Assignment II - Syntax Analyzer

Compilers II (CS3423)

Deadline: 19th September 2023

1 Objective

This programming assignment requires you to generate a parser for a C-like language. This includes devising a grammar to describe the language's syntax. The end result will be a program that reads the source file and produces a `.parsed` file (explained in Section 4.1). You may need to get help from your first assignment, too.

The details of the input program format are provided in Section 3. The deliverables of this assignment are mentioned in Section 4, and the grading pattern is mentioned in Section 6.

2 Parser

The job of your parser is to parse a `.clike` source file. Your parser should require the appropriate syntax for the kind of file it is parsing. It should output a `.parsed` for each given input file. Your parser must be implemented using an **LALR(1)** parser generator.

Your compiler should behave as follows:

1. If there is a lexical or syntax error within the source, the compiler should indicate this by printing to standard output an error message that includes the position of the error. Please follow section 3 for a more detailed explanation.

2. If the program is syntactically valid, the compiler should terminate normally exit (code 0) without generating any standard output.

3 Sample Program and Output

Consider a sample program in Figure 1. We outline the fundamental principles of the program in the subsequent points. A program in this C-like language can have any number of methods and class definitions in any arbitrary order but must not have any statements outside of a class or a method.

1. `method or function definitions:`

   (a) A method definition always starts with the scope type. Two types of scope are possible: (1) `global`, and (2) `local`.

   (b) `return_type` of the function always follows `scope`, and the valid `return_type` can be either a pre-defined data type or a user-defined data type. The pre-defined data types in the given language are: { `int, char, string, bool, void` }.

(c) `function_name` is followed by the number of arguments passed to the function. The number of arguments passed will always be mentioned inside square braces. If a function has no arguments, the number of arguments will be empty (**not** zero). *example:* `local int bar[1](int x)` and `local int bar()` both are valid function definition statements.

(d) A function body always contains at least one return statement.

2. `Class definition:`

   (a) A class definition always starts with the `class` keyword and followed by `class_name`.

   (b) The number of methods defined in the class are specified in squared braces after the class name. Note that if no method is defined inside the class scope, then the number of methods will be empty (**not** zero). *Example:* `class class_foo[1]` and `class class_bar` both are valid syntax.

3. `Statement:`

   (a) `Declaration:` A variable or object declaration can be made using a declaration statement. `datatype` of the variable or object will follow the `declare` keyword. Note that the variable's initialization cannot be done in a declaration statement. *example:* `declare int v;` is valid, but `declare int v = 0;` is not.

   (b) `Expression:` A variable or object can be initialized or assigned using an `expression` statement. Initialization or the assignment will always be preceded by `expr` keyword. Note that `expression` statement must contain equal-to (=) operator and may contain binary and unary operators, but must not contain any `datatypes` for typecasting.

   (a) `Call Statement`

      i. A function/method call will always start with `call` keyword. If the callee method (the target method of the call statement) is defined within a class, the object name or `this` keyword is required to call the method. The syntax for the same are: `call object name` → `methodname[number_of_arguments](...);`, and `call this` → `methodname[number_of_arguments](...);` Note that the arrow operator is a must in this statement. Also, note that `objectname` can be `'this'` keyword when both the caller and callee methods are in the same class.

      ii. If a callee method is defined outside the class, the `objectname` is not required during the call statement. The valid syntax for this case is: `call methodname [number_of_arguments](...)`. Note that if no arguments need to be passed to a method, the `number_of_arguments` field can be left empty (**not** zero).

   (b) `Conditional Statement:` Conditional statements are defined using the following syntax: `in case that(predicate) do {stmt} otherwise {stmt}`. Note that `in case that` statement will always be followed by a `do` keyword. However, having a `do` keyword following `otherwise` will result in an invalid syntax.

   (c) `Loops:` Two types of loops are possible in the given language: `for` and `while`.

      i. `while:` `while` statement will always start with a `loop` keyword and is followed by a `do` keyword. However, `for` statement does not need any such keywords. *example:* `loop while(p lt 3) do {...}`.

ii. `for:` Unlike `while`, `for` takes three arguments separated by semicolons, (1) `expression`, (2) `predicate`, and (3) `unary operator`. However, `unary operator` is optional. *example:* `for(expr v = 0; v gt 3; postincr(v)){...}` are valid syntax. Similarly, `for(expr v = 0; v gt 3;){...}` is also a valid syntax.

(d) `Return Statement:` The return statement ends the function execution and specifies a value to be returned to the function caller. Note that each `method` must have at least one `return` statement and can occur anywhere within the function. If a `return` statement does not occur inside a function, you should print `invalid statement` after you parse the complete function.

4. `RHS of an Expression:` The RHS of an expression may contain (1) a variable, (2) a constant, (3) `binary operation`, and (4) `unary operation`

   (a) The binary operation will have the following syntax: `operator(`v_1, v_2`)`. The valid binary operators are as follows: {`add, sub, mul, div`}. Note that the binary operators can never be used as a variable name.

   (b) The unary operator also has a similar syntax to the binary operator, except that the unary operator take only one variable as argument. The valid unary operators are as follows: {`postincr, postdecr`}.

   (c) The binary operators can be called only at the RHS of an `expression`, however the `unary` operators can be called on their own. Moreover, binary and unary operators can never be used in a `declaration` statement. *example:* `expr v = mul(2, add(2,c))`, `expr v = postdecr(c)`, and `postincr(2)` are examples of valid syntax.

5. `Predicate:` A predicate is any unary, binary, logical, or arithmetical operation that returns true or false. The valid keywords that are used for logical operations in the predicate are {`lt, gt, geq, leq, ne, eq, and neg`}. The C++ equivalent for the mentioned keywords are {`<, >, >=, <=, !=, ==, and !`}. Remember that, nested operations are also allowed in the predicate. Moreover, the predicates can also contain the method call statements.

Note: Each statement (except conditional statements and loops) should end with a **semicolon**.

4 Assignment Submission Guidelines

4.1 Files to be Submitted

You should submit the following items:

1. `overview.pdf`: Your overview document for the assignment. This file should contain

   (a) your name

   (b) your college ID

   (c) end-to-end compilation steps of your analyzer

   (d) clear instructions on how to run your analyzer on an input program

   (e) all known issues you have with your implementation

```
global void foo()
{
    declare int x;
    declare int y;
    expr x = 8;
    declare int i;
    loop while(i lt x)
    do
    {
        expr y = i;
        postincr(i);
    }
}
class class_bar[1]
{
    declare int x;
    local int bar[1](int x)
    {
        postincr(x);
        in case that((x gt 0) and (x geq 3))
        do
        {
            expr x = 3;
        }
        otherwise
        {
            expr x = 4;
        }
        return x;
    }
}
global void main()
{
    declare class_bar obj;
    call obj->bar[1](20);
    return void;
}
```

Figure 1: An example format of the input programming language (`example.clike`)

2. `yacc Source Program`, `y.tab.c`, `Lex Source Program`, and `lex.yy.c`: You should include everything required to compile and run the project. Please ensure that the directory structure of your source files is maintained within the archive so that your code can be compiled upon extraction. Your build process must **not** download anything from the internet.

3. `Sequence of tokens`: You should also provide the sequence of tokens generated by your lexical analyzer on the public test cases in a file `seq_tokens_i.txt` (`i` is the test case number) and submit the same. The file format of `seq_tokens_i.txt` is the same as in Programming Assignment I.

   If you discover a problem with your lexer (from Assignment I), you must devise one or more test cases that clearly expose the bug. After you have done this and confirmed that your Assignment 1 implementation indeed fails these tests, fix the bug. Discuss these tests in your overview document, and explain the problem briefly.

   After running it on the input files, the file should contain first **your name** and **college ID**

and the corresponding output from the lexical analyzer. The output data will contain multiple lines, where each line first describes the nature of the token and then followed by a colon and the token itself. The nature of the tokens is the same as that of Assignment I. Violating the format may lead to the exclusion of the same during evaluation.

4. `Parsed File:` A `parser_i.parsed` file should be provided for each of the input `i.clike` program file (where `i` is the input test case number). The `parser_i.parsed` file should contain multiple lines, where each line prints the program line number and the corresponding grammar rule (in `small case`) for that. *Example:* Consider an example program in the Figure 1. Given the input program to your parser, the output file should consist of the following lines:

```
global void foo()[space]:[space]function definition
{
declare int x;[space]:[space]declaration statement
declare int y;[space]:[space]declaration statement
expr x = 8;[space]:[space]expression statement
...
}
invalid statement
```

In case of an invalid statement, the parser should print the statement and print it as invalid. *example*: `declare int x = 0; : invalid statement`. Note that, till the invalid statement is encountered, the parser should print all the statements and corresponding rules in the `.parsed` file. An auto checker will check your `.parsed` file. Hence, please strictly adhere to the submission format of the output files. The parsed file should strictly identify only the following types: {`function definition, class definition, declaration statement, expression statement, call statement, call statement with object, loop, conditional statement, return statement`}

4.2 Assignment Folder Format

1. The project directory must be named XXXXXXXX (your ID).
2. The project directory must contain subdirectory named PY, where Y is the homework number.
3. The directory PY should contain only source files, a TPY folder, and a TPP folder. No binary files should be present in the directory.
4. The directory TPY should contain an overview file, file called `seq_tokens_i.txt`, and `parser_i.parsed` file (`i` is the input file number). We may not evaluate an assignment if these files are not present.
5. The TPP sub-directory should contain the public test cases (to be provided by you) and corresponding output files. For further details please check section 5.
6. Tar the XXXXXXXX directory along with the contents and then gzip it to **XXXXXXXX.PY.tar.gz**. Submit the XXXXXXXX.PY.tar.gz file.

5 Test Case Submission Guidelines

`Public Test Cases`: You should provide exactly **three** test cases with programs that follow the basic construct of the program defined in Figure 1. You may add extra features to the program but

cannot delete or modify any existing features. If you are adding extra features, do clearly mention that in **overview.pdf** file. Your parser will be analyzed on your public test cases first, and then on private test cases.

1. The test cases directory must be named XXXXXXXX (your ID).
2. The test case directory must contain only a subdirectory named TPP.
3. The TPP sub-directory should contain the public test cases (to be provided by you) and a sub-directory named TPPO. The test cases should be renamed as `public_test_i.clike`. TPPO should contain the output files of the corresponding public test cases. The corresponding output files should be renamed as `pt_seq_tokens_i.txt` and `pt_parser_i.parsed`.
4. Tar the XXXXXXXX directory along with the contents and then gzip it to XXXXXXXX.TY.tar.gz. Submit the XXXXXXXX.TY.tar.gz file.

6 Grading

Solutions will be graded on implementation efforts, correctness, and proper documentation. A **correct** program compiles without errors or warnings and behaves according to the requirements given. Your lexical analyzer will be tested against multiple private test cases (along with the public test cases) to verify its correctness; failure to do this may result in a lower grade on your assignment. Documentation includes the overview document and comments in the code.

The total 100 marks are distributed for each section in the following manner:

1. `Contribution of public test cases:` 5 marks
2. `Correctness on testcases:` 70 marks (5 each: 1.5 for token gen and 3.5 for parsed file gen)
3. `Proper Documentation:` 25 marks

7 Late Submission Policy

1. We expect that all the students will meet the given deadline.
2. A delayed submission will automatically incur a 20% penalty for each day of delay.
3. A student may make as many submissions (before the given deadline or delayed submission) as he/she wants. But we will use only the last submitted one for evaluation.

8 Plagiarism and GPT Policy

1. Plagiarism in any form will not be tolerated.
2. We will not differentiate between the source (giver) and the sink (taker). We will consider both parties to have plagiarised.
3. Any student suspected of plagiarism for an assignment will get a zero on that assignment (may lead to FR grade also) and will be reported to the institute disciplinary committee.
4. Copying directly from the GPT tool will not be tolerated. You must not rephrase your answers using the GPT tool also.