Computer Science 3 - 2013

Programming Language Translation

Practical for Week 24, beginning 7 October 2013 - Solutions

Once again, there were some very good solutions handed in, but a few groups seemed rather at a loss here and there. Full sources for the solutions can be found in the file `PRAC24A.ZIP` or `PRAC24AC.ZIP` for those who wish to experiment further.

Tasks 2 and 3 - Checking on Tokens

This is harder to get right than it at first appears. One has to be able to recognise and distinguish (assume spaces precede and follow the patterns below):

```
 30      (integer 30)
 0       (integer 0)
 030     (integer 0 followed by integer 30)
 30.     (integer 30 followed by period)
 0.      (integer 0 followed by period)
 9.      (integer 9 followed by period)
 .0      (period followed by integer 0)
 .9      (period followed by integer 9)
 0.3     (double 0.3)
 3.0     (double 3.0)
 3.3     (double 3.3)
```

Something like this seems to be required:

```
import library.*;
import java.util.*;

COMPILER TokenTests $CN
/* Test scanner construction and token definitions - C# version
   The generated program will read a file of words, numbers, strings etc
   and report on what characters have been scanned to give a token,
   and what that token is (as a magic number).  Useful for experimenting
   when faced with the problem of defining awkward tokens!

   P.D. Terry, Rhodes University, 2013 */

/* Some code to be added to the parser class */

  static void display(Token token) {
  // Simply reflect the fields of token to the standard outpul
    IO.write("Line " + token.line + " Column");
    IO.write(token.col, 4);
    IO.write(":  Kind");
    IO.write(token.kind, 3);
    IO.writeLine("  Val |" + token.val.trim() + "|");
  } // display

CHARACTERS  /* You may like to introduce others */

  sp         = CHR(32) .
  backslash  = CHR(92) .
  control    = CHR(0) .. CHR(31) .
  letter     = "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz" .
  digit09    = "0123456789" .
  digit19    = "123456789" .
  stringCh   = ANY - '"' - control - backslash .
  charCh     = ANY - "'" - control - backslash .
  printable  = ANY - control .

TOKENS      /* You may like to introduce others */

  ident      = letter { {"_"} (letter | digit09) } .

  integer    =   digit19 { digit09 }
               | digit19 { digit09 } CONTEXT(".")
               | "0"
               | "0" CONTEXT(".") .
```

```
  double     = ( digit19 { digit09 } "." | "0." ) digit09 { digit09 }
               [ "E" [ "+" | "-" ] digit09 { digit09 } ] .

  string     = '"' { stringCh | backslash printable } '"' .

  char       = "'" ( charCh   | backslash printable ) "'" .

IGNORE control

PRODUCTIONS

   TokenTests
   = { (    ANY
          | "."
          | ".."
          | "ANY"
       )                        (. display(token); .)
     } EOF                      (. display(token); .)
   .

END TokenTests.
```

There may be a lurking bug in Coco/R that needs to be probed further. The alternative below does not work, although I see no reason why not!

```
  integer    =   ( digit19 { digit09 } | "0" )
               | ( digit19 { digit09 } | "0" ) CONTEXT(".")
```

Only two groups got all the way. Most fell down on one or more of those examples, possibly without noticing, and probably because they had not thought of exhaustive test data.

Task 4 - The Parva Cross reference generator.

Once again, this is capable of a very simple elegant solution (hint: most Pat Terry problems admit to a simple elegant solution; the trick is to find it, so learn from watching the Expert in Action, and pick up the tricks for future reference).

All the references to identifiers in the Parva grammar are channelled through the `Ident` non-terminal, so all we have to do is add the actions here to add the identifier to the table, suitably tagged according to how it has been encountered,

This should suggest attributing the `Ident` production with a value parameter. The details follow of the other productions that refer to it. Note the careful way in which we attribute the `ForStatement` production, which has the option of incorporating a highly local control identifier.

```
COMPILER Parva $CN
/* Parva level 1 grammar  - Coco/R for Java (EBNF)
   P.D. Terry, Rhodes University, 2013
   Extended for prac 24
   Cross reference generator */

  static final boolean
    declared = true,
    used     = false;

PRODUCTIONS
  Parva                              (. Table.clearTable(); .)
  = "void" Ident<declared>
    "(" ")" Block                    (. Table.printTable(); .) .

  Assignment
  =   Designator
     (   "=" Expression | "++" | "--" )
       | "++" Designator
       | "--" Designator .

  Designator
  = Ident<used> [ "[" Expression "]" ] .
```

```
ForStatement                        (. boolean how = used; .)
= "for" "("
    [ [ BasicType                   (. how = declared; .)
      ] Ident<how>
      "=" Expression
    ] WEAK ";"
    [ Condition ] WEAK ";"
    [ Assignment ]
  ")" Statement .

Ident<boolean declared>
= identifier                        (. Table.addRef(token.val, declared, token.line); .) .
```

Of course, we need to have a table handler. In this case we can just write very simple code like the following. Running the search loop from the bottom makes for a very simple `addRef` method. Note that this handler allows us to enter an identifier that has been used before being declared. While this may be "wrong", it prevents crashes of the cross-referencer itself.

```
// Handle cross reference table for Parva
// P.D. Terry, Rhodes University, 2013

package Parva;

import java.util.*;
import library.*;

  class Entry {                     // Cross reference table entries
    public String name;             // The identifier itself
    public ArrayList<Integer> refs; // Line numbers where it appears
    public Entry(String name) {
      this.name = name;
      this.refs = new ArrayList<Integer>();
    }
  } // Entry

  class Table {
    static ArrayList<Entry> list = new ArrayList<Entry>();

    public static void clearTable() {
    // Clears cross-reference table
      list.clear();
    }

    public static void addRef(String name, boolean declared, int lineRef) {
    // Enters name if not already there, and then adds another line reference (negative
    // if at a declaration point in the original source program)
      int i = 0;
      while (i < list.size() && !name.equals(list.get(i).name)) i++;
      if (i >= list.size()) list.add(new Entry(name));
      list.get(i).refs.add(declared ? -lineRef : lineRef);
    }

    public static void printTable() {
    // Prints out all references in the table
      for (int i = 0; i < list.size(); i++) {
        Entry e = list.get(i);
        IO.write(e.name, -16);                    // left justify in 16 spaces
        for (int j = 0; j < e.refs.size(); j++)
          IO.write(e.refs.get(j), 4);             // left justify in 4 spaces
        IO.writeLine();
      }
    }

  } // Table
```

The `printTable` method above suffers from a possible disadvantage in that multiple occurrences of an identifier on one line, as in

```
i = i + (i - list[i] * i);
```

create unwanted duplicate entries. These can be eliminated in various ways; the simplest might be to do so at the output stage, rather than when they are added by the `addRef()` method. Please yourself; here is a suggestion.

```
  public static void printTable() {
  // Prints out all references in the table (eliminate duplicate line numbers)
    for (int i = 0; i < list.size(); i++) {
      Entry e = list.get(i);
      IO.write(e.name, -16);                    // left justify in 16 spaces
      int last = 0;                             // impossible line number
      for (int j = 0; j < e.refs.size(); j++) {
        int line = e.refs.get(j);
        if (line != last) {
          IO.write(line, 4);                    // left justify in 4 spaces
          last = line;                          // remember we have printed this line number
        }
      }
      IO.writeLine();
    }
  }
```

Several solutions revealed that people either had not thought that far or were confused about the point of the `declared` argument to the `addRef` method. In particular, they might not have realized that an identifier may legally be declared several times, in different scopes, as in the following example:

```
void main () {
  int i = 1;
  while (i < 10) {
    int k = 1;              // first declaration
    while (k < i) {
      write(k); k = k - 1;
     }
    i = i + 1;
  }
  bool k = false;           // second declaration
  write(k);
}
```

Tasks 5 - 7 - The Extended Calculator

Tasks 5 - 7 provided more of a challenge. There were some excellent submissions, but some showed that not nearly enough effort had been expended on the exercise. Pity!

It is possible to solve this problem by embedding the table handling into the start of the ATG file, and the solution kit has versions showing this. However, it may be worth introducing a proper (even if simple) table handler.

It makes for enhanced readability to assign names to a set of magic numbers useful for mapping the types. Besides the obvious `intType` and `boolType`, it is convenient to introduce a fictitious `noType` that can be used to denote the resulting type when an expression is badly formed.

The `Entry` class defines the nodes needed in the symbol table, which need to record the name, type, and value for each variable.

We introduce a wrapper class `ValNode` to allow the various "nodes" in an expression to pass back both the value and type of an operand. Though similar, this class is not the same as the `Entry` class, because it does not require a name field. Note how the default constructor assigns a value of 1 - this will prevent problems if you try to divide by an undefined variable. Clearly one could arrange define these classes in other ways as well.

```
package Calc3;

  import java.util.*;
  import library.*;

  class Entry {
    static final int // enumerations
      noType   =  0,    intType  =  1,    boolType =  2;

    public String name;
    public ValNode value;                                   // other fields could be added

    public Entry(String name, ValNode value) {              // constructor
      this.name = name;
      this.value = value;
    }
  } // Entry
```

```
class ValNode {
  public int type = Entry.noType;
  public int value = 1;

  public ValNode() {                                        // constructor
    this.type = Entry.noType;
    this.value = 1;
  }

  public ValNode(int type, int value) {                     // constructor
    this.type = type;
    this.value = value;
  }
} // ValNode

public class Table {

  static ArrayList<Entry> symTable = new ArrayList<Entry>();

  public static Entry find(String name) {
  // Returns entry with search key name in list, or null if not found
    int i = symTable.size() - 1;                            // index of last entry
    while (i >= 0) {
      if (name.equals(symTable.get(i).name)) return symTable.get(i);
      i--;                                                  // will reach -1 if no match
    }
    return null;
  } // find

  public static void add(Entry e) {
    symTable.add(e);
  } // add

} // Table
```

Here is a complete grammar itself, heavily commented to make various points that may have been missed (these comments are not in the solution sources). Note that all error handling uses the `SemError` interface - do not simply write to the standard output stream directly.

```
  import library.*;

  COMPILER Calc3 $NC
  // Integer/Boolean calculator
  // P.D. Terry, Rhodes University, 2013

  static int toInt(boolean b) {
  // return 0 or 1 according as b is false or true
    return b ? 1 : 0;
  }

  static boolean toBool(int i) {
  // return false or true according as i is 0 or 1
    return i == 0 ? false : true;
  }

/* It makes for enhanced readability to assign names to the magic numbers useful for mapping the
   operators.  Rather use this idea than work with many tedious string comparisons! */

  static final int // enumerations
    opnop =  1,    opadd =  2,    opsub =  3,   opor  =  4,
    opmul =  5,    opdvd =  6,    opmod =  7,   opand =  8,
    opeql =  9,    opneq = 10,    oplss = 11,   opgeq = 12,
    opgtr = 13,    opleq = 14;

  CHARACTERS
    digit      = "0123456789" .
    letter     = "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz" .

  TOKENS
    number     = digit { digit } .
    identifier = letter { letter | digit } .

  IGNORE CHR(0) .. CHR(31)

  PRODUCTIONS
    Calc3
    = {   Print
        | Assignment
      } "quit" .
```

```
        Print
        = "print"
          OneExpr { WEAK "," OneExpr }
          SYNC ";"                    (. IO.writeLine(); .) .

    /* OneExpr is introduced as a simple way of cutting down on actions that might otherwise clutter up
       the Print production.  It becomes responsible for evaluating a single expression that is to be
       printed rather than stored.  Note that some care is taken to ensure that Boolean values are
       printed in a user friendly form.  */

        OneExpr                       (. ValNode expValue; .)
        = Expression<out expValue>    (. switch(expValue.type) {
                                           case Entry.intType:
                                             IO.write(expValue.value); break;
                                           case Entry.boolType:
                                             IO.write(toBool(expValue.value)); break;
                                           default:
                                             IO.write(" bad value"); break;
                                         } .) .

    /* Note that in the Assignment production we do not meddle with the list of variables until the
       expression has been evaluated.  In particular we do not want to enter the incomplete entry into
       the table after parsing the Variable to find its name.  In this way we avoid treating an initial
       definition of, say ABC = ABC + 1, as being correct when it is not (as the ABC on the right side
       has no defined value yet).  Some people missed this. */

        Assignment                    (. ValNode value;
                                         String name; .)
        = Variable<out name>
          "=" Expression<out value>   (. Entry e = Table.find(name);
                                         if (e == null)  // first definition
                                           Table.add(new Entry(name, value));
                                         else // updating a value
                                           e.value = value; .)
        SYNC ";" .

    /* Note the form of the type checking - several people omitted it or had erroneous variations on the
       lines of    if (value.type != Entry.boolType && val2.type != Entry.boolType) ... Note how the
       toInt and toBool methods are used to provide "type casting" of a sort between integer and Boolean
       values.  Notice how the return type is set to the "best guess" (even in the case of an error) -
       this can have beneficial effects later on.  Alternatively one could make more use of the noType idea */

        Expression<out ValNode value>  (. ValNode val2; .)
        = AndExp<out value>
          { "||"
            AndExp<out val2>          (. if (value.type != Entry.boolType || val2.type != Entry.boolType)
                                           SemError("boolean operands required");
                                         value.type = Entry.boolType;
                                         value.value = toInt(toBool(value.value) || toBool(val2.value)); .)
          } .

    /* The rest of the "binary" operations follow in much the same way */

        AndExp<out ValNode value>     (. ValNode val2; .)
        = EqlExp<out value>
          { "&&"
            EqlExp<out val2>          (. if (value.type != Entry.boolType || val2.type != Entry.boolType)
                                           SemError("boolean operands required");
                                         value.type = Entry.boolType;
                                         value.value = toInt(toBool(value.value) && toBool(val2.value)); .)
          } .

    /* But the EqlExp production needs further comment.  Note that the presence of an EqlOp between two
       integer operands yields a type "change" to Boolean - some people seem not to have taken advantage
       of this.  Note also that the == and != operators have meaning only between two operands of
       exactly the same type.  */

        EqlExp<out ValNode value>     (. ValNode val2;
                                         int op; .)
        = RelExp<out value>
          { EqlOp<out op>
            RelExp<out val2>          (. if (value.type != val2.type)
                                           SemError("incomparable operands ");
                                         value.type = Entry.boolType;
                                         switch(op) {
                                           case opeql:
                                             value.value = toInt(value.value == val2.value); break;
                                           case opneq:
                                             value.value = toInt(value.value != val2.value); break;
                                           default:
                                             break;
```

```
                                        } .)
          } .

/* The RelExp production needs further comment.  Note that the presence of an RelOp between two
   integer operands yields a type "change" to Boolean - some people did not appreciate this.  The
   < <= > and >= operators have meaning only between two operands of the integer type.  Note that the
   production only allows for one or two AddExp terms - we use [ ] and not { }.  Do you see why? */

        RelExp<out ValNode value>       (. ValNode val2;
                                           int op; .)
        = AddExp<out value>
          [ RelOp<out op>
            AddExp<out val2>            (. if (value.type != Entry.intType || val2.type != Entry.intType)
                                             SemError("integer operands required");
                                           value.type = Entry.boolType;
                                           switch(op) {
                                             case oplss:
                                               value.value = toInt(value.value < val2.value); break;
                                             case opleq:
                                               value.value = toInt(value.value <= val2.value); break;
                                             case opgtr:
                                               value.value = toInt(value.value > val2.value); break;
                                             case opgeq:
                                               value.value = toInt(value.value >= val2.value); break;
                                             default:
                                               break;
                                           } .)
          ] .

/* The AddExp parser is easy and needs no further comment */

        AddExp<out ValNode value>       (. ValNode val2;
                                           int op; .)
        = MultExp<out value>
          { AddOp<out op>
            MultExp<out val2>           (. if (value.type != Entry.intType || val2.type != Entry.intType)
                                             SemError("integer operands required");
                                           value.type = Entry.intType;
                                           switch(op) {
                                             case opadd:
                                               value.value = value.value + val2.value; break;
                                             case opsub:
                                               value.value = value.value - val2.value; break;
                                             default:
                                               break;
                                           } .)
          } .

/* The MultExp parser needs to check for division by zero for both the / and the % operator - most
   people checked only the obvious one involving / */

        MultExp<out ValNode value>      (. ValNode val2;
                                           int op; .)
        = UnaryExp<out value>
          { MulOp<out op>
            UnaryExp<out val2>          (. if (value.type != Entry.intType || val2.type != Entry.intType)
                                             SemError("integer operands required");
                                           value.type = Entry.intType;
                                           switch(op) {
                                             case opmul:
                                               value.value = value.value * val2.value; break;
                                             case opdvd:
                                               if (val2.value == 0) SemError("division by zero");
                                               else value.value = value.value / val2.value; break;
                                             case opmod:
                                               if (val2.value == 0) SemError("division by zero");
                                               else value.value = value.value % val2.value; break;
                                             default:
                                               break;
                                           } .)
          } .

/* The unary operators need careful type checking.  Note that a unary + does not require any arithmetic */

        UnaryExp<out ValNode value>     (. value = new ValNode(); .)
        =   Factor<out value>
          | "+" UnaryExp<out value>     (. if (value.type != Entry.intType)
                                             SemError("integer operand required");
                                           value.type = Entry.intType; .)
          | "-" UnaryExp<out value>     (. if (value.type != Entry.intType)
                                             SemError("integer operand required");
```

```
                                        value.type = Entry.intType;
                                        value.value = -value.value; .)
          | "!" UnaryExp<out value>     (. if (value.type != Entry.boolType)
                                             SemError("boolean operand required");
                                        value.type = Entry.intType;
                                        value.value = toInt(!toBool(value.value)); .) .

/* The Factor parser is the one ultimately responsible for starting to pass value and type information "up the
   tree".  Note that we regard an undeclared identifier at this stage to be an error - but we will still have
   assigned a value and type to the return argument to make sure that something "sensible" propagates up from here
   (using the default ValNode constructor).  Several people added to the table if an undeclared identifier was
   encountered, but this may not be a wise thing to do. */

        Factor<out ValNode value>       (. String name;
                                           int n;
                                           value = new ValNode(); .)
        =   Variable<out name>          (. Entry e = Table.find(name);
                                           if (e == null) {
                                             SemError("undeclared variable");
                                             // maybe symTable.add(new Entry(name, value));
                                           }
                                           else
                                             value = new ValNode(e.value.type, e.value.value); .)

                                           /* *********** why can we NOT simply assign value = e.value; ? ******* */

          | Number<out n>               (. value = new ValNode(Entry.intType, n); .)
          | "true"                      (. value = new ValNode(Entry.boolType, 1); .)
          | "false"                     (. value = new ValNode(Entry.boolType, 0); .)
          | "(" Expression<out value> ")" .

/* The production for Variable does nothing more than extract the name for us.  Note the exception
   handling in Number - some submissions omitted this. */

        Variable<out String name>
        = identifier                    (. name = token.val; .) .



        Number<out int n>
        =  number                       (. try {
                                             n = Integer.parseInt(token.val);
                                           } catch (NumberFormatException e) {
                                             n = 0; SemError("number too large");
                                           } .) .

/* It is convenient for readability to define the operators by their own non-terminals and for them simply to
   return an appropriate value from an enumeration.  Note the use of a default opnop to mop up errors that might
   otherwise creep in from undefined values */

        MulOp<out int op>               (. op = opnop; .)
        =   "*"                         (. op = opmul; .)
          | "%"                         (. op = opmod; .)
          | "/"                         (. op = opdvd; .) .

        AddOp<out int op>               (. op = opnop; .)
        =   "+"                         (. op = opadd; .)
          | "-"                         (. op = opsub; .) .

        RelOp<out int op>               (. op = opnop; .)
        =   "<"                         (. op = oplss; .)
          | "<="                        (. op = opleq; .)
          | ">"                         (. op = opgtr; .)
          | ">="                        (. op = opgeq; .) .

        EqlOp<out int op>               (. op = opnop; .)
        =   "=="                        (. op = opeql; .)
          | "!="                        (. op = opneq; .) .

      END Calc3.
```