Harry Bassi and Tony Doan

Hw3

In our assignment we learned how to parse nanoAda, and translate code written in nanoAda into Java. We were provided a parser specification written in SableCC. The SableCC framework creates a parse tree that we traverse using a visitor pattern. The framework came with classes that we extended in order to complete the language translation tasks.

Part 1: Symbol Tables

For part one, we used most of our SymbolEntry and Symbol Table from the past hw2, and removed anything we did not need from it. For example we did not use Chario, and did not use the append and next functionalities of the original. We also implemented the functions for finding and entering identifiers and functions relating to role analysis in our Semantic Analyzer.

Part 2: Semantic Analysis

For part 2 (semantic analysis), we extended the DepthFirstAdapter, and made sure to correctly check for scope and role analysis by implementing enterID, findID, setRole, and acceptRole (similarly to the previous assignment). The purpose of this part was to make sure we set the correct roles to identifiers, and that they follow the rules they are given. For example, we cannot use variables without declaring them first, and we cannot end a procedure with a variable, since it would have to be a procedure identifier that ends the procedure.

Part 3: Code Generation

For part 3 (code generation), we extended the DepthFirstAdapter, and overrode functions in order to convert the nanoAda code and translate it into Java code. We were successfully able to convert it, and ran many test cases. It correctly makes the main class, from the first procedure, and we create functions for all of the following procedures, and call them in our constructor for the main class. We handled all of the changes in converting nanoAda code to java code. Unfortunately we were not able to figure out how to do the nested procedures, but everything else is working. The java code is also correctly indented.

Part 4: Pix and Primes

For part 4, we learned more nanoAda code, in order to correctly produce Pix, and Primes. Primes is supposed to create a delimited list that produces a one or zero for every number between one to one-hundred. Pix creates a delimited list that prints out prime numbers that are below an inputted number. That number can only be from one to one-hundred. We ran Pix and Primes through our semantic analyzer and code generation, and they produce correct java files. We used javac to create class files for them and executed those. The output was exactly correct.

APPENDIX:

PRIMES:

```
procedure PRIMES is
        I, J : INTEGER;
        COUNTER : INTEGER;
        MODRESULT : INTEGER;
        begin
        write("\r");
 I := 1;
        while I <= 100 loop
                COUNTER := 0;
                J := 1;
                while J <= I loop
                        MODRESULT := I mod J;
                        if MODRESULT = 0 then
                                COUNTER := COUNTER + 1;
                        endif;
                        J := J + 1;
                end loop;
                if I = 100 then
                        write("0");
                elseif COUNTER = 2 then
                        write("1,");
                else
                        write("0,");
                endif;
                I := I + 1;
        end loop;
 write("\r");
end;
```

PIX:

```
procedure Pix is
        procedure Pi(X : INTEGER) is
                I, J : INTEGER;
                COUNTER : INTEGER;
                MODRESULT : INTEGER;
                begin

                 I := 1;

                while I <= X loop
                COUNTER := 0;
                J := 1;
                while J <= I loop
                        MODRESULT := I mod J;
                        if MODRESULT = 0 then
                                COUNTER := COUNTER + 1;
                        endif;
                        J := J + 1;
                 end loop;

                 if COUNTER = 2 then
                        if I = 2 then
                                write("2");
                        else
                                write(",");
                                write(I);
                        endif;
                 endif;

                 I := I + 1;
                 end loop;
        end;

begin
        write("\r");
        Pi(100);
        write("\r");
end;
```