ASSIGNMENT 2: Java program
Due: January 30, 2018

Overview

You will be given a handout for a previous class's assignment, which describes the E programming language and its implementation. You will also be given a student's solution to that problem. You are to modify that solution.

The code you will be given works, but it isn't necessarily written as clearly as you might prefer. Also, some design decisions made were appropriate for the previous project, but might have been made differently if the designer knew then the requirements for the current project. Both of these situations are what you are likely to encounter in large software projects, e.g., in a real job.

Parts in the previous assignment were numbered 1-6. To avoid confusion with those parts, parts in this class's assignment are numbered starting with Part 10.

Before reading the rest of this handout, read the previous assignment to get an idea of what was required. Below outlines the differences and the individual parts for this year's assignments.

The changes below will require you to modify parts of the scanner, parser, symtab (semantic checks), and code generator.

Part 10: Previous Solution

Get the previous solution (on the class webpage). Look over the given code and tests. Build *e2c*, run it, and test it to get familiar with how to do so and what the output looks like.

Part 11: skip statement

Add a new statement, **skip.** It has no effect. The modified syntax is:

> statement ::= assignment | print | skip | if | do | fa
> skip ::= **skip**

This statement is sometimes useful, say, for the programmer to indicate explicitly that s/he has considered a case of an if statement and no action is needed, e.g.,

> if i /= 10 → x := x+1
> else → skip
> fi

Of course, the entire "else skip" could be omitted, but some programmers prefer including it. And, adding this statement is a great warm-up part for this project.

Part 12: stop Statement

Add a new statement, **stop.** It simply stops the program. (Hint: generate the C code "exit(0)". To use exit, your generated C code will need to "#include <stdlib.h>") The modified syntax is:

> statement ::= assignment | print | skip | stop | if | do | fa
> stop ::= **stop**

Also, give a translation-time warning message if any statements follow the stop statement within the same block (i.e., statement_list). Those statements will never be reached, but just give the warning and process them as usual. (Hint: don't change the grammar, but add some checking code within the parser.)

Part 13: Remainder Operator

Change the syntax of multop to:

multop ::= '*' | '/' | '%'

This new, remainder operation has a meaning identical to that in C.

Part 14: Modulo Operator

Change the syntax of factor to:

factor ::= '(' expression ')' | id | number | predef
predef ::= **mod** '(' expression ',' expression ')'

First, note that **mod** appears as a predefined function rather than as an operator (i.e., more like a prefix operator than an infix operator) to ease your implementation. (That might not be apparent until after you actually finish this part; think about it.) Also, for the same reason, **mod** is a reserved word (keyword).

The modulo and remainder operators are related in that they both return a result that represents the remainder from the division of their two operands. However, they differ in exactly what they return. The sign of *mod(a,b)* is that of *b*, while the sign of *a%b* depends on how integer division is performed on the underlying machine. For example, consider a circular buffer of size *n* with slots numbered 0, 1, ..., *n-1*. The two slots adjacent to slot *i* have indices *mod(i+1,n)* and *mod(i-1,n)*. When *i* is zero, the value of this last expression is *n-1*, whereas the value of *(i-1) % n* can be either *n-1* or *-1*. Accordingly, the modulo operator is generally more useful than the remainder operator.

Also, if *b* is 0 in *mod(a,b)*, then, during E program execution, your generated C code should print an error message (to stdout) and terminate E program execution without attempting to further evaluate the expression, which would result in an error from the C code (e.g., that shows up on our systems as floating exception or Trace/BPT errors).

Hint: You'll need to generate a few lines of C code for the E expression *mod(a,b)*. The modulo operator can appear within arbitrary expressions, so whatever code you generate must be valid within a C expression, i.e., it cannot contain multiple expressions (separated by semicolons). So, the easiest way is to generate (once before the C main function) a C function that performs the check for zero. If the check is okay, it returns the appropriate value; otherwise, it prints the error message and exits the program.[1] The code generated for *mod(a,b)* then just invokes that function.

Write your own modolo function that considers the different possible combinations of signs of its arguments described above. Do *not* use any C or other library function (e.g., *fmod*).

[1] To exit the C program, invoke the C library function *exit* with a status of 1. See "man 3 exit" for details; be sure to *#include <stdlib.h>* in the C code you generate.

Part 15: Maximum function

Change the syntax of predef to also allow a **max** expression:

predef ::= **mod** '(' expression ',' expression ')' | **max** '(' expression ',' expression ')'

The value of the **max** expression is the maximum of all the values of its expressions.

Note that, for simplicity, **max** takes exactly two arguments.

To make this part more interesting (and different from the previous part), the code you generate must not use any procedure call. One way to do so is to generate a *max* macro (once before the C main function) and use that. The *max* macro can use C's exp?exp:exp to compute the maximum of its arguments.

Also, to make this part slightly more interesting, **max** expressions may be nested at most 5 deep. If they are nested more deeply, your translator is to print an error and exit.

Part 16: break Statement

Add a new statement, **break.** It breaks out of the nearest enclosing loop (just like C's **break**, but see the next part). The modified syntax is:

> statement ::= assignment | print | skip | stop | if | do | fa | break
>
> break ::= **break**

Also, give a translation-time warning message and ignore the break statement if it appears outside a loop. Also, for a break statement that appears within a loop, give a translation-time warning message if any statements follow the break statement within the same block (i.e., statement_list). Those statements will never be reached, but just give the warning and process them as usual. (Hint: don't change the grammar, but add some checking code within the parser.)

Part 17: break Statement revisited

Revise the break statement so that it takes an optional integer specifying the number of levels of nested loops out of which to break. The modified syntax is:

> break ::= **break** [number]

Also, give a translation-time warning message and ignore the break statement if the specified number is 0 or exceeds the current loop nesting depth.

A multi-level break statement is useful, e.g., for searching a two-dimensional array for the position of its first zero to break out of the inner loop. However, our E doesn't have arrays, so here's pseudocode in an *extended* (beyond the scope of this HW!) E:

```
fa r := 1 to n →
  fa c := 1 to n →
    if a[r,c] = 0 → break 2  fi
  af
af
print r  print c
```

Hint: use C's **goto** statement in your generated code. Further hint: most modern C compilers require a statement label to appear on an actual statement; to ensure that, you might find it easiest to always generate an empty statement following a statement label (e.g., 'label:;').

To convert String s to its integer equivalent and store the result in int num, use

```
num = Integer.parseInt(s);
```

Part 18: dump Statement

Add a new statement, **dump.**

> statement ::= assignment | print | skip | stop | if | do | fa | break | dump
>
> dump ::= **dump**

The **dump** statement prints the current values of all currently active variables, i.e., those variables that are accessible in the program where the **dump** statement appears.

Print the variables in *order* of their declaration. For each variable, give, in this order, its value, line number on which it was declared, level in which it was declared, and its name. To match the 'correct' output, use *printf* format

> "%12d %3d %3d %s\n"

Part 19: dump Statement extended

Extend the **dump** statement so that it optionally specifies the level of variables to be printed. The new rule is

> dump ::= **dump** [number]

The number gives the block nesting level; if specified, only variables currently active at that level number are printed. Use the same output format as in Part 18. (Note that the level number will appear for each variable and in the 'begin' and 'end' of dump messages, so it is redundant, but that makes the code slightly easier.) If the specified number exceeds the current block nesting level, give a translation-time warning message, and, in place of the specified number, use the current block nesting level.

Also, output from each **dump** statement is now to include the line number on which the **dump** statement appears.

Part 20: Statement Execution Counting

Add a new statement, **EXCNT.**

> statement ::= assignment | print | skip | stop | if | do | fa | break | dump | excnt
>
> excnt ::= **EXCNT**

The **EXCNT** statement indicates to keep a count of the number of times this particular statement is reached during execution. For a program that contains any **EXCNT** statements (even if none are actually executed), a table of all **EXCNT** and their counts is output at the end of execution (i.e., by reaching the end of the program, by executing a **stop**, but not a run-time error such as "mod(a,0)"). For a program that contains no **EXCNT**, no such table is output. So, you'll need to include in the generated C code 'instrumentation' code that counts as each **EXCNT** is encountered and code that outputs the table as noted above.

This functionality gives a primitive form of program execution profile information, which for more realistic programs would be useful for tuning program performance and even debugging.

For simplicity, restrict the E program to a maximum of 100 **EXCNT** statements. For the output table, number such statements beginning with 1 in the order in which they're encountered lexically.

Part 21: Statement Execution Counting (improved)

Also include in the **EXCNT** table the line number on which each **EXCNT** occurs.

Notes

- Grading will be divided as follows.

Percentage	
0	Part 10: Previous Solution
5	Part 11: **skip** statement
10	Part 12: **stop** Statement
10	Part 13: Remainder Operator
10	Part 14: Modulo Operator
15	Part 15: Maximum function
10	Part 16: **break** Statement
10	Part 17: **break** Statement revisited
10	Part 18: **dump** Statement
5	Part 19: **dump** Statement extended
10	Part 20: Statement Execution Counting
5	Part 21: Statement Execution Counting (improved)

- All the notes on the previous class's assignment apply here too.
  - The Java setup on CSIF has changed since the previous assignment. Both javac and java are now in /usr/bin, which should already be in your search path. But, check just to be sure. The latest Java version is:

    1.8.0_131

    Test that you're using the correct Java by the commands

    javac -version

    java -version

    Their output should show the same version number as above.
  - Note well: You will be expected to turn in all working parts along with your attempt at the next part. So, be sure to save your work for each part. No credit will be given if the initial working parts are not turned in. You must develop your program in parts as outlined above *and in that order*. Each part builds on its immediately preceding part; the parts are, with respect to the test files, *not* independent.
  - Be sure to use the provided test files and test scripts. Your output must match the "correct" output, except for the wording of and the level of detail in the error messages. (Your messages can be more or less detailed as you wish, provided they are reasonably understandable. Don't spend a lot of time making fancy error messages, though.) There is one test file for which the order in which you check things might be different from the order I check things, so the specific complaint might differ; that test file has a comment to that effect. Similarly, for Part 17, you might give different errors depending on which you check for first: break statement outside of a loop or a bad number on a break. Note that the test files and their "correct" output might differ from part to part, so be sure to work in the order given above and to use the right test files for each part.

- Get started **now** to avoid the last minute rush.