Computer Science 3 - 2017

Programming Language Translation

Practical 7 - Week beginning 4 September 2017 - solutions

Sources of full solutions for these problems may be found on the course web page as the file `PRAC7A.ZIP`.

Task 2 - Use of the debugging and other pragmas

The extra pragmas needed in the refined Parva compiler are easily introduced. We need some static fields:

```
      public static boolean
        debug = false,
*       optimize  = false,
*       listCode  = false,
*       warnings  = true,
```

The definitions of the pragmas are done in terms of these:

```
      PRAGMAS
*       CodeOn      = "$C+" .               (. listCode  = true; .)
*       CodeOff     = "$C-" .               (. listCode  = false; .)
        DebugOn     = "$D+" .               (. debug     = true; .)
        DebugOff    = "$D-" .               (. debug     = false; .)
*       OptimizeOn  = "$O+" .               (. optimize  = true; .)
*       OptimizeOff = "$O-" .               (. optimize  = false; .)
*       WarningsOn  = "$W+" .               (. warnings  = true; .)
*       WarningsOff = "$W-" .               (. warnings  = false; .)
        StackDump   = "$SD" .               (. CodeGen.Stack(); .)
        HeapDump    = "$HD" .               (. CodeGen.Heap(); .)
        TableDump   = "$ST" .               (. Table.PrintTable(OutFile.StdOut); .)
```

It is convenient to be able to set the options with command-line parameters as well. This involves straightforward changes to the `Parva.frame` file:

```
          if      (args[i].ToLower().Equals("-l")) mergeErrors = true;
*         else if (args[i].ToLower().Equals("-c")) Parser.listCode = true;
          else if (args[i].ToLower().Equals("-d")) Parser.debug = true;
          else if (args[i].ToLower().Equals("-g")) immediate = true;
          else if (args[i].ToLower().Equals("-n")) execution = false;
*         else if (args[i].ToLower().Equals("-o")) Parser.optimize = true;
*         else if (args[i].ToLower().Equals("-w")) Parser.warnings = false;
          else inputName = args[i];
        }
        if (inputName == null) {
          Console.WriteLine("No input file specified");
*         Console.WriteLine("Usage: Parva [-c] [-d] [-g] [-l] [-n] [-o] [-w] source.pav");
*         Console.WriteLine("-c produce .COD file");
          Console.WriteLine("-d turns on debug mode");
          Console.WriteLine("-g execute immediately after compilation (StdIn/StdOut)");
          Console.WriteLine("-l directs source listing to listing.txt");
          Console.WriteLine("-n no execution after compilation");
*         Console.WriteLine("-o optimize LDA LDG LDC");
*         Console.WriteLine("-w turn off warning messages");
          System.Environment.Exit(1);
        }
```

Finally, the following change to the frame file gives the ability to suppress the generation of the `.COD` listing.

```
*       if (Parser.listCode) PVM.ListCode(codeName, codeLength);
```

Task 3 - Sometimes it makes sense, as you start, to be told how you will stop

Extending *HaltStatement* to have an optional parameter that can issue a report before execution ceases (a useful way of indicating how a program has ended prematurely) is almost trivially easy. It is useful to incorporate a full *WriteList* for maximum usability, rather than just a fixed string.

```
      HaltStatement
      =  "halt"
*       [ "(" [ WriteList ] ")"             (. CodeGen.WriteLine(); .)
*       ]                                   (. CodeGen.LeaveProgram(); .)
        WEAK ";" .
```

Task 4 - Things are not always what they seem

Simply issuing warnings for empty statements or empty blocks is quite easy. As a start we could try:

```
*  Block<StackFrame frame>
   =                                    (. Table.openScope();
*                                          bool empty = true; .)
*     "{" { Statement<frame, funcType>  (. empty = false; .)
*         }                            (. if (empty && warnings) Warning("empty {} block");
       WEAK "}"                         (. Table.closeScope(); .) .

   Statement<StackFrame frame, int funcType>
   =  SYNC (   Block<frame, funcType>
             | ConstDeclarations
             | VarDeclarations<frame>
             | CallOrAssignmentStatement
             | IfStatement<frame, funcType>
             | WhileStatement<frame, funcType>
             | HaltStatement
             | ReturnStatement<funcType>
             | ReadStatement
             | WriteStatement
*            | ";"                      (. if (warnings) Warning("empty statement"); .) .
```

Note that the Boolean variable `empty` is declared locally. Some students declared it globally (as a static variable in the atg file), but static variables are a bit tricky in heavily recursive systems like this. They are acceptable for the flags set/unset/used by the pragma system, however - passing all those as parameters between every production would be extremely tedious.

Spotting an empty block, or the empty statement in the form of a stray semicolon, is partly helpful. Detecting blocks that really have no effect might be handled in several ways. One suggestion would be to count the *executable* statements in a *Block*. This would mean that the *Statement* parser would have to be attributed so as to return this count, and this would have a knock-on effect in various other productions as well. Since we might have all sorts of nonsense like

```
{ { int k; } { { int j; } int i; ; ;    { } {{}} } }     { ; }
```

counting would have to proceed carefully. Details are left as a further exercise - but it is probably not worth doing. Stray semicolons and totally empty blocks are simple and their detection could well be helpful. Once you have started seeing how stupid some code can be, you can develop a flare for writing bad code suitable for testing compilers without asking your friends in CSC 102 or even CSC 301 to do it for you!

Task 5 - How long is a piece of string?

The prac sheet asked why languages generally impose a restriction that a literal string must be contained on a single line of code. The reason is quite simple - it becomes difficult for a human to see or track the control characters and spaces that would otherwise be buried in the string. It is easier and safer for language designers to use the escape sequence idea if they need to cater for non-graphic characters in strings and character literals.

In C++, two or more literal strings that appear in source with nothing but white space between them are automatically concatenated into a single string. Such concatenation is simple. The place to do it is in the *StringConst* production which calls on a *OneString* parser to obtain the substrings (which have had their leading quotes and internal escape characters processed by the time the concatenation takes place):

```
StringConst<out String str>          (. string str2; .)
= OneString<out str>
  { [ "+" ] OneString<out str2>      (. str = str + str2; .)
  } .

OneString<out string str>
=  stringLit                         (. str = token.val;
                                        str = Unescape(str.Substring(1, str.Length - 2)); .) .
```

This feature is not needed in languages like C# and Java, which have proper strings, as the concatenation can be done with a + operator. Just for fun, the code above allows this concatenation operator as an option between string literals that are to be concatenated.

Task 6 - You had better do this one or else….

We can begin by looking at the addition of an *else* option to the *IfStatement*. Doing this efficiently is easy once you see the trick of associating an empty alternative - as in `( something | )` with an action - rather than using the meta brackets `[ something ]`. This is a very useful technique to remember.

```
        IfStatement<StackFrame frame, int funcType>
                                                (. Label falseLabel = new Label(!known);
                                                   Label outLabel = new Label(!known); .)
        =  "if" "(" Condition ")"               (. CodeGen.BranchFalse(falseLabel); .)
*          Statement<frame, funcType>
*       (   "else"                              (. CodeGen.Branch(outLabel);
*                                                  falseLabel.Here(); .)
*           Statement<frame, funcType>          (. outLabel.Here(); .)
*         | /* empty else part */               (. falseLabel.Here(); .)
*       ) .
```

Many - perhaps most - people in attempting this problem come up with the following sort of code instead. This can generate `BRN` instructions where none are needed. Devoid of checking, just to save space:

```
        IfStatement<StackFrame frame, int funcType>
                                                (. Label falseLabel = new Label(!known);
                                                   Label outLabel = new Label(!known); .)
        =  "if" "(" Condition ")"               (. CodeGen.BranchFalse(falseLabel); .)
           Statement<frame, functype>           (. CodeGen.Branch(outLabel);
                                                   falseLabel.Here(); .)
          [ "else"  Statement<frame, funcType>  ]
                                                (. outLabel.Here(); .) .
```

Using this strategy, source code like

```
        if (i == 12) k = 56;
```

would lead to object code like

```
        12    LDA  0
        14    LDV
        15    LDC  12
        17    CEQ
        18    BZE  27
        20    LDA  5
        22    LDC  56
        24    STO
        25    BRN  27        // unnecessary
        27    ....
```

Handling the *elsif* clauses uses the same sort of idea. Note that. after defining `falseLabel.Here()`, the label is "re-used" by assigning it another instance of an "unknown" label. If you don't do this you will get all sorts of bad code or funny messages from the label handler!

```
        IfStatement<StackFrame frame, int funcType)
                                                (. Label falseLabel = new Label(!known);
                                                   Label outLabel = new Label(!known); .)
        =  "if" "(" Condition ")"               (. CodeGen.BranchFalse(falseLabel); .)
           Statement<frame, int funcType)
*       {                                       (. CodeGen.Branch(outLabel);
*                                                  falseLabel.Here();
*                                                  falseLabel = new Label(!known); .)
*          "elsif" "(" Condition ")"            (. CodeGen.BranchFalse(falseLabel); .)
*             Statement<frame, funcType>
*       }
        (  "else"                               (. CodeGen.Branch(outLabel);
                                                   falseLabel.Here(); .)
           Statement<frame, funcType>
         | /* no else part */                   (. falseLabel.Here(); .)
        )                                       (. outLabel.Here(); .) .
```

Task 7 Something to do - while you wait for inspiration

I think this was discussed in class, and I am surprised that some people had trouble with it. We need a single label, and a single conditional branch that goes to the start of the loop body. The only trick is that we don't have

a "branch on true" opcode - but all we have to do is to generate a "negate boolean" operation that will be applied to the computed value of the *Condition* at run time before the conditional branch takes effect:

```
DoWhileStatement<StackFrame frame, int funcType>
                                    (. Label loopStart = new Label(known); .)
=  "do"
     Statement<frame, funcType>
   WEAK "while"
   "(" Condition ")" WEAK ";"       (. CodeGen.NegateBoolean();
                                       CodeGen.BranchFalse(loopStart); .) .
```

Task 8 - This has gone on long enough - time for a break (and then continue)

The syntax of the *BreakStatement* and *ContinueStatement* is, of course, trivial. The catch is that one has to allow these statements only in the context of loops. Trying to find a context-free grammar with this restriction is not worth the effort.

One approach that incorporates context-sensitive checking in conjunction with code generation, as hopefully you know, is based on passing information as parameters between subparsers. The pieces of information we need to pass here are `Label` objects. We change the parser for *Statement* and for *Block* as follows:

```
*  Block<StackFrame frame, int funcType, Label breakLabel, Label continueLabel>
   =                                (. Table.openScope();
                                       bool empty = true; .)
*     "{" { Statement<frame, funcType, breakLabel, continueLabel>
                                    (. empty = false; .)
         }                          (. if (empty && warnings) Warning("empty {} block");
       WEAK "}"                     (. Table.closeScope(); .) .

*  Statement<StackFrame frame, int funcType, Label breakLabel, Label continueLabel>
*  =  SYNC (   Block<frame, breakLabel, continueLabel>
             | ConstDeclarations
             | VarDeclarations<frame>
             | AssignmentOrCall
*            | IfStatement<frame, funcType, breakLabel, continueLabel>
             | WhileStatement<frame, funcType>
             | DoWhileStatement<frame, funcType>
             | LoopStatement<frame, funcType>
*            | BreakStatement<breakLabel>
*            | ContinueStatement<continueLabel>
             | HaltStatement
             | ReturnStatement<funcType>
             | ReadStatement
             | WriteStatement
             | ";"                  (. if (warnings) Warning("empty statement"); .)
          ) .
```

The very first call to *Statement* from within the *Body* of a function passes `null` as the value for each of these labels:

```
Body<StackFrame frame, int funcType>
=                                   (. Label DSPLabel = new Label(known);
                                       int sizeMark = frame.size;
                                       CodeGen.OpenStackFrame(0); .)
*     "{" { Statement<frame, funcType, null, null> }
      WEAK "}"                      (. CodeGen.FixDSP(DSPLabel.Address(), frame.size - sizeMark);
                                       if (funcType == Types.voidType)
                                         CodeGen.LeaveVoidFunction();
                                       else CodeGen.FunctionTrap(); .) .
```

and the labels can be inherited by the parsers for *Block* and *IfStatement* and passed through these to the parsers for *BreakStatement* and *ContinueStatement* that will generate the branch opcodes themselves.

The productions that are concerned with breaking and continuing are simple. The tests for null labels serve to catch the context-sensitive requirement that these statements have meaning only when one is executing the body of a loop.

```
      BreakStatement<Label breakLabel>
*     =  "break"                          (. if (breakLabel == null)
*                                              SemError("break is not allowed here");
*                                           else CodeGen.Branch(breakLabel); .)
         WEAK ";" .

      ContinueStatement<Label continueLabel>
*     =  "continue"                       (. if (continueLabel == null)
*                                              SemError("continue is not allowed here");
*                                           else CodeGen.Branch(continueLabel); .)
         WEAK ";" .
```

Each looping construct must define a point to which a break would transfer control and a point to which a continue would transfer control. These are conveniently handled through the use of *Label* objects, which must be declared locally to each production and then passed to the *Statement* that defines the body of the loop. Few students seemed to be aware of exactly where the continue point would be. A "continue" shortcircuits the rest of the loop body, but in the case of the *DoWhile* or *Repeat* loop transfers control to the *Condition* evaluation, which is *not* at the "start" of the loop. This may be made clearer by considering the following diagrams

```
          While loop                              DoWhile loop                    Infinite loop

start  ┌─> Condition                  start    ┌──> // start body          start  ┌──> // start body
       │  branch false ->┐                    │   { ....                        │   { ....
       │  // start body  │                    │      break ──────>┐             │      break ──────>┐
       │   { .....       │                    │     ....          │             │     ....          │
      ↑│     break ──────>┤                    │┌─<── continue    v             │┌──<── continue    v
       │     ....        │                    │└   .....          │             │└      .....       │
       │┌─<── continue    v                    │     break ──────>┤             │       break ──────>┤
       │└     .....      │                    │     .....         │             │       .....       │
      ↑│     break ──────>┤                    │┌─<── continue    v             │┌──<── continue    v
       │     .....       │                    │└    ....          │             │└      ....        │
       │┌─<── continue    v                    │  } // end body                │    } // end body
      ↑│     ....        │           continue │└──>  Condition                │
       └└── } // end body │                    └└──── branch true              └└────── branch
                         │                                │                                │
exit                     ┌─────────┘           exit       ┌─────────┘           exit       ┌─────────┘
                         v                                v                                v
```

The attributed productions follow. Note carefully how the labels are declared, passed as parameters, and where the `Here()` calls are made to fix labels that have been "used" before they were "known". *DoWhileStatement* requires three of these *Label* objects.

```
      WhileStatement<StackFrame frame, int funcType>
                                          (. Label loopExit = new Label(!known);
*                                            Label loopStart = new Label(known); .)
      =  "while" "(" Condition ")"        (. CodeGen.BranchFalse(loopExit); .)
*        Statement<frame, funcType, loopExit, loopStart>
                                          (. CodeGen.Branch(loopStart);
*                                            loopExit.Here(); .) .

*     DoWhileStatement<StackFrame frame, int funcType>
                                          (. Label loopExit = new Label(!known);
*                                            Label loopContinue = new Label(!known);
                                             Label loopStart = new Label(known); .)
      =  "do"
*          Statement<frame, funcType, loopExit, loopContinue>
*        WEAK "while"                     (. loopContinue.Here(); .)
         "(" Condition ")" WEAK ";"       (. CodeGen.NegateBoolean();
                                             CodeGen.BranchFalse(loopStart);
*                                            loopExit.Here(); .) .
```

Finally, the labels may have to be inherited and then retransmitted through intervening *IfStatement* productions (without modification in this case):

```
      IfStatement<StackFrame frame, int funcType, Label breakLabel, Label continueLabel>
                                          (. Label falseLabel = new Label(!known);
                                             Label outLabel = new Label(!known); .)
      =  "if" "(" Condition ")"           (. CodeGen.BranchFalse(falseLabel); .)
*           Statement<frame, funcType, breakLabel, continueLabel>
         {                                (. CodeGen.Branch(outLabel);
                                             falseLabel.Here();
                                             falseLabel = new Label(!known); .)
           "elsif" "(" Condition ")"      (. CodeGen.BranchFalse(falseLabel); .)
*             Statement<frame, funcType, breakLabel, continueLabel>
         }
```

```
        (  "else"                          (. CodeGen.Branch(outLabel);
                                              falseLabel.Here(); .)
*          Statement<frame, funcType, breakLabel, continueLabel>
         | /* no else part */               (. falseLabel.Here(); .)
        )                                  (. outLabel.Here(); .) .
```

There are other ways of solving the problem. One involves using local variables in the parsing methods to "stack" up old global labels, assigning new ones to these globals, and then restoring the old globals from the local copies afterwards. Another way involves setting up global *List* structures and operating on these as stacks. But the method suggested here seems the neatest. All three of these are essentially "stack" based. By now I hope you know that function calls stack up "stack frames", and one may as well use that automatic stack/unstack mechanism rather than recreate it in dangerous global structures. Avoid the temptation of global structures in highly recursive systems!

Task 9 - Here we go loop - de - loop (You may be too young to remember that song?)

As an alternative way of writing loops, and a very easy one, add an indefinite loop to Parva, as exemplified by

```
loop
  writeLine("Keep it as simple as you can, but no simpler");
```

This is trivial - almost an insult to your intelligence! Here the "continue" branches straight back to the "start".

```
LoopStatement<StackFrame frame, int funcType>
                                       (. Label loopExit = new Label(!known);
                                          Label loopStart = new Label(known); .)
=  "loop"
     Statement<frame, funcType, loopExit, loopStart>
                                       (. CodeGen.Branch(loopStart);
                                          loopExit.Here(); .) .
```

Task 10 - Your professor is quite a character

Obvious simple extensions are needed to introduce a `charType` and the keyword `char` into the grammar and support modules. These are not given in full here in a document that is already too long.

The major part of this extension is concerned with the changes needed to apply various constraints on operands of the `char` type. Essentially, and annoyingly perhaps, in the C family of languages it is a sort of arithmetic type when this is convenient (this is called "auto-promotion"). Explicitly, it ranks as an arithmetic type, in that expressions of the form

character + character	(int)	character != integer	(bool)
character > character	(bool)	integer == character	(bool)
character + integer	(int)	integer + character	(int)
character > integer	(bool)	integer > character	(bool)

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

```
   static bool IsArith(int type) {
*    return type == Types.intType || type == Types.charType || type == Types.noType;
   } // IsArith

   static bool Compatible(int typeOne, int typeTwo) {
*  // Returns true if typeOne is compatible (and comparable for equality) with typeTwo
     return    typeOne == typeTwo                              // obvious
*            || IsArith(typeOne) && IsArith(typeTwo)           // concession char/int
             || typeOne == Types.noType                        // concession for bad type
             || typeTwo == Types.noType                        // concession for bad type
             || IsArray(typeOne) && typeTwo == Types.nullType  // array pointers
             || IsArray(typeTwo) && typeOne == Types.nullType; // array pointers
   } // Compatible
```

Even stricter is the idea of being comparable for "ordering", for which both operands must be of an arithmetic

type, ruling out pointer and Boolean operands.

The preceding discussion relates to *expression compatibility*. However, *assignment compatibility* is more restrictive. Assignments of the form

integer = integer expression
integer = character expression
character = character expression

are allowed, but

character = integer expression

is not allowed. This may be checked with the aid of a further helper method, `Assignable()`.

```
*  static bool Assignable(int typeOne, int typeTwo) {
*  // Returns true if a variable of typeOne may be assigned a value of typeTwo
*    return    typeOne == typeTwo                                    // obvious
*            || typeOne == Types.intType && typeTwo == Types.charType  // one way - can assign char to int
*            || typeOne == Types.noType                              // concession for bad type
*            || typeTwo == Types.noType                              // concession for bad type
*            || IsArray(typeOne) && typeTwo == Types.nullType;       // array pointers
*  }  // Assignable
```

The `Assignable()` function call now takes the place of the `Compatible()` function call in several places in *OneVar* and *AssignmentStatement* where, previously, calls to `Compatible()` might have sufficed.

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

(char) integer // returns the character with internal value of integer
(int) character // returns an integer with internal value for character

is allowed, and for completeness, so are the following redundant operations

(int) integer
(char) character

The cast operator can only be applied to arithmetic values, and casting may be seen as an operation whereby the type of an operand is "changed" for the purposes of semantic analysis of expressions, without seemingly needing code to change its value (but read on).

Some students may make the mistake of thinking that a cast can only appear in the context of a simple assignment, that is, must be restricted to being found in statements like:

```
char  ch;
int x, y, z;
ch = (char) SomeExpression
ch = (char) x + y + z;  // understood wrongly to "mean"
                        // ch = (char) (x + y + z);  // a character assignment
```

but that is not the case. Casting applies only to a **component** of an *Expression*, so that the above "means":

```
ch = ((char) x)  + y + z;    // an incorrect integer assignment again
                             // integer expressions are incompatible with
                             // character target designators
```

and as a further example, stressing the importance of thinking of casting as belonging *within* expressions, it is quite legal to write

```
bool b = 'A' < (char) (x + (int) 'B') ;
```

Of course, the syntax in the C family is crazy (in spite of what some of my colleagues think). It would have been infinitely better to have been allowed to use a notation like

```
bool b = 'A' > char(x + int('B'));
```

but it would not have been possible to define that form easily (do you see why? It might be an exam question; you never know my devious mind).

To get it right requires that casting be handled within the *Primary* production, which has to be factored to deal with the potential LL(1) trap in distinguishing between components like `"("` `"int"` `")"`, `"("` `"char"` `")"` and `"("` *Expression* `")"`:

Casting operations are accompanied by a type *check* (you cannot cast "anything", only arithmetic values) and a type *conversion* (you fool the compiler into it is dealing with a different type for this component of an expression). However, since the PVM stores both characters and integers as 32 bit values, for safety, the `(char)` cast should generate code for checking that the *run-time* integer value to be "converted" lies within the range 0 .. 255 (the 8-bit ASCII character set). And let's not forget to introduce the functions for `cap(ch)`, `low(ch)` and `isLet(ch)`.

```
        Factor<out int type>                  (. type = Types.noType;
                                                 int size;
                                                 DesType des;
                                                 ConstRec con;
                                                 bool upper = true; .)
        =    (  IF (IsCall(out des))             // /* use resolver to handle LL(1) conflict */
                identifier                    (. if (des.type == Types.voidType)
                                                   SemError("void function call not allowed here");
                                                 CodeGen.FrameHeader(); .)
                "(" Arguments<des> ")"        (. CodeGen.Call(des.entry.entryPoint); .)
             |  Designator<out des>           (. switch (des.entry.kind) {
                                                   case Kinds.Var:
                                                     CodeGen.Dereference();
                                                     break;
                                                   case Kinds.Con:
                                                     CodeGen.LoadConstant(des.entry.value);
                                                     break;
                                                   default:
                                                     SemError("wrong kind of identifier");
                                                     break;
                                                 } .)
           )                                  (. type = des.type; .)
          | Constant<out con>                 (. type = con.type;
                                                 CodeGen.LoadConstant(con.value); .)
          | "new" BasicType<out type>         (. type++; .)
              "[" Expression<out size>        (. if (!IsArith(size))
                                                   SemError("array size must be integer");
                                                 CodeGen.Allocate(); .)
              "]"
*         | "("
*             (   "char" ")"
*                 Factor<out type>            (. if (!IsArith(type))
*                                                   SemError("invalid cast");
*                                                 else type = Types.charType;
*                                                 CodeGen.CastToChar(); .)
*               | "int" ")"
*                 Factor<out type>            (. if (!IsArith(type))
*                                                   SemError("invalid cast");
*                                                 else type = Types.intType; .)
*               | Expression<out type> ")"
*             )
*         | ( "cap" | "low"                   (. upper = false; .) )
*           "(" Expression<out type>          (. if (type != Types.charType)
*                                                   SemError("character argument needed");
*                                                 type = Types.charType;
*                                                 CodeGen.ChangeCase(upper); .)
*           ")"
*         | "isLet"
*           "(" Expression<out type>          (. if (type != Types.charType)
*                                                   SemError("character argument needed");
*                                                 type = Types.charType;
*                                                 CodeGen.IsLetter(); .)
*           ")" .
```

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

Various of the other productions need modification. 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). So, for example:

```
        AddExp<out int type>                  (. int type2;
                                                 int op; .)
        =  MultExp<out type>
           { AddOp<out op>
             MultExp<out type2>               (. if (!isArith(type) || !isArith(type2)) {
                                                   SemError("arithmetic operands needed");
                                                   type = Types.noType;
                                                 }
*                                                else type = Types.intType;
                                                 CodeGen.BinaryOp(op); .)
           } .
```

Similarly a prefix + or - operator applied to either an integer or a character creates a new value, but always deemed to be of integer type (so `-'a'` yields the integer -96).

The extra code generation method we need is as follows:

```
public static void CastToChar() {
// Generates code to check that TOS is within the range of the character type
  Emit(PVM.i2c);
} // CodeGen.CastToChar
```

and within the `switch` statement of the `Emulator` method we need:

```
case PVM.i2c:           // check (char) cast is in range
  if (mem[cpu.sp] < 0 || mem[cpu.sp] > maxChar) ps = badVal;
  break;
```

The interpreter has another opcode for checked storage of characters, but if the `i2c` opcodes are inserted correctly it appears that we might never really need `stoc`. Think about this in a quiet moment.

```
case PVM.stoc:          // character checked store
  tos = Pop(); adr = Pop();
  if (InBounds(adr))
    if (tos >= 0 && tos <= maxChar) mem[adr] = tos;
    else ps = badVal;
  break;
```

Task 11 - Make the change; enjoy life; upgrade now to Parva++ (Ta-ra!)

It might not at first have been obvious, but I fondly hoped that everyone would realize that this extension is handled at the initial level by clever modifications to the *AssignmentOrCall* production, which has to be factorized in such a way as to avoid LL(1) conflicts. The code below handles this task (including the tests for assignment compatibility and for the the operators to be applicable only to arithmetic variables rather than constants or pointer variables, tests that several students omitted). It generates the few new machine opcodes introduced in Practical 2. There was some awful confusion in some submissions which attempted, incorrectly, to tack the ++ or -- operator in front and/or behind a designator in the *Designator* production. Where did that idea come from?

```
        AssignmentOrCall                      (. int expType;
                                                 DesType des;
                                                 bool inc = true; .)
        =  ( (  IF (IsCall(out des))               // /* use resolver to handle LL(1) conflict */
                identifier                    (. if (des.type != Types.voidType)
                                                   SemError("non-void function call not allowed here");
                                                 CodeGen.FrameHeader(); .)
                "(" Arguments<des> ")"        (. CodeGen.Call(des.entry.entryPoint); .)
              | Designator<out des>           (. if (des.entry.kind != Kinds.Var)
                                                   SemError("cannot assign to " + Kinds.kindNames[des.entry.kind]); .)
*               (   AssignOp
*                   Expression<out expType>   (. if (!Assignable(des.type, expType))
*                                                SemError("incompatible types in assignment");
*                                                CodeGen.Assign(des.type); .)
*                 | ( "++" | "--"             (. inc = false; .)
*                   )                         (. if (!IsArith(des.type))
*                                                SemError("arithmetic type needed");
*                                                CodeGen.IncOrDec(inc, des.type); .)
*               )
*           )
```

```
*      |  ( "++" | "--"                  (. inc = false; .)
*         ) Designator<out des>          (. if (des.entry.kind != Kinds.Var)
*                                             SemError("variable designator required");
*                                          if (!IsArith(des.type))
*                                             SemError("arithmetic type needed");
*                                          CodeGen.IncOrDec(inc, des.type); .)
      ) WEAK ";" .
```

The extra code generation routine is straightforward, but note that the operators are different for integers and for characters (for which range checks should be incorporated at run time).

```
   public static void IncOrDec(bool inc, 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 == Types.charType) Emit(inc ? PVM.incc : PVM.decc);
*    else Emit(inc ? PVM.inc : PVM.dec);
   } // CodeGen.IncOrDec
```

As usual, the extra opcodes in the PVM make all this easy to achieve at run time. Some submissions might have forgotten to include the check that the address was "in bounds". I suppose one could argue that if the source program were correct, then the addresses could not go out of bounds, but if the interpreter were to be used in conjunction with a rather less fussy assembler (as we had in earlier practicals) it would make sense to be cautious.

```
case PVM.inc:           // integer ++
  adr = Pop();
  if (InBounds(adr)) mem[adr]++;
  break;

case PVM.dec:           // integer --
  adr = Pop();
  if (InBounds(adr)) mem[adr]--;
  break;

case PVM.incc:          // character ++ (checked)
  adr = Pop();
  if (InBounds(adr))
    if (mem[adr] < maxChar) mem[adr]++;
    else ps = badVal;
  break;

case PVM.decc:          // character -- (checked)
  adr = Pop();
  if (InBounds(adr))
    if (mem[adr] > 0) mem[adr]--;
    else ps = badVal;
  break;
```

Task 12 - Let's operate like C#

Parva is looking closer to C/C++/C# with each successive long hour spent in the Hamilton Labs. Seems a pity not to get even closer. The operator precedences in Parva as supplied resemble those in Pascal and Modula, where only four basic levels are supported. Task 12 required you to modify Parva so that it would use a precedence structure based on that in C++, C# or Java, for which a suggested set of (unattributed) productions was supplied.

In general this was not well done, and I received several badly hacked and incomplete solutions suggesting that their authors had given up on what was a straightforward, if slightly tedious exercise. A few explicit comments follow:

The weakest operators are *or* and *and*. These insist on Boolean operands, and if such an operation is performed, it will result in a Boolean subexpression. Note how the short circuit "jumping code" is generated, and that there is no mention of the rest of the *MulOp* operators from the original grammar!

```
Expression<out int type>              (. int type2;
                                         Label shortcircuit = new Label(!known); .)
=  AndExp<out type>
   { "||"                             (. CodeGen.BooleanOp(shortcircuit, CodeGen.or); .)
     AndExp<out type2>                (. if (!IsBool(type) || !IsBool(type2))
                                             SemError("Boolean operands needed");
                                          type = Types.boolType; .)
   }                                  (. shortcircuit.Here(); .) .
```

```
AndExp<out int type>                    (. int type2;
                                           Label shortcircuit = new Label(!known); .)
=  EqlExp<out type>
   { "&&"                               (. CodeGen.BooleanOp(shortcircuit, CodeGen.and); .)
     EqlExp<out type2>                  (. if (!IsBool(type) || !IsBool(type2))
                                             SemError("Boolean operands needed");
                                           type = Types.boolType; .)
   }                                    (. shortcircuit.Here(); .) .
```

At the next level of precedence come the comparisons for equality/inequality which can be applied to compatible operands (essentially of the same type, or perhaps arithmetic (int/char) type), and the comparisons for "ordering" which can only be applied to arithmetic operands (int/char). Again, if these operators appear, the resulting subexpression yields a value of the Boolean type. In the C family there are two levels of precedence to handle what in the Pascal/Modula system was accomplished all at the same level. Notice that no use has been made of the `Types.noType` type in these last four productions - effectively the application of any of these operators is coerced to generating a Boolean expression regardless. Do you suppose this is a good idea?

```
EqlExp<out int type>                    (. int type2;
                                           int op; .)
=  RelExp<out type>
   [ EqualOp<out op>
     RelExp<out type2>                  (. if (!Compatible(type, type2))
                                             SemError("incomparable operand types");
                                           CodeGen.Comparison(op, type);
                                           type = Types.boolType; .)
   ] .

RelExp<out int type>                    (. int type2;
                                           int op; .)
=  AddExp<out type>
   [ RelOp<out op>
     AddExp<out type2>                  (. if (!IsArith(type) || !IsArith(type2))
                                             SemError("incomparable operand types");
                                           CodeGen.Comparison(op, type);
                                           type = Types.boolType; .)
   ] .
```

The rest of the expression hierarchy deals mainly with arithmetic operators. *AddExp* and *MulExp* introduce the possibility of pretending that an incorrect subexpression is of the fictitious `Types.noType` (do you remember why?) and require their operands to being arithmetic (int/char), normally returning an integer subexpression.

```
AddExp<out int type>                    (. int type2;
                                           int op; .)
=  MultExp<out type>
   { AddOp<out op>
     MultExp<out type2>                 (. if (!IsArith(type) || !IsArith(type2)) {
                                             SemError("arithmetic operands needed");
                                             type = Types.noType;
                                           }
                                           else type = Types.intType;
                                           CodeGen.BinaryOp(op); .)
   } .

MultExp<out int type>                   (. int type2;
                                           int op; .)
=  Factor<out type>
   { MulOp<out op>
     Factor<out type2>                  (. if (!IsArith(type) || !IsArith(type2)) {
                                             SemError("arithmetic operands needed");
                                             type = Types.noType;
                                           }
                                           else type = Types.intType;
                                           CodeGen.BinaryOp(op); .)
   } .
```

Too many submissions did not get the production involving the prefix operators anywhere near correct. It appears at a rather different place in the grammar from where it was found in the Pascal/Modula grammar:

```
Factor<out int type>                    (. type = Types.noType; .)
=    Primary<out type>
   | "+" Factor<out type>               (. if (!IsArith(type)) {
                                             SemError("arithmetic operand needed");
                                             type = Types.noType;
                                           }
                                           else type = Types.intType; .)
```

```
   | "-" Factor<out type>            (. if (!IsArith(type)) {
                                          SemError("arithmetic operand needed");
                                          type = Types.noType;
                                       }
                                       else type = Types.intType;
                                       CodeGen.NegateInteger(); .)
   | "!" Factor<out type>            (. if (!IsBool(type))
                                          SemError("Boolean operand needed");
                                       type = Types.boolType;
                                       CodeGen.NegateBoolean(); .) .
```

Finally, *Primary* is essentially the old *Factor* but with the code for "prefix *not*" removed. You might like to consider whether the casting operator should be applied to a *Factor* (as here) or restricted to a *Primary*, rather more like what was suggested earlier. Does it make any practical difference?

```
Primary<out int type>                (. type = Types.noType;
                                        int size;
                                        DesType des;
                                        ConstRec con;
                                        bool upper = true; .)
=    (  IF (IsCall(out des))            // /* use resolver to handle LL(1) conflict */
        identifier                   (. if (des.type == Types.voidType)
                                          SemError("void function call not allowed here");
                                        CodeGen.FrameHeader(); .)
        "(" Arguments<des> ")"       (. CodeGen.Call(des.entry.entryPoint); .)

      |  Designator<out des>         (. switch (des.entry.kind) {
                                          case Kinds.Var:
                                            CodeGen.Dereference();
                                            break;
                                          case Kinds.Con:
                                            CodeGen.LoadConstant(des.entry.value);
                                            break;
                                          default:
                                            SemError("wrong kind of identifier");
                                            break;
                                        } .)
     )                               (. type = des.type; .)
   | Constant<out con>               (. type = con.type;
                                        CodeGen.LoadConstant(con.value); .)

   | "new" BasicType<out type>       (. type++; .)
       "[" Expression<out size>      (. if (!IsArith(size))
                                          SemError("array size must be integer");
                                        CodeGen.Allocate(); .)
       "]"

   | "("
       (   "char" ")"
           Factor<out type>          (. if (!IsArith(type))
                                          SemError("invalid cast");
                                        else type = Types.charType;
                                        CodeGen.CastToChar(); .)
         | "int" ")"
           Factor<out type>          (. if (!IsArith(type))
                                          SemError("invalid cast");
                                        else type = Types.intType; .)
         | Expression<out type> ")"
       )

   | ( "cap" | "low"                 (. upper = false; .) )
      "(" Expression<out type>       (. if (type != Types.charType)
                                          SemError("character argument needed");
                                        type = Types.charType;
                                        CodeGen.ChangeCase(upper); .)
     ")"

   | "isLet"
      "(" Expression<out type>       (. if (type != Types.charType)
                                          SemError("character argument needed");
                                        type = Types.charType;
                                        CodeGen.IsLetter(); .)
      ")" .
```

Of course, the productions defining the various grouping of operators at a given level are slightly different:

```
AddOp<out int op>                    (. op = CodeGen.nop; .)
=    "+"                             (. op = CodeGen.add; .)
```

```
    | "-"                              (. op = CodeGen.sub; .) .

MulOp<out int op>                      (. op = CodeGen.nop; .)
=    "*"                               (. op = CodeGen.mul; .)
    | "/"                              (. op = CodeGen.div; .)
    | "%"                              (. op = CodeGen.rem; .) .

EqualOp<out int op>                    (. op = CodeGen.nop; .)
=    "=="                              (. op = CodeGen.ceq; .)
    | "!="                             (. op = CodeGen.cne; .) .

RelOp<out int op>                      (. op = CodeGen.nop; .)
=    "<"                               (. op = CodeGen.clt; .)
    | "<="                             (. op = CodeGen.cle; .)
    | ">"                              (. op = CodeGen.cgt; .)
    | ">="                             (. op = CodeGen.cge; .) .
```

If you did not do so before, you might like to think what advantages or disadvantages are to be found in having fewer (Pascal-like) or more (C#-like) levels of precedence in the *Expression* hierarchy.

Task 13 (Bonus) - Generating tighter PVM code

Way back in earlier exercises we added some specialized opcodes like `LDC_1, LDA_2` and so on to the PVM. The problem asked for these to be used, and for a `$O` pragma or `-o` command line option to be added to the system so that these "optimized" opcodes would be used only on request (or *not* used on request - suit yourself).

The extensions to the grammar and frame files were illustrated earlier.

The code generator can respond to the pragma setting with various routines modified on the following lines (it does not seem necessary to give them all in full at this point). However, some of these optimized opcodes are, when you think about it, of precious little use for the multifunction compiler. The first four elements of a stack frame are used for "housekeeping" - RV, DL, RA and SM, and the first parameter or local variable has an offset of 4 from the frame pointer FP. So it would be a better idea to introduce several more of these special codes, as in the extract below, and this will be done in the source code in future.

```
public static void LoadAddress(Entry var) {
// Generates code to push address of local variable with known offset onto evaluation stack
  if (var.level == Entry.global)
    if (Parser.optimize)
      switch (var.offset) {
        case 0:  Emit(PVM.ldg_0); break;
        case 1:  Emit(PVM.ldg_1); break;
        case 2:  Emit(PVM.ldg_2); break;
        case 3:  Emit(PVM.ldg_3); break;
        case 4:  Emit(PVM.ldg_4); break;
        case 5:  Emit(PVM.ldg_5); break;
        case 6:  Emit(PVM.ldg_6); break;
        case 7:  Emit(PVM.ldg_7); break;
        case 8:  Emit(PVM.ldg_8); break;
        default: Emit(PVM.ldga); Emit(var.offset); break;
      }
      else {
        Emit(PVM.ldga); Emit(var.offset);
      }
  else // local variables
    if (Parser.optimize)
      switch (var.offset) {
        case 0:  Emit(PVM.lda_0); break;
        case 1:  Emit(PVM.lda_1); break;
        case 2:  Emit(PVM.lda_2); break;
        case 3:  Emit(PVM.lda_3); break;
        case 4:  Emit(PVM.lda_4); break;
        case 5:  Emit(PVM.lda_5); break;
        case 6:  Emit(PVM.lda_6); break;
        case 7:  Emit(PVM.lda_7); break;
        case 8:  Emit(PVM.lda_8); break;
        default: Emit(PVM.lda); Emit(var.offset); break;
      }
      else {
        Emit(PVM.lda); Emit(var.offset);
      }
} // CodeGen.LoadAddress
```

Task 14 - If you survive this I'll pass you a reference so that employers will know your value

As supplied, the Parva compiler could only pass parameters "by value", and you were challenged to extend it on the lines of the approach adopted in C#, allowing you to write impressive methods like

```
void Swap(ref int i, ref int j) {
// Interchange values of i and j
  int k = i;
  i = j;
  j = k;
} // Swap

void Main() {
  int a, b;
  readLine(a, b);
  Swap(ref a, ref b);
  writeLine(a, b);
} // Main
```

This is remarkably easy to do. We need an extra field in the `Entry` class which will normally have the value of *false*, and will be *true* only for formal parameters that were qualified with the *ref* keyword at their point of declaration.

```
class Entry {
// All fields initialized, but are modified after construction (by semantic analyser)
  public int     kind        = Kinds.Var;
  public string  name        = "";
  public int     type        = Types.noType;
  public int     value       = 0;       // constants
  public int     offset      = 0;       // variables
  public bool    declared    = true;    // true for all except sentinel entry
  public Entry   nextInScope = null;    // link to next entry in current scope
  public int     nParams     = 0;       // functions
*   public bool    byRef       = false;   // true only for parameters passed by reference
  public Label   entryPoint  = new Label(false);
  public Entry   firstParam  = null;
} // end Entry
```

Where needed, the `byRef` field is set true within the *OneParam* production:

```
 OneParam<out Entry param, Entry func>
 =                                  (. param = new Entry();
                                       param.level = Entry.local;
                                       param.kind = Kinds.Var;
                                       param.offset = CodeGen.headerSize + func.nParams;
                                       func.nParams++; .)
*    [ "ref"                         (. param.byRef = true; .)
     ] Type<out param.type>          (. if (param.type == Types.voidType)
                                          SemError("parameters may not be of void type"); .)
     Ident<out param.name>           (. Table.Insert(param); .) .
```

When a function activation is in progress, a choice is made within the *OneArg* production between pushing an address (for an argument denoted by a designator and qualified by `ref` for a reference argument) or the value of an expression (for a value argument). Note the careful check that the correct passing mechanism is being selected, as well as the assignment compatibility checks between formal and actual parameters. Finally, your attention is drawn to the lookout for *null* entries in the parameter list.

```
 OneArg<Entry fp>                   (. int argType;
                                       DesType des; .)
 =  (  Expression<out argType>      (. if (fp != null) {
                                          if (!Assignable(fp.type, argType))
                                            SemError("argument type mismatch");
*                                       if (fp.byRef)
*                                         SemError("this argument must be passed by reference");
                                       } .)
      |
*       "ref" Designator<out des>   (. if (fp != null) {
                                       if (!Assignable(fp.type, des.type))
                                         SemError("argument type mismatch");
*                                       if (!fp.byRef)
*                                         SemError("this argument must be passed by value");
                                       } .)
     ) .
```

If a variable has been passed "by reference" (that is, where the actual argument is its address rather than its value)

then, when the *Designator* parser encounters the corresponding formal parameter within the function body, an extra *dereference* opcode must be generated, to be able to follow the address found as the argument to find the address of the actual variable being referenced:

```
        Designator<out DesType des>        (. string name;
                                              int indexType; .)
        =  Ident<out name>                 (. Entry entry = Table.Find(name);
                                              if (!entry.declared)
                                                SemError("undeclared identifier");
                                              des = new DesType(entry);
                                              if (entry.kind == Kinds.Var)
                                                CodeGen.LoadAddress(entry);
   *                                          if (entry.byRef)
   *                                            CodeGen.Dereference(); .)
           [  "["                          (. if (IsArray(des.type)) des.type--;
                                              else SemError("unexpected subscript");
                                              if (des.entry.kind != Kinds.Var)
                                                SemError("unexpected subscript");
                                              CodeGen.Dereference(); .)
                 Expression<out indexType>  (. if (!IsArith(indexType))
                                                SemError("invalid subscript type");
                                              CodeGen.Index(); .)

              "]"
           ] .
```

The following snapshots attempt to clarify the two modes of parameter passing for a very simple case. Firstly, consider the situation where parameter passing is by value. The source code is given, as is most of the PVM code, and a sequence of snapshots indicates the state of the stack frames as the execution takes place:

```
  void One(int p) {                      LDA  p    ; p is a local variable
  // Passing by value                    LDV
    int j = 5, k = 12, l = 'a';          PRNI      ; write(p) (14)
    write(p);                            LDA  p    ; address of local/parameter p (189)
    p = j;                               LDA  j
  } // function                          LDV
                                         STO       ; p = j  (5)
                                         RETV      ; return

  void Main() {                          FHDR      ; allocate frame header for call to One
    int a = 6, b = 8, c = a + b;         LDA  c
    One(c);                              LDV       ; push value of c (14) as the value for parameter p
    write(c);                            CALL One
  } // Main                              LDA  c
                                         LDV
                                         PRNI      ; write(c)  (still 14)
```

```
                                                      c    b    a   Main frame header
|    |    |    |    |    |    |    |    |    |    || 14 |  8 |  6 | SM | RA | DL | RV ||  Main initializes
                                                                                         a, b and c in 194
 offsets from FP                                   |  6    5    4    3    2    1    0 |  ... 196
 actual     186  187  188  189  190  191  192  193 |194  195  196  197  198  199  200 |
 for illustration                                 SP                                  FP
```

```
                                p  One frame header   c    b    a   Main frame header
|    |    |    |    |    | 14 | SM | RA | DL | RV || 14 |  8 |  6 | SM | RA | DL | RV ||  Main activates
                                                                                         One, pushing the
                                                                                         value of c as the
 offsets from FP         |  4    3    2    1    0 |  6    5    4    3    2    1    0    arg corresponding
 actual     186  187  188| 189  190  191  192  193| 194  195  196  197  198  199  200   to p before calling
                        SP                       FP                                     One to take control
```

```
                  l    k    j    p  One frame header   c    b    a   Main frame header
|    |    | 96 | 12 |  5 | 14 | SM | RA | DL | RV || 14 |  8 |  6 | SM | RA | DL | RV ||  One is called and
                                                                                         initializes local
                                                                                         j, k and l in 186
 offsets   |  7    6    5    4    3    2    1    0 |  6    5    4    3    2    1    0    ... 188
 actual    | 186  187  188  189  190  191  192  193| 194  195  196  197  198  199  200
          SP                                      FP
```

```
                  l    k    j    p  One frame header   c    b    a   Main frame header
|    |    | 96 | 12 |  5 | 14 | SM | RA | DL | RV || 14 |  8 |  6 | SM | RA | DL | RV ||  To print the value
                                                                                         of formal param p
                                                                                         One dereferences
 offsets   |  7    6    5    4    3    2    1    0 |  6    5    4    3    2    1    0    location 189 to get 14
 actual    | 186  187  188  189  190  191  192  193| 194  195  196  197  198  199  200
          SP                                      FP
```

```
                  l    k    j    p  One frame header      c    b    a   Main frame header
------------------------------------------------------------------------------------------- One assigns j to
|         |    |  96|  12|   5|   5|  SM|  RA|  DL|  RV‖  14|   8|   6|  SM|  RA|  DL|  RV‖ formal parameter
------------------------------------------------------------------------------------------- p at 189.  Arg c in
  offsets      |             5    4    3    2    1    0|   6    5    4    3    2    1    0   Main is not affected.
  actual       | 186  187  188  189  190  191  192  193| 194  195  196  197  198  199  200
               SP                                      FP
```

```
                                                          c    b    a   Main frame header
------------------------------------------------------------------------------------------- One relinquishes
|         |    |    |    |    |    |    |    |    |    ‖  14|   8|   6|  SM|  RA|  DL|  RV‖ control and its stack
------------------------------------------------------------------------------------------- frame is dscarded as
  offsets                                              |   6    5    4    3    2    1    0|  Main resumes control
  actual         186  187  188  189  190  191  192  193| 194  195  196  197  198  199  200|
                                                       SP                                  FP
```

Secondly, consider the situation where parameter passing is by reference. The source code is given, as is most of the PVM code, and a sequence of snapshots indicates the state of the stack frames as the execution takes place:

```
void One(ref int p) {               LDA  p    ; p is a local variable storing the address of c
// Passing by reference             LDV       ; find the address stored in p
  int j = 5, k = 12, l = 'a';       LDV       ; dereference to find the value of variable c
  write(p);                         PRNI      ; write(p)
  p = j;                            LDA  p    ; address of parameter p (189)
} // function                       LDV       ; find the address stored in formal parameter p (194)
                                    LDA  j
                                    LDV       ; p = j (equivalent to c = j)
                                    STO

void Main() {                       FHDR      ; allocate frame header for call to One
  int a = 6, b = 8, c = a + b;      LDA  c    ; push address of c (194) as the value for parameter p
  One(ref c);                       CALL One
  write(c);                         LDA  c
} // Main                           LDV
                                    PRNI      ; write(c)
```

```
                                                          c    b    a   Main frame header
------------------------------------------------------------------------------------------- Main initializes
|         |    |    |    |    |    |    |    |    |    ‖  14|   8|   6|  SM|  RA|  DL|  RV‖ a, b and c in 194
------------------------------------------------------------------------------------------- ... 196
  offsets                                              |   6    5    4    3    2    1    0|
  actual         186  187  188  189  190  191  192  193| 194  195  196  197  198  199  200|
  for illustration                                     SP                                  FP
```

```
                                 p  One frame header      c    b    a   Main frame header
------------------------------------------------------------------------------------------- Main activates One,
|         |    |    |    |    | 194|  SM|  RA|  DL|  RV‖  14|   8|   6|  SM|  RA|  DL|  RV‖ pushing the address
------------------------------------------------------------------------------------------- of local c as the
  offsets                     |   4    3    2    1    0|   6    5    4    3    2    1    0   arg corresponding
  actual         186  187  188| 189  190  191  192  193| 194  195  196  197  198  199  200   to formal parameter
                              SP                       FP                                   p before calling One
```

```
                  l    k    j    p  One frame header      c    b    a   Main frame header
------------------------------------------------------------------------------------------- One is called and
|         |    |  96|  12|   5| 194|  SM|  RA|  DL|  RV‖  14|   8|   6|  SM|  RA|  DL|  RV‖ initializes local
------------------------------------------------------------------------------------------- j, k and l in 186
  offsets      |   7    6    5    4    3    2    1    0|   6    5    4    3    2    1    0   ... 188
  actual       | 186  187  188  189  190  191  192  193| 194  195  196  197  198  199  200
               SP                                      FP
```

```
                  l    k    j    p  One frame header      c    b    a   Main frame header
------------------------------------------------------------------------------------------- To print the value of
|         |    |  96|  12|   5| 194|  SM|  RA|  DL|  RV‖  14|   8|   6|  SM|  RA|  DL|  RV‖ formal parameter one
------------------------------------------------------------------------------------------- first dereferences
  offsets      |             5    4    3    2    1    0|   6    5    4    3    2    1    0   location 189 to obtain
  actual       | 186  187  188  189  190  191  192  193| 194  195  196  197  198  199  200   address 194 and then
               SP                                      FP                                   dereferences 194 to get 14
```

```
                  l    k    j    p  One frame header      c    b    a   Main frame header
------------------------------------------------------------------------------------------- One assigns the value
|         |    |  96|  12|   5| 194|  SM|  RA|  DL|  RV‖   5|   8|   6|  SM|  RA|  DL|  RV‖ of local j to variable
------------------------------------------------------------------------------------------- c at the address 194
  offsets      |             5    4    3    2    1    0|   6    5    4    3    2    1    0   still stored in formal
  actual       | 186  187  188  189  190  191  192  193| 194  195  196  197  198  199  200   parameter p
               SP                                      FP
```

```
                                                          c    b    a   Main frame header
------------------------------------------------------------------------------------------- One relinquishes
|         |    |    |    |    |    |    |    |    |    ‖   5|   8|   6|  SM|  RA|  DL|  RV‖ control and its stack
------------------------------------------------------------------------------------------- frame is dscarded as
  offsets                                              |   6    5    4    3    2    1    0|  Main resumes control
  actual         186  187  188  189  190  191  192  193| 194  195  196  197  198  199  200|
                                                       SP                                  FP
```