ASSIGNMENT 2: Java program
Due: October 11, 2016

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 as described below.

The code you will be given works (or so is our intent and belief), 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 11. (The Part 6 solution to the previous assignment is given to you as "part10" in the "Given" code.)

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 11: 'not equal' token

Change E's 'not equal' token from '/=' to '!='.

Part 12: printnl Statement

Add a new statement, **printnl.** It simply prints a newline. The modified syntax is:

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

Part 13: Variable Declaration

Change the syntax of variable declaration to:

> declarations ::= **var** id { ',' id }

Note that **rav** is no longer a keyword.

Part 14: Print Statement

Change the syntax of the print statement to allow multiple expressions:

> print ::= **print** expression { ',' expression}

Values of the expressions are output on a single line, separated by a single blank. (Do not put a blank after the last value.)

Part 15: swap Statement

Add a new statement, **swap.** The modified syntax is:

> statement ::= assignment | swap | print | printnl | if | do | fa
>
> swap ::= **swap** id id

It swaps the values of the two variables.

Semantic checks:

- If either variable is undeclared, give an appropriate error message via "System.err.println" and then stop your program via "System.exit(1)".
- If the two variables are the same, give a warning message (via "System.err.println") and just continue.

A variable that appears in a swap statement should be shown as both assigned and used for the variable reference table (see Part 4 of the previous class's assignment).

Part 16: fa Statement

Modify the **fa** statement so to also allow the loop counter to count down. The modified syntax is:

> fa ::= **fa** id ':=' expression (**to**|**downto**) expression [**st** expression] commands **af**

So, a **to** loop counts up; a **downto** loop counts down. Examples:

> **fa** j := 3 **to** 5 → **print** j **af** # outputs 3, 4, 5
>
> **fa** j := 5 **to** 3 → **print** j **af** # outputs nothing
>
> **fa** j := 3 **downto** 5 → **print** j **af** # outputs nothing
>
> **fa** j := 5 **downto** 3 → **print** j **af** # outputs 5, 4, 3

Part 17: An If Expression

E already allows 'if statements'. In contrast, this part adds 'if expressions' to E. Such expressions will now be allowed to appear where other E expressions may appear, as detailed below. As simple examples:

max := {x>y → x **else** y} # sets max to maximum of x and y

sgn := {x>0 → 1 [] x<0 → 0-1 **else** 0} # sets sgn to sign(x)

See the provided test files for more examples.

The modified syntax is:

factor ::= '(' expression ')' | id | number | '^' expression | '@' expression
| '{' guarded_expressions '}'
guarded_expressions ::= guarded_expression { '[]' guarded_expression } **else** expression
guarded_expression ::= expression '–>' expression

Such an expression is evaluated similarly to how an if statement is evaluated: Guard expressions are evaluated in lexical order until one is found true or the **else** is reached; the value of the entire if expression is then the value of the expression after the associated '–>' or the value of the expression after **else.**

Hint: use C's exp?exp:exp in your generated code; a C if statement won't work here (why?).

Part 18: Run-time Checking for Unassigned Variables

It is now a run-time error if the E program attempts during its execution to use the value of an unassigned variable. Generate additional C code so that at E program execution time, the E program will stop execution with an appropriate error message (including the offending variable name) if it encounters such an unassigned variable.

Note that each variable is already initialized to -12345, but for our purposes do not treat that as an assignment. (Also, do not change that initialization; it might help in debugging, rather than getting some garbage value.) Note that variables can be assigned any legal integer value, even -12345.

Hint: use an additional flag variable for each E variable.

Another hint: Recall that variables can appear within arbitrary expressions, so whatever code you generate must be valid within a C expression. One way is to generate (once, before the C main function) a C function that performs the check of the additional flag variable. If the check is okay, it returns 1; otherwise, it prints the error message and exits the program. To exit the program, use the C library function *exit*. See "man 3 exit" for details; be sure to *#include <stdlib.h>* in the C code you generate.

Part 19: Run-time Checking for Unassigned Variables: Improved Diagnostics

Add to the output more information about the error reported in Part 18. Specifically,

- the line number on which the error occurred.
- the line number and nesting depth of the declaration for the offending variable.

See the provided correct output for examples of the required format for this output.

Notes

- All the notes on the previous class's assignment apply here too.
  - Following the notes on the previous assignment should give you the latest Java:

    javac 1.8.0_60

  - 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.
  - Additional notes on your output vs. "correct" output: There is one Part 18 and Part 19 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. 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.

- Grading will be divided as follows.

Percentage	
5	Part 11: 'not equal' token
10	Part 12: **printnl** Statement
10	Part 13: Variable Declaration
10	Part 14: Print Statement
15	Part 15: **swap** Statement
10	Part 16: **fa** Statement
20	Part 17: An If Expression
15	Part 18: Run-time Checking for Unassigned Variables
5	Part 19: Run-time Checking for Unassigned Variables: Improved Diagnostics

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