RHODES UNIVERSITY

Computer Science 301 - 2011 - Programming Language Translation

You lucky people! Here is some more free information - a complete solution to the problem posed earlier today.

If one can assume that the Parva programs to be submitted to it are completely correct, a basic pretty-printer for Parva programs can be developed from the following grammar (to be found in the auxiliary kit as `PrettyParva.atg`). We assume the existence of a `CodeGen` class - which essentially consists of the same methods as you saw in the earlier EBNF example.

```
import library.*;

COMPILER PrettyParva $CN
/* Parva level 1.5 grammar for examination - Coco/R for Java
   PrettyPrinter actions
   Java operator precedences
   Supplied Parva Compiler matches this grammar (and has a few extensions)
   P.D. Terry, Rhodes University, 2011 */

  public static boolean indented = true;

CHARACTERS
  lf           = CHR(10) .
  backslash    = CHR(92) .
  control      = CHR(0) .. CHR(31) .
  letter       = "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz" .
  digit        = "0123456789" .
  nonZeroDigit = "123456789" .
  stringCh     = ANY - '"' - control - backslash .
  charCh       = ANY - "'" - control - backslash .
  printable    = ANY - control .

TOKENS
  identifier   = letter { letter | digit | "_" { "_" } ( letter | digit ) } .
  number       = "0" | nonZeroDigit { digit } .
  stringLit    = '"' { stringCh  | backslash printable } '"' .
  charLit      = "'" ( charCh    | backslash printable ) "'" .

COMMENTS FROM "//" TO lf
COMMENTS FROM "/*" TO "*/"

IGNORE CHR(9) .. CHR(13)

PRODUCTIONS
  PrettyParva                         (. String name; .)
  = "void"                            (. CodeGen.append("void "); .)
    Ident<out name>                   (. CodeGen.append(name); .)
    "(" ")"                           (. CodeGen.append("()"); .)
    Block .

  Block
  = "{"                               (. CodeGen.append(" {"); CodeGen.indent(); .)
       { Statement<!indented> }
    "}"                               (. CodeGen.exdentNewLine();
                                         CodeGen.append("}"); .) .


  Statement<boolean indented>
  =   Block
    | ";"                             (. CodeGen.append(";"); .)
    |                                 (. if (indented) CodeGen.indent();
                                         CodeGen.newLine(); .)
     (   ConstDeclarations
       | VarDeclarations
       | AssignmentStatement
       | IfStatement
       | WhileStatement
       | DoWhileStatement
       | ForStatement
       | BreakStatement
       | HaltStatement
       | ReadStatement
       | ReadLineStatement
       | WriteStatement
       | WriteLineStatement
     )                                (. if (indented) CodeGen.exdent(); .) .
```

```
ConstDeclarations
= "const"                          (. CodeGen.append("const "); CodeGen.indentNewLine(); .)
  OneConst
  { WEAK ","                       (. CodeGen.append(", "); CodeGen.newLine(); .)
    OneConst
  } WEAK ";"                       (. CodeGen.append(";"); CodeGen.exdent(); .) .

OneConst                           (. String name; .)
= Ident<out name>                  (. CodeGen.append(name); .)
  "="                              (. CodeGen.append(" = "); .)
  Constant .

Constant                           (. String value; .)
=   IntConst<out value>            (. CodeGen.append(value); .)
  | CharConst<out value>           (. CodeGen.append(value); .)
  | "true"                         (. CodeGen.append("true");  .)
  | "false"                        (. CodeGen.append("false"); .)
  | "null"                         (. CodeGen.append("null");  .) .

VarDeclarations
= Type                             (. CodeGen.indentNewLine(); .)
  OneVar
  { WEAK ","                       (. CodeGen.append(", "); .)
    OneVar
  }
  WEAK ";"                         (. CodeGen.append(";"); CodeGen.exdent(); .) .

Type
= BasicType
  [ "[]"                           (. CodeGen.append("[]"); .)
  ] .

BasicType
=   "int"                          (. CodeGen.append("int");  .)
  | "bool"                         (. CodeGen.append("bool"); .)
  | "char"                         (. CodeGen.append("char"); .) .

OneVar                             (. String name; .)
= Ident<out name>                  (. CodeGen.append(name); .)
  [ "="                            (. CodeGen.append(" = "); .)
    Expression
  ] .

AssignmentStatement
= Assignment WEAK ";"              (. CodeGen.append(";"); .) .

Assignment
=   Designator
    (   AssignOp Expression
      | "++"                       (. CodeGen.append("++"); .)
      | "--"                       (. CodeGen.append("--"); .)
    )
  | "++"                           (. CodeGen.append("++"); .)
    Designator
  | "--"                           (. CodeGen.append("--"); .)
    Designator .

Designator                         (. String name; .)
= Ident<out name>                  (. CodeGen.append(name); .)
  [  "["                           (. CodeGen.append("["); .)
        Expression
     "]"                           (. CodeGen.append("]"); .)
  ] .

WhileStatement
= "while"                          (. CodeGen.append("while "); .)
  "("                              (. CodeGen.append("("); .)
     Condition
  ")"                              (. CodeGen.append(")"); .)
  Statement<indented> .
```

```
IfStatement
= "if"                              (. CodeGen.append("if "); .)
  "("                               (. CodeGen.append("("); .)
     Condition
  ")"                               (. CodeGen.append(")"); .)
  Statement<indented>
  { "elsif" "("                     (. CodeGen.newLine(); CodeGen.append("elsif ("); .)
      Condition
    ")"                             (. CodeGen.append(")"); .)
    Statement<indented>
  }
  [ "else"                          (. CodeGen.newLine(); CodeGen.append("else"); .)
    Statement<indented>
  ] .

DoWhileStatement
= "do"                              (. CodeGen.append("do"); .)
    Statement<indented>
  WEAK "while"                      (. CodeGen.newLine(); CodeGen.append("while "); .)
  "("                               (. CodeGen.append("("); .)
     Condition
  ")"                               (. CodeGen.append(")"); .)
  WEAK ";"                          (. CodeGen.append(";"); .) .

ForStatement
= "for"                             (. CodeGen.append("for "); .)
  ForControl
  Statement<indented> .

ForControl                          (. String name; .)
=   "("                             (. CodeGen.append("("); .)
       [ [ BasicType                (. CodeGen.append(" "); .)
         ]
         Ident<out name>            (. CodeGen.append(name); .)
         "="                        (. CodeGen.append(" = "); .)
         Expression
       ] ";"                        (. CodeGen.append("; "); .)
       [ Condition ] ";"            (. CodeGen.append("; "); .)
       [ Assignment ]               (. CodeGen.append(")"); .)
    ")"
  | Ident<out name>                 (. CodeGen.append(name); .)
    "="                             (. CodeGen.append(" = "); .)
    Expression
    (   "to"                        (. CodeGen.append(" to "); .)
      | "downto"                    (. CodeGen.append(" downto "); .)
    )
    Expression .

BreakStatement
= "break"                           (. CodeGen.append("break"); .)
  WEAK ";"                          (. CodeGen.append(";"); .) .

HaltStatement
= "halt"                            (. CodeGen.append("halt"); .)
  WEAK ";"                          (. CodeGen.append(";"); .) .

ReadStatement
= "read"                            (. CodeGen.append("read"); .)
  "("                               (. CodeGen.append("("); .)
     ReadElement
     { WEAK ","                     (. CodeGen.append(", "); .)
       ReadElement
     }
  ")"                               (. CodeGen.append(")"); .)
  WEAK ";"                          (. CodeGen.append(";"); .) .

ReadLineStatement
= "readLine"                        (. CodeGen.append("readLine"); .)
  "("                               (. CodeGen.append("("); .)
     [ ReadElement
       { WEAK ","                   (. CodeGen.append(", "); .)
         ReadElement
       }
     ]
  ")"                               (. CodeGen.append(")"); .)
  WEAK ";"                          (. CodeGen.append(";"); .) .

ReadElement                         (. String str; .)
= (   StringConst<out str>          (. CodeGen.append(str); .)
    | Designator
  ) .
```

```
WriteStatement
= "write"                          (. CodeGen.append("write"); .)
  "("                              (. CodeGen.append("("); .)
     WriteElement
     { WEAK ","                    (. CodeGen.append(", "); .)
       WriteElement
     }
  ")"                              (. CodeGen.append(")"); .)
  WEAK ";"                         (. CodeGen.append(";"); .) .

WriteLineStatement
= "writeLine"                      (. CodeGen.append("writeLine"); .)
  "("                              (. CodeGen.append("("); .)
     [ WriteElement
       { WEAK ","                  (. CodeGen.append(", "); .)
         WriteElement
       }
     ]
  ")"                              (. CodeGen.append(")"); .)
  WEAK ";"                         (. CodeGen.append(";"); .) .

WriteElement                       (. String str; .)
= (   StringConst<out str>         (. CodeGen.append(str); .)
    | Expression
  ) .

Condition
= Expression .

Expression
= AndExp
  { "||"                           (. CodeGen.append(" || "); .)
    AndExp
  } .

AndExp
= EqlExp
  { "&&"                           (. CodeGen.append(" && "); .)
    EqlExp
  } .

EqlExp = RelExp { EqualOp RelExp } .

RelExp = AddExp [ RelOp AddExp ] .

AddExp = MulExp { AddOp MulExp } .

MulExp = Factor { MulOp Factor } .

Factor
=   Primary
  | "+"                            (. CodeGen.append(" +"); .)
    Factor
  | "-"                            (. CodeGen.append(" -"); .)
    Factor
  | "!"                            (. CodeGen.append("!"); .)
    Factor .

Primary
=   Designator
  | Constant
  | "new"                          (. CodeGen.append("new "); .)
    BasicType
    "["                            (. CodeGen.append("["); .)
       Expression
    "]"                            (. CodeGen.append("]"); .)
  | "("                            (. CodeGen.append("("); .)
    (   "char" ")"                 (. CodeGen.append("char)"); .)
        Factor
      | "int" ")"                  (. CodeGen.append("int)"); .)
        Factor
      | Expression
        ")"                        (. CodeGen.append(")"); .)
    ) .

AddOp
=   "+"                            (. CodeGen.append(" + "); .)
  | "-"                            (. CodeGen.append(" - "); .) .
```

```
  MulOp
  =   "*"                           (. CodeGen.append(" * "); .)
    | "/"                           (. CodeGen.append(" / "); .)
    | "%"                           (. CodeGen.append(" % "); .) .

  EqualOp
  =   "=="                          (. CodeGen.append(" == "); .)
    | "!="                          (. CodeGen.append(" != "); .) .

  RelOp
  =   "<"                           (. CodeGen.append(" < "); .)
    | "<="                          (. CodeGen.append(" <= "); .)
    | ">"                           (. CodeGen.append(" > "); .)
    | ">="                          (. CodeGen.append(" >= "); .) .

  AssignOp
  =   "="                           (. CodeGen.append(" = "); .) .

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

  StringConst<out String name>
  = stringLit                       (. name = token.val; .) .

  CharConst<out String name>
  = charLit                         (. name = token.val; .) .

  IntConst<out String name>
  = number                          (. name = token.val; .) .

END PrettyParva.
```

Your attention is drawn to the use of the `indented` variable for fine control of the spacing.

Moving right along ….

A few years ago the compiler course was done using small languages that more closely resembled Pascal or Modula-2 code than C-like languages. Here is a syntactic description of one that I invented and refined, which I called Mikra (*Mikra* being the Greek word for "little", just as *Parva* is the Latin word for "little").

```
COMPILER Mikra $CN
/* Mikra level 1.5 grammar
   P.D. Terry, Rhodes University, 2011 */

CHARACTERS
  lf           = CHR(10) .
  backslash    = CHR(92) .
  control      = CHR(0) .. CHR(31) .
  letter       = "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz" .
  digit        = "0123456789" .
  nonZeroDigit = "123456789" .
  stringCh     = ANY - '"' - control - backslash .
  charCh       = ANY - "'" - control - backslash .
  printable    = ANY - control .

TOKENS
  identifier   = letter { letter | digit | "_" { "_" } ( letter | digit ) } .
  number       = "0" | nonZeroDigit { digit } .
  stringLit    = '"' { stringCh  | backslash printable } '"' .
  charLit      = "'" ( charCh    | backslash printable ) "'" .

COMMENTS FROM "(*" TO "*)"
IGNORE CHR(9) .. CHR(13)

PRODUCTIONS
  Mikra              = "program" Ident ";" Block Ident "." .
  Block              = { ConstDeclarations | VarDeclarations }
                       "begin" StatementSequence "end" .
  StatementSequence  = Statement { ";" Statement } .
  Statement          = [   AssignmentStatement | IfStatement     | WhileStatement
                         | RepeatStatement     | LoopStatement   | ForStatement
                         | HaltStatement       | BreakStatement  | IncOrDecStatement
                         | ReadStatement       | WriteStatement
                       ] .
```

```
  ConstDeclarations   = "const" OneConst { "," OneConst } ";" .
  OneConst            = Ident "=" Constant .
  Constant            = IntConst | CharConst | "true" | "false" | "null" .
  VarDeclarations     = "var" VarList ";" { VarList ";" } .
  VarList             = Ident { "," Ident } ":" Type .
  Type                = [ "array" "of" ] BasicType .
  BasicType           = "int" | "bool" | "char" .
  AssignmentStatement = Designator AssignOp Expression .
  Designator          = Ident [ "[" Expression "]" ] .
  IfStatement         = "if" Condition "then" StatementSequence
                        { "elsif" Condition "then" StatementSequence }
                        [ "else" StatementSequence ]
                        "end" .
  WhileStatement      = "while" Condition "do" StatementSequence "end" .
  RepeatStatement     = "repeat" StatementSequence "until" Condition .
  LoopStatement       = "loop" StatementSequence "end" .
  ForStatement        = "for" Ident ":=" Expression ( "to" | "downto" ) Expression
                         "do" StatementSequence "end" .
  HaltStatement       = "halt" .
  BreakStatement      = "break" .
  IncOrDecStatement   = ( "inc" | "dec" ) "(" Designator ")" .
  ReadStatement       = "read"  ReadList | "readLn" [ ReadList ] .
  ReadList            = "(" ReadElement { "," ReadElement } ")" .
  ReadElement         = StringConst | Designator .
  WriteStatement      = "write" WriteList | "writeLn" [ WriteList ] .
  WriteList           = "(" WriteElement { "," WriteElement } ")" .
  WriteElement        = StringConst | Expression .
  Condition           = Expression .
  Expression          = AddExp [ RelOp AddExp ] .
  AddExp              = [ "+" | "-" ] MulExp { AddOp MulExp } .
  MulExp              = Factor { MulOp Factor } .
  Factor              =   Designator | Constant | "new" BasicType "[" Expression "]"
                        | "char" "(" Expression ")" | "int"  "(" Expression ")"
                        | "(" Expression ")" | NotOp Factor .
  NotOp               = "not" .
  MulOp               = "*" | "/" | "mod" | "and" .
  AddOp               = "+" | "-" | "or" .
  RelOp               = "=" | "<>" | "<" | "<=" | ">" | ">=" .
  AssignOp            = ":="  .
  Ident               = identifier .
  StringConst         = stringLit .
  CharConst           = charLit .
  IntConst            = number .
END Mikra.
```

Mikra programs look quite like their Parva equivalents, and the "strengths" of the two languages, and their semantics, are essentially the same.

There are some very obvious simple differences in syntax - for example, Mikra uses the operators `:=, =, <>` and `mod, and, or, not` where Parva uses `=, ==, !=` and `%, &&, ||, !` respectively, while the Mikra operator precedences follow the system used in Pascal and Modula-2 rather than the one used in Java, C# and Parva (as refined in your practical course).

Casting is done with the notation `int(ch)` and `char(n)` in place of the `(int) ch` and `(char) n` used in the C family of languages.

Mikra only supports one form of the *for* loop.

As an illustrative example, here is a Mikra program which does the same thing as the Parva program repeatedly used as an example in The Book (see page 90, for example). There are some further familiar looking small examples in the updated exam kits now available on the web page.

```
program Voter;
(* Simple voter example from page 90 of the book coded into Mikra 2011 *)
  const
    votingAge = 18,
  var
    age, eligible, total : int;
    allEligible : bool;
    canVote : bool;
    voters : array of int;
  begin
    eligible := 0;
    total := 0;
    allEligible := true;
    voters := new int[100];
    readLn(age);
    while age > 0 do
      canVote := age >= votingAge;
      allEligible := allEligible and canVote;
      if canVote then
        voters[eligible] := age;
        inc(eligible);
        total := total + voters[eligible - 1];
      end;
      readLn(age);
    end;
    writeLn(eligible, " voters.  Average age is ", total / eligible);
    if allEligible then
      writeLn("Everyone was above voting age")
    end
  end Voter.
```

To prepare yourself to answer the examination tomorrow, you are encouraged to study these grammars in depth and, if you like, to experiment with them further. Questions in the examination will probe this understanding, and you might be called on to make some modifications and extensions to one or other or both.

That gives me a lot of scope, does it not? Here are some things to think about. An exam some years ago required people to produce HTML output - how might you modify the Parva prettyprinter to do that, highlighting the key words in some colour? Or suppose one wanted to choose the indentation level by using a pragma, rather than using a fixed value of 2? Or suppose somebody had the misguided idea that they wanted to convert Mikra programs from a syntax where the key words were given in lower case to one where they were given in UPPER CASE? Or suppose one wanted to generate a pretty version of a program, adding line numbers as in the example below, possibly followed by a cross reference listing of all the identifiers, and a statement count.

```
/*   1 */  void main() {
/*   2 */    int b, c, d;
/*   3 */    b = 10;
/*   4 */    while (b > 0) {
/*   5 */      write("b = ", b);
/*   6 */      b--;
/*   7 */      if (b == 4)
/*   8 */        write(" hit four!\n");
/*   9 */      else
/*  10 */        write(" on we go\n");
/*  11 */    }
/*  12 */  }
```

As before, you are quite at liberty to continue discussions with your class mates, but not with staff or demonstrators.

I have a vivid imagination. Do you?

Have fun, and get a good night's sleep!

You will receive printed copies of these grammars tomorrow, as well as copies in machine-readable form.