ASSIGNMENT 2: Java program

Due: October 12, 2010

Overview

The E programming language is similar to C, C++, Java, Pascal, Modula-2, etc. but it differs syntactically and semantically, and is considerably simpler.

You are to write a Java program that translates E programs to their semantically equivalent C programs. That is, the input to your program—henceforth, the *translator*—is an E program; the output from your program is a C program—henceforth, the *generated code*, or simply *GC*. To verify that your translator works correctly, you are to compile and execute the GC. This assignment should help you develop a better understanding of parsing and compiling, and some familiarity and practical programming experience with Java.

The general approach to translation you will be using does all its work in "one pass". That is, its three activities of syntactic analysis, semantic analysis, and code generation are intermixed. As your translator reads a token or a small group of tokens in the E program, it will determine if the token(s) is syntactically and semantically valid, and generate code for the token(s). The E language is designed to facilitate one-pass translation. In contrast, many real compilers perform multiple passes, e.g., one pass for each of the three activities.

The E Language

The following grammar partially describes E's syntax. In the BNF below, reserved words are typeset in bold and other terminal symbols are enclosed in single quotes.

program ::= block

block ::= {declaration} {statement}

declaration ::= var_decl | const_decl

var_decl ::= **var** id { ‘,’ id }

const_decl ::= **const** id ‘=’ number

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

assignment ::= id ‘:=’ expression

print ::= **print** expression

if ::= **if** expression **then** block {**elsif** expression **then** block} [**else** block] **end**

while ::= **while** expression **do** block **end**

for ::= **for** id ‘:=’ expression **to** expression **do** block **end**
  | **for** id ‘:=’ expression **downto** expression **do** block **end**

expression ::= simple [relop simple]

simple ::= term {addop term}

term ::= factor { multop factor }

factor ::= ‘(’ expression ‘)’ | id | number

relop ::= ‘=’ | ‘<’ | ‘>’ | ‘/=’ | ‘<=’ | ‘>=’

addop ::= ‘+’ | ‘−’

multop ::= ‘*’ | ‘/’

The following rules complete the syntactic definition of E.

- An E program is followed by an end-of-file indicator; extra text is not legal.
- The terminal (token) "id" represents a nonempty sequence of letters other than one of the keywords. Similarly, the terminal (token) "number" represents a nonempty sequence of digits.
- As is the case in most languages, tokens are formed by taking the longest possible sequences of constituent characters. For example, the input "abcd" represents a single identifier, not several identifiers. Whitespace (i.e., one or more blanks, tabs, or newlines) may precede or follow any token. E.g., "x=10" and " x = 10" are equivalent. Note that whitespace delimits tokens; e.g., "abc" is one token whereas "a bc" is two.
- A comment in E begins with a "#" and consists of all characters up to, but not including, a newline or end-of-file.

E's semantics follow for the most part C's semantics, but there are significant differences. The important semantic points are:

- E has only one type: integer. Unlike local variables in C, the initial value of E variables is defined; it is 8888. As in C, variables and constants must be declared before use, redeclaration of an id (in the same block) is an error (though see later how to handle such errors—some just cause warnings), and the declaration of an id in an inner block hides the declaration of id(s) of the same name declared in outer blocks; i.e., E has the same scoping rules as C, which allows such hiding (whereas Java doesn't).
- An expression containing relational operators has the same meaning as in C; e.g., the value of a<b is 0 if false and 1 if true.
- A conditional expression—i.e., the expression immediately following **if**, **elsif**, or **while**—is considered true if it is nonzero and false otherwise.
- E's **for** statement is syntactically similar to Pascal's or Modula-2's but semantically closer to C's. A **for** statement using **to** implies an increasing sequence of index values; a **for** statement using **downto** implies a decreasing sequence of index values. The body of a **for** is not executed if its implied range is empty (e.g., as in **for** i := 1 **to** -3). E's **for** statement: requires that the index variable be declared prior to the **for** statement; does *not* allow the index variable to be modified within the **for**'s body (although don't worry about that until Part 5); reevaluates the final expression of the loop (i.e., the expression following **to** or **downto**) before each loop iteration to determine whether the loop should terminate; and guarantees that the value of the index variable after the loop completes is the last value assigned to the index variable as implied by the **for.**

Part 1: The Scanner

A scanner is sometimes called a lexer—since it does lexical analysis—or a tokenizer—since it breaks up its input into tokens. For example, the *GetToken* procedure described in Louden is a scanner.

You will be provided with a complete scanner. You need to

- Answer a few questions in the code itself. (Search for "QUESTION" in Scan.java.) Type each answer as comments directly under the question. Be concise.
- Test your scanner thoroughly by using the provided main program (which simply calls the scanner repeatedly and outputs the current line number, the token's string value, and the token's kind) and the provided test scripts and test programs..

The scanner imposes a limit on the length of tokens. If the scanner encounters a token whose length exceeds that limit, it prints an appropriate error message and ignores the extra characters. Also, if the scanner encounters a character that is not in E's character set, it prints an appropriate error message, ignores the unknown character[1], and continues by examining the next character in the input. Finally, the scanner simply discards whitespace and comments.

Part 2: The Parser

First, rewrite the grammar using syntax graphs. Then, determine the *first* sets; this should be straightforward because the grammar is simple. Finally, translate the syntax graph into a parser. See the textbook for details. If an error—e.g., missing ":=" in an assignment statement—is encountered in parsing, print an appropriate error message and stop your translator via "System.exit(1)".

Test your parser to see that it recognizes syntactically legal E programs and complains about syntactically illegal programs.

[1] More precisely, such a character is treated as whitespace as it delimits tokens.

Part 3: The Symbol Table

Up to this point, we have dealt with syntax. Now, we need to enforce the semantic constraints that deal with variables and constants: we must detect undeclared and redeclared variables and constants, and prevent assignments to constants.

A table of id's and their kinds in the current scope needs to be maintained. (For this program, we define the *kind* of an id to indicate whether it is a variable or constant.) Each time a variable or constant is declared, the table is checked. If the id is already in the table entries for the *current* block, then it is being redeclared, which is an error. Otherwise, the id and its kind is added to the table. Each time an id is referenced, the table is checked to ensure that the id has been declared, and that if the id appears on the left-hand side of an assignment, that it is a variable (not a constant).

This symbol table is maintained at translation time. It can be represented as follows. Variables (and constants) defined within a block can be kept in a list, one list for each block. Since E's scoping rules are the same as C's, variables (and constants) defined within a block disappear at the end of the block. Hence, a stack of such lists is a natural representation for the symbol table. A new entry on the stack is created whenever a new block is encountered; the most recent entry on the stack is discarded whenever a block is exited.

An undeclared id, then, is one that does not appear anywhere in the entire stack of lists. Note that the stack should be searched beginning with the *newest* entry to locate the most recent symbol table entry with a given name. Consider, for example, the following legal program.

```
const N = 10
var i
for i := 1 to N do
   var N
   N := i*i+8*i
   print N
end
```

When parsing the inner block, the most recent symbol table entry for N indicates that N is a variable, whereas the older symbol table entry for N indicates that N is a constant. Thus, N's most recent entry is the correct one to use.

For a redeclared variable or constant, give an appropriate error message, and then continue by ignoring the redeclaration. For an undeclared variable or an attempted assignment to a constant, give an appropriate error message, and then stop your translator by "System.exit(1)".

Since the number of id's in the program is neither known in advance nor bounded, your program must use a dynamically allocated data structure.

Part 4: Generating Code

Modify your parser so that it outputs appropriate C code. For example, for the E program

```
var i
i := 1
if i /= 10 then print 99 # this is a comment
end
```

your translator might produce something like

```
#include <stdio.h>
main(){
int x_i = 8888;
x_i = 1;
if( x_i != 10 ) {
printf("%d\n", 99);
}
return 0;
}
```

Note that since the scanner discards whitespace and comments, the output is not as neatly formatted as the E program. In fact, the actual output from your translator will probably be less formatted than shown above; e.g., it is fine to output a single C token per line. Also note that the translator has prepended each E variable name with x_ to avoid conflicts with C reserved words. Also be sure to *#include <stdio.h>* in the C code you generate.

To test your translator, examine its C output (the GC). Then, compile the GC and execute the resulting program. Verify that it does indeed execute as the source E program specifies.

Part 5: Unmodifiable Index Variables

Modify your code so it disallows index variables of “for loops” to be modified within the body of the loop (except, of course, they can be modified according the definition of the “for” loop). As part of this part, you need to identify what E statement(s) can modify index variables and prevent them from doing so. (That’s not too challenging and, hint, you can check the given test programs.) For such a translation-time error, give an appropriate error message, and then stop your translator by “System.exit(1)”.

Notes

- You must use Java 1.6, not Java 1.4. On CSIF use

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

  /usr/java/latest is a link to the latest java version installed, which is:

  ```
  /usr/java/jdk1.6.0_21
  ```

  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.
- Do not be overly concerned with efficiency. On the other hand, do not write a grossly inefficient program.
- Strive for simplicity in your programming. Do not try to be fancy. For example, the syntax graphs translate directly into code. Learn and use that technique—do not spend your time trying to improve that code. You will find it most helpful to use the names of the nonterminals as the names of your parsing procedures (except, e.g., "if" is a keyword in Java).
- Express your program neatly. Part of your grade will be based on presentation. Indentation and comments are important. Be sure to define each variable used with a concise comment describing its purpose. In general, each procedure should have a comment at its beginning describing its purpose. In this program, however, the parsing procedures do not need such introductory comments; a single explanatory comment before the first parsing procedure should suffice.

- The provided program is written so that it works on standard input. Thus, your program, named e2c, will be invoked as “java e2c” if you want input to come from the keyboard, or as “java e2c < t01.e” if you want input to come from the file t01.e; use “java e2c < t01.e > t01.c” to take input from t01.e and to output to t01.c. The provided program can also take input from a file, e.g, “java e2c t01.e” (which might facilitate easier testing within Eclipse). Output the generated code to stdout via “System.out.println” (or “System.out.print”). So as not to intermix generated code and error/warning messages, output error messages to stderr via “System.err.println”.
- Your program must work with the provided Makefile and shell scripts.
- The provided shell scripts depend on the exit status of your program. If your program detects a "serious" error, your program should exit via “System.exit(1)”, as described earlier. If your program finishes normally, your main method should just end normally or equivalently “System.exit(0)”.
- You must 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. By “match”, we mean match exactly character by character, including blanks, on each line; also, do not add or omit any blank lines. Your error 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.
- If you run your program on Windows, the output might look the same, but a file comparison program might complain due to extra \r’s in Windows’ output. Retest it on CSIF to be sure, and as is required.
- Create several of your own E test programs. You will probably want different test data for the different parts. Be sure that your program works for boundary conditions; e.g., the shortest legal E program is empty.
- Grading will be divided as follows.

Percentage	
10	Part 1: The Scanner
20	Part 2: The Parser
30	Part 3: The Symbol Table
30	Part 4: Generating Code
10	Part 5: Unmodifiable Index Variables

- Points will be deducted for not following instructions, such as the above.
- A message giving exact details of what to turn in, where the provided source and test files are, etc. will be posted.
- Get started **now** to avoid the last minute rush.