Project 1 - miniC Frontend

COSC 57/257

Spring Term 2022

1 Introduction

In this project you will implement the frontend of a miniC compiler. The input to your executable will be a filename given as a command line argument; the file contains a program in the miniC language.

2 miniC

miniC is a just a vastly simplified version of C.

All variables, parameters and constants in a miniC program are integers. Functions can return *void* or *int* and take a maximum of one *void* or *int* parameter.

2.1 Function Declarations

The ability to parse function declarations such as:

```
extern int read(void);
void print(int);
```

Function declarations have to be at the top of the file. (This should make parsing easier.) The functions being called do not have to exist or be compiled.

2.2 Expressions

A *term* is a variable or constant. An expression (*expr*) can be a unary expression, binary expression or a term.

- Binary arithmetic expression ('*term b_op term*'). *b_op* is a binary operator and can be `+` | `-` | `*` | `/`.
- Unary arithmetic expression ('*u_op term*'). *u_op* is a unary operator and can be `-` (minus sign).
- Relational expression ('*expr c_op expr*'). *c_op* is a comparison operator and can be `>` | `<` | `>=` | `<=` | `==` | `!=`.

2.3 Statements

Valid statements for miniC are:

- assignment (`variable =` *b_expr* or `variable =` *u_expr*)
- `while (`*r_expr*`)` *stmt*
- `if (`*r_expr*`)` *stmt*
- `if (`*r_expr*`)` *stmt* `else` *stmt*
- A block statement: {*stmt*+}

We assume all the declarations of variable occur before any statements in a block. No declaration statements in assignments (i.e., no `int b = 0;`)

2.4 Not Included

Things that you do not have to handle. This list is not exhaustive, so if you're not sure if you have to implement it, ask.

- Multiple function definitions
- Preprocessor commands (`#` commands)
- Comments
- Any statements that are not included in the above given 'statements' list
- Syntactic sugar like: `++`, `--`, `+=`, `-=`, `*=`, `/=`
- Empty blocks and statements ({}, `;`)

2.5 Input

A file containing a miniC program that meets the requirements given above.

2.6 Output

Your parser code should build an AST. The main function in your parser should call the `ast_print()` function defined in the AST code to print the constructed AST.

2.7 Semantic Analysis

Your frontend needs to check for just two semantic issues. These errors should display an error message. The frontend should try to find all errors, not find one and exit.

- It should generate an error on multiple declarations of a variable. Since we are not allowing declaration statements in assignments and only allowing variables to be declared at the top of a function (i.e., no variables inside a block for separate scope), so this should simplify the checks. For example,

```
int triangle (int n)
{
    int a;
    int b;
    int c;
    int a;
}
```

  The frontend should catch the duplicated variable *a*.

- It should generate an error on the use of undeclared variables. For example,

```
int triangle (int n)
{
    val = 0;
    while (n != 0)
    ...
}
```

  The frontend should catch val was not defined.

3 Implementation

You have a choice to use either flex/bison or use ANTLR4 (not ANTLR2). This also gives you choice for the language you can write in:

- flex/bison - you can use either C or C++
- ANTLR4 - you can use C++, Java, or Python

We provide the AST file for each of the languages so you don't have to write that part; you may modify the AST file if you choose. You should use a Makefile to make builds easier.

4 What to Submit

Your source code and a working executable in a zip or tarball file. Here is the list of what it should contain:

- minic.l and minic.y for flex/bison ***OR*** minic.g4 for ANTLR4.
- All the supporting source code and Makefile.
- The AST code you used with any changes made to it marked as comments.

5 Example

I know I write C oddly since I don't put the "open" bracket at the end of the line. (I've done it this way forever and it's impossible for me to change!) Your code should be able to parse it anyway.

```
int triangle (int n)
{
    int val;
    val = 0;

    while (n != 0)
    {
        val = val + n;
        n = n - 1;
    }
    return val;
}
```

I will create a set of test code that you can use to validate your frontend.

6 Due Date

Project 1 will be due **Wednesday, May 4, 2022**.

7 Rubric

Project 1 will count for 30% of your total grade. Project 1 will be graded out of 100 points:

- We will use five test cases to test your frontend code. These will be mini C routines with no errors, one error or multiple errors.
- Lexer specification: 25 points maximum. All necessary tokens created and ordered correctly. Plenty of comments to explain sections of the lexer file. Lexer builds and operates correctly.
- Parser specification: 40 points maximum. Parser handles all commands as per the project specification. Comments to explain sections of the parser. Parser code builds and operates correctly.

  If you use ANTLR, the lexer and parser are in one file, so we will look for completeness in each area and award points as if they were separate files.
- Semantic analysis: 20 points.
  - 5 points will be awarded if your frontend disallows multiple declarations of a variable.
  - 5 points will be awarded if your frontend disallows the use of undeclared variables.
  - 5 points will be awarded for displaying the correct error messages. Errors for multiple declarations and undeclared variables should be printed out.
  - 5 points for printing out the AST for all test cases. One point will be deducted for each non-working case.
- Bench test: 15 points. A maximum of 5 points will be deducted if the Makefile doesn't work properly. 10 points will be awarded if all test cases; 2 points will be deducted for each failing test case.

A maximum of 10 points will be deducted for unreadable "spaghetti code" or for no comments.

Late work will have 5 points deducted per day for the first week; work more than a week late will receive a zero.