COMP 321: Design of programming languages
Homework 3: Lexer and parser specifications for a fragment of C

---

1. Written problems

There are no written problems for this assignment.

2. Coding problems (20 points)

For this assignment, you will write specifications for a lexical analyzer and a parser for a fragment of the C programming language. C is an imperative language. A program consists of a sequence of definitions, the most common of which is a function definition. A function definition in turn consists primarily of a sequence of statements such as variable declarations, conditionals, and loops. If you are familiar with any imperative language, then you will recognize most of the constructs that you must handle for this assignment.

You must implement only `ml-ulex` and `ml-antlr` specifications for the language, along with the abstract syntax type definition. The language you must handle is the CPP language from the *Implementing Programming Languages* website (you will write a type-checker, interpreter, and compiler for this language in later assignments). The language is described in the "Type Checker for CPP" project on the *IPL* website (note that we are *not* implementing the fragment of C++ that is used in the "Parser for C++" project).

Some comments on some of the tokens you will need to recognize in the lexer:

- `int` literals consist of one or more digits optionally preceded by `~` (for negative literals).
- `double` literals consist of one or more digits followed by `.` followed by one or more digits, optionally preceded by `~` (for negative literals) and optionally followed by `e`n where n is an integer literal (for scientific notation).
- `string` literals consist of any sequence of printable characters delimited by the double-quote character. If you want to get fancy, allow strings that have escaped double-quote characters of the form `\"`.
- Identifiers consist of sequences of alphabetic characters, digits, and `_`, not starting with a digit.
- There are two kinds of comments:
  - *End-of-line comments* start with `//` and continue to the end of the line.
  - *Block comments* are delimited by `/*` and $*/$. They may extend over multiple lines.

  Comments should be skipped by the lexer. You need not handle nested block comments.
- A preprocessor directive is any line that starts with `#`. Treat preprocessor directives like comments.

Some comments on the grammar and abstract syntax:

- You must define a structure named `Ast` with types `exp` (for expressions) and `program` (for programs), and functions for converting each to strings. You will almost certainly add many more types corresponding to other syntactic categories in your grammar.
- A *program* is a sequence of definitions (also possibly comments and preprocessor directives, but those should have been filtered out by your lexer). The first non-terminal of your grammar specification must parse a program and yield a value of type `Ast.program`.

- A *definition* is a function definition; function prototype; variable declaration; or a variable initialization.
- A *function definition* consists of a type, name, parameter list enclosed in parentheses, and a function body that is enclosed in left- and right-braces.
- A *function declaration* consists of a type, name, and either a parameter list or a prototype list enclosed in parentheses. A function declaration is ended with a semicolon.
- A *variable declaration* consists of a type and one or more variable names separated by commas, ending with a semicolon.
- A *variable initialization* consists of a type and one or more bindings separated by commas, ending with a semicolon. A *binding* consists of a variable followed by `=` followed by an expression.
- A *parameter list* is a comma-separated sequence of parameter declarations. A *parameter declaration* consists of a type and a variable name.
- A *prototype list* is a comma-separated sequence of types.
- A *function body* is a sequence of statements.
- A *statement* is as described on the "Type Checker for CPP" project webpage. Ignore the comments about typing. Include the following statement forms:
  - Do-while loops, which have the test expression after the body of the loop, as in

    ```
    do ++i ; while (i < 10) ;
    ```

    Note that the body (`++i` in this example) can be any statement.
  - For loops, which have a variable declaration and two expressions in parenthesis, followed by a statement, as in

    ```
    for (int i=0; i != 10; ++i) k = k + i ;
    ```

  - Conditionals without an `else` branch.
- An *expression* is as described on the "Type Checker for CPP" project webpage. Ignore the comments about typing. Make the following changes:
  - Ternary conditional: $e \,?\, e : e$ at precedence 2.5. For intuition, the evaluation if $e \,?\, e_0 : e_1$ is as follows: if e evaluates to `true`, then evaluate to e_0; otherwise evaluate to e_1. The ternary conditional is right associative, which means that, e.g., the expression `a ? b : c ? d : e ? f : g` is parenthesized as `a ? b : (c ? d : (e ? f : g))`.
  - Shift expressions e `<<` e and e `>>` e at precedence level 10, left associative.
  - Mod expressions $e \,\%\, e$ at precedence level 12, left associative.
  - Logical negation expressions `!`e at precedence level 13, right associative.
  - Function call expressions are to be precedence level 14 (same as post-increment/decrement).

  The starting non-terminal in your grammar for parsing an expression must be `exp`, and the yield of the `exp` production must be a value of type `Ast.exp`.
- The types are `int`, `double`, `string`, `bool`, and `void`.

A few more comments are in order:

- C allows both function *definitions* and function *prototypes*. The former has a (return) type, name, parameter list, and body, as in

  ```
  double f(int x, double y) {
      return 0.0 ;
  }
  ```

The latter has a (return) type, name, parameter list or parameter type list, and no body. In the following code fragment, the first line is a function prototype with a parameter list and the second is a prototype with a parameter type list:

```
double f(int x, double y) ;
double g(int, double) ;
```

A function prototype is used to declare the type of a function before defining it. A function prototype with a parameter list is equivalent to a prototype with just the corresponding parameter type list—i.e., when parameters are specified in a prototype, they are ignored. You must parse both kinds of prototypes, but you should have only one kind of metalanguage value for function prototypes with associated data being the return type, name, and parameter types. I found that handling function definitions and function prototypes simultaneously to be a bit tricky. I recommend focusing on just handling function definitions and deal with prototypes last.

- You need not handle mixes of variable declarations and initializations, as in `int x, y, z=7, w ;`. A statement will either be all declarations or all initializations.
- You must handle both `if` and `if−else` statements. The naive approach has a problem with finite lookahead, and the standard solution is to left factor. In my solutions, I notice a couple of issues when I do this. First, `ml−antlr` complains about a left-recursion, but generates the parser anyway. Second, if I use `ml−antlr`'s ? facility in the natural way to handle an optional `else` clause, I again get finite-lookahead errors, and so I have to write the equivalent rule with two productions, one which starts with `else`, and another with an empty sequence.
- Somewhat oddly, when I try parsing just expressions (using the `parseexp` function generated by `ml−antlr`, parses always fail if the expression ends in an identifier. I can fix the problem by adding and additional production that says that an identifier followed by `EOF` yields an identifier, but I cannot explain the problem or the fix. Accordingly, we will only test your expression parsing on expressions that do not end with identifiers.

3. Approach

This project may look large and impenetrable at first. I recommend you take it in very small pieces. Here are some suggestions:

(1) Start by defining the `Ast.exp` type for just some very simple arithmetic expressions: maybe integer literals and sums and products. Make sure that `Ast.expToString` does something reasonable, since the driver program uses that function to print out the results of an attempted parse. Add to the lexer specification the tokens for just that, and then write the grammar specification starting from the `exp` non-terminal (i.e., don't worry about the `pgm` non-terminal). A lot of this you can copy from the sample code. Add a few test files in the `testfiles/expr` directory of the code distribution and make sure you are comfortable both running the test suite and using the driver program. Of course, the test suite will try to parse all the files under `testfiles`, and so you will see several test failures. You want to make sure that the ones you were just starting with pass. Don't forget that you can use the `--lex` option for the driver program to see how your input is tokenized.

(2) Now add identifiers to your language. You will need a constructor for `Ast.exp`, a new clause for `Ast.expToString`, a specification for lexing identifiers, and a production in your grammar to parse them (or add a rule to an existing production). Write some more test files with identifiers. Don't forget that you may or may not run into problems with expressions

in files that end with an identifier (like `3+x`), so you may want to end all expressions with literals.

(3) Now try to add variable declarations like `int x ;` to the language. That means defining the `Ast.pgm` type. I suggest making `Ast.pgm` a list of definitions, where a definition is defined by some `datatype` definition that ought to include some sort of constructor like `DDecl` for a variable declaration. Add appropriate token definitions to your lexer (for example, tokens for the types like `int`). Now start working on the `pgm` non-terminal in your grammar definition. It needs to parse as a sequence of definitions, where a definition is defined by a new non-terminal. And one possibility for a definition is a declaration, which consists of a type followed by an identifier followed by a semicolon. Write some test files in `testfiles/programs` that consist only of declaration statements, each of which only declares a single variable.

(4) Patch up your work on variable declarations to handle one-line declarations of multiple variables like `int x, y, z ;`. Write more test files.

(5) Add function definitions and write more test files. A function definition has a body that consists of statements, so you will also need to define statements. Just define variable declaration statements. This will be almost identical to the work you just did to add variable declaration definitions, except that you will be adding constructors to a new type (some type for statements), and you will be adding rules to the grammar for a different non-terminal (something corresponding to statements). Write some test files with functions that consist of nothing but declarations like

```
int f(int x) {
    int x, y, z;
}
```

(6) Add one of the simpler forms of statements like while loops.

(7) And so on, incrementally adding a feature and testing it before moving on.

Remember that while there are many features that you need to handle, there are really only a few kinds of features. For example, there are many operators for building expressions. But once you have figured out how to go from one precedence level to the next and you have figured out how to handle left- and right-associativity for one operator, handling all the other similar operators will follow that same pattern. So while my grammar specification has about 20 non-terminals, over half of them are for the operators, and almost all of them have the same form, which boils down to "parse something of the next highest precedence, then zero or more things of that next highest precedence, and combine them all together using the correct form of associativity to produce the yield." A few of them are different.

4. Code distribution and submission

I have provided skeleton files for `ast.sml`, `cpp.lex`, and `cpp.grm` on which you should base your solutions. Make sure to read the comments in those files, as there are certain things you must do and certain things you must not change.

I have provided both a driver program and a test suite for your code. The driver program (structure `Driver` in `driver.sml`) defines several parsing functions that print out the parse tree of an expression or program supplied as a string argument, or of a program that is in a file named by the argument. The `make` target is `driver`. Documentation is in the header comment in `driver.sml`.

The test suite attempts to parse a number of files, which are located in the `good` and `bad` subdirectories of `testfiles/programs` (for parsing complete programs) and `testfiles/expr` (for

parsing expressions). The test suite just verifies that the files in `good` successfully parse and the files in `bad` do not. The `make` target is `tests` and the CM build file is `tests.cm`.

As a point of comparison, my lexer specification produces a DFA with about 110 states and my grammar specification about 20 non-terminals (although `ml-antlr` reports about 50 non-terminals, so it must be creating some on the fly).

You will submit `cpp.lex` (your lexer specification), `cpp.grm` (your grammar specification), and `ast.sml` (your `Ast` implementation).