Phase 2: Parsing

In the second phase of our term project, we construct a handwritten predictive parser for the SnuPL/1 language.

The output of the parser is an abstract syntax tree (AST) in textual or graphical form. No semantical checks (i.e., type checking, number of parameter checking, etc) are required in this phase. Syntactical checks must be done (for example., the declaration and end identifiers must match for the module, functions, and procedures).

A skeleton for the parser is provided so that you can focus on the interesting parts. The working example, as before, implements the parser for the "SnuPL/-1" language as defined in the scanner assignment. You may use your own scanner from phase 1, or use our scanner binary for SnuPL/1 provided in the directory scanner/.

The parser skeleton can be found in snupl/src/parser.[h/cpp]. The parser already outputs an AST and contains a type manager and a nested symbol table. We advise you to use the existing code, but of course you can write your own code. In its original form, the parser parses and builds an AST for SnuPL/-1.

In the parser, you will have to modify/code the methods of the predictive parser. For SnuPL/-1, the following methods are implemented:

```
/// @name methods for recursive-descent parsing
/// @{

CAstModule*       module(void);

CAstStatement*    statSequence(CAstScope *s);

CAstStatAssign*   assignment(CAstScope *s);

CAstExpression*   expression(CAstScope *s);
CAstExpression*   simpleexpr(CAstScope *s);
CAstExpression*   term(CAstScope *s);
CAstExpression*   factor(CAstScope *s);

CAstConstant*     number(void);

/// @}
```

The call sequence of the method represents a parse tree. The AST is constructed from the return values of the methods called during the parse. The AST is implemented in snuplc/src/ast.[h/cpp].

In a first step, you may want to simply build a predictive parser that only consumes the tokens but does not build the AST. Once your parser is working correctly, you can then start to return the correct AST nodes in a second step.

The type manager, implemented in snuplc/src/type.[h/cpp], does not need to be modified, you can use it to retrieve types for integer, character, and boolean variables and the composite types pointer and array.
Call CTypeManager::Get()→GetInt()/GetChar()/GetBool()/GetPointer()/GetArray() to retrieve a reference to integer, character, boolean, pointer or array types.

The symbol table is implemented in snuplc/src/symbol.[h/cpp]. Again, you do not need to modify this file, the functionality provided will be enough for this phase of the project. Symbol tables are nested, you need to create a new nested symbol table whenever you parse a function/procedure and insert the symbols into the symbol table of the current scope.

A test program that prints the AST is provided. Build and run it as follows:

snuplc $ make test_parser
snuplc $ ./test_parser ../test/parser/test04.mod

In the directory test/parser/ you can find a number of test files for your parser. We advise you to create your own test cases to test special cases; we have our own set of test files to test (and grade) your parser.

A reference implementation can be found in reference/. Compare the output of the reference implementation to your own implementation. Note that also the reference implementation may contain bugs (if you discover a bug, please let us know.)

Materials to submit:

- source code of your compiler (use Doxygen-style comments)
- report describing your implementation of the parser and AST (PDF)

Submission:

- the deadline for the second phase is **Friday, October 20, 2017 at 14:00**.
- email your submission to the TA (compiler@csap.snu.ac.kr). The arrival time of your email counts as the submission time.

As usual: start early, ask often! We are here to help.

Happy coding!