CS 345 - Programming Languages
Fall 2010

Homework #1

Due: 2pm CDT (in class), September 21, 2010

YOUR NAME: _____

Collaboration policy

No collaboration is permitted on this assignment. Any cheating (*e.g.*, submitting another person's work as your own, or permitting your work to be copied) will automatically result in a failing grade. The Computer Science Department Code of Conduct can be found at `http://www.cs.utexas.edu/academics/conduct/`.

Late submission policy

This homework is due at the **beginning of class** on **September 21**. All late submissions will be subject to the following policy.

You start the semester with a credit of 3 late days. For the purpose of counting late days, a "day" is 24 hours starting at 2pm on the assignment's due date. Partial days are rounded up to the next full day. You are free to divide your late days among the take-home assignments any way you want: submit four assignments 1 day late, submit one assignment 3 days late, *etc.* After your 3 days are used up, no late submissions will be accepted and you will automatically receive 0 points for each late assignment.

You may submit late assignments to Vitaly Shmatikov (CSA 1.114—slide under the door if the office is locked). **If you are submitting late, please indicate how many late days you are using.**

Write the number of late days you are using: _____

Homework #1 (35 points)

Problem 1 (4 points)

Describe some design tradeoffs between efficiency and safety or reliability in any programming language you know.

Problem 2 (4 points)

By hand, using a top-down recursive descent approach, derive the complete parse tree for the simple C program below, showing each non-terminal interior node and each terminal leaf note.

```
int main() { int x; if (x == 0) return 1; else return 2*x; }
```

Problem 3

The purpose of this problem is to introduce you to lexical and syntactic analysis, and to help you gain some experience with Flex, Bison, and C. You will start with a simple integer calculator. Its Flex and Bison specifications can be found here, in the files called, respectively, `icalc.l` and `icalc.y`:

`http://www.cs.utexas.edu/~shmat/courses/cs345/icalc.tar.gz`

Your goal is to extend the calculator by adding new lexical specifications and BNF grammar rules for correctly evaluating new types of expressions.

The operators and their associativity rules are given below, listed from lowest to highest precedence. Parentheses "( )" can be used to override the default precedence rules.

Operators	Category	Associativity
$=, +=, -=, *=, /=, \%=$	binary assignment	right-to-left
$? :$	ternary conditional	right-to-left
$\|\|$	binary logical OR	left-to-right
&&	binary logical AND	left-to-right
$<, >, <=, >=$	binary relational	left-to-right
$==, !=$	binary equality/inequality	left-to-right
$+, -$	binary additive	left-to-right
$*, /, \%$	binary multiplicative	left-to-right
^	binary exponentiation	right-to-left
$-, +$	unary minus/plus	right-to-left
$++, --$	postfix increment/decrement	left-to-right
$++, --$	prefix increment/decrement	right-to-left

Submission instructions

1. Submit a paper printout of your Flex and Bison code, stapled to the first page of this homework (the one showing your name and the number of late days you are using, if any). The printout **must** be processed using the following command:

   ```
   enscript -C -2Gr -Ec <yourfile> -o <outputfile.ps>
   ```

2. Submit your source code electronically using the following command:

   ```
   turnin --submit austin hw2 <filename1> <filename2> ...
   ```

Problem 3a (6 points)

Add twenty-six 32-bit registers, labeled a through z. Users should be able to store values in registers and use them in subsequent expressions in the calculator. A register variable by itself evaluates to the value currently stored in the register. Registers should all be initialized to 0 when the calculator starts.

You should also implement assignment expressions of the form `reg = expr`, `reg += expr`, `reg -= expr`, `reg *= expr`, `reg /= expr`, and `reg %= expr`. Each assignment expression evaluates to the value of the expression on the right-hand side, updates the register on the left-hand side, and prints the result. Note the **associativity** of the assignment operators (see the table above).

```
icalc: b
0
icalc: b=3
3
icalc: b *= 5 + a
15
icalc: a = b = c = 7
7
icalc: z += a*b + c
56
```

Problem 3b (4 points)

Add C-style unary increment and decrement operators `++` and `--`, ensuring that they can only be applied to registers (*i.e.*, `++2` is a syntax error). Pay attention to the precedence of these operators vs. binary operators. These operators should have the same semantics as in C.

```
icalc: x=1
1
icalc: x++
1
icalc: x
2
icalc: ++x
3
icalc: ---x
syntax error
icalc: -(--x)
-2
```

Problem 3c (3 points)

Implement built-in constants `MAXINT` and `MININT`, assuming signed two's-complement 32-bit integers.

```
icalc: MAXINT
2147483647
```

```
icalc: MININT
-2147483648
```

Problem 3d (4 points)

Implement a binary exponentiation operator ^ that raises the first argument to the power of the second argument (note the **associativity** of this operator in the table above).

Feel free to look at the standard C function `pow(x,y)`, as defined in `math.h`. It computes `x^y` up to `MAXINT` and `-x^y` up to `MININT`, and `pow(x,0)=1` for all `x`.

```
icalc: 2^5
32
icalc: 2^0
1
icalc: -2^31
-2147483648
icalc: 2^31
2147483647
icalc: 2^100
2147483647
```

Problem 3e (4 points)

Implement built-in functions `abs`, `min` and `max` as prefix operators which take a comma-separated, parenthesized list of arguments that can be constants, registers, or expressions.

```
icalc: abs(-5)
5
icalc: a
0
icalc: x=abs(a-2*3)
6
icalc: min(abs(-2),min(5,1))
1
icalc: max(x,2+2)
6
```

Problem 3f (6 points)

Implement relational operators `==, !=, <, <=, >, >=`, logical operators `||` and `&&`, and the ternary conditional expression operator `?:` Their semantics should be the same as in C.