COP4020 Programming Assignment 2: CALC interpreter/translator
Due: Feb 25 (phase 1), March 4 (phase 2)

Purpose

This project is intended to give you experience in using a parser generator(YACC), writing syntax specification (grammar) for a language, performing parsing and semantics analysis (attribute grammar), and practicing error handling in a compiler.

Project Summary

Your task is to write an interpreter/translator for a simple calculator whose programming language, *CALC*, contains basic language constructs such as variables and assignment statements. The interpreter/translator will be written using a compiler generator (YACC). Your program should perform the functions of both interpreter and translator. The program should call the lexical analyzer (lex) for the next token, parse the token stream, report grammatical errors, perform static and dynamic semantic checks, interpret the program statement-by-statement (including print statements that produce outputs), and translate the CALC program into a working C++ program, called mya.cpp, with equivalent semantic and calculation.

Lexical specification

Table 1 list all tokens in CALC. The set of tokens is a subset of the tokens in PASC in Project 1. Identifiers in CALC are case-insensitive.

Table 1: Token numbers to be returned by yylex

Token	Symbolic name	Token	Symbolic name
;	SEMInumber	,	COMMAnumber
(	LPARENnumber	)	RPARENnumber
integer constant	ICONSTnumber	identifier	IDnumber
begin	BEGINnumber	end	ENDnumber
program	PROGRAMnumber	is	ISnumber
-	MINUSnumber	+	PLUSnumber
*	TIMESnumber	div	DIVnumber
var	VARnumber	print	PRINTnumber
integer	INTnumber	=	EQnumber
end of file	EOFnumber		

Syntax Specification

The syntax of the CALC language is described by a set of syntax diagrams in Figure 1. A syntax diagram is a directed graph with one entry and one exit. Each path through the graph defines an allowable sequence of symbols. For example, the structure of a valid CALC program is defined by the first syntax diagram. The occurrence of the name of another diagram such as declaration and compound statement indicates that any sequence of symbols defined by the other diagram may occur at that point. The following is an example valid CALC program.

```
program xyz is
begin
  var a, b;
```

```
  a = 2;
  b = 3;
  var c;
  c = a + b;
  print c
end
```

Your first task in this assignment is to develop a context free grammar for the CALC language from the syntax diagrams.

YACC specification

Your second task in this assignment is to express your grammar as a YACC specification. You will want to run your specification through YACC to ensure that the grammar produces no parsing conflicts (compiled with "yacc -v"). If conflicts are indicated by YACC, you should alter your grammar to eliminate them without changing the language accepted by your grammar, or ensure that YACC's handling of the conflict agrees with the CALC language specification. In the first phase of the project, you will develop a parser that merely prints out ACCEPT for syntactically correct CALC programs and REJECT with error messages for incorrectly structured CALC programs. Your parser may call your lexical analyzer (yylex) from Project 1 to obtain the next token.

In the second phase of the project, you will extend your YACC grammar by adding attributes and semantic rules with actions to develop an interpreter/translator for CALC that does various static and dynamic semantics checks, performs the calculation specified in the CALC program, and translates the CALC program into a C++ program. Besides reporting grammatical errors, the interpreter/translator also performs the following semantics checks:

- *Duplicated declaration* when a variable is declared multiple times.
- *Undeclared variable* when a variable is used in the program, but not declared or before it is declared.
- *Uninitialized variable* when a variable is referenced before initialized.
- *Divided by 0 error* when the denominator in a division is 0.

The program can exit after reporting one semantic/syntax error. If there is no error, the program should (1) execute each statement and output the result of the expression in each *print* statement (interpreter function), and (2) produce a semantically equivalent C++ program called mya.cpp with equivalent expressions/assignments (translator function).

To facilitate semantic checks and program interpretation and translation, a symbol table must be created to store variables and the related information. The symbol table will need to have at least three fields: the name of the variable, the value of the variable (integer type only in CALC), and a flag indicating whether the variable has been initialized. When the parser encounters an identifier in the declaration, the identifier must be inserted into the symbol table. When the parser encounters an identifier in an expression, it must look up the symbol table to check if the variable has been initialized and obtain its value (if initialized). After the parser processes an assignment statement, the value of the variable in the left hand side of the assignment statement must be updated in the symbol table.

The generated mya.cpp file should have the same number of statements as the source program. Each statement in mya.cpp should have the same expression (may differ only in notation) as the original program. For the example CALC program described, the corresponding C++ program would be similar to the following.

```
#include <iostream>
using namespace std;
```

```
main()
{
  int a, b;
  a = 2;
  b = 3;
  int c;
  c = a + b;
  cout << c << ``\n'';
}
```

Interpreting and translating CALC with YACC is relatively simple. Basically, for every assignment statement, the program first evaluates the value of the expression in the right hand side of the statement. This is similar to the cal_trans example we gave in class. After that, the interpreter updates the value of the variable in the left hand side of the statement. When processing a print statement, the interpreter evaluates the value of the expression and prints the result to the standard output. To facilitate translation, each expression can have an string attribute that stores the string representation of the C++ expression.

Error Handling:

Your parser should print appropriate error messages. You do not have to implement any error recovery. Your program should have similar behavior as the sample executable provided.

Assignment Submission

There are two phases in this project, the first phase is due on Feb. 25, 11:59pm. The second phase (the whole project) is due on March 4, 11:59pm, when you should submit all related files to the blackboard (that allows a 'make' command to produce the executable for the project).

- Phase 1: The program accepts `p1.cal` and rejects `p2_err.cal` (20).
- Phase 2: Recognize correct programs and detect incorrect programs (30).
- Phase 2: Semantic checks and error reporting (20).
- Phase 2: Correct calculation (10).
- Phase 2: Correct translation (15).
- Phase 2: Correct translation with minimal parentheses in expressions (5).
- Extra 10 points for generating correct dynamic divide-by-zero checking code in the target C++ program.
- Extra 3 points for the person who first reports a new error in the sample executable.

Your program will be tested with a series of programs. Some of the testing programs are provided in the project package. If the code cannot handle any input file, 0 point will be given (programs that cannot produce the executable with a 'make' command due to whatever error such as wrong makefile, compiling errors, and missing files automatically get 0 point). If your code can process some programs in the test suite, the grade will be assigned based on the number of programs that your code processes correctly.

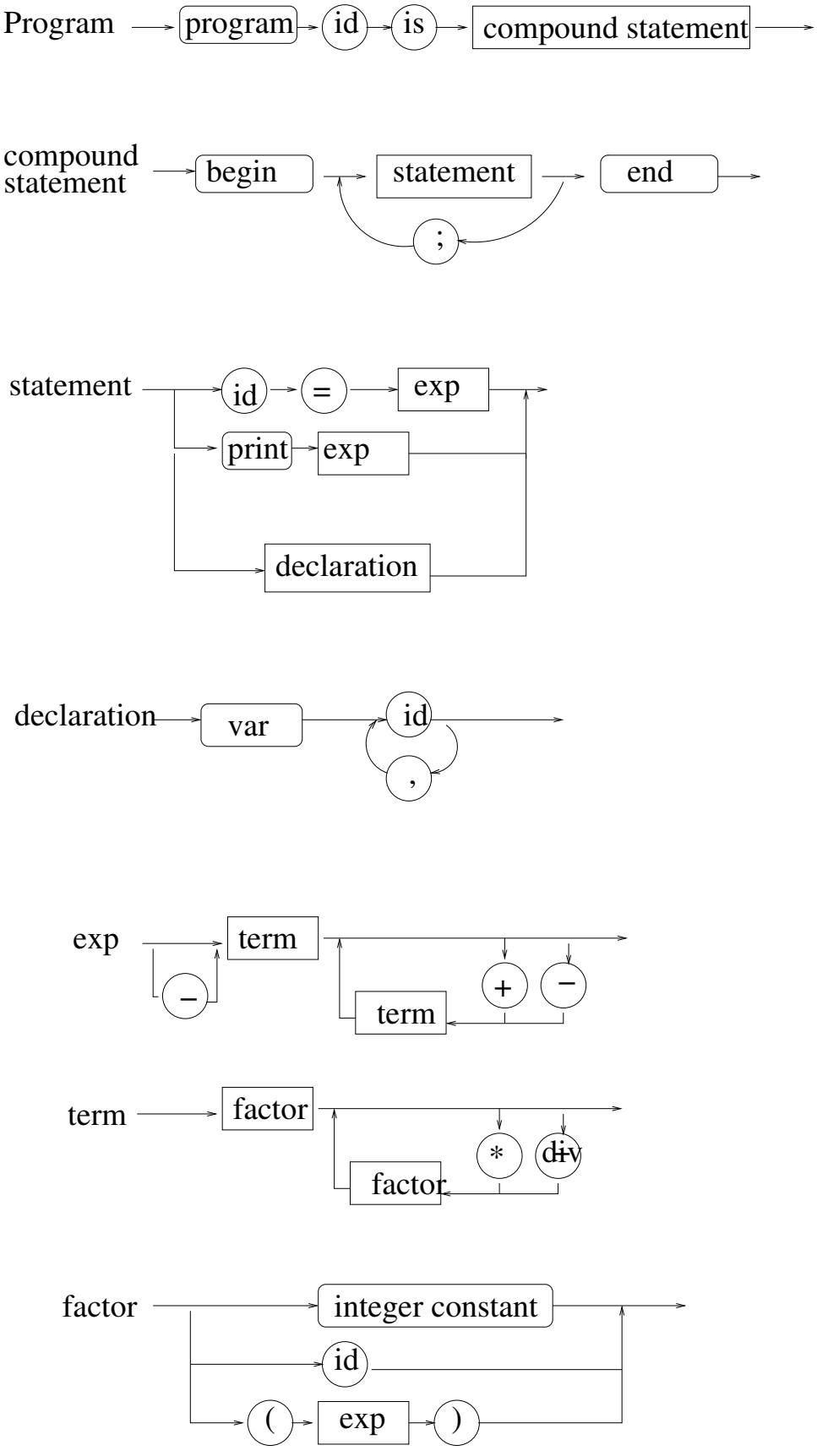

Figure 1: CALC syntax diagrams