ASSIGNMENT 2: Java program
Due: October 16, 2012

Overview

Attached is a handout for a previous class's assignment, which describes the E programming language and its implementation. You will 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 (static 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: Strings in Print

E now allows the printing of string literals. String literals can appear only within a **print** statement. The rule for print is now

print ::= **print** print_expression
print_expression ::= expression | string

The token "string" represents a possibly empty sequence of characters enclosed within a pair of double quote characters (e.g., "hi"). For simplicity:

- There are no special interpretations given to any "escape" characters in strings, as for, e.g., \n or %s in C; such are just treated as characters.
- Each string must be contained entirely on one line.
- There is no limit on the length of a string.

So, the above means that the characters % and \ can appear within an string and they have no special meaning. It also means that the character " cannot, since it indicates to end the string and there is no escape character.

Your program should print a fatal error message and terminate immediately if it encounters a would-be string that is missing its closing quote. The surrounding quotes are not printed by the E program.

Hint: as the revised grammar suggests, string should be a new token.

Part 12: Repeat-Until Statement

E now allows a repeat-until statement, similar in syntax and semantics to those in Pascal or Modula-2. The rule for statement is now

statement ::= assignment | print | if | while | for | repeat

The form of the repeat-until statement is

repeat ::= **repeat** block **until** expression

Its meaning is as follows. Execute the block of statements and then evaluate the boolean expression. If its value is true, then the repeat-until statement terminates and execution continues with the following statement. Otherwise, execution of the block, evaluation of the expression, and testing is repeated until the expression does evaluate to true. Hint: C has a **do-while** statement.

Part 13: Arrays

E now allows arrays to be declared, and array elements to be assigned to and used within expressions. Each array element is to have an initial value of 4444. An array is a special form of a variable, as opposed to a constant; E does not allow constant arrays. An array is declared as part of a var_decl, as follows:

var_decl ::= **var** decl_id { ‘,’ decl_id }
decl_id ::= id [‘[’ bound ‘:’ bound ‘]’]
bound ::= [‘-’] number

An element of an array is referenced as part of assignment or factor, as follows:

assignment ::= id_sub ‘:=’ expression
factor ::= ‘(’ expression ‘)’ | id_sub | number
id_sub ::= id [‘[’ expression ‘]’]

This part requires a few new semantic checks, specifically compile-time checks (i.e., performed by the translator):

- the size of the array is positive, i.e., upperbound-lowerbound+1>0.[1]
- a reference to an identifier (i.e., via id_sub) has a subscript *iff* the identifier was declared to be an array.

For any violation of the above, give an appropriate error message via “System.err.println” and then terminate the translator via “System.exit(1)”.

In this part, assume that each array index is within the declared bounds of the array. (In Part 14, we remove that assumption.)

Two programming notes:

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

  num = Integer.parseInt(s);

[1] We require the size to be strictly positive since we’re generating C code and C doesn’t allow 0-sized arrays. (Java and some other languages do.)

- An example of C's array initializer:

```
int b[4] = {99, 33, 2, 5, };
```

  Note that the last comma above is allowed (which you can use in code generation for initialization of E arrays).

Part 14: Array Bounds Checking

This part requires one new semantic check: each array index is within the declared bounds of its array. This check must be performed at E run time (i.e., performed by the generated C code), specifically:

- the subscript value for each reference to an array falls between the array's lowerbound and upperbound, i.e., lowerbound≤subscript≤upperbound. This check is performed just before the array is to be accessed.

For any violation of the above, give an appropriate error message via "stderr" and terminate the executable via "exit(1)".

It's not simple to generate code for bounds checking since the array reference can appear in the middle of an arbitrary expression and array references can be nested (e.g., "8*(b-a[w*c[t-6]])/3"), and we're using a one-pass translator combined with code generator. Hint: Generate a boundscheck function, say *bc*, in your C code. Invoke *bc* before each array reference. Have *bc* check whether the subscripting expression is within bounds. If it is, then have *bc* return the value of the subscripting expression; if it isn't, have *bc* print an error and exit (and *not* return).

Part 15: Every Statement

E now allows an every statement. It is motivated by Java's "enhanced for" statement that allows iteration through arrays and collection types. As a simple example, the E every statement

```
var a[1:10]
every x: a do
   print x
end
```

prints the values of a[1], a[2], ..., a[10].

The rule for statement is now

```
statement ::= assignment | print | if | while | for | repeat | every
```

The form of the every statement is

every ::= **every** [**element**|**index**] [**forward**|**reverse**] id ':' id **do** block **end**

The first "id" is the every's index variable; the second "id" is the every's array. The every statement allows iteration through either the elements or indexes for the array; the default is **element.** The direction of iteration is either **forward** or **reverse**; the default is **forward.** As another example,

```
var a[1:10]
every index reverse x: a do
   print x
end
```

prints the values 10, 9, ..., 1.

The index variable of an every statement

- is a new variable whose scope is the body of the every statement.[2]
- is not allowed to be modified (in the same sense as for the index variable of an E for statement).
- is computed once at the beginning of each iteration of the every (as opposed to each time the index variable appears in the body). This makes a difference only for **element** every statements. In short, view the meaning of an every statement by translation into a for statement (which is exactly how Java's enhanced for statement is defined)[3]. The "simple example" above translates to roughly the E for statement

```
for z := 1 to 10 do
  x := a[z]
  print x
end
```

  where *z* is a new variable (created by the translator). This meaning explains several otherwise tricky situations (which also arise with Java's enhanced for statement and are defined similarly). For example, consider the following code fragment

```
var a[1:10], w
 ...
w := 1
every x: a do
   a[w] := 999
   w := w+1
   print x
end
```

  It prints the values of *a* before they were changed by the every statement because, for example, *a*[1] isn't changed until after the first iteration of the every begins, which is when *x*'s value is set. On the other hand, if a value in *a* is changed by a previous iteration of the every, that value is used. For example,

```
every x: a do
   a[2] := 999
   print x
end
```

  prints 999 as the second value output, because *a*[2] was changed to 999 on the *first* iteration of the every. (If these two examples aren't clear, write out the for statements to which they're conceptually translated.)

---

[2] Note that this differs from the index variable for an E for statement, which must be previously declared. From a language design perspective, such an inconsistency is not good, but it's good in this assignment to make you deal with something a little different.

[3] In at least one Java compiler, the enhanced for statement is actually translated to a basic for statement by generating an appropriate parse tree for the for statement. Since your translator works all in one pass and doesn't keep parse trees, that's not an option. So, this translation is really more conceptual. But, your tranlator's code for the every statement will be similar to your code for the for statement.

Notes

- Grading will be divided as follows.

Percentage	
0	Part 10: Previous Solution
10	Part 11: Strings in Print
20	Part 12: Repeat-Until Statement
35	Part 13: Arrays
15	Part 14: Array Bounds Checking
20	Part 15: Every Statement

- All the notes on the previous class's assignment apply here too.
  - One change: You must use Java 1.7. On CSIF use

    /usr/java/latest/bin/javac
    /usr/java/latest/bin/java

    which are links to latest version installed, which is:

    /usr/java/jdk1.7.0/bin/javac
    /usr/java/jdk1.7.0/bin/java

    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. If not, then you need to change your PATH environment variable as follows:

    - For csh (or tcsh) users, *carefully* put in your .cshrc file

      set path = (/usr/java/latest/bin $path)

    - For bash users, *carefully* put in your .bashrc file

      export PATH=/usr/java/latest/bin:$PATH

    For either, you need to logout and login again to have this change take effect.
  - Translation by javac of your Java code must not generate any warning messages. You will be given a Makefile for each part. Your code for each part must work with the provided Makefile. (You won't need to modify the Makefiles.) It asks the Java compiler to give all possible warnings about your code. Your code for each part must compile without any warnings.
  - 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.) 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.