CS202: PROGRAMMING PARADIGMS & PRAGMATICS

Semester II, 2021 – 2022

Lab 8: Compiler Design for CUCU

- ## Aim:

  Learning to write a compiler for a simple language

- ## Note

  Although we talked about Lex/Yacc in the class, the newer and currently available versions are called Flex / Bison. There are some technical differences between them, but for all practical purposes of this lab, they'll function exactly the same. Only difference is that instead of calling lex or yacc for compiling you'll be calling flex and bison as follows:

```
% bison -d cucu.y

% flex cucu.l

% g++ cucu.tab.c lex.yy.c -lfl -o cucu
```

- ## Introduction

  CUCU (A Compiler U Can Understand) is a toy compiler for a toy language. As writing a compiler for the whole ANSI C standards is very difficult, CUCU is a very small subset of the C language. Following is a sample CUCU code snippet:

```
int cucu_strlen(char *s) {
    int i = 0;
    while (s[i]) {
        i = i + 1;
    }
    return i;
}
```

- ## Language

  Here's a top-to-bottom description of grammar for CUCU:

  - CUCU source files contains a **program**
  - A program is a list of **variable declarations**, **function declarations** and **function definitions**

```
int func(char *s, int len); /* function declaration */
int i;                      /* variable declaration */

int func(char *s, int len) { /* function definition */
    ...
}
```

- Variable declaration is a **type** name followed by the **identifier** followed by a semicolon, like we usually do in C

```
int i;
char *s;
```

- Function declaration is a bit more complicated. It is a type name followed by identifier followed by an optional list of **function arguments** inside the parenthesis and ending in a semicolon (see the example above)
- Function arguments list, in turn, is a sequence of comma-separated "type identifier", like:

```
char *s, int from, int to
```

- The supported data types are only `int` and `char *`
- Identifier is a sequence of letters, digits and an underscore symbol starting with a letter!
- Function definition is a type name followed by identifier followed by an optional list of function arguments inside the parenthesis (just like function declaration) and then followed by **function body** enclosed in braces
- Function body is just a bunch of valid **statements**
- **Statement** is a smallest standalone element of the language. Here are valid statements of our CUCU language:

```
/* These are simple statements */
i = 2 + 3; /* assignment statement */
my_func(i); /* function call statement */
return i; /* return statement */

/* These are compound statements */
if (x > 0) { .. } else { .. }
while (x > 0) { .. }
```

- **Expression** is a smaller part of the statement. Unlike statements, expressions always return a value. Usually, it's just the arithmetic. For example in the statement `func(x[2], i + j)` the expressions are `x[2]` and `i+j`
  - Expressions are limited to Boolean and arithmetic expressions
  - Boolean expressions are used as tests in **control statements** (if and while)
  - **Relational operators** are: == (equals) and != (not equals)
  - **Arithmetic operators** are: +, - and *, /
  - **Parenthesis** () can be used for grouping **terms**
  - **Precedence Order (High to Low)**
    - Parenthesis
    - * and /

    - + and –
    - == and !=
    - Assignment (=)

- Control statements (if and while) form is as shown in the examples above
- **Comments** are enclosed in /* and */ Do not worry about dealing with nested comments
- And that's it!! **If any aspects of the language are ambiguous or missing, make an assumption and state it clearly in the README file under a section called 'ASSUMPTIONS'**

Lexer Output:

- Use the 'Action' part of the rules (for patterns) to do two things:
    - Print the token type along with it's value in the output file (`Lexer.txt`), one per line.
      Example: `int i = 2 + 3;`
      Output:
      ```
      TYPE : int
      ID : i
      ASSIGN : =
      NUM : 2
      PLUS : +
      NUM : 3
      SEMI : ;
      ```
    - Return the token type and it's value to Yacc (use Lex's predefined variables like yytext, yylval to accomplish this)
    - Any lexical errors should be reported using yyerror and displayed in the `Lexer.txt` file

Parser Output:

- Use the 'Action' part of the grammar rules to print to an output file (`Parser.txt`)
- Sample format of the `Parser.txt` is shown in the example below
    - You don't have to replicate this format
    - You can come up with your format for the parser output file that lists all the terminals/non-terminals/expressions and statements that it parses successfully from the source code!
    - Any syntactical errors should be reported using yyerror and displayed in the `Parser.txt` file

Compiling and Running:

- Given the following source code in file called `sample.cu`

```
int main(int argc, char **argv) {
    int i = 2 + 3;
    char *s;
    func(i+2, i == 2 + 2, s[i+2]);
    return i & 34 + 2;
}
```

- and compiled using the cucu compiler as follows: `./cucu sample.cu`

- should generate a **`Lexer.txt`** file with above mentioned format and **`parser.txt`** file with format **similar** to the following:

```
identifier: main
function argument: argc
function argument: argv
function body
local variable: i
 const-2  const-3  +  :=
local variable: s
 var-func  var-i  const-2  +  FUNC-ARG
 var-i  const-2  const-2  +  ==  FUNC-ARG
 var-s  var-i  const-2  +  []  FUNC-ARG
 FUNC-CALL
 var-i  const-34  const-2  +  AND RET
```

- **Submitting your work:**
  - All source files and class files as one tar-gzipped archive.
    - When unzipped, it should create a directory with your ID. Example: **2008CS1001** (NO OTHER FORMAT IS ACCEPTABLE!!! Case sensitive!!!)
    - Should include: `cucu.l` and `cucu.y` and README file
    - Should also include `Sample1.cu` and `Sample2.cu` files with sample CUCU code with correct syntax and incorrect syntax respectively to demonstrate that your compiler works on the specified language and can recognize errors.
    - ***Negative marks if the TA has to manually change this to run his/her scripts!!***
  - ***Negative marks for any problems/errors in running your programs***
  - Submit/Upload it to Google Classroom
- ***WARNING: Remember, PLAGIARISM will get you an "F" for the whole course!!!***