Compilers

Lab 4

Piotr Błaszyński

22nd March 2022

Tasks (explained later in the document):

- modify the source files and Makefile so that the syntax analyzer and the entire compiler is compiled using the C++ compiler,
- modify the grammar so that a program consisting of multiple expressions (multiple lines) can be compiled,
- add the code that writes the triples to the file.

Modification to allow compilation using C++:

- rename the file def.y to def.yy (so bison will generate us the files def.tab.cc and def.tab.hh),
- in the Makefile:
  - we change references to def.y to def.yy,
  - change references to def.tab.c to def.tab.cc,
  - we add a new variable at the beginning: CXX=g++
  - and replace references to CC with CXX in two places
    - def.tab.cc compilation
    - compilation of the whole thing (we still compile the lexical analyzer using the C compiler),
- add compile option -std=c++11 to Makefile (both places where we use g++),
- to def.yy file add in header section:
  - extern "C" int yylex();

  - extern "C" int yyerror(const char *msg, ...);
  - using namespace std;
  - if we have *yyin* and *yyout* declarations, we also add *extern* to them (without C),

- in the zx.l file we change:
  - at the beginning: we change the included header file (def.tab.h to def.tab.hh),
  - at the beginning: int yyerror(const char *msg, ...); (actually we change the header of the yyerror function - we add const)
  - at the end: int yyerror(const char *msg, ...){; (actually we are changing the header of the yyerror function - we are adding const)

For defining a grammar to handle multiple expressions, we can use the analogy of arithmetic expressions, which can also be arbitrarily long.

As triples we will at this stage consider expressions consisting of the variable *result* (numbered in the future) and two arguments and an operator. For example, for the expression $a = b + c * 9$; we will obtain three triples:

- $result = c\ 9\ *$
- $result = b\ result\ +$
- $result = a\ result\ =$

This is just an example form of notation, different compilers do it in their own ways. To get this form of notation, numbers and identifiers should be put on a stack (this stack should store objects of some structure/class - in addition to the value, the type should also be stored). Whereas in semantic actions:

- for arithmetic operators should pull from the stack the appropriate number of arguments (usually 2),
- write to the stack a variable storing the result,
- all these elements (arguments, operator and result variable) to a file.

Reminder: the $std :: stack$ class has methods:

- *push* - insert a value to the top of the stack,
- *top* - retrieve a value from the top of the stack,
- *pop* - removing a value from the top of the stack.