Phase 2: Parsing

In the second phase of our term project, we implement a handwritten predictive parser for the SnuPL/2 language.

The output of the parser is an abstract syntax tree (AST) in textual and graphical form. No semantical checks (type checking, number of parameter checking, etc.) are required in this phase with one exception: symbols must be defined before use. In addition, constraints that cannot be expressed with a context-free grammar but are part of the syntax must be checked. These include, for example, that the declaration and end identifiers must match for the module and subroutines.

A skeleton for the parser is provided so that you can focus on the interesting parts. The working example, as before, implements a 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 recommend to use the existing code, but of course you can write your own class hierarchy. 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 provided functionality is sufficient 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/test01.mod
```

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

This handout does not include a reference implementation. A reference implementation will be provided after the submission deadline.

Materials to submit:

- source code of the scanner (use Doxygen-style comments)
- a brief report describing your implementation of the parser (PDF)
  (the report is almost as important as your code. Make sure to put sufficient effort into it!)

Submission:

- the deadline for the second phase is **Sunday, October 18, 2020 at 14:00**.
- submission instructions will be given later

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

Happy coding!