CS 352 Spring 2016

Compiler Project 1

Posted Feb. 8, 2016, Due Feb. 19, 2016, 11:59pm

This project is to be done by each student individually. Read the Academic Honesty Policy posted on Piazza carefully.

0. Introduction

We plan four projects in this semester. Projects 1 and 2 use Lex/Yacc to implement (i.e. to interpret) a simple scripting language. Projects 3and 4 use GCC to examine abstract syntax trees and to analyze the dataflow of a program in order to recognize potentially uninitialized variables.

Since these projects progress in the degree of difficulty, they carry different weights. Our of 100, Project 1 counts 10, Project 2 counts 20, Project 3 counts 25, and Project 4 counts 35. We may post Project n before project n-1 is due, so that students can have a preview what is coming up and to start early if they finish project n-1 early.

1. Objective and Scope

This part of our compiler project is to implement a parser for a scripting language, called *miniscript*, which is similar to a subset of JavaScript. (The following page shows an example of miniscript.)

1.1 A Note on Syntax Requirements versus Assumptions about Test Cases

In the project descriptions this semester, when we say "so-and-so is not allowed …" or "so-and-so is illegal …", then your compiler/interpreter must be able to detect any existence of so-and-so, and report it as an error.

If, we say "the input is assumed to contain no so-and-so …", then it means, although so-and-so is not good, we do not require your compiler/interpreter to check for its existence. You can assume that so-and-so does not appear in the test cases.

1.2 Tokens

An identifier starts with a letter (either case), after which it can contain letters (either case), numbers, or underscores. A constant can be either an unsigned integer or a quoted string. An unsigned integer is allowed to have leading zeroes.

1.3 Other Syntax Rules

```
<script type="text/JavaScript">
var two = 2 ; var ten = 10
var linebreak = "<br />"

document.write("two plus ten = ")
var result = two + ten
document.write(result)
document.write(linebreak)

result = ten * ten
document.write("ten * ten = ", result)
document.write(linebreak);

document.write("ten / two = ")
result = ten / two
document.write(result)
var ID; ID = result;
document.write(linebreak)
document.write(ID)

</script>
```

The script may contain arbitrary white spaces, such as spaces, tabs, and new lines. Unless spaces and tabs are inside a quoted string, they have no use in the program other than separating tokens. We assume that no quotation marks appear inside a quoted string. Line breaks are not allowed in the middle of a quoted string. Unlike spaces and tabs, newlines have significance in the grammar, as stated below.

The first line and the last line in the example shown above are called the opening tag and the closing tag, respectively. The opening tag must be the first line of the script, but newlines are allowed to follow the closing tag. Between the opening and the closing tags, there are zero or more lines. Each line contains zero or more statements. *A statement is not allowed to be split across multiple lines.* For example, the following statement is considered a syntax error:

```
var result =
two + ten;
```

Different statements are separated by either a *single* semicolon or by one or more newlines (i.e. \n). (Of course, a newline may or may not follow some spaces and tabs.) A statement ending with a semicolon may also be followed by newlines. A statement cannot be followed by two or more semicolons, and a semicolon must follow a statement in the same line.

We reemphasize that the newlines have significance in the grammar rules, unlike the grammar examples in the lectures. The students need to carefully take this into consideration to avoid writing an ambiguous grammar.

We consider only three kinds of statements as shown in the example above:

(1) Document write statements, in the exact form of *document.write(<param_list>)* where <param_list> represents zero or more parameters. Different parameters are separated by commas. Each parameter is either a quoted string, which never contains a linebreak (not to be confused by the identifier "linebreak" in the example), or an arithmetic expression. We only consider arithmetic operators +, -, *, /. Arbitrarily complex expressions are allowed (multiple operations, parentheses nested to arbitrary depth, etc). (We have studied production rules for expressions in class, at least

partially. The reference book also has example grammar.) A single variable or a constant is also viewed as an expression. (For this project, we do not define and check type rules, e.g. whether 1 + “abc” violates any type rule. We just recognize it as an arithmetic expression.)

(2) Assignment statements, with the left-hand side to be an identifier and the right-hand side to be an arithmetic expression described in (1)

3) Declaration statements in the form of *var ID* or in the form of *var ID = <expr>* where <expr> an arithmetic expression described above.

2. Project Requirement

You are to write a lex/flex program and a yacc/bison program to parse a *miniscript* program defined above, with no parsing conflicts reported by the yacc/bison tool. Do not use any precedence defining instructions such as %left to resolve parsing conflicts. **Do not add any output statements to the semantic actions in the yacc/bison program. Your code is expected to produce no output on correct programs, and the YACC-generated parser will automatically print “Syntax errors” for incorrect input.** (Your temporary version may have additional printing statements added to yyerror() to generate more information for your own debugging purpose. Remove those print statements from the submitted copy to avoid penalty.)

Submission instruction

1) No offline submission (such as email) is accepted.
2) Use the following command at CS lab machines, e.g. the XINU machines (i.e., xinu01.cs ~ xinu20.cs) to submit your homework.
turnin –c cs352 –p p1 [your working directory]

(Your home directories are shared amongst all CS lab machines.)

3) You MUST provide ‘Makefile’ for every programming question.
a. For example, your ‘Makefile’ of this assignment may look like

```
  parser:y.tab.c lex.yy.c
    gcc y.tab.c lex.yy.c -o parser -lfl
  y.tab.c : parser.y
    bison -y -d -g --verbose parser.y
  lex.yy.c:parser.l
    lex parser.l
 clean:
   rm -f lex.yy.c y.tab.c
```

NOTE that in the Makefile, a “tab” is required as the first character on each command line, e.g. “<tab> lex paser.l”. Otherwise you will see an error saying “missing separator”.

4) Your program MUST compile and run without any error at CS lab's Linux machines. Please make sure your Makefile and program runs properly on such machines.
5) Make the final executable program's name 'parser', as shown in the sample Makefile listed above. TA will run your program by executing:

```
> ./parser program_name
```

Where "program_name" is the file name containing the input program.

NOTE: Deviation from the above requirement will get a 10 point of penalty. (For example, if your code receives 90 points, the final score on the Blackboard for this assignment will be 80 points.)

You may find information on the following links posted on piazza to be useful, including the following:
http://ds9a.nl/lex-yacc/cvs/lex-yacc-howto.html

3. Discussion Online

Students are encouraged to share and discuss test cases on our course's piazza site. However, no part of solution is allowed to be posted. The teaching staff will diligently check submissions against each other and against submissions from previous semesters (if applicable).