Computer Secience and Engineering - Compiler

# **Assignment** 3

Soumik Roy 150101074
Ayush singh 150101013
Bishwendra choudhary Roll 150101017
Aman Agarwal 150101005

March 19, 2018

## 1. Perser Code

```
%{
    #include <stdio.h>
    #include <stdlib.h>
    #include <iostream>
    #include <fstream>
    #include <string>
    #include "functions.cpp"
    using namespace std;
    int yylex(void);
    void yyerror(char *s)
    {

        fprintf(stderr,"Unknown errors detected.\n");
        extern int yylineno;
    }
    int flag = 0;


    extern int  yylex();
    extern int yylineno;
    extern int lineno;
%}

%union{
        node *Node;
}

%token <Node>   NUMCONST SCOPE_SPECIFIER TYPE_SPECIFIER RETURN PRINT IF ELSE WHILE FOR BREAK
BOOLCONST READ IDENTIFIER COMMA
%token <Node>   SEMICOLON OPENBRACES OPENPAREN CLOSEDBRACES CLOSEDPAREN
OPENBRACKETS CLOSEBRACKETS
%token <Node>   MINUS PLUS BY TIMES MOD NOT
%token <Node>   GREQ LEQ LESS GREATER NEQ EQUALS AND OR ASSIGN
%start   program
%expect 6

%type <Node> program declaration_list declaration empty scoped_type_specifier scoped_variable_dec
 variable_dec variable_dec_list function_dec parameters parameter_list parameter_type_list
 statement print_stmt compound_stmt stmt_list expression_stmt conditional_stmt iteration_stmt
 return_stmt break_stmt expression expr_suffix read_expr simple_expr and_expr unary_rel_expr
 rel_expr sum_expr sumop term mulop factor call args arg_list constant array array_elem openparen
 closedparen closedbraces comma semicolon id

%%

program : declaration_list
{
     $$ = add_node("program",$1);

    if(flag==0)
    {
        printf("Compilation successful.\n");
        ofstream output;
            output.open("tree.txt");
        print_nodes($$,output);
```

```
        output.close();
    }
    else
    {
        printf("Syntax errors found.\n");
    }

}

        ;
empty :
{
    $$ = NULL;
}

;

declaration_list : declaration_list declaration
{
    $$ = add_node("declaration_list",$1,$2);
}
                 | declaration
{
    $$ = add_node("declaration_list",$1);
}
                 ;

declaration : scoped_variable_dec
{
   $$ = add_node("declaration",$1);
}
            | function_dec
{
   $$ = add_node("declaration",$1);
}
            | variable_dec
{
   $$ = add_node("declaration",$1);
}
            | error {printf("Error: Missing Type Specifier at line %d\n", lineno); flag = 1;$$ = NULL;}
            ;


scoped_type_specifier    : SCOPE_SPECIFIER TYPE_SPECIFIER
{
    $$ = add_node("scoped_type_specifier",$1,$2);
}
                         | SCOPE_SPECIFIER error {printf("Error: Missing Type Specifier at line
                         %d\n", lineno); flag = 1;$$ = NULL;}

;

scoped_variable_dec      : scoped_type_specifier variable_dec_list semicolon
{
    $$ = add_node("scoped_variable_dec",$1,$2,$3);
};
```

```
variable_dec            : TYPE_SPECIFIER variable_dec_list semicolon
{
    $$ = add_node("variable_dec",$1,$2,$3);
}
;


variable_dec_list       : variable_dec_list comma id
{
    $$ = add_node("variable_dec_list",$1,$2,$3);
}
                        | id

{
    $$ = add_node("variable_dec_list",$1);
}
;

function_dec            : TYPE_SPECIFIER IDENTIFIER OPENPAREN parameters
closedparen compound_stmt
{
    $$ = add_node("function_dec",$1,$2,$3,$4,$5,$6);
}


;


parameters              : parameter_list
{
    $$ = add_node("parameters",$1);
}
| empty
{
    $$ = add_node("parameters",$1);
}
;

parameter_list          : parameter_list comma parameter_type_list
{
    $$ = add_node("parameter_list",$1,$2,$3);
}
                        | parameter_type_list
{
    $$ = add_node("parameter_list",$1);
}
;

parameter_type_list     : TYPE_SPECIFIER IDENTIFIER
{
    $$ = add_node("parameter_type_list",$1,$2);
}
                        | IDENTIFIER {printf("Error: Missing Type Specifier at line
                        %d\n", lineno); flag = 1; $$ = NULL;};

;
```

```
statement                : print_stmt
{
    $$ = add_node("statement",$1);
}
                         | expression_stmt
{
    $$ = add_node("statement",$1);
}
                         | compound_stmt
{
    $$ = add_node("statement",$1);
}

                         | conditional_stmt
{
    $$ = add_node("statement",$1);
}
                         | iteration_stmt
{
    $$ = add_node("statement",$1);
}
                         | return_stmt
{
    $$ = add_node("statement",$1);
}
                         | break_stmt
{
    $$ = add_node("statement",$1);
}
                         | scoped_variable_dec
{
    $$ = add_node("statement",$1);
}
                         | variable_dec
{
    $$ = add_node("statement",$1);
}
                         ;

print_stmt               : PRINT  expression semicolon
{
    $$ = add_node("print_stmt",$1,$2,$3);
}
;

compound_stmt            : OPENBRACES stmt_list closedbraces
{
    $$ = add_node("compound_stmt",$1,$2,$3);
}
;

stmt_list                : stmt_list statement
{
```

```
    $$ = add_node("stmt_list",$1, $2);
}
                        | empty
{
    $$ = add_node("stmt_list",$1);
}
                        ;

expression_stmt         : expression semicolon
{
    $$ = add_node("expression_stmt",$1,$2);
}
                         ;

conditional_stmt        : IF  openparen expression closedparen statement
{
    $$ = add_node("conditional_stmt",$1,$2,$3,$4,$5);
}

                        | IF openparen expression closedparen statement ELSE statement
{
    $$ = add_node("conditional_stmt",$1,$2,$3,$4,$5,$6,$7);
}
                        ;

iteration_stmt          : WHILE openparen expression closedparen statement
{
    $$ = add_node("iteration_stmt",$1,$2,$3,$4,$5);
}
                        | FOR openparen expression SEMICOLON expression SEMICOLON
expression closedparen statement
{
    $$ = add_node("iteration_stmt",$1,$2,$3,$4,$5,$6,$7,$8,$9);
}
;

return_stmt             : RETURN  semicolon
{
    $$ = add_node("return_stmt",$1,$2);
}
                        | RETURN  expression semicolon
{
    $$ = add_node("return_stmt",$1,$2,$3);
};

break_stmt              : BREAK semicolon
{
    $$ = add_node("break_stmt",$1,$2);
}
;
expression              : read_expr
{
    $$ = add_node("expression",$1);
}
                        | IDENTIFIER ASSIGN simple_expr
{
    $$ = add_node("expression",$1,$2,$3);
```

```
}
                        | IDENTIFIER PLUS expr_suffix
{
    $$ = add_node("expression",$1,$2,$3);
}
                        | IDENTIFIER MINUS expr_suffix
{
    $$ = add_node("expression",$1,$2,$3);
}
                        | IDENTIFIER TIMES expr_suffix
{
    $$ = add_node("expression",$1,$2,$3);
}
                        | IDENTIFIER BY expr_suffix
{
    $$ = add_node("expression",$1,$2,$3);
}
                        | IDENTIFIER MOD expr_suffix
{
    $$ = add_node("expression",$1,$2,$3);
}
                        | simple_expr
{
    $$ = add_node("expression",$1);
}
                        ;

expr_suffix             : simple_expr
{
    $$ = add_node("expr_suffix",$1);
}
                        | ASSIGN simple_expr
{
    $$ = add_node("expr_suffix",$1,$2);
}
                        ;

read_expr               : READ IDENTIFIER
{
    $$ = add_node("read_expr",$1,$2);
}

;

simple_expr             : OPENPAREN simple_expr OR and_expr closedparen
{
    $$ = add_node("simple_expr",$1,$2,$3,$4,$5);
}
                        | and_expr
{
    $$ = add_node("simple_expr",$1);
}
                        ;

and_expr                : and_expr AND unary_rel_expr
{
```

```
    $$ = add_node("and_expr",$1,$2,$3);
}
                        | unary_rel_expr
{
    $$ = add_node("and_expr",$1);
}
                        ;

unary_rel_expr          : NOT unary_rel_expr
{
    $$ = add_node("unary_rel_expr",$1,$2);
}
                        | rel_expr
{
    $$ = add_node("unary_rel_expr",$1);
}
                        ;

rel_expr                : sum_expr GREATER sum_expr
{
    $$ = add_node("rel_expr",$1,$2,$3);
}
                        | sum_expr LESS sum_expr
{
    $$ = add_node("rel_expr",$1,$2,$3);
}
                        | sum_expr EQUALS sum_expr
{
    $$ = add_node("rel_expr",$1,$2,$3);
}
                        | sum_expr GREEQ sum_expr
{
    $$ = add_node("rel_expr",$1,$2,$3);
}
                        | sum_expr LEQ sum_expr
{
    $$ = add_node("rel_expr",$1,$2,$3);
}
                        | sum_expr NEQ sum_expr
{
    $$ = add_node("rel_expr",$1,$2,$3);
}
                        | sum_expr
{
    $$ = add_node("rel_expr",$1);
}
                        ;

sum_expr                : sum_expr sumop term
{
    $$ = add_node("sum_expr",$1,$2,$3);
}
                        | term
{
    $$ = add_node("sum_expr",$1);
}
                        ;
```

```
sumop                   : PLUS
{
    $$ = add_node("sumop",$1);
}
                        | MINUS
{
    $$ = add_node("sumop",$1);
}
                        ;

term                    : term mulop factor
{
    $$ = add_node("term",$1,$2,$3);
}
                        | factor
{
    $$ = add_node("term",$1);
}
                        ;

mulop                   : TIMES
{
    $$ = add_node("mulop",$1);
}

                        | BY
{
    $$ = add_node("mulop",$1);
}
                        | MOD
{
    $$ = add_node("mulop",$1);
}
;

factor                  : IDENTIFIER
{
    $$ = add_node("factor",$1);
}
                        | OPENPAREN expression closedparen
{
    $$ = add_node("factor",$1,$2,$3);
}
                        |  call
{
    $$ = add_node("factor",$1);
}
                        | constant
{
    $$ = add_node("factor",$1);
}
                        | array
{
    $$ = add_node("factor", $1);
}
                        ;
```

```
call                    : IDENTIFIER OPENPAREN args closedparen
{
    $$ = add_node("call",$1,$2,$3,$4);
}
;

args                    : arg_list
{
    $$ = add_node("args",$1);
}

                        | empty
{
    $$ = add_node("args",$1);
}
                        ;

arg_list                : arg_list comma expression
{
    $$ = add_node("arg_list",$1,$2,$3);
}
                        | expression
{
    $$ = add_node("arg_list",$1);
}
                        ;

constant                : NUMCONST
{
    $$ = add_node("constant",$1);
}
                        | BOOLCONST
{
    $$ = add_node("constant",$1);
}
                        ;

array                   : OPENBRACKETS CLOSEBRACKETS
{
    $$ = add_node("array",$1,$2);
}
                        | OPENBRACKETS array_elem CLOSEBRACKETS
{
    $$ = add_node("array",$1,$2,$3);
};

array_elem              : constant comma array_elem
{
    $$ = add_node("array_elem", $1,$2,$3);
}
                        | constant
{
    $$ = add_node("array_elem",$1);
};

openparen               : OPENPAREN
```

```
{
    $$ = add_node("openparen",$1);
}
                        | error
{printf("Error: '(' expected after IF or WHILE at line %d\n", lineno);
flag = 1;
$$ = NULL;
};


closedparen             : CLOSEDPAREN

{
    $$ = add_node("closedparen",$1);
}
                        | error {printf("Error: Missing ')' at line %d\n", lineno); flag = 1; $$ = NULL;};

closedbraces            : CLOSEDBRACES
{
    $$ = add_node("closedbraces",$1);
}
                        | error {printf("Error: Missing '}' at line %d\n", lineno); flag = 1; $$ = NULL;};

comma                   : COMMA
{
    $$ = add_node("comma",$1);
}
                        ;

semicolon               : SEMICOLON
{
    $$ = add_node("semicolon",$1);
}
                        | error {printf("Error: Missing ';' or ',' near line %d\n", lineno);
                         flag = 1; $$ = NULL;};

id                      : IDENTIFIER
{
    $$ = add_node("id",$1);
}
                        | comma {printf("Error: Extra ',' at line %d\n", lineno); flag = 1;}
                        | error {printf("Error: Missing identifier name at line %d\n", lineno); flag = 1;
                        $$ = NULL;};
```

```
%%
int main(){
    //yydebug = 1;
    yyparse();
    return 0 ;
}
```

## 2. Lexer Code

```
%{
#include "functions2.cpp"
```

```
#include "y.tab.h"
using namespace std;
extern "C" int yywrap() { }
int lineno = 1;
%}

%option yylineno

DIGIT [0-9]
NUMBER {DIGIT}+
FLOAT_NUMBER {DIGIT}*"."{DIGIT}+
TEXT [a-zA-Z ]+
TEXT_NUMBERS [a-zA-Z0-9]*
SCOPE_SPECIFIER "static"
TYPE_SPECIFIER "int"|"float"|"char"|"bool"|"Processor"|"JOB"|"MEMORY"|"LINK"
RETURN "return"
PRINT "print"
IF "if"
ELSE "else"
WHILE "while"
FOR "for"
BREAK "break"
BOOLCONST "true"|"false"
READ "read"
WHITESPACE [; :\t\n]
IDENTIFIER [a-zA-Z]{TEXT_NUMBERS}|[a-zA-Z]{TEXT_NUMBERS}[[{NUMBER}+]]

%%

{NUMBER}            { yylval.Node = add_node_leaf("NUMBER", string(yytext)); return NUMCONST; }
{FLOAT_NUMBER}      { yylval.Node = add_node_leaf("FLOAT_NUMBER", string(yytext)); return NUMCONST; }
{SCOPE_SPECIFIER}   { yylval.Node = add_node_leaf("STATIC", "static"); return SCOPE_SPECIFIER; }
"int"               { yylval.Node = add_node_leaf("TYPE_SPECIFIER", "int"); return TYPE_SPECIFIER; }
"float"             { yylval.Node = add_node_leaf("TYPE_SPECIFIER", "float"); return TYPE_SPECIFIER; }
"char"              { yylval.Node = add_node_leaf("TYPE_SPECIFIER", "char"); return TYPE_SPECIFIER; }
"bool"              { yylval.Node = add_node_leaf("TYPE_SPECIFIER", "bool"); return TYPE_SPECIFIER; }
"Processor"         { yylval.Node = add_node_leaf("TYPE_SPECIFIER", "Processor"); return TYPE_SPECIFIER; }
"JOB"               { yylval.Node = add_node_leaf("TYPE_SPECIFIER", "JOB"); return TYPE_SPECIFIER; }
"MEMORY"            { yylval.Node = add_node_leaf("TYPE_SPECIFIER", "MEMORY"); return TYPE_SPECIFIER; }
"LINK"              { yylval.Node = add_node_leaf("TYPE_SPECIFIER", "LINK"); return TYPE_SPECIFIER; }
{RETURN}            { yylval.Node = add_node_leaf("RETURN", "return"); return RETURN; }
{PRINT}             { yylval.Node = add_node_leaf("PRINT", "print"); return PRINT; }
{IF}                { yylval.Node = add_node_leaf("IF", "if"); return IF; }
{ELSE}              { yylval.Node = add_node_leaf("ELSE", "else"); return ELSE; }
{WHILE}             { yylval.Node = add_node_leaf("WHILE", "while"); return WHILE; }
{FOR}                 { yylval.Node = add_node_leaf("FOR", "for"); return FOR; }
{BREAK}             { yylval.Node = add_node_leaf("BREAK", "break"); return BREAK; }
"true"              { yylval.Node = add_node_leaf("BOOLCONST","true"); return BOOLCONST; }
"false"             { yylval.Node = add_node_leaf("BOOLCONST","false"); return BOOLCONST; }
{READ}              { yylval.Node = add_node_leaf("READ", "read"); return READ; }
{IDENTIFIER}        { yylval.Node = add_node_leaf("IDENTIFIER", string(yytext)); return IDENTIFIER; }
","                                 { yylval.Node = add_node_leaf("COMMA", ","); return COMMA; }
";"                 { yylval.Node = add_node_leaf("SEMICOLON", ";"); return SEMICOLON; }
"{"                 { yylval.Node = add_node_leaf("OPENBRACES", "{"); return OPENBRACES; }
"}"                 { yylval.Node = add_node_leaf("CLOSEDBRACES", "}"); return CLOSEDBRACES; }
"("                 { yylval.Node = add_node_leaf("OPENPAREN", "("); return OPENPAREN; }
")"                 { yylval.Node = add_node_leaf("CLOSEDPAREN", ")"); return CLOSEDPAREN; }
```

```
"["                     { yylval.Node = add_node_leaf("OPENBRACKETS", "["); return OPENBRACKETS; }
"]"                     { yylval.Node = add_node_leaf("CLOSEBRACKETS", "]"); return CLOSEBRACKETS; }
"="                     { yylval.Node = add_node_leaf("ASSIGN", "="); return ASSIGN; }
"+"                     { yylval.Node = add_node_leaf("PLUS", "+"); return PLUS; }
"-"                     { yylval.Node = add_node_leaf("MINUS", "-"); return MINUS; }
"/"                     { yylval.Node = add_node_leaf("BY", "/"); return BY; }
"*"                     { yylval.Node = add_node_leaf("TIMES", "*"); return TIMES; }
">"                     { yylval.Node = add_node_leaf("GREATER", ">"); return GREATER; }
"<"                     { yylval.Node = add_node_leaf("LESS", "<"); return LESS; }
">="                    { yylval.Node = add_node_leaf("GREQ", ">="); return GREQ; }
"<="                    { yylval.Node = add_node_leaf("LEQ", "<="); return LEQ; }
"=="                    { yylval.Node = add_node_leaf("EQUALS", "=="); return EQUALS; }
"!="                    { yylval.Node = add_node_leaf("NEQ", "!="); return NEQ; }
"&"                     { yylval.Node = add_node_leaf("AND", "&"); return AND; }
"|"                     { yylval.Node = add_node_leaf("OR", "|"); return OR; }
"!"                     { yylval.Node = add_node_leaf("NOT", "!"); return NOT; }
"%"                     { yylval.Node = add_node_leaf("MOD", "%"); return MOD; }
[\n]                    {yylineno++; lineno++;}

%%
```

## 3. Function_1

```
#include "tree.h"
#include <iostream>
node* add_node(string name, node* a=NULL, node* b=NULL, node* c=NULL, node* d=NULL,
 node* e=NULL, node* f=NULL, node* g=NULL, node* h=NULL, node* i=NULL, node* j=NULL) {
        static int no = 1;
        node *new_node;
        new_node = new node();

        new_node->children[0] = a;
        new_node->children[1] = b;
        new_node->children[2] = c;
        new_node->children[3] = d;
        new_node->children[4] = e;
        new_node->children[5] = f;
        new_node->children[6] = g;
        new_node->children[7] = h;
        new_node->children[8] = i;
        new_node->children[9] = j;
        new_node->children[10] = NULL;
        new_node->node_name=name;
        new_node->node_no = no * 11;
        no++;
        // new_node->line_no=line;

        return new_node;
    }


void print_nodes(struct node* root, std::ofstream& myfile)
{
        if(root == NULL)
                return;
        else
        {
```

```
cout << "Parent␣node␣" << root->node_no << "(␣" << root->node_name << "␣)␣:␣";
 myfile << "Parent␣node␣" << root->node_no << "(␣" << root->node_name << "␣)␣:␣";
if(root->children[0] == NULL)
{
        cout << root->node_val;
        myfile << root->node_val;
}
for(int i=0 ; i<11 ; i++)
{
        if(root->children[i] != NULL)
        {
                cout << root->children[i]->node_no << "␣(" << root->children[i]->node_name <
                 myfile << root->children[i]->node_no << "␣(" << root->children[i]->node_nam
        }
        else
        {
                cout << "\n";
                myfile << "\n";
                break;
        }
    }
    for(int i=0;i<11; i++)
    {
            if(root->children[i] != NULL)
            {
                    print_nodes(root->children[i],myfile);
                    //break;
            }
    }
  }

}
```

## 4. Function_2

```
#include "tree.h"

node *add_node_leaf(string name, string value)
{
    static int no = 1;
    node *new_node;
    new_node = new node();
    new_node->children[0] = NULL;
    new_node->children[1] = NULL;
    new_node->children[2] = NULL;
    new_node->children[3] = NULL;
    new_node->children[4] = NULL;
    new_node->children[5] = NULL;
    new_node->children[6] = NULL;
    new_node->children[7] = NULL;
    new_node->children[8] = NULL;
    new_node->children[9] = NULL;
    new_node->children[10] = NULL;
    new_node->node_name=name;
    new_node->node_val=value;
    new_node->node_no = no * 11 + 1;
```

```
    no++;
    return new_node;
}
```

## 5. Tree_h

```
#include <iostream>
#include <vector>
#include <cstdio>
#include <cstring>
#include <string>

using namespace std;

struct node{
    node* children[11];
    string node_name;
    string node_val;
    int line_no;
    int node_no;
    bool is_int;
    bool is_bool;
    bool is_job;
    bool is_processor;
    bool is_memory;
    bool is_link;

    node() {
        is_int=false;
        is_bool=false;
        is_link=false;
        is_memory=false;
        is_processor=false;
        is_job=false;
    }
}; new_node;
}
```