BRAC UNIVERSITY

Inspiring Excellence

Department of Computer Science and Engineering

Course Code: CSE 420	**Credits: 1.5**
Course Name: Compiler Design	**Semester:** **Spring 2024**

1 Introduction

In the previous assignments, we have built a lexical analyzer, constructed a syntax analyzer with a symbol table, and performed semantic analysis for our subset of the C language. Now, we will move to the next phase of compilation: intermediate code generation.

Intermediate code is a representation of the program that bridges the gap between the high-level source code and the low-level target code. One common form of intermediate representation is three-address code, where each instruction has at most three operands. By generating an intermediate representation, we can simplify the code generation phase and enable various optimizations.

In this assignment, we will first build an Abstract Syntax Tree (AST) during parsing and then traverse this AST to generate three-address code.

2 Tasks

In this assignment, you have to perform the following tasks:

2.1 Abstract Syntax Tree (AST)

You need to implement an AST for the C subset we have been working with. Your implementation should include:

1. A base **ASTNode** class that all other node types will inherit from.
2. **ExprNode** classes for different expressions:
    - **VarNode** for variable references

    - **ConstNode** for constants
    - **BinaryOpNode** for binary operations (+, -, *, /, etc.)
    - **UnaryOpNode** for unary operations (-, !, etc.)
    - **AssignNode** for assignment operations
    - **FuncCallNode** for function calls
3. **StmtNode** classes for different statements:
    - **ExprStmtNode** for expression statements
    - **BlockNode** for compound statements (blocks)
    - **IfNode** for if-else statements
    - **WhileNode** for while loops
    - **ForNode** for for loops
    - **ReturnNode** for return statements
    - **DeclNode** for variable declarations
4. **ProgramNode** as the root of the AST representing the entire program

The skeleton of these classes is provided in the "ast.h" file. You need to complete the implementation, especially focusing on the generate_code method for each node type.

2.2 AST Construction During Parsing

Modify your parser to build the AST while parsing the input code. For each grammar rule, create the corresponding AST nodes and connect them appropriately.

Here's an example of how to modify the action part of a grammar rule to build an AST node:

```
expression : logic_expression
{
   $$ = $1; // Pass through the AST node
}
| variable ASSIGNOP logic_expression
{
   // Create an assignment node
   AssignNode* assignNode = new AssignNode(
      (VarNode*)$1->get_ast_node(),
      (ExprNode*)$3->get_ast_node(),
      $$->getvartype()
   );
   $$->set_ast_node(assignNode);
}

;
```

2.3 Three-Address Code Generation

Implement the generate_code method for each AST node type to output three-address code instructions. The three-address code should follow this format:

- Temporary variables: t0, t1, t2, ...
- Labels: L0, L1, L2, ...
- Operations: +, -, *, /, etc.
- Control flow: if, goto, call, return

Here are some examples of three-address code instructions:

- t0 = a + b
- t1 = t0 * c
- if t1 goto L0
- goto L1
- L0:
- a = b
- L1:

Specifically, implement code generation for:

1. Arithmetic operations (+, -, *, /)
2. Logical operations (&&, ||, !)
3. Relational operations (<, >, ==, !=, <=, >=)
4. Assignment operations (=)
5. Array access (a[i])
6. Control structures (if-else, while, for)
7. Function calls and returns

4 Input

The input will be a file with .c extension containing a c source program. File name will be given from the command line.

5 Output

In this assignment, there will be three output files:

1. **log.txt**: Contains matching grammar rules and corresponding segments of source code as in previous assignments. Also includes information about the constructed AST.

2. **error.txt**: Contains any error messages encountered during parsing or semantic analysis.
3. **code.txt**: Contains the generated three-address code.

Print the line count and error count at the end of the log file. Print the error count at the end of the error file.

The three-address code should be well-formatted and include comments to enhance readability. Each instruction should be on a separate line. Include appropriate labels for control flow instructions.

For more clarification about input-output, check the supplied sample I/O files given in the lab folder. You are highly encouraged to produce output exactly like the sample ones

6 Submission

1. In your local machine, create a new folder whose **name is your student id**.
2. Put the following files in the folder:
    - The lex file named as **<your_student_id>.l**
    - The Yacc file **<your_student_id>.y**
    - A script named **script.sh** (modified with your own filenames)
    - The **symbol_info.h** file
    - The **scope_table.h** file
    - The **symbol_table.h** file
    - The **ast.h** file
    - The **three_addr_code.h** file
    - A suitable input file named **input.c**
    - **DO NOT** put any output file, generated lex.yy.c/y.tab.h/y.tab.c file, or any executable file in this folder
3. Compress the folder in a **.zip file** which should be **named as your student id**.
4. Submit the .zip file.

Failure to follow these instructions will result in a penalty.