Computer Science 3 - 2010

Programming Language Translation

Practical for Weeks 25 - 26, beginning 11 October 2010

This extended prac is designed to take you the best part of two weeks. Hand in your solutions *before* lunch time on **Monday 25 October**, correctly packaged in a transparent folder with your cover sheet and individual assessment sheets. Since the practical will have been done on a group basis, please hand in one copy of the cover sheet for each member of the group. These will be returned to you in due course, signed by the marker. **Please make it clear whose folder you have used for the electronic submission, for example g03A1234.** Please resist the temptation to carve up the practical, with each group member only doing one or two tasks. The group experience is best when you work on tasks together.

The reason for requiring all submissions by 25 October is to free you up during swot week to prepare for the final examinations. I shall try to get the marking done as soon as possible after that.

Objectives:

In this practical you are to

- familiarize yourself with the compiler described in chapters 12 and 13 that translates Parva to PVM code
- extend this compiler in numerous ways, some a little more demanding than others.

You will need this prac sheet and your text book. As usual, copies of the prac sheet are also available at `http://www.cs.ru.ac.za/CSc301/Translators/trans.htm`.

Outcomes:

When you have completed this practical you should understand

- several aspects of semantic constraint analysis in an incremental compiler
- code generation for a simple stack machine.

Hopefully after doing these exercises (and studying the attributed grammar and the various other support modules carefully) you will find you have learned a lot more about compilers and programming languages than you ever did before (and, I suspect, a lot more than undergraduates at any other university in this country). I also hope that you will have begun to appreciate how useful it is to be able to base a really large and successful project on a clear formalism - namely the use of attributed context-free grammars - and will have learned to appreciate the use of sophisticated tools like Coco/R.

To hand in:

By the hand-in date you are required to hand in, besides the cover sheets (one per group member):

- Listings of your `Parva.atg` file and the source of any auxiliary classes that you develop. Please print these on the laser printer using the LPRINT utility and *not* using UltraEdit in two-column mode, as the listings get wide. Please ensure that the lines do all "fit" - lay out your grammars neatly! It would also help if you could use a highlighter to show where you have made changes.
- Some examples of very short test programs and the code generated by your systems.
- Electronic copies of your solutions.

I do NOT require listings of any Java or C# code produced by Coco/R.

Keep the prac sheet and your solutions until the end of the semester. Check carefully that your mark has been entered into the Departmental Records.

You are referred to the rules for practical submission which are clearly stated in our Departmental Handbook. However, for this course pracs must be posted in the "hand-in" box outside the laboratory and

not given to demonstrators.

A rule not stated there, but which should be obvious, is that you are not allowed to hand in another group's or student's work as your own. Attempts to do this will result in (at best) a mark of zero and (at worst) severe disciplinary action and the loss of your DP. You are allowed - even encouraged - to work and study with other students, but if you do this you are asked to acknowledge that you have done so. You are expected to be familiar with the University Policy on Plagiarism, which you can consult at:

`http://www.scifac.ru.ac.za/plagiarism_policy.pdf`

Before you begin

The tasks are presented below in an order which, if followed, should make the practical an enjoyable and enriching experience. Please do not try to leave everything to the last few hours, or you will come horribly short. You must work consistently, and with a view to getting an overview of the entire project, as the various components and tasks all interact in ways that will probably not at first be apparent. Please take the opportunity of coming to consult with me at any stage if you are in doubt as how best to continue. By all means experiment in other ways and with other extensions if you feel so inclined.

This version of Parva has been restricted so as not to include functions. This means that there will be no practical work set on chapter 14 of the text. Because of the timing of our courses this is unavoidable, if highly regrettable. You should be warned that some of the material of that chapter may be examinable.

The operator precedences in Parva as supplied use a precedence structure based on that in C++ or Java, rather than the "Pascal-like" ones in the book. Study these carefully and note how the compiler provides "short-circuit" semantics correctly (see page 365) and deals with type compatibility issues (see section 12.6.8)

You are advised that it is in your best interests to take this opportunity of really studying the code in the Parva grammar and its support files. The exercises have been designed to try to force you to do that, but it is always tempting just to guess and to hack. With a program of this size that often leads to wasting more time than it saves. Finally, remember the advice given in an earlier lecture:

Keep it as simple as you can, but no simpler.

A note on test programs

Throughout this project you will need to ensure that the features you explore are correctly implemented. This means that you will have to get a feel for understanding code sequences produced by the compiler. The best way to do this is usually to write some very minimal programs, that do not necessarily do anything useful, but simply have one, or maybe two or three statements, the object code for which is easily predicted and understood.

When the Parva compiler has finished compiling a program, say `SILLY.PAV`, you will find that it creates a file `SILLY.COD` in which the stack machine assembler code appears. Studying this is often very enlightening.

Useful test programs are small ones like the following. There are some specimen test programs in the kit, but these are deliberately incomplete, wrong in places, too large in some cases and so on. Get a feel for developing some of your own.

```
$D+ // Turn on debugging mode
void main (void) {
  int i;
  int[] List = new int[10];
  while (true) { // infinite loop, can generate an index error
    read(i);
    List[i] = 100;
  }
}
```

The debugging pragma

It is useful when writing a compiler to be able to produce debugging output - but sometimes this just clutters up a production quality compiler. The `PARVA.ATG` grammar makes use of the `PRAGMAS` option of Coco/R (see text, page 250) to allow pragmas like those shown to have the desired effect.

```
$D+ /* Turn debugging mode on  */
$D- /* Turn debugging mode off */
```

Task 1 - Create a working directory and unpack the prac kit

There are several files that you need, zipped up in the file `PRAC25.ZIP` (Java) or `PRAC25C.ZIP` (C#).

- Immediately after logging on, get to the command line level by using the `Start -> All Programs -> Accessories -> Command prompt` option from the tool bar.

- Copy the prac kit into a newly created directory/folder in your file space

  ```
  J:
  md  prac25
  cd  prac25
  copy  i:\csc301\trans\prac25.zip
  unzip  prac25.zip
  ```

  This will create several other directories "below" the `prac25` directory:

  ```
  J:\prac25
  J:\prac25\library
  J:\prac25\Parva
  ```

  containing the Java classes for the I/O library, and for the code generator and symbol table handler.

  You will also find the executable version of Coco/R and batch files for running it, frame files, and various sample data, code and the Parva grammar, contained in files with extensions like

  ```
  *.ATG,    *.PAV
  ```

- UltraEdit is probably your editor of choice. The version in the lab is configured to run various of the compilers easily, and has been tweaked to run Coco/R in much the same sort of way (see below). *To get this to work properly, start UltraEdit from a command window by giving the command* `UEDIT32`*, rather than by clicking on an icon on the desktop.*

- As usual, you can use the `CMAKE` and `CRUN` commands to build and run the compiler. The kit also supplied a `PARVA.BAT` file to allow you to give a command like `parva voter.pav` more easily than by using `CRUN`.

Task 2 - Better use of the debugging pragma

We have already commented on the `$D+` pragma. At present it is only used to request the printout of a symbol table. How would you change the system so that one would have to use a similar pragma or command line option if one wanted to obtain the assembler code file - so that the "`.COD`" file with the assembler code listing would only be produced if it were really needed?

```
$C+ /* Request that the .COD file be produced */
$C- /* Request that the .COD file not be produced */
```

Another useful (run-time) debugging aid is the undocumented `stackdump` statement. Compilation of this is also controlled by the $D pragma (in other words, the stack dumping code is only generated when in debug mode - much of the time you are testing your compiler you will probably be working in "debugging" mode, I expect).

Hint: This addition is almost trivially easy. You will also need to look at (and probably modify) the `Parva.frame` file, which is used as the basis for constructing the compiler proper (see page 268).

Task 3 - More variety in identifier names

As supplied, the compiler allows identifiers to start with a letter and then be followed by any number of letters and digits and underscores (sometimes called lowlines). Make a trivial change to the definition of the identifier token so as to prevent an identifier from having an underscore as its final character.

Task 4 - Learning many languages is sometimes confusing

If, like me, you first learned programming in languages other than Java, you may persist in making silly mistakes when writing Parva programs. For example, programmers familiar with Pascal and Modula-2 easily confuse the roles played by `==`, `!=` and `=` with the similar operators in Pascal denoted by `=`, `<>` and `:=`, and are prone to introduce words like `then` into *IfStatements*, giving rise to code like

```
if (a = b) then c := d;
if (x <> y) then p := q;
```

instead of

```
if (a == b) c = d;
if (x != y) p = q;
```

Can you think of (and implement) ways to handle these errors sympathetically - that is to say, to report them, but then "recover" from them without further ado?

(Confusing `=` with `==` in Boolean expressions is also something C, C++ and Java programmers can do. It is particularly nasty in C/C++, where a perfectly legal statement like

```
 if (a = b) x = y;
```

does not compare `a` and `b`, but assigns `b` to `a`, as you probably know).

Task 5 - The final word in declarations

In Parva, the key word *const* is not used in the sense that it is used in C#, or in the sense that 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 cannot be modified later. If you check the grammar for Parva you will come 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 will not handle code of the form some people might prefer, or a variation perhaps, like

```
final int max = 2;
int i = 5;
final int iPlusMax = i + max;
final int[] list = new int[iPlusMax];
```

Add this feature, while leaving the *ConstDeclarations* production alone.

Hint: This involves making small changes to the symbol table handler. Be careful, as there may be side effects of making the "obvious" changes which you might not at first realize.

Task 6 - Better string handling

Strings only appear in Parva in read and write lists, and they have been implemented by stacking them in the top part of memory. This would get messy if we really wanted to add a string type to Parva. You may be relieved to learn that we don't want to do that at this stage (your predecessors in 2007 were faced with this exercise in the 24 hour exam). But as a prelude, consider how we might, instead, store literal strings in an "generic" ArrayList structure defined within the PVM and then modify the PRNS opcode to index this list.

Task 7 - Your professor is quite a character

Parva will get closer to C/C++/Java with each successive long hour spent in the Hamilton Labs. Seems a pity to stop now, so go right on and extend the system to allow for a character type as well as the integer and Boolean ones, enabling you to develop classic programs 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++;
    read(ch);
  }
  while (i > 0) {      // output loop
    i--;
    write(sentence[i]);
  }
}
```

Hint: A major part of this exercise is concerned with the changes needed to apply various constraints on operands of the `char` type. In some ways it ranks as an arithmetic type, so that expressions of the form

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

are all allowable. However, assignment compatibility is more restricted. Assignments like

integer = integer
integer = character
character = character

are all allowed, but

character = integer

is not allowed. Following Java and C#, introduce a casting mechanism to handle the situations where it is necessary explicitly to convert integer values to characters, so that

character = (char) integer

would be allowed, and for completeness, so would

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

But be careful. Parva uses an ASCII character set, so that executing code generated from statements like

```
int i = -90;
char ch = (char) 1000;
char ch2 = (char) 2 * i;
```

should lead to run-time errors.

Task 8 - What if you don't like short-circuit evaluation of Boolean expressions?

As you are hopefully aware, most languages implement "short-circuit semantics" in generating code for the evaluation of complex Boolean expressions. Extend the system to allow a user to use a `$S` pragma or `-s` command line parameter to choose between code generation using short-circuit semantics or code generation using a Boolean operator approach (see the textbook, pages 22 and 365). All the opcodes you need are already in the source kit.

Task 9 - Make the change; enjoy life; upgrade now to Parva++ (Ta-ra!)

At last! Let's really make Parva useful and turn it into Parva++ by adding the increment and decrement statement forms exemplified by

```
int parva;
int [] list = new int[10];
char ch = 'A';
...
parva++;
--ch;
list[10]--;
```

Suggestions for doing this - specifically by introducing new operations into the PVM - are made in section 13.5.1 of the text. Be careful - only integer and character variables (and array elements) can be handled in this way.

Task 10 - This has gone on long enough - time for a break

The *BreakStatement* is syntactically simple, but takes a bit of thought. Give it some! Be careful - breaks can currently only appear within *while* loops, but there might be several break statements inside a single loop, and loops can be nested inside one another.

Task 11 - What are we doing this for?

Things are getting more interesting by this stage, and more challenging.

Many languages provide for a *ForStatement* in one or other form. Although most people are familiar with these, their semantics and implementation can actually be quite tricky.

A possible syntax for a *ForStatement*, suggested by the C family of languages, might be:

```
ForStatement = "for" (Ident "=" Expression ";" Condition ";" Assignment) Statement .
```

or, allowing for optional components:

```
ForStatement = "for" ([ Ident "=" Expression ] ";" [ Condition ] ";" [ Assignment] ) Statement .
```

and, of course, we can even declare the control variable locally, with a scope extending over the *ForStatement* only:

```
ForStatement = "for" ([ [BasicType] Ident "=" Expression ] ";" [ Condition ] ";" [ Assignment] )
               Statement .
```

The semantics of this statement are deemed to be essentially equivalent to

```
Ident "=" Expression;
while ( Condition ) {
  Statement;
  Assignment;
}
```

If the *Condition* is omitted it is equivalent to *true*.

Implement this version of a *ForStatement*.

Hint: You will not need any new opcodes in the PVM. But you will have to exercise some ingenuity to handle the fact that, syntactically, the *Assignment* component appears (and is parsed) before the *Statement* part - in an incremental compiler this can be handled by generating a suitable arrangement of branches, even though this leads to what is often known as "spaghetti code". You will also need to pay attention to handling the scoping issue if the control variable is (very sensibly) declared locally to the *ForStatement*.

Task 12 - For what it's worth, Pascal did it differently

Suppose we were to add a simple Pascal-style *ForStatement* to Parva, to allow statements whose concrete syntax is defined by

```
ForStatement = "for" Ident "=" Expression ("to" | "downto") Expression Statement .
```

for example

```
for i = 1 to 10 write(i);         // forwards  1 2 3 4 5 6 7 8 9 10
for i = 10 downto 1 write(i);     // backwards 10 9 8 7 6 5 4 3 2 1
for i = 10 to 5 write(i);         // no effect
for i = i - 1 to i + 6 write(i);  // what does this do?
for i = 1 to 5 read(i);           // should we allow this?
for i = 1 to 5 i = i + 1;         // should we allow this?
```

The dynamic semantics at first sight look easy. The *ForStatement* is often explained to beginners as being a shorthand form of writing a *while* loop - indeed in the C family of languages it is deemed to be just that. For example, the statements

```
for i = 1 to 10  write(i);
for i = 12 downto 4 write(i);
```

seem to be equivalent to

```
i = 1;                          i = 12;
while (i <= 10) {               while (i >= 4) {
  write(i);                       write(i);
  i = i + 1;                      i = i - 1;
}                               }
```

However, it is not quite as simple as that. Consider an example like

```
for i = i + 4 to i + 10  write(i);
```

Here the obvious equivalent code leads to an potentially infinite loop

```
i = i + 4;
while (i <= i + 10) {
  write(i);
  i = i + 1;
}
```

In principle, the condition `i <= i + 10` would now always be *true*, although the program would probably misbehave when the value assigned to `i` overflowed the capacity of an integer.

Pascal was a much "safer" language than C, and the semantics of the Pascal-style *ForStatement* are better described as follows. The statements

```
for Control = Expression1 to Expression2  Statement
for Control = Expression1 downto Expression2  Statement
```

should be regarded as more closely equivalent to

```
         Temp1 := Expression1                        Temp1 := Expression1
         Temp2 := Expression2                        Temp2 := Expression2
         IF Temp1 > Temp2 THEN GOTO EXIT             IF Temp1 < Temp2 THEN GOTO EXIT
         Control := Temp1;                           Control := Temp1;
BODY:    Statement                          BODY:    Statement
         IF Control = Temp2 THEN GOTO EXIT           IF Control = Temp2 THEN GOTO EXIT
         Control := Control + 1                      Control := Control - 1
         GOTO BODY                                   GOTO BODY
EXIT:                                       EXIT:
```

respectively, where `Temp1` and `Temp2` are temporary variables. This code will not assign a value to the control variable at all if the loop body is not executed, and will leave the control variable with the "obvious" final value if the loop body is executed. Code generation for these semantics may appear to be a little awkward for an incremental compiler, since there are now multiple apparent references to extra variables and to the control variable.

Hint: The simplest way of handling all these issues in the PVM system is to note that the obvious calls to the parsing routines to handle the sequence

```
Control
Expression1
Expression2
```

will ensure that code to push the address of the control variable and the values of the temporary variables will have been generated by the time the *Statement* forming the loop body is encountered. At this point code must be generated for the initial test, and if we avail ourselves of the ability to define extensions to the opcode set of the PVM we can generate a special opcode at this point that will perform the test, but leave these three values on the stack so that they can be used ag/ain later. Similarly, after generating the code for the *Statement* we can generate a second special opcode that can be responsible for the final test and possible increment of the control variable. One last complication is that once the for loop completes its execution we have to discard these three elements from the stack, which suggests a variation on the use of the DSP opcode.

Develop these ideas in detail. Ensure that your loop also supports the *break* statement. Finally, insist on type compatibility between the control variable and the expressions defining its initial and final values.

Something else to think about. In Pascal the word `do` is also required, as illustrated below. Would it be a good idea to insist on it in Parva as well?

```
ForStatement = "for" Ident "=" Expression ("to" | "downto") Expression "do" Statement .
```

Task 13 - Beg, borrow and steal ideas from other languages

Suppose we extend Parva to adopt an idea used in Pascal, where a statement like

```
write(X : 5, X + A : 12, X - Y : 2 * N);
```

will write the values of `X`, `X+A` and `X-Y` in fields of widths `5`, `12` and `2*N` respectively.

Note that a statement like

```
write(3 + 4, A < B);
```

should produce numeric output for the first element, and Boolean output for the second one. But one like this

```
write(3 + 4 : A < B)
```

should not be allowed, because the (optional) formatting expression is meaningless unless it is numeric.

Hint: This is best handled by producing new opcodes based on the `PRNI` and `PRNB` opcodes already in the PVM, and these can make use of the formatted output routines which are in the `library` supplied with the distribution kits. But leave the original opcodes in the system as well!

Task 14 - Switch to Parva - satisfaction guaranteed! (This week's Big Bonus if you get this far.)

The very useful *SwitchStatement* is found in various forms in various languages. One possibility might be described by the productions:

```
SwitchStatement
=  "switch" "(" Expression ")" "{"
      { CaseLabelList Statement { Statement } }
      [ "default" ":" { Statement } ]
   "}" .
CaseLabelList = CaseLabel { CaseLabel } .
CaseLabel     = "case" [ "+" | "-" ] Constant ":" .
```

as exemplified by

```
switch (i + j) {
  case 2   : if (i === j) break; write("i = " , i); read(i, j);
  case 4   : write("four"); i = 12;
  case 6   : write("six");
  case -9  :
  case 9   :
  case -10 :
  case 10  : write("plus or minus nine or ten"); i = 12;
  default  : write("not 2, 4, 6, 9 or 10");
}
```

The semantics here require that the *Expression* be evaluated once, and its value compared for membership of one of the sets of *CaseLabelLists*, a match being followed by execution of the associated statement list. If no match is found, the statement list associated with the optional *default* clause is executed (or, in the absence of the *default* option, no action is taken at all). There are many variations on this theme, but for the one illlustrated here we require that (a) there is an automagic *BreakStatement* introduced at the end of each statement list so that there is no need to include an explicit *BreakStatement* in each statement list, although these may occur if needed (b) the concept of "falling through" in the absence of an explicit *BreakStatement* thus does not apply (c) more than one *CaseLabel* may be associated with a given statement list and (d) each *CaseLabelList* must be separated from the next one by at least one statement in its statement list (even if this is an empty statement).

One way to implement this would be to generate code matching an equivalent set of *IfStatement*s

```
    if    (i + j == 2) { if (i === j) goto out; write("i = " , i); read(i, j); goto out; }
    elsif (i + j == 4) { write("four"); i = 12; goto out; }
    elsif (i + j == 6) { write("six"); goto out; }
    elsif (i + j in (-9, 9, -10, 10)) { write("plus or minus nine or ten"); i = 12; goto out; }
    else write("not 2, 4, 6, 9 or 10");
out: ...
```

but it should not take much imagination to see that we might initially be at a loss for a way repeatedly to inject the code for the selector *Expression* into the code generation process if we are restricted to use an incremental compilation technique. One approach is to use the DUP opcode already suggested for the PVM, which will duplicate the element on the top of stack. This is applied before each successive comparison or test for list membership is effected, so as to make sure that the value of the expression is preserved, in readiness for the next comparison.

Although this idea does not lead to a highly efficient implementation of the *SwitchStatement*, it is relatively easy to implement - the complexity arising from the need, as usual, to impose semantic checks that all labels are unique, that the type of the selector is compatible with the type of each label, and from a desire to keep the number of branching operations as low as possible.