RHODES UNIVERSITY

November Examinations - 2003

Computer Science 301 - Paper 2 - solutions

Examiners:	Time 3 hours
Prof P.D. Terry	Marks 180
Prof E.H. Blake	Pages 10 (please check!)

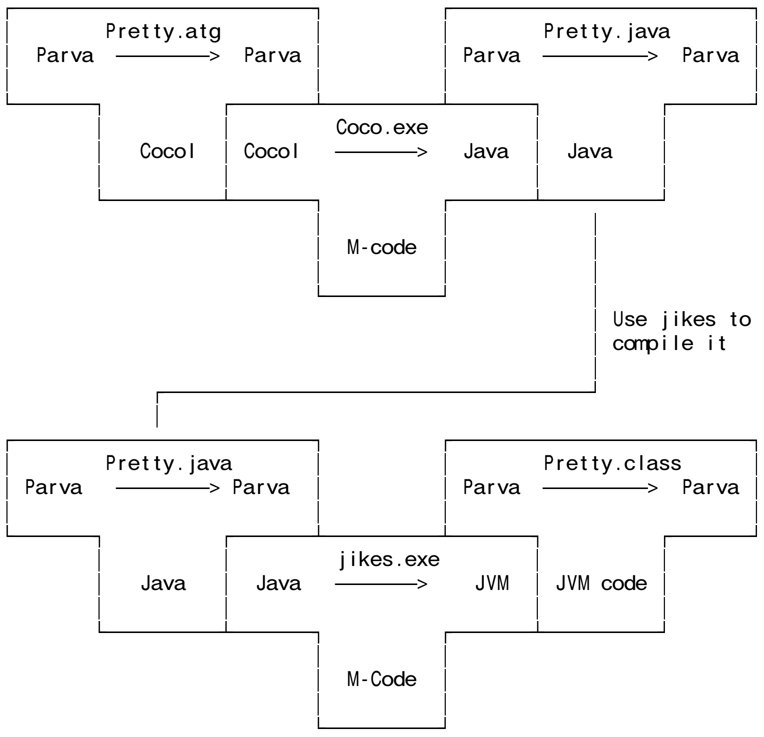
Answer all questions. Answers may be written in any medium except red ink.

(For the benefit of future readers of this paper, various free information was made available to the students 24 hours before the formal examination. This included the full text of Section B, summaries of useful Java library classes, and the source listing of a Parva compiler. During the examination, candidates were given machine executable versions of the Coco/R compiler generator, access to a computer and machine readable copies of the questions.)

Section A [100 marks]

A1. (a) A pretty-printer is a form of compiler that will take a source program and reformat it to make it look "pretty", without actually changing the source in any way other than by inserting some spaces and line breaks and deleting other spaces and line breaks.

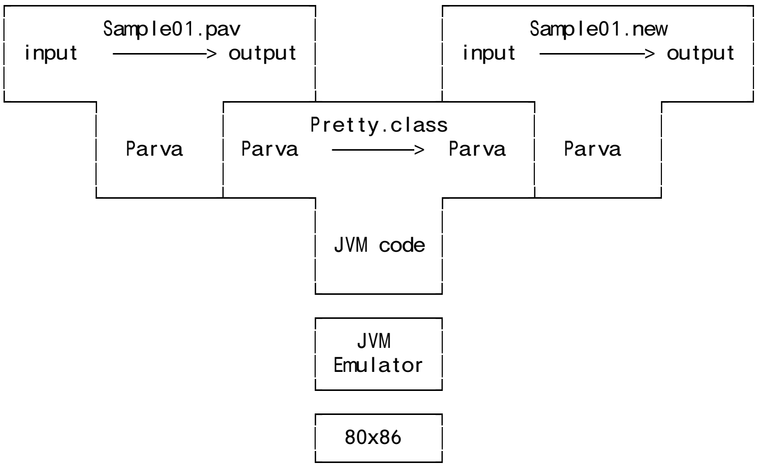
Suppose you are asked to write a pretty-printer for Parva, using the familiar Java version of Coco/R to provide the scanner, parser and driver components of the system. Complete the T diagrams below to show the steps you would follow to do this. *(A copy of these T-diagrams appears as an attachment to this paper, which you can detach and hand in with your answer book.)* [6 marks]



(b) Do you suppose a pretty-printer would need to make use of a symbol table handler? Give a reason for your answer. [3 marks]

Not really. Pretty printers can be concerned only with improving the appearance based purely on the context-free syntax. There would be no real need to do semantic analysis. I suppose a very sophisticated pretty printer might use semantic information, though I cannot see how.

(c) Develop a further T diagram to show how the pretty-printer would be applied to process a Parva program like SAMPLE01.PAV. [3 marks]



A2. Formally, a grammar G is defined by a quadruple $\{ N, T, S, P \}$ with the four components

(a) N - a finite set of **non-terminal** symbols,
(b) T - a finite set of **terminal** symbols,
(c) S - a special **goal** or **start** or **distinguished** symbol,
(d) P - a finite set of **production rules** or, simply, **productions**.

where a production relates to a pair of strings, say α and β, specifying how one may be transformed into the other:

$$\alpha \rightarrow \beta \text{ where } \alpha \in (N \cup T)^* N (N \cup T)^* \ , \ \beta \in (N \cup T)^*$$

and we can then define the language $L(G)$ produced by the grammar G by the relation

$$L(G) = \{ w \mid S \Rightarrow^* w \land w \in T^* \}$$

(a) In terms of this style of notation, define **precisely** (*that is to say, mathematically; we do not want a long essay or English description*) what you understand by [2 marks each]

(1) FIRST(σ) where $\sigma \in (N \cup T)^+$

$a \in \text{FIRST}(\sigma)$ if $\sigma \Rightarrow^* a\tau$ $(a \in T \ ; \ \sigma, \tau \in (N \cup T)^*)$

(2) FOLLOW(A) where $A \ \varepsilon \ N$

$a \in \text{FOLLOW}(A)$ if $S \Rightarrow^* \xi A a \zeta$ $(A, S \in N \ ; \ a \in T \ ; \ \xi, \zeta \in (N \cup T)^*)$

(b) State the two rules that must be satisfied by the productions of the grammar in order for it to be classified as an LL(1) grammar. [6 marks]

Rule 1

For each non-terminal $A_i \in N$ that admits alternatives

$$A_i \rightarrow \xi_{i1} \mid \xi_{i2} \mid \ . \ . \ . \ \xi_{in}$$

the sets of initial terminal symbols of all strings that can be generated from each of the alternative ξ_{ik} must be disjoint, that is

$$\text{FIRST}(\xi_{ij}) \cap \text{FIRST}(\xi_{ik}) = \varnothing \qquad \text{for all } j \neq k$$

Rule 2

For each non-terminal $A_i \in N$ that admits alternatives

$$A_i \rightarrow \xi_{i1} \mid \xi_{i2} \mid \ldots \xi_{in}$$

and but where $\xi_{ik} \Rightarrow \varepsilon$ for some k, the sets of initial terminal symbols of all sentences that can be generated from each of the ξ_{ij} for $j \neq k$ must be disjoint from the set FOLLOW(A_i) of symbols that may follow any sequence generated from A_i, that is

$$\text{FIRST}(\xi_{ij}) \cap \text{FOLLOW}(A_i) = \varnothing, \qquad j \neq k$$

or, rather more loosely,

$$\text{FIRST}(A_i) \cap \text{FOLLOW}(A_i) = \varnothing$$

(c) The Cocol grammar below (`RE.ATG`) describes a sequence of Regular Expressions (written one to a line) using the conventions discussed during the course.

```
COMPILER RE   /* Regular expression grammar */

CHARACTERS
  lf       = CHR(10) .
  control  = CHR(1) .. CHR(31) .
  noquote1 = ANY - control - "'" .
  noquote2 = ANY - control - '"' .
  meta     = "()*|[]-?+" .
  simple   = ANY - control - "'" - '"' - meta .

IGNORE  CHR(1) .. CHR(9) + CHR(11) .. CHR(31)

TOKENS
  atomic  = simple .
  escaped = "'" noquote1 "'" | '"' noquote2 '"' .
  EOL     = lf .

PRODUCTIONS
  RE         = { Expression EOL } EOF .
  Expression = Term { "|" Term } .
  Term       = Factor { Factor } .
  Factor     = Element [ "*" | "?" | "+" ] .
  Element    = Atom | Range | "(" Expression ")" .
  Range      = "[" OneRange { OneRange } "]" .
  OneRange   = Atom [ "-" Atom ] .
  Atom       = atomic | escaped .
END RE.
```

Determine the following FIRST and FOLLOW sets: [8 marks]

FIRST(*Factor*)	FOLLOW(*Factor*)
FIRST(*OneRange*)	FOLLOW(*OneRange*)

It follows fairly simply that

FIRST(*Factor*) = { atomic , escaped , "(" , "[" }
FOLLOW(*Factor*) = { atomic , escaped , EOL , "|" , "(" , ")" , "[" }

FIRST(*OneRange*) = { atomic , escaped }
FOLLOW(*OneRange*) = { atomic , escaped , "]" }

(d) Does the production for *Factor* lead to LL(1) errors? Justify your answer. [3 marks]

No it does not. *Factor* is not nullable. The bit that follows is, but we could rewrite the system of productions to have

Factor	= *Element TailFactor* .
TailFactor	= "*" | "?" | "+" | ε .

Rule 1 is easily seen to be satisfied for *TailFactor*, and for Rule 2 we have that

FIRST(*TailFactor*) = { ")" , "?" , "+" }

while

FOLLOW(TailFactor) = FOLLOW(Factor)
= { atomic , escaped , EOL , "|" , "(" , ")" , "[" }

so that Rule 2 is satisfied as well.

A3. Chomsky classified grammars into four types, sometimes known as types 0, 1, 2, 3. The classification depended on the form of the productions. Computer Scientists tend to use more descriptive names for these types - for example a type 0 grammar is often called "unrestricted".

(a) What are the numbers corresponding to the other types? [3 marks]

Your answer should take the form

A Regular grammar is also called ... Type 3
A Context-free grammar is also called ... Type 2
A Context-sensitive grammar is also called ...Type 1

(b) Consider the following simple grammars defined on a vocabulary with two non-terminals { A, B }, four terminals { a, b, c, d }. Assume that A is the goal symbol.

The grammars are not supposed to describe any sensible languages, of course. Classify each grammar as being of one of the above types. [6 marks]

```
Grammar 1
    A   = "a" B | "b" .
    B   = "c" A | "d" .

Grammar 2
    A   = "a" B | A B "b" .
    A B = "a" B "b" .
    B   = B "a" "b" "c" "d" .

Grammar 3
    A   = "a" A | "b" .
    B   = "b" A | "c" A "d" .
```

Grammar 1 is Regular (and by implication is of all the other types as well, though it would clearly be classified as Regular)

Grammar 2 is Context-sensitive because of the second production. (By implication it is also of Type 0, but would clearly be classified as Context-sensitive.

Grammar 3 is Context-free as each production has only a single non-terminal on the left. (By implication it is also of Types 0 and 1, but would clearly be classified as Context-free).

(c) Do any of the grammars incorporate "useless productions"? If so, which grammar(s) have useless productions, and which productions are the useless ones? [3 marks]

The second production in Grammar 3 is useless, as *B* cannot be reached in any sentential form derived from an initial *A*

A4. You will recall that my brother occasionally appears on television in various roles. The following set of productions attempts to describe some exciting viewing on the SABC.

```
SABCTV      = { Programme } "Closedown" .
Programme   = "Announcement" { QualityShow  }
              [ "Reminder" ] /* you know it's the right thing to do */ .
QualityShow = ( "Frasier" | "MyFamily" | "Generations" ) { Advert } .
Advert      = "CTMTiles" | "Domestos" | "VodaCom" | "Nando's" | "LGDigital" .
```

Assuming that you have available a suitable scanner method called `getSym` that can recognize the terminals of this language and classify them appropriately as members of an enumeration

```
EOFSym, noSym, closeSym, announceSym, remindSym, frasierSym,
myFamilySym, generationsSym, ....
```

develop a hand-crafted recursive descent **parser** for watching a typical boring evening on television. Your parser can take drastic action if an invalid sequence is detected - simply produce an appropriate error message and then terminate parsing (in effect, switch off the television set). *(You are not required to write any code to implement the getSym method, nor need you take any precautions to check on my brother's appearances.)* [20 marks]

This can be done in several ways, one of which would be:

```
static void accept(int wantedSym, String message) {
  if (sym.kind == wantedSym) getSym();
  else {
    IO.writeString(message); System.exit(1);
  }
}

static void abort(String message) {
  IO.writeString(message); System.exit(1);
}

SymSet firstShow
  = new SymSet(new int[] {frasierSym, myFamilySym, generationsSym});
SymSet firstAdvert
  = new SymSet(new int[] {ctmtileSym, domestosSym, vodacomSym, nandoSym});

static void SABCTV() {
//  SABCTV = { ProgrammelgdigitalSym:  } "Closedown" .
  while (sym.kind == announceSym) Programme();
  accept(closeSym, "close down expected");
}

static void Programme() {
//  Programme = "Announcement" { QualityShow  } [ "Reminder" ] .
  accept(announceSym, "announcement expected");
  while (firstShow.contains(sym.kind)) QualityShow();
  if (sym.kind == remindSym) getSym();
}

static void QualityShow() {
//  QualityShow = ( "Frasier" | "MyFamily" | "Generations" ) { Advert } .
  switch (sym.kind) {
    case frasierSym:
    case myFamilySym:
    case generationsSym:
      getSym();
      break;
    default:
      abort("unexpected quality show");
      break;
  }
  while (firstAdvert.contains(sym.kind)) Advert();
}
```

```
static void Advert() {
//  Advert = "CTMTiles" | "Domestos" | "VodaCom" | "Nando's" | "LGDigital" .
  switch (sym.kind) {
    case ctmtileSym:
    case domestosSym:
    case vodacomSym:
    case nandoSym:
    case lgdigitalSym:
      getSym();
      break;
    default:
      abort("bad advert");
      break;
  }
}
```

A5. Consider the grammar given in question A2 for describing regular expressions. For this next question ignore any potential LL(1) problems with the grammar.

How would you add appropriate actions and attributes so that you could generate a program that would parse a sequence of regular expressions and report on the **alphabets** used in each one. For example, given input like

```
a | b c d | ( x y z )*
[a-g A-G] [x - z]?
a? "'" z+
```

the output should be something like

```
Alphabet = a b c d x y z
Alphabet = A B C D E F G a b c d e f g x y z
Alphabet = ' a z
```

A machine readable version of the grammar can be found in the exam kit (`RE.ATG`), should you prefer to develop a machine readable solution. A printed version of the productions is given as an attachment. [24 marks]

Hints:

It will be simplest to use a simple static array to store each alphabet.

"Keep your grammar as simple as possible, but no simpler".

Pay particular attention to where you introduce the actions/attributes; do not simply write them in random positions in the grammar.

A complete solution follows:

```
import Library.*;

COMPILER RE
/* Regular expression grammar */

static boolean[] alphabet = new boolean[256];

CHARACTERS
  lf       = CHR(10) .
  control  = CHR(1) .. CHR(31) .
  noquote1 = ANY - control - "'" .
  noquote2 = ANY - control - '"' .
  meta     = "()*|[]-?+" .
  simple   = ANY - control - "'" - '"' - meta .

IGNORE  CHR(1) .. CHR(9) + CHR(11) .. CHR(31)

TOKENS
  atomic  = simple .
  escaped = "'" noquote1 "'" | '"' noquote2 '"' .
  EOL     = lf .
```

```
PRODUCTIONS
  RE                              (. int i; .)
  = {                             (. for (i = 1; i < 256; i++)
                                       alphabet[i] = false; .)
      Expression EOL              (. IO.write("Alphabet = ");
                                     for (i = 1; i < 256; i++)
                                       if (alphabet[i]) IO.write( (char) i);
                                     IO.writeLine(); .)
    } EOF .

  Expression
  = Term { "|" Term } .

  Term
  = Factor { Factor } .

  Factor
  = Element [ "*" | "?" | "+" ] .

  Element                         (. char ch; .)
  =   Atom<↑ch>                   (. alphabet[ch] = true; .)
    | Range | "(" Expression ")" .

  Range
  = "[" OneRange { OneRange } "]" .

  OneRange                        (. char ch, ch1, ch2; .)
  = Atom<↑ch1>                    (. alphabet[ch1] = true; .)
    [ "-" Atom<↑ch2>              (. if (ch2 < ch1)
                                       SemError("invalid range");
                                     for (ch = ch1; ch <= ch2; ch++)
                                       alphabet[ch] = true; .)
    ] .

  Atom<↑char ch>                  (. ch = 0; .)
  = (   atomic                    (. ch = token.str.charAt(0); .)
      | escaped                   (. ch = token.str.charAt(1); .)
    )
  .
END RE.
```

A6. Consider the following small Parva program

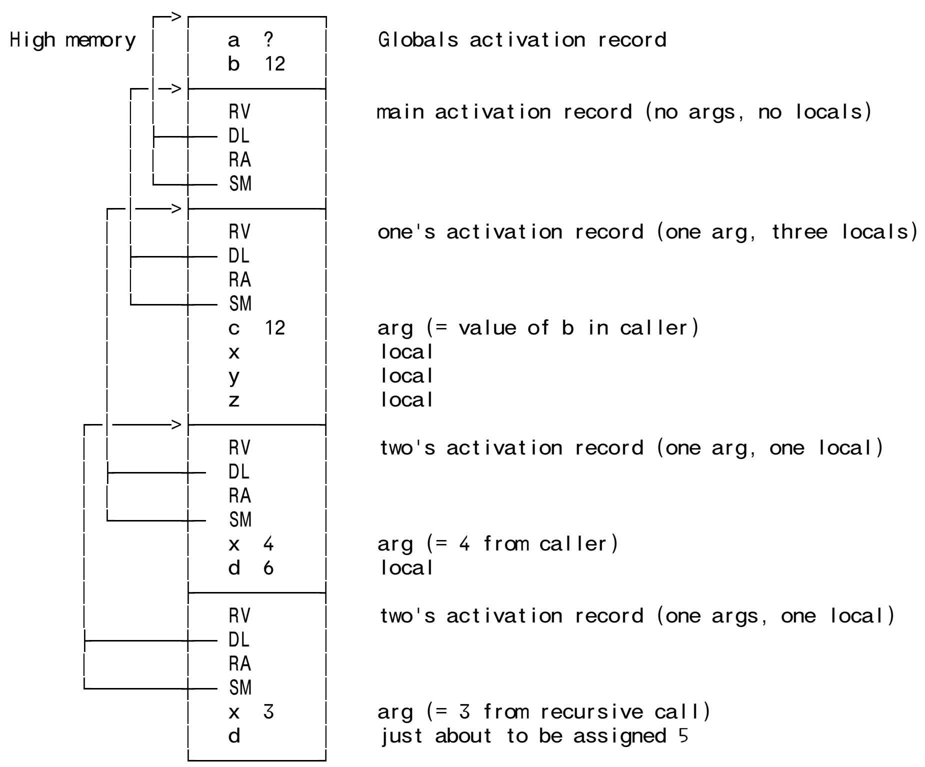
```
int a, b = 12;

void two(int x) {
  // here ------------------------------
  int d = x + 2;
  if (x > 1) two(x-1);
  write(d);
}

void one(c) {
  int x, y, z;
  two(4);
}

void main () {
  one(b);
}
```

(a) The programmer has introduced the identifier `x` in more than one function. Does this represent an error? Explain your answer. [3 marks]

There is no problem with this. The scope of the first `x` is within function two while the scope of the second `x` is within function `one`.

(b) Assuming that the program is, in fact, correct, show that you understand the general concept of using activation records to implement function calls, by drawing a diagram showing what these records would look like at run-time when the point marked `// here` is reached for the second time. [8 marks]

Conceptually the setup would be something like this:


High memory
a ?
b 12
Globals activation record
RV
DL
RA
SM
main activation record (no args, no locals)
RV
DL
RA
SM
c 12
x
y
z
one's activation record (one arg, three locals)
arg (= value of b in caller)
local
local
local
RV
DL
RA
SM
x 4
d 6
two's activation record (one arg, one local)
arg (= 4 from caller)
local
RV
DL
RA
SM
x 3
d
two's activation record (one args, one local)
arg (= 3 from recursive call)
just about to be assigned 5

Section B [80 marks]

Please note that there is no obligation to produce a machine readable solution for this section. Coco/R and other files are provided so that you can enhance, refine, or test your solution if you desire. If you choose to produce a machine readable solution, you should create a working directory, unpack EXAM.ZIP, modify any files that you like, and then copy all the files back onto an exam folder on the network.

A minor crisis has developed in the Computer Science department. This year, as an experiment, the first year students have been taught programming using the influential Parva language. This has proved eminently successful, and the students are due to write a practical examination on Monday. Today is Sunday. Earlier this morning the lecturer in charge of the class discovered, to her horror, that the existing Parva compiler makes provision only for integer and boolean data types, and not for the character type that would be required if students were asked to write code like the following:

```
void main () {
// Read a sentence and write it backwards
  char[] sentence = new char[1000];
  int i = 0;
  char ch;
  read(ch);
  while (ch != '.') {  // input loop
    sentence[i] = ch;
    i = i + 1;
    read(ch);
  }
  while (i > 0) {      // output loop
    i = i - 1;
    write(sentence[i]);
  }
}
```

or similar applications which require simple character manipulations, including the ability to mix integers and characters appropriately in expressions, detect type incompatibilities, cast between integers and characters when required, increment and decrement character variables and so on.

She had intended to set a simple problem in which the students were required to print a set of multiplication tables. When testing the solution, she made a typing error and submitted the following code to the compiler, with rather unexpected results:

```
void main () {
// Print a set of multiplication tables
  int i, j;
  for i = 1 to 10 do { // <---------------- should be j not i
    for i = 1 to 10 do write (i * j, "\t");
    write("\n");
  }
}
```

But worse is still to come! The lecturer has also been under the illusion that the key word *const* is used in the sense that it is used in C# and the word *final* is used in Java, namely as a modifier in a variable declaration that indicates that when a variable is declared it can be given a value which can then not be modified later. On checking the grammar for Parva she comes across the productions

```
Statement         = Block | ConstDeclarations | VarDeclarations | ...
ConstDeclarations = "const" OneConst { "," OneConst } ";" .
OneConst          = identifier "=" Constant .
Constant          = number | charLit | "true" | "false" | "null" .
VarDeclarations   = Type OneVar { "," OneVar } ";" .
OneVar            = identifier [ "=" Expression ] .
```

which she realizes will not correctly handle code of the form she needs, like

```
const int max = 2;
int i = 5;
const int iPlusMax = i + max;
```

In desperation she turns to the members of the third year class, imploring them to come up with a reliable new version of the Parva compiler in 24 hours that will provide the character type, treat the keyword *const* as she intends, and detect situations where attempts are made to alter a for loop control within the loop body. She offers to provide them with a few examples of the sort of code that she expects the compiler to be able to handle. For example, besides the example involving character types, she provides a commented test program:

```
void main () {
// Rather meaningless program to illustrate variable declarations
// that incorporate a const modifier
  const int i = 10;              // acceptable
  const char terminator;         // unacceptable
  const int[] list = new int[i]; // acceptable
  int j = i;
  while (j < 100) {
    const int k = j;             // acceptable
    read(i);                     // unacceptable
    i = j;                       // unacceptable
    list[4] = 5;                 // acceptable
    j = j + list[4];             // acceptable
    list = new int[12];          // unacceptable
  }
}
```

Rise to the challenge! Produce a new compiler, and avoid a disaster for the department.

Note:

(a) In the exam kit (`EXAM.ZIP`) you will find the executables and support files for Coco/R, as used in the practical course. You will also find the complete attribute grammar and support files for the Parva language as developed by the end of your lecture and practical course.

(b) In the exam kit will be found some other example programs like those above to assist in your understanding of the requirements and your development of the system, and an executable derived from a model solution that you are free to use to compile these example programs or any others you may devise.

(c) Depending on your approach, your solution may require modifications to any or all of the grammar and support files in the exam kit.

(d) It is not particularly difficult to provide "first approximations" to these extensions and modifications. The examiners will, however, be looking for maturity in your solution and, in particular for signs that you have seen past the obvious "quick fix".

Clearly we have to be able to recognize and represent the types denoted by `char` and `char[]`. This necessitates a simple change to the *Entry* class, as shown. To handle the new form of constant declaration and the problem of checking that a for loop control variable is not abused it is convenient to add a new field `canChange` to both the *Entry* and the *DesType* classes, and to modify the *DesType* constructor accordingly:

```
          class Entry {
            public static final int
              Con = 0,                      // identifier kinds
              Var = 1,
              Fun = 2,

              noType   =  0,                // identifier types.  The numbering is significant
              nullType =  2,                // as array types are denoted by these numbers + 1
              intType  =  4,
              boolType =  6,
  ******      charType =  8,
  ******      voidType = 10;

            public int     kind;
            public String  name;
            public int     type;
            public int     value;           // constants
            public int     offset;          // variables
            public Entry   nextInScope;     // link to next entry in current scope
            public boolean found;           // true for all except sentinel entry
  ******    public boolean canChange;       // true except for constants and for loop controls
          } // Entry


          class DesType {
          // Objects of this type are associated with l-value and r-value designators
            public Entry entry;          // the identifier properties
            public int type;             // designator type (not always the entry type)
  ******    public boolean canChange;

            public DesType(Entry entry) {
              this.entry = entry;
              this.type = entry.type;
  ******      this.canChange = entry.canChange;
            }
          } // DesType
```

Corresponding to this is the need to modify the *BasicType* production to recognize the keyword `char`:

```
            BasicType<↑int type>            (. type = Entry.noType; .)
            =   "int"                       (. type = Entry.intType; .)
              | "bool"                      (. type = Entry.boolType; .)
  ******      | "char"                      (. type = Entry.charType; .)
            .
```

The new form of variable declaration requires that we eliminate the old *ConstDeclaration* production completely, so that the production for *Statement* becomes simply

```
            Statement<↑int execCount, StackFrame frame>
                                            (. execCount = 0; .)
            =  SYNC (   Block<↑execCount, frame>
                      | VarDeclarations<frame>
                      | ";"                 (. if (warnings) Warning("empty statement"); .)
                      |                     (. execCount = 1; .)
                        ( Assignment
                          | IfStatement<frame>
                          | WhileStatement<frame>
                          | DoWhileStatement<frame>
                          | ForStatement<frame>
                          | BreakStatement
                          | ContinueStatement
                          | HaltStatement
                          | ReturnStatement
                          | ReadStatement
                          | WriteStatement
                          | "stackdump" ";"       (. if (debug) CodeGen.dump(); .)
                        )
                    ) .
```

The *VarDeclarations* production is changed to incorporate an optional `const` modifier, which is used to set a flag that can be passed to the *OneVar* production:

```
         VarDeclarations<StackFrame frame>     (. int type;
******                                            boolean canChange = true; .)
******   =  [ "const"                          (. canChange = false; .)
******      ] Type<↑type>
******      OneVar<frame, type, canChange>
******      { WEAK "," OneVar<frame, type, canChange> }
            WEAK ";" .
```

The *OneVar* production inherits this attribute and marks the entries in the symbol table accordingly. Note that a defining *Expression* is now required for variables marked as `const`, needing a further constraint check:

```
         OneVar<StackFrame frame, int type, boolean canChange>
                                               (. int expType; .)
         =                                     (. Entry var = new Entry(); .)
            Ident<↑var.name>                   (. var.kind = Entry.Var;
                                                  var.type = type;
******                                            var.canChange = canChange;
                                                  var.offset = frame.size;
                                                  frame.size++; .)
            ( "="                              (. CodeGen.loadAddress(var); .)
              Expression<↑expType>             (. if (!assignable(var.type, expType))
                                                    SemError("incompatible types in assignment");
                                                  CodeGen.assign(var.type); .)
******      |                                  (. if (!canChange)
                                                    SemError("defining expression required"); .)
            )                                  (. Table.insert(var); .)
         .
```

The `canChange` field is retrieved by the production that parses a designator, and used to mark the designator object accordingly. A subtle point is that if a reference variable has been marked as immutable, it may not be altered, although an element of the array referred to can still be changed:

```
         Designator<↑DesType des>              (. String name;
                                                  int indexType; .)
         =  Ident<↑name>                       (. Entry entry = Table.find(name);
                                                  boolean notDeclared = !entry.found;
                                                  if (notDeclared) {
                                                    SemError("undeclared identifier");
                                                    entry = new Entry(); // new is critical
                                                    entry.name = name;
                                                    entry.kind = Entry.Var;
                                                    entry.type = Entry.noType;
                                                    entry.offset = 0;
                                                    Table.insert(entry);
                                                  }
                                                  des = new DesType(entry);
                                                  if (entry.kind == Entry.Var)
                                                    CodeGen.loadAddress(entry); .)
******      [  "["                             (. des.canChange = true;
                                                  if (notDeclared) entry.type++;
                                                  else if (isRef(des.type)) des.type--;
                                                  else SemError("unexpected subscript");
                                                  if (entry.kind != Entry.Var)
                                                    SemError("unexpected subscript");
                                                  CodeGen.dereference(); .)
                   Expression<↑indexType>      (. if (!isArith(indexType))
                                                    SemError("invalid subscript type");
                                                  CodeGen.index(); .)
               "]"
            ] .
```

The immutable property of a designator must be tested in various places, in particular in *ForStatement, Assignment* and *ReadElement*. The last of these is shown next, along with the change needed to allow code to be generated for reading single characters:

```
         ReadElement                           (. String str;
                                                  DesType des; .)
         =   StringConst<↑str>                 (. CodeGen.writeString(str); .)
           | Designator<↑des>                  (. if (des.entry.kind != Entry.Var)
                                                    SemError("wrong kind of identifier");
******                                            if (!des.canChange)
                                                    SemError("may not alter this variable");
```

```
                                                      switch (des.type) {
                                                        case Entry.intType:
                                                        case Entry.boolType:
******                                                  case Entry.charType:
                                                          CodeGen.read(des.type); break;
                                                        default:
                                                          SemError("cannot read this type"); break;
                                                      } .) .
```

A similar addition is needed to *WriteElement* to allow for single characters to be output:

```
          WriteElement                      (. int expType;
                                               String str; .)
          =   StringConst<↑str>             (. CodeGen.writeString(str); .)
            | Expression<↑expType>          (. switch (expType) {
                                                 case Entry.intType:
                                                 case Entry.boolType:
******                                           case Entry.charType:
                                                   CodeGen.write(expType); break;
                                                 default:
                                                   SemError("cannot write this type"); break;
                                               } .) .
```

The major part of this exercise is concerned with the changes needed to apply various constraints on operands of the `char` type. Essentially it ranks as an arithmetic type, in that expressions of the form

character + character
character > character
character + integer
character > integer

are all allowable. This can be handled by modifying the helper methods in the parser as follows:

```
          static boolean isArith(int type) {
******      return type == Entry.intType || type == Entry.charType || type == Entry.noType;
          }

******    static boolean compatible(int typeOne, int typeTwo) {
          // Returns true if typeOne is compatible (comparable) with typeTwo
            return    typeOne == typeTwo
******             || isArith(typeOne) && isArith(typeTwo)
                   || typeOne == Entry.noType || typeTwo == Entry.noType
                   || isRef(typeOne) && typeTwo == Entry.nullType
                   || isRef(typeTwo) && typeOne == Entry.nullType;
          }
```

However, assignment compatibility is more restricted

integer = integer
integer = character
character = character

is allowed, but

character = integer

is not allowed. This may be checked within the *Assignment* and *ForStatement* productions with the aid of a further helper method `assignable`:

```
******    static boolean assignable(int typeOne, int typeTwo) {
******    // Returns true if typeOne may be assigned a value of typeTwo
******      return    typeOne == typeTwo
******             || typeOne == Entry.intType && typeTwo == Entry.charType
******             || typeOne == Entry.noType || typeTwo == Entry.noType
******             || isRef(typeOne) && typeTwo == Entry.nullType;
******    }
```

As well as checking this new aspect of compatibility, the *Assignment* production also has to check that the variable specified as the designated target can, in fact, be changed. Furthermore, the compound `+=`, `-=`, `*=`, `/=` and `%=` operators can only be applied when the designated target is of strictly integer type:

```
            Assignment                         (. int expType;
                                                  DesType des;
                                                  int op;
                                                  Label shortcircuit = new Label(!known);.)
            =  Designator<↑des>                (. if (des.entry.kind != Entry.Var)
                                                    SemError("invalid assignment");
   ******                                         if (!des.canChange)
   ******                                           SemError("may not alter this variable"); .)
               (  AssignOp<↑op>                (. switch (op) {
                                                    case CodeGen.nop:
                                                      break;
                                                    case CodeGen.add:
                                                    case CodeGen.sub:
                                                    case CodeGen.mul:
                                                    case CodeGen.div:
                                                    case CodeGen.rem:
   ******                                             if (des.type != Entry.intType)
                                                        SemError("integer destination required");
                                                      CodeGen.duplicate();
                                                      CodeGen.dereference();
                                                      break;
                                                    case CodeGen.and:
                                                    case CodeGen.or:
                                                      if (!isBool(des.type))
                                                        SemError("boolean destination required");
                                                      CodeGen.duplicate();
                                                      CodeGen.dereference();
                                                      CodeGen.booleanOp(shortcircuit, op);
                                                      break;
                                                  } .)
   ******         Expression<↑expType>         (. if (!assignable(des.type, expType))
                                                    SemError("incompatible types in assignment");
                                                  switch (op) {
                                                    case CodeGen.nop:
                                                      break;
                                                    case CodeGen.and:
                                                    case CodeGen.or:
                                                      shortcircuit.here();
                                                      break;
                                                    case CodeGen.add:
                                                    case CodeGen.sub:
                                                    case CodeGen.mul:
                                                    case CodeGen.div:
                                                    case CodeGen.rem:
                                                      CodeGen.binaryOp(op);
                                                      break;
                                                  }
                                                  CodeGen.assign(des.type); .)
                 | "++"                        (. if (!isArith(des.type))
                                                    SemError("arithmetic type required");
                                                  CodeGen.increment(des.type); .)
                 | "--"                        (. if (!isArith(des.type))
                                                    SemError("arithmetic type required");
                                                  CodeGen.decrement(des.type); .)
               )
               WEAK ";" .
```

Assignment compatibility and assignment to control variables that are not marked as `const` are both checked within the production for *ForStatement*. There is a further refinement needed - the control variable's original `canChange` status must be preserved before the *Statement* forming the body of the loop is parsed, and during this parse the for loop control variable is temporarily marked as immutable, and afterwards restored to the preserved value:

```
   ******   ForStatement<StackFrame frame>     (. boolean up = true, canChange;
                                                  int count, expType;
                                                  String name;
                                                  loopLevel++;
                                                  Label oldContinue = loopContinue;
                                                  Label oldExit = loopExit;
                                                  loopContinue = new Label(!known);
                                                  loopExit = new Label(!known); .)
            =  "for" Ident<↑name>              (. Entry control = Table.find(name);
                                                  if (!control.found) {
                                                    SemError("undeclared identifier");
                                                    control = new Entry(); // new is critical
                                                    control.name = name;
                                                    control.kind = Entry.Var;
                                                    control.type = Entry.noType;
   ******                                           control.canChange = true;
```

```
                                                            control.offset = 0;
                                                            Table.insert(control);
                                                          }
******                                                    canChange = control.canChange;
******                                                    if (isRef(control.type) || control.kind != Entry.Var
******                                                        || !canChange)
******                                                      SemError("illegal control variable");
                                                          CodeGen.loadAddress(control);
******                                                    control.canChange = false; .)
******      "=" Expression<↑expType>                   (. if (!assignable(control.type, expType))
                                                            SemError("incompatible with control variable"); .)
            ( "to" | "downto"                          (. up = false; .)
            )
******      Expression<↑expType>                       (. if (!assignable(control.type, expType))
******                                                      SemError("incompatible with control variable");
                                                          CodeGen.startForLoop(up, loopExit);
                                                          Label startLoop = new Label(known); .)
            "do" Statement<↑count, frame>              (. if (count == 0 && warnings)
                                                            Warning("empty statement part");
                                                          loopContinue.here();
                                                          CodeGen.endForLoop(up, startLoop);
                                                          loopExit.here();
                                                          CodeGen.pop3();
                                                          loopExit = oldExit;
                                                          loopContinue = oldContinue;
******                                                    control.canChange = canChange;
                                                          loopLevel--; .)
          .
```

We turn finally to consideration of the changes needed to the various sub-parsers for expressions.

A casting mechanism is introduced to handle the situations where it is necessary explicitly to convert integer values to characters, so that

character = (char) integer

is allowed, and for completeness, so are

integer = (int) character
integer = (char) character
character = (char) character

Casting operations are accompanied by a type conversion; the `(char)` cast also introduces the generation of code for checking that the integer value to be converted lies within range.

This is all handled within the *Primary* production, which has to be factored to deal with the potential LL(1) trap in distinguishing between components of the form `"(" "char" ")"` and `"(" Expression ")"`:

```
          Primary<↑int type>                  (. int value = 0;
                                                 type = Entry.noType;
                                                 int size;
                                                 DesType des;
                                                 ConstRec con; .)
          =    Designator<↑des>                (. type = des.type;
                                                 switch (des.entry.kind) {
                                                   case Entry.Var:
                                                     CodeGen.dereference();
                                                     break;
                                                   case Entry.Con:
                                                     CodeGen.loadConstant(des.entry.value);
                                                     break;
                                                   default:
                                                     SemError("wrong kind of identifier");
                                                     break;
                                                 } .)
            | Constant<↑con>                   (. type = con.type;
                                                 CodeGen.loadConstant(con.value); .)
            | "new" BasicType<↑type>           (. type++; .)
              "[" Expression<↑size>            (. if (!isArith(size))
                                                   SemError("array size must be integer");
                                                 CodeGen.allocate(); .)
              "]"
******      | "("
******        (   "char" ")"
******           Factor<↑type>                (. if (!isArith(type))
******                                             SemError("invalid cast");
******                                           else type = Entry.charType;
******                                           CodeGen.castToChar(); .)
******         | "int" ")"
******           Factor<↑type>                (. if (!isArith(type))
******                                             SemError("invalid cast");
******                                           else type = Entry.intType; .)
******         | Expression<↑type> ")"
******       )
           .
```

Strictly speaking the above grammar departs slightly from the Java version, where the casting operator is regarded as weaker than the parentheses around an *Expression*, but in practice it makes little difference.

A minor change to the *Constant* production is needed to allow character literals to be regarded as of the new `charType`:

```
           Constant<↑ConstRec con>          (. con = new ConstRec(); .)
           =   IntConst<↑con.value>         (. con.type = Entry.intType; .)
******       | CharConst<↑con.value>        (. con.type = Entry.charType; .)
             | "true"                       (. con.type = Entry.boolType; con.value = 1; .)
             | "false"                      (. con.type = Entry.boolType; con.value = 0; .)
             | "null"                       (. con.type = Entry.nullType; con.value = 0; .)
           .
```

Various of the other productions need modification. For example, the presence of an arithmetic operator correctly placed between character or integer operands must result in the sub-expression so formed being of integer type (and never of character type), and similarly a prefix + or - operator applied to an integer or character *Factor* creates a new factor of integer type:

```
           AddExp<↑int type>                (. int type2;
                                               int op; .)
           =  MultExp<↑type>
              { AddOp<↑op>
                MultExp<↑type2>             (. if (!isArith(type) || !isArith(type2)) {
                                                 SemError("arithmetic operands required");
                                                 type = Entry.noType;
                                               }
******                                         else type = Entry.intType;
                                               CodeGen.binaryOp(op); .)
              } .

           MultExp<↑int type>               (. int type2;
                                               int op; .)
           =  Factor<↑type>
              { MulOp<↑op>
                Factor<↑type2>              (. if (!isArith(type) || !isArith(type2)) {
                                                 SemError("arithmetic operands required");
                                                 type = Entry.noType;
                                               }
******                                         else type = Entry.intType;
                                               CodeGen.binaryOp(op); .)
              } .

           Factor<↑int type>                (. type = Entry.noType; .)
           =    Primary<↑type>
              | "+" Factor<↑type>           (. if (!isArith(type)) {
                                                 SemError("arithmetic operand required");
                                                 type = Entry.noType;
                                               }
******                                         else type = Entry.intType; .)
              | "-" Factor<↑type>           (. if (!isArith(type)) {
                                                 SemError("arithmetic operand required");
                                                 type = Entry.noType;
                                               }
******                                         else type = Entry.intType;
                                               CodeGen.negateInteger(); .)
              | "!" Factor<↑type>           (. if (!isBool(type))
                                                 SemError("boolean operand required");
                                               type = Entry.boolType; CodeGen.negateBoolean(); .)
           .
```

The remaing changes to the system are easily detailed. The *Table* class requires a small change to introduce the new type names needed if the symbol table is to be displayed:

```
               static String[] typeName = {
                 "none", "none[]", "null", "null[]", "int ", "int[] ",
    ******       "bool", "bool[]", "char", "char[]", "void", "void[]" };
```

and the initialization of the `sentinelEntry` requires it to be marked as changeable:

```
               public static void init() {
               // Clears table and sets up sentinel entry
                 sentinelEntry = new Entry();
                 sentinelEntry.name = "";
                 sentinelEntry.type = Entry.noType;
                 sentinelEntry.kind = Entry.Var;
                 sentinelEntry.nextInScope = null;
                 sentinelEntry.found = false;
    ******       sentinelEntry.canChange = true;
                 topScope = null;
                 openScope();
               }
```

The new code generation methods are

```
               public static void read(int type) {
               // Generates code to read a value of specified type
               // and store it at the address found on top of stack
                 switch (type) {
                   case Entry.intType:  emit(PVM.inpi); break;
                   case Entry.boolType: emit(PVM.inpb); break;
    ******         case Entry.charType: emit(PVM.inpc); break;
                 }
               }

               public static void write(int type) {
               // Generates code to output value of specified type from top of stack
                 switch (type) {
                   case Entry.intType:  emit(PVM.prni); break;
                   case Entry.boolType: emit(PVM.prnb); break;
    ******         case Entry.charType: emit(PVM.prnc); break;
                 }
               }

               public static void increment(int type) {
               // Generates code to increment the value found at the address currently
               // stored at the top of the stack.
               // If necessary, apply character range check
    ******       if (type == Entry.charType) emit(PVM.incc); else emit(PVM.inc);
               }

               public static void decrement(int type) {
               // Generates code to decrement the value found at the address currently
               // stored at the top of the stack.
               // If necessary, apply character range check
    ******       if (type == Entry.charType) emit(PVM.decc); else emit(PVM.dec);
               }

    ******     public static void castToChar() {
    ******     // Generates code to check that TOS is within the range of the character type
    ******       emit(PVM.i2c);
               }

               public static void assign(int type) {
               // Generates code to store value currently on top-of-stack on the address
               // given by next-to-top, popping these two elements.
               // If necessary, apply character range check
    ******       if (type == Entry.charType) emit(PVM.stoc); else emit(PVM.sto);
               }
```

Finally the changes to the interpreter class may be summarized:

```
                // Machine opcodes

                  public static final int
                    ...
******              inpc   = 63,
******              prnc   = 64,
******              incc   = 65,
******              decc   = 66,
******              stoc   = 67,
******              i2c    = 68,

                    nul    = 99;              // leave gap for future
```

Within the `switch` statement of the `emulator` method we need:

```
******      case PVM.inpc:        // character input
              if (inBounds(mem[cpu.sp])) {
                mem[mem[cpu.sp]] = data.readChar();
                if (data.error()) ps = badData; else cpu.sp++;
              }
              break;

******      case PVM.prnc:        // character output
              if (tracing) results.write(padding);
              results.write((char) (Math.abs(mem[cpu.sp]) % (maxChar + 1)));
              cpu.sp++;
              if (tracing) results.writeLine();
              break;

******      case PVM.incc:        // character checked ++
              if (inBounds(mem[cpu.sp]))
                if (mem[mem[cpu.sp]] >= 0 && mem[mem[cpu.sp]] < maxChar) {
                  mem[mem[cpu.sp]]++; cpu.sp++;
                }
                else ps = badVal;
              break;

******      case PVM.decc:        // character checked --
              if (inBounds(mem[cpu.sp]))
                if (mem[mem[cpu.sp]] > 0 && mem[mem[cpu.sp]] <= maxChar) {
                  mem[mem[cpu.sp]]--; cpu.sp++;
                }
                else ps = badVal;
              break;

******      case PVM.stoc:        // character checked store
              cpu.sp++;
              if (inBounds(mem[cpu.sp]))
                if (mem[cpu.sp - 1] >= 0 && mem[cpu.sp - 1] <= maxChar)
                  mem[mem[cpu.sp]] = mem[cpu.sp - 1];
                else ps = badVal;
              cpu.sp++;
              break;

******      case PVM.i2c:         // check convert character to integer
              if (mem[cpu.sp] < 0 || mem[cpu.sp] > maxChar) ps = badVal;
              break;
```

Mnemonics are needed for the new opcodes:

```
                  public static void init() {
                    ...
                    mnemonics[PVM.inpc]   = "INPC";
                    mnemonics[PVM.prnc]   = "PRNC";
                    mnemonics[PVM.incc]   = "INCC";
                    mnemonics[PVM.decc]   = "DECC";
                    mnemonics[PVM.stoc]   = "STOC";
                    mnemonics[PVM.i2c]    = "I2C";
                  }
```