.
3. We process some information and sometime modify it in the semantic analysis

How Implement Symbol table.

We can implement symbol table as a **hashtable** using chain implementation.

Basic Operations on a symbol table.

1. Adding identifiers (insert).
2. Searching identifiers(lookup).

Using the Hash function and ASCII characters of name of ID's we generate a integer/position and return for hashtable.

hash(name) returns a position for hash table.

- Insert() :- We create a new entry for the symbol table when we recognize a new identifier if it does not already exist in symbol table.
- Lookup() :- During syntax or semantic analysis search for an identifier and return it's symbol table entry.

insert()

This function use for inserting a new identifier to the Symbol Table. The parameter are the name of identifier that will first undefined(UNDEF) and change afterwards and lastly the line number where it was found. For inserting a identifier we use hash() function for generating hash index. After that we search at the corresponding list. If the identifier is not found then we can create a new entry that will contain the current known information and we initialize the reference list. If the identifier is found we will add the line number to the reference list.

lookup()

This function is used for searching. First we find the hash index of the input by using the hash() function, and then start searching for the corresponding linked list. If identifier was found then we will return the pointer of that identifier otherwise we will return NULL, which mean identifier was not found in list.

Syntax Analysis

Syntax analysis is a step in which we check the correctness of the syntax of our program. It combines the tokens from lexical analysis to form a syntax tree based on grammar/production rules.

Some other things that we do in this step:

1. Syntax error message
2. Insert information to our symbol table using the information that we get through the parsing of our program.

Syntax analysis uses a context-free grammar, which can be expressed using the production rules, which become push-down or shift reduce automata.

Bison uses LALR parser.

Grammars

```
program: declarations statements;
declarations: declarations declaration | declaration ;

declaration: type names SEMI
;

type: INT
    | CHAR
    | FLOAT
    | DOUBLE
    | VOID
;

names: names COMMA variable
    | names COMMA init
    | variable
    | init
;

variable: ID ;

init:
    var_init
;

var_init : ID ASSIGN constant
;

statements:
```



```

statements statement
| statement
;

statement:
    while_statement
| assignment SEMI
| CONTINUE SEMI
| BREAK SEMI
| ID INCR SEMI
| INCR ID SEMI
;

while_statement: WHILE LPAREN expression RPAREN tail
;

tail: LBRACE statements RBRACE
;

expression:
    expression ADDOP expression
| expression MULOP expression
| expression DIVOP expression
| ID INCR
| INCR ID
| expression OROP expression
| expression ANDOP expression
| NOTOP expression
| expression EQUOP expression
| expression RELOP expression
| LPAREN expression RPAREN
| var_ref
| constant
| ADDOP constant %prec MINUS
;

```

```
constant:
    ICONST
    | FCONST
    | CCONST
;

assignment: var_ref ASSIGN expression
;

var_ref: variable
    | REFER variable
;
```

Semantic Analysis

Until now the lexical analyzer (lexer) is responsible for validity of the tokens, token are supplied to the syntax analyzer. Then syntax analyzer combines these “valid” tokens to check if the syntax of the program is correct.

But parser can't determine if:

- A token is valid
- A token was declared before being used
- A token was initialized before being used
- A operation performed on the specific token is allowed or not

These operations are accomplished by semantic analyzer. The program is correct. Semantic analysis also performs **Type checking** which recognizes type mismatch, identifier misuse.

AST(abstract syntax tree):-

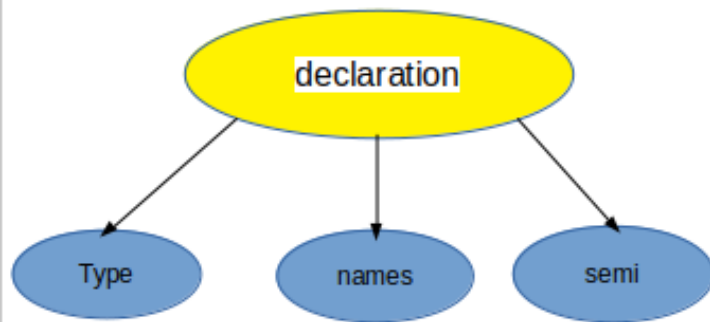
Abstract syntax tree is used for representing the syntax of a programming language as a tree-like structure, which is similar to a parse tree.

Program will generate syntax tree from the tokens. Nodes are created for every production.

ast.h contain definition of nodes and ast.c contain declaration of nodes .

Semantic Action for Type declaration

```
// declaration
int a;
int b=10;
char c;
float f=2.5
```



At initial stage variable are declared in program when new variable is found we add name of variable in list after that a declaration node is created in which identifier type and name of identifier is defined.

For production

- declaration : type names SEMI

`$$=new_ast_decl_node($1,names,nc)`

- `new_ast_decl_node ()` create a declaration node
- `nc` is stand for name count

Semantic action for declaration

```
declarations: declarations declaration | declaration ;  
declaration: type { declare = 1; } names { declare = 0; } SEMI  
  
    {  
        int i;  
  
        $$ = new_ast_decl_node($1, names, nc);  
  
        nc = 0;
```

```

AST_Node_Decl *temp = (AST_Node_Decl*) $$;

for(i=0; i < temp->names_count; i++){

    if(temp->names[i]->st_type == UNDEF){

        set_type(temp->names[i]->st_name, temp->data_type, UNDEF);

    }

    else if(temp->names[i]->st_type == POINTER_TYPE){

        set_type(temp->names[i]->st_name, POINTER_TYPE,
temp->data_type);

    }

    else if(temp->names[i]->st_type == ARRAY_TYPE){

        set_type(temp->names[i]->st_name, ARRAY_TYPE,
temp->data_type);

    }

}

ast_traversal($$); /* just for testing */

}

;

```

Semantic action for type declaration

```

type: INT      { $$ = INT_TYPE;  }
    | CHAR     { $$ = CHAR_TYPE; }
    | FLOAT    { $$ = REAL_TYPE; }
    | DOUBLE   { $$ = REAL_TYPE; }
    | VOID     { $$ = VOID_TYPE; }
;

```

Semantic action for variable names

```
names: names COMMA variable
    {
        add_to_names($3);
    }
| names COMMA init
    {
        add_to_names($3);
    }
| variable
    {
        add_to_names($1);
    }
| init
    {
        add_to_names($1);
    }
;
```

add_to name function is used for making entry of Identifier whenever a new entry is found it is inserted in the list.

```
add_to_name(list_t *entry)
{
    //first entry
    If nc is 0{
        nc=1;
        names=(list_t**)malloc(1*size_of (list_t*));
        names[0]=entry;
    }
    //general entry
    else{
        nc++;
        names=(list_t **)realloc(names,nc*sizeof(list_t *));
        names[nc-1]=entry;
    }
}
```

```
}
```

Only declaration of a variable

- int a;

Semantic action for variable production

```
variable: ID { $$ = $1; };
```

Initialization of a variable during declaration

```
int a=10;  
char c='f';
```

tem_no is associated with the identifier which is used for the generating 3 address code. When a constant value is found we create a constant node and add the type of constant in the constant node we create a constant node using AST_Node_Const .

```
init:  
    var_init { $$ = $1; }  
;  
var_init : ID ASSIGN constant  
{  
    AST_Node_Const *temp = (AST_Node_Const*) $$;  
    $1->val = temp->val;  
    $1->st_type = temp->const_type;  
    $$ = $1;  
    AST_Node_Const *t=(AST_Node_Const *)$3;  
    AST_Node *t2=(AST_Node *)$3;  
    $1->tem_no=t2->tem_no;  
}  
;
```

Semantic action for Expression

An expression can be

-
1. Increment expression
 2. Arithmetic expression node
 3. Boolean expression node
 4. Equality expression node
 5. Relational expression node
 6. var_ref , which is a variable with or without reference
 7. An expression with parentheses ”()”.

All the arithmetic expressions are applied on two variables; they have two child (left child and right child) and an operator depending on their type which can be arithmetic,boolean,equality and relational. We create an arithmetic node which contain information of left child right child and operator type

- \$\$=new_ast_arithm_node(OP,\$1,\$3)
- Op stands for operator \$1 is left child and \$3 is right child.
- tem_no is used for creating three address code generation.

Semantic Action for Expression

```
expression:
    expression ADDOP expression
    {
        $$ = new_ast_arithm_node($2.ival, $1, $3);
        AST_Node *temp=(AST_Node *)$$;
        AST_Node *temp1=(AST_Node *)$1;
        AST_Node *temp2=(AST_Node *)$3;
        temp->tem_no=tem_no;
        tem_no++;
    }
    | expression MULOP expression
    {
        $$ = new_ast_arithm_node(MUL, $1, $3);
        AST_Node *temp=(AST_Node *)$$;
        AST_Node *temp1=(AST_Node *)$1;
```



```

        AST_Node *temp2=(AST_Node *)$3;
        temp->tem_no=tem_no;
        tem_no++;
    }
    | expression DIVOP expression
    {
        $$ = new_ast_arithm_node(DIV, $1, $3);

        AST_Node *temp=(AST_Node *)$$;
        AST_Node *temp1=(AST_Node *)$1;
        AST_Node *temp2=(AST_Node *)$3;
        temp->tem_no=tem_no;
        tem_no++;
    }
    | ID INCR
    {
        /* increment */
        if($2.ival == INC){
            $$ = new_ast_incr_node($1, 0, 0);
        }
        else{
            $$ = new_ast_incr_node($1, 1, 0);
        }
    }
    | INCR ID
    {
        /* increment */
        if($1.ival == INC){
            $$ = new_ast_incr_node($2, 0, 1);
        }
        else{
            $$ = new_ast_incr_node($2, 1, 1);
        }
    }
    | expression OROP expression
    {
        $$ = new_ast_bool_node(OR, $1, $3);
    }

```

```

}
| expression ANDOP expression
{
    $$ = new_ast_bool_node(AND, $1, $3);

}
| NOTOP expression
{
    $$ = new_ast_bool_node(NOT, $2, NULL);
}
| expression EQUOP expression
{
    $$ = new_ast_equ_node($2.ival, $1, $3);
}
| expression RELOP expression
{
    $$ = new_ast_rel_node($2.ival, $1, $3);
}
| LPAREN expression RPAREN
{
    $$ = $2; /* just pass information */
}
| var_ref
{
    $$ = $1; /* just pass information */
    AST_Node_Ref *t=(AST_Node_Ref *)$1;
    AST_Node *t2=(AST_Node *)$$;
    t2->tem_no=(t->entry->tem_no);
}
| constant
{
    $$ = $1; /* no sign */
    AST_Node *t=(AST_Node *)$1;
    t->tem_no=tem_no;
}

```

```

    AST_Node *t2=(AST_Node *)$$;
    t2->tem_no=tem_no;
    AST_Node_Const *t3=(AST_Node_Const *)$1;
    tem_no++;

}
| ADDOP constant %prec MINUS
{
    /* plus sign error */
    if($1.ival == ADD){
        fprintf(stderr, "Error having plus as a sign!\n");
        exit(1);
    }
    else{
        AST_Node_Const *temp = (AST_Node_Const*) $2;

        /* inverse value depending on the constant type */
        switch(temp->const_type){
            case INT_TYPE:
                temp->val.ival *= -1;
                break;
            case REAL_TYPE:
                temp->val.fval *= -1;
                break;
            case CHAR_TYPE:
                /* sign before char error */
                fprintf(stderr, "Error having sign before character
constant!\n");
                exit(1);
                break;
        }

        $$ = (AST_Node*) temp;
    }
}

```

Semantic Action for referenced variable

```

var_ref: variable
{
    $$ = new_ast_ref_node($1, 0); /* no reference */
}
| REFER variable
{
    $$ = new_ast_ref_node($2, 1); /* reference */
}
;

```

Semantic action for assignment expression

```

assignment: var_ref ASSIGN expression
{
    AST_Node_Ref *temp = (AST_Node_Ref*) $1;
    $$ = new_ast_assign_node(temp->entry, temp->ref, $3);

    AST_Node *t=(AST_Node *)$3;
}
;

```

Action rule for statements

- increment/Decrement statement
 - int i=10;
 - I++;
 - ++i;

For increment or decrement we create a increment node which contain the entry of variable and increment **type 0 stand for increment and 1 stand for decrement.**

Semantic Action for increment decrement statment

```
statement:

    while_statement

| assignment SEMI

| CONTINUE SEMI

| BREAK SEMI

| ID INCR SEMI

{

    /* increment */

    if($2.ival == INC){

        $$ = new_ast_incr_node($1, 0, 0);

    }

    else{

        $$ = new_ast_incr_node($1, 1, 0);

    }

}

| INCR ID SEMI

{

    /* increment */

    if($1.ival == INC){

        $$ = new_ast_incr_node($2, 0, 1);

    }

    else{

        $$ = new_ast_incr_node($2, 1, 1);

    }

}
```

```
}  
;
```

Semantic action for Continue and break

There are two simple statements : continue and break and both represented by AST simple node, by only having difference in the statement type 0 stands for continue and 1 stands for break.

```
statement:while_statement  
| assignment SEMI  
| CONTINUE SEMI  
{  
    $$ = new_ast_simple_node(0);  
}  
| BREAK SEMI  
{  
    $$ = new_ast_simple_node(1);  
}  
| ID INCR SEMI  
| INCR ID SEMI  
;
```

Semantic Action for Assignment statement

In the assignment right part is an expression that will be simply passed as the assign_val node entry of the AST assign node. AST assign node store in ST entry, assign value and ref.

Example:-

```
int i=5;  
a=10;
```

```

                                a=i;
statement:
    while_statement
    {
        $$ = $1; /* just pass information */
    }
    | assignment SEMI
    {
        $$ = $1; /* just pass information */
    }
    | CONTINUE SEMI
    | BREAK SEMI
    | ID INCR SEMI
    | INCR ID SEMI
;
assignment: var_ref ASSIGN expression
{
    AST_Node_Ref *temp = (AST_Node_Ref*) $1;
    $$ = new_ast_assign_node(temp->entry, temp->ref, $3);

    AST_Node *t=(AST_Node *)$3;
}
;
var_ref: variable
{
    $$ = new_ast_ref_node($1, 0); /* no reference */
}
| REFER variable
{
    $$ = new_ast_ref_node($2, 1); /* reference */
}
;

```

Action Rule while Loop statements

We only use While statement for loop while statement contain condition for loop execution and body during. while node contain the condition and while branch (tail)

- tail is used for production of while branch.

```
statement:
    While_statement
{
    $$ = $1; /* just pass information */
}
.....
;

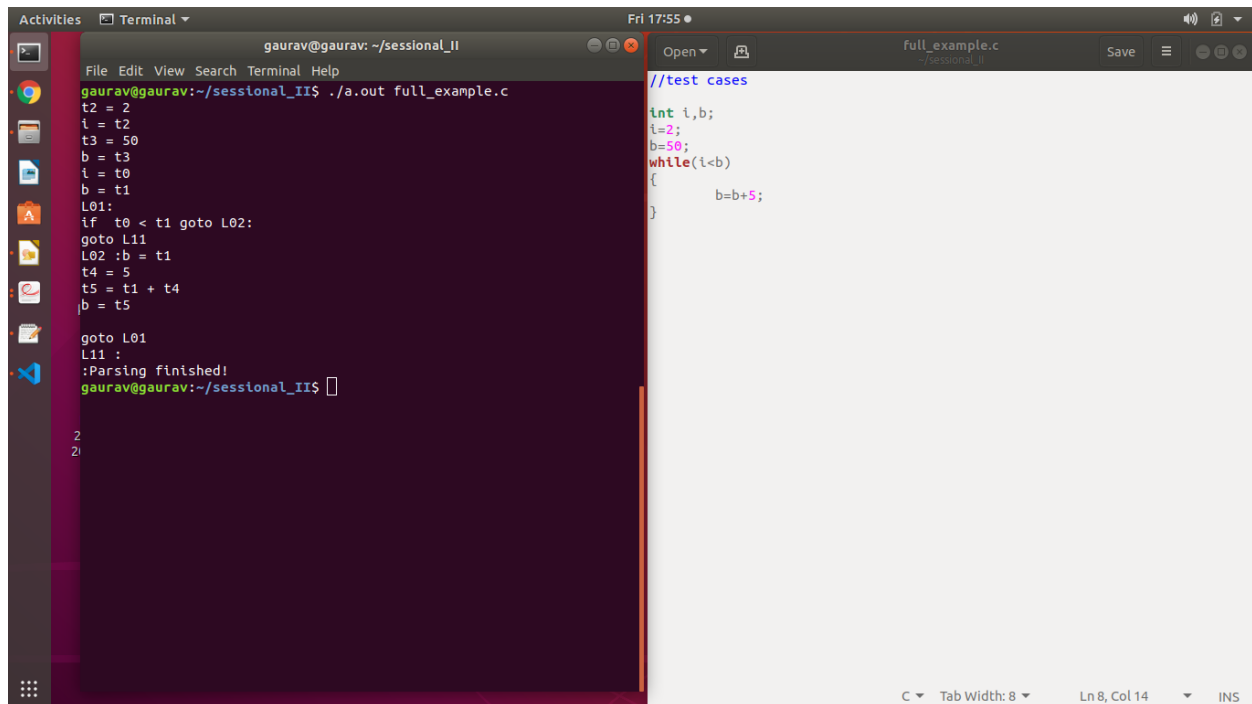
while_statement: WHILE LPAREN expression RPAREN tail
{
    $$ = new_ast_while_node($3, $5, &loop_no);
};

tail: LBACE statements RBACE
{
    $$ = $2; /* just pass information */
}
;
```


Three-address code

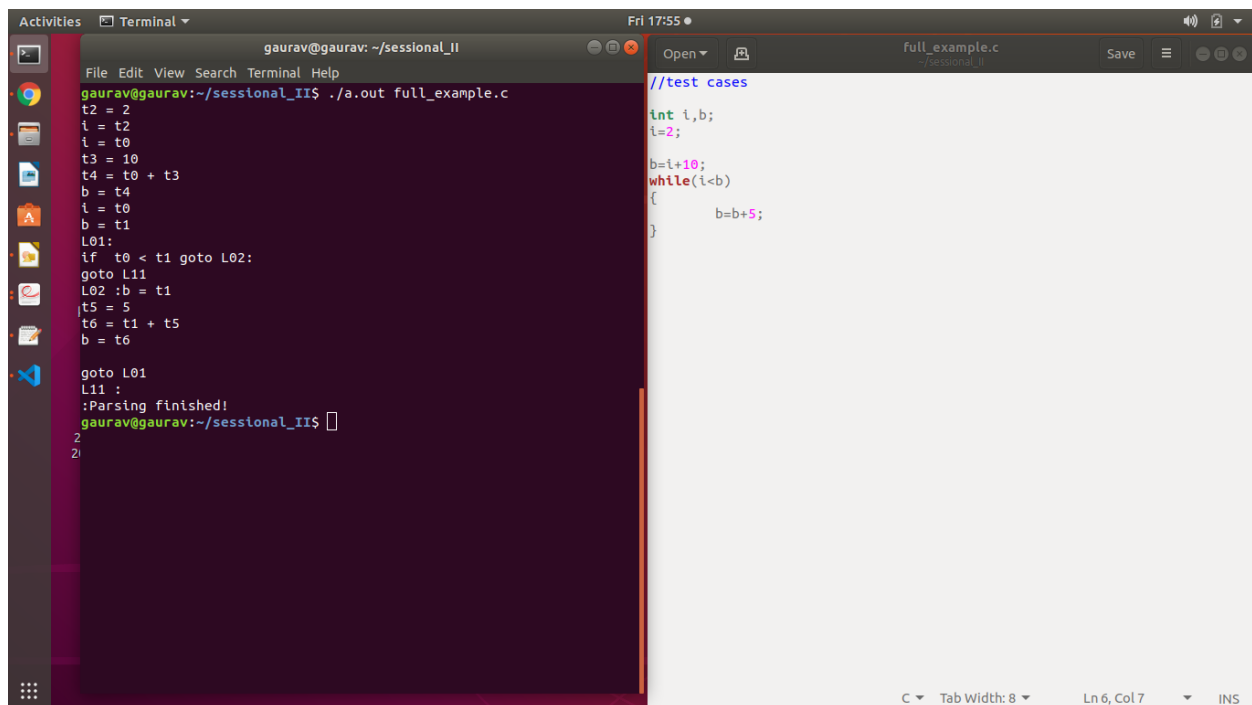
Three address code is generated during the tree traversal

Outputs of program :



```
gaurav@gaurav: ~/.sessional_II$ ./a.out full_example.c
t2 = 2
i = t2
t3 = 50
b = t3
i = t0
b = t1
L01:
if t0 < t1 goto L02:
goto L11
L02 :b = t1
t4 = 5
t5 = t1 + t4
b = t5

goto L01
L11 :
:Parsing finished!
gaurav@gaurav: ~/.sessional_II$
```



```
gaurav@gaurav: ~/.sessional_II$ ./a.out full_example.c
t2 = 2
i = t2
t3 = 10
t4 = t0 + t3
b = t4
i = t0
b = t1
L01:
if t0 < t1 goto L02:
goto L11
L02 :b = t1
t5 = 5
t6 = t1 + t5
b = t6

goto L01
L11 :
:Parsing finished!
gaurav@gaurav: ~/.sessional_II$
```

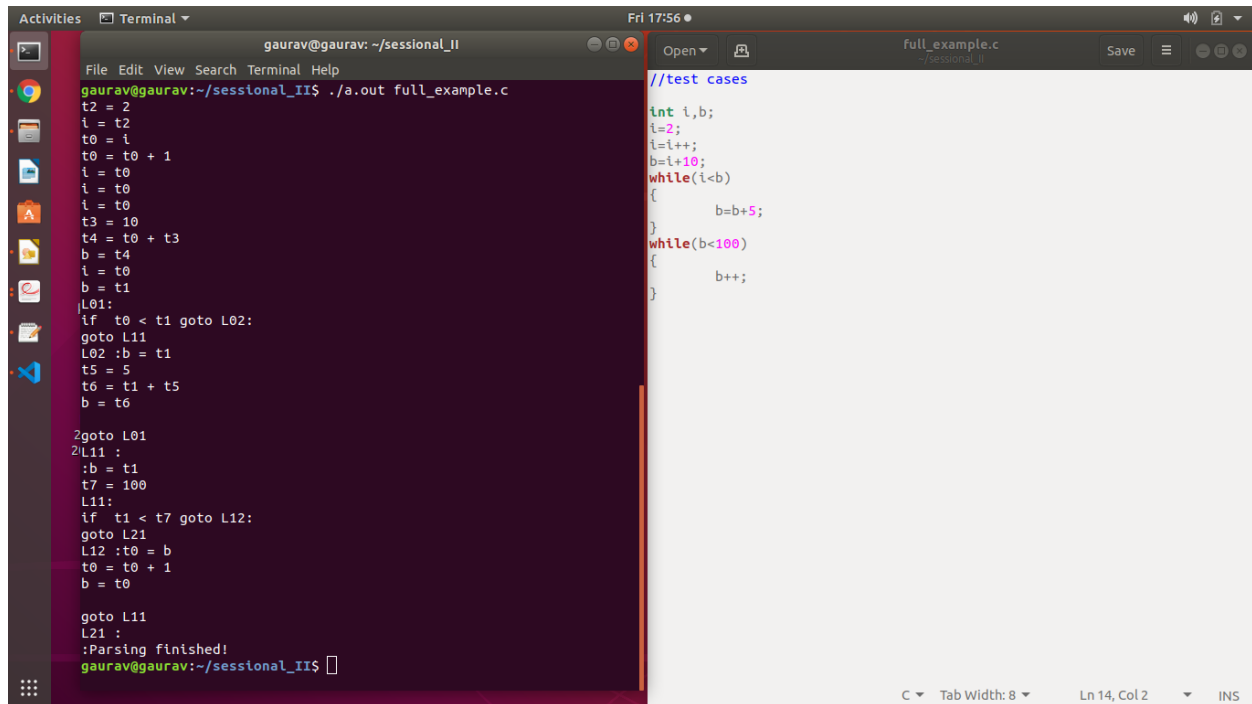
```
gaurav@gaurav: ~/sessional_II
File Edit View Search Terminal Help
gaurav@gaurav:~/sessional_II$ ./a.out full_example.c
t2 = 2
i = t2
i = t0
t3 = 10
t4 = t0 + t3
b = t4
i = t0
b = t1
L01:
if t0 < t1 goto L02:
goto L11
L02 :b = t1
t5 = 5
t6 = t1 + t5
b = t6

goto L01
L11 :
:b = t1
t7 = 100
2L11:
2if t1 < t7 goto L12:
goto L21
L12 :t0 = b
t0 = t0 + 1
b = t0

goto L11
L21 :
:Parsing finished!
gaurav@gaurav:~/sessional_II$
```

```
//test cases
int i,b;
i=2;
b=i+10;
while(i<b)
{
    b=b+5;
}
while(b<100)
{
    b++;
}
```

C Tab Width: 8 Ln 14, Col 2 INS



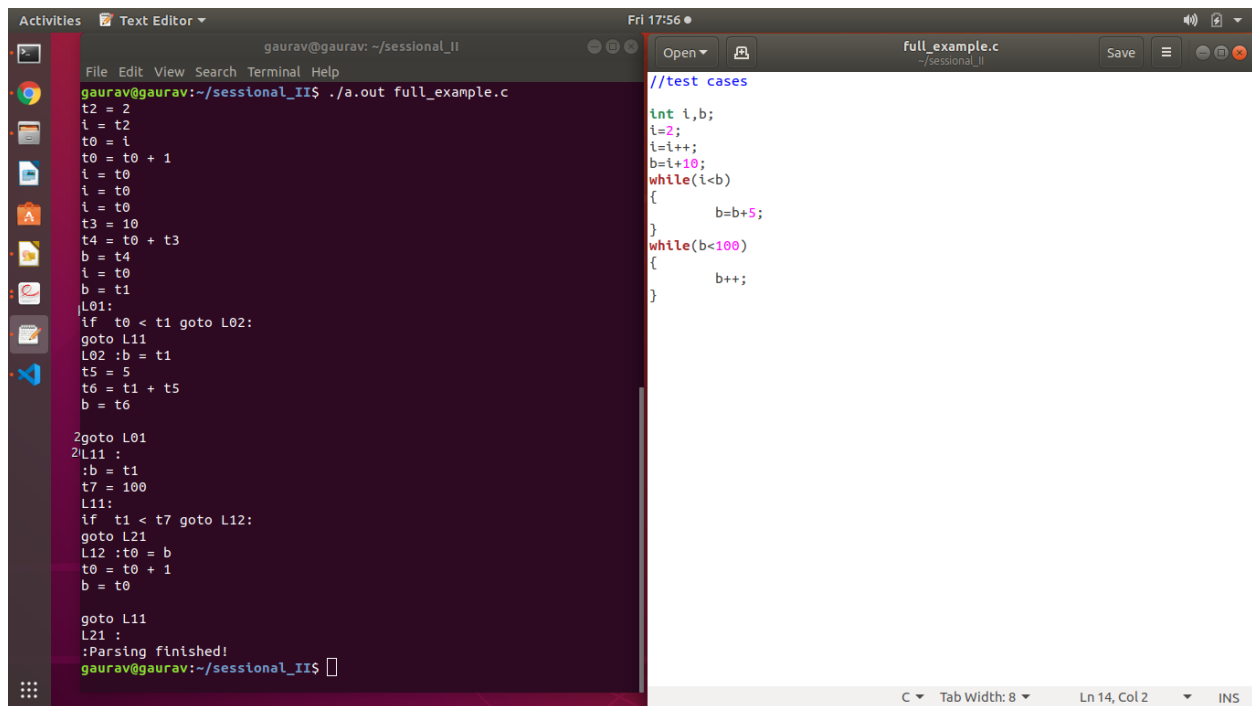
```
gaurav@gaurav: ~/sessional_II
File Edit View Search Terminal Help
gaurav@gaurav:~/sessional_II$ ./a.out full_example.c
t2 = 2
i = t2
t0 = i
t0 = t0 + 1
i = t0
i = t0
i = t0
t3 = 10
t4 = t0 + t3
b = t4
i = t0
b = t1
L01:
if t0 < t1 goto L02:
goto L11
L02 :b = t1
t5 = 5
t6 = t1 + t5
b = t6

2goto L01
2L11 :
:b = t1
t7 = 100
L11:
if t1 < t7 goto L12:
goto L21
L12 :t0 = b
t0 = t0 + 1
b = t0

goto L11
L21 :
:Parsing finished!
gaurav@gaurav:~/sessional_II$
```

```
//test cases
int i,b;
i=2;
i=i++;
b=i+10;
while(i<b)
{
    b=b+5;
}
while(b<100)
{
    b++;
}
```

C Tab Width: 8 Ln 14, Col 2 INS



```
gaurav@gaurav: ~/sessional_II
File Edit View Search Terminal Help
gaurav@gaurav:~/sessional_II$ ./a.out full_example.c
t2 = 2
i = t2
t0 = i
t0 = t0 + 1
i = t0
i = t0
i = t0
t3 = 10
t4 = t0 + t3
b = t4
i = t0
b = t1
L01:
if t0 < t1 goto L02:
goto L11
L02 :b = t1
t5 = 5
t6 = t1 + t5
b = t6

2goto L01
2L11 :
:b = t1
t7 = 100
L11:
if t1 < t7 goto L12:
goto L21
L12 :t0 = b
t0 = t0 + 1
b = t0

goto L11
L21 :
:Parsing finished!
gaurav@gaurav:~/sessional_II$
```

```
//test cases
int i,b;
i=2;
i=i++;
b=i+10;
while(i<b)
{
    b=b+5;
}
while(b<100)
{
    b++;
}
```

C Tab Width: 8 Ln 14, Col 2 INS

-
- Node creation function and definition of nodes, list creation ,Tree traversal codes are in **ast.c** file.
 - Parser.y for bison program
 - Lexer.l for the token creation
 - semantic .c for type checking and error handling
 - Symtab.c for insert and lookup of Identifiers.