Computer Science 3 - 2016

Programming Language Translation

Practical 7: Week beginning 5 September 2016

This extended prac is designed to take you the best part of two weeks. Hand in your solutions *before* lunch time on **Wednesday 21 September**, 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 g14A1234.** 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.

I shall try to get the marking done as soon as possible after 21 September, and may release solutions before then.

Objectives:

In this practical you are to

- familiarize yourself with a compiler largely described in chapters 12 to 14 that translates Parva to PVM code.
- extend this compiler in numerous ways, some a little more demanding than others.

This prac sheet is at `http://www.cs.ru.ac.za/courses/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 by using the LPRINT utility, as the listings get wide. Please ensure that the lines do all "fit" (into less than 120 characters) - lay out your grammars neatly! **It would also help if you could use a highlighter to show where you have made changes, or else just edit out the changed sections and print those**.
- Some examples of very short test programs and the corresponding PVM code as generated by your compiler.
- Electronic copies of your complete solutions.

I do NOT require listings of any 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 on the university website.

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 may 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.

Please resist the temptation simply to copy code from model answers issued in previous years.

This version of Parva has been developed from the system described in Chapters 12 and 13, extended on the lines suggested in Chapter 14.4, so as incorporate void functions with value parameters.

The operator precedences in Parva as supplied use a precedence structure based on that in C++. 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 172) and deals with type compatibility issues (see section 12.6.8). The supplied compiler also incorporates the `char` type.

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, especially as they deal with operator precedence and the `char` type. 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, hacking often leads to wasting more time than it saves. Finally, *please* 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 130) to allow pragmas like those shown to have the desired effect (see the sample program above).

```
$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 `PRAC7.ZIP`.

- Immediately after logging on, get to the command line level as usual!

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

```
J:
md  prac7
cd  prac7
copy  i:\csc301\trans\prac7.zip
unzip  prac7.zip
```

  This will create another directory "below" the `prac7` directory:

```
J:\prac7
J:\prac7\Parva
```

  containing the C# classes for the code generator, symbol table handler and the PVM.

  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,     Examples\*.PAV
```

- As usual, you can use the `CMAKE` command to rebuild the compiler, and a command like `Parva EG.PAV` to run it.

You should attempt all of Tasks 2 through 11. Perfect answers to these would earn you a mark of 100% for this prac. Bonus marks can be earned for attempting any or all of Tasks 12, 13 and 14. You are urged to do these - more for the insight they will give you in preparing for the final examination than for the marks!

Task 2 - Use of the debugging and other pragmas

We have already commented on the `$D+` pragma. How would you add to 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?

Suggested pragmas would be (to be included in the source being compiled at a convenient point):

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

while the effect of `$C+` might more usefully be achieved by using the compiler with a command like

```
Parva Myprog.pav -c          (produce .COD file)
Parva Myprog.pav -c -l       (produce .COD file and merge error messages)
```

Other useful debugging aids are provided by the `$ST` pragma, which will list out the current symbol table. Two more are the `$SD` and `$HD` pragmas, which will generate debugging code that will display the state of the run-time stack area and the runtime heap area at the point where they were encountered in the source code. Modify the system so that these are also dependent on the state of the `$D` pragma. In other words, the stack dumping code would only be 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: These additions are almost trivially easy. You will also need to look at (and modify) the `Parva.frame`

file, which is used as the basis for constructing the compiler proper (see page 143).

Task 3 - 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 in C# by `==`, `!=` and `=` with the similar operators in Pascal denoted by `=`, `<>` and `:=`, and are prone to introduce words like `then` into an *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 that 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` and then assigns y to x if a is non-zero, as you probably know).

The remaining tasks all involve coming to terms with the code generation process.

Task 4 - Two of the three Rs - Reading and Writing

Extend the *WriteStatement* to allow a variation introduced by a new key word `writeLine` that automatically appends a line feed to the output after the last *WriteElement* has been processed.

Extend the *ReadStatement* to allow a variation introduced by a new key word `readLine` that causes the rest of the current data line to be discarded, so that the next *ReadStatement* will start reading from the next line of data.

Extend the *HaltStatement* to have an optional string parameter that will be output just before execution ceases (a useful way of indicating how a program has ended prematurely).

Task 5 - Repeat discussions with your team mates until you all get the idea

As very easy exercise, add a Pascal-like *Repeat* loop to Parva, as exemplified by

```
repeat  a = b; c = c + 10; until (c > 100);
```

If you feel so inclined, add a *DoWhile* loop as well, as exemplified by

```
do { a = b; c = c + 10; } while (c < 100);
```

There is no real reason why your compiler cannot allow both statement forms, of course.

Task 6 - It should only take a MIN or two to derive MAX benefit from these tasks

Earlier in the course we explored adding opcodes like `MIN`, `MAX` and `SQR` to the PVM. This week, modify Parva and the PVM to allow for expressions in which might appear `Min()` and `Max()` functions, which can take one or more arguments, for example

```
Write(Max(90), Min(3, 5, 2, 12), Max(min(2, 4), Min(21, 4)));
```

You should start by checking that you know what output might be expected from the execution of that statement!

When you have gone sqr-eyed from all that effort, you should find adding the `Sqr()` function to be a breeze.

Task 7 - You had better do this one or else….

Add the *else* option to the *IfStatement*. Oh, yes, it is trivial to add it to the grammar. But be careful. Some *IfStatements* will have *else* parts, others may not, and the code generator has to be able to produce the correct code for whatever form is actually to be compiled. The following silly examples are all valid.

```
if (a == 1) { c = d; }
if (a == 1) {}
if (a == 1) {} else {}
if (a == 1) ; else { b = 1; }
```

Implement this extension (make sure all the branches are correctly set up). By now you should know that the obvious grammar will lead to LL(1) warnings, but that these should not matter.

Task 8 - 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 loops, but there might be several break statements inside a single loop, and loops can be nested inside one another.

Task 9 - Make the change - upgrade now to Parva++

At last! Let's really make Parva useful and turn it into Parva++ by adding the increment and decrement *statement* forms (as variations on assignments, of course), 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, and the necessary opcodes are already waiting for you in the PVM. Be careful - only integer and character variables (and array elements) can be handled in this way. **Do not bother with trying to handle the ++ and -- operators within terms, factors, primaries and expressions**.

Task 10 - The array is the object of the exercise

This one should be easy. Add a little function into your expression parsers to allow you to determine the length of an array, which might be useful in writing code like

```
void DisplayList(int [] list) {
// Display all the values in list[0] .. list[length - 1]
  int i = 0;
  while (i < Length(list)) (
    write(list[i]);
    ++i;
  }
} // DisplayList
```

Hint: You may have forgotten (or have yet to discover) that the heap manager, when allocating space for an array, also records the length of that array. As a further hint, you shouldn't need any new opcodes in the PVM.

Hopefully you will understand that the output from the following program would be as given in the comments

```
void main() {
// Some misconceptions about arrays
  int[] array1, array2;
  array1 = new int[45];
  array2 = new int[45];
  write(array1 == array2);        // false, even though of the same length
  array1[0] = 6;
  array2 = array1;
  array2[0] = 8;
  write(array1 == array2, array1[0], array2[0]);   // true 8 8   and not  false 6 8
} // main
```

because the manipulations of `array1` and `array2` are dealing with *references* to the array objects and not to the *contents* of the objects being referenced.

Very well. This would confuse beginners, and that Won't Do. Extend Parva so that the following variation on the program would produce the output shown:

```
void main() {
// Some misconceptions about arrays
  int[] array1, array2;
  array1 = new int[45];               // all elenents originally zero
  array2 = new int[45];               // all elenents originally zero
  write(Equals(array1, array2));      // true
  array1[0] = 6;
  array2 = Copy(array1);
  array2[0] = 8;
  write(Equals(array1, array2), array1[0], array2[0]);   // false 6 8
} // main
```

You might like to think whether "OOP-like" syntax more like `array.Length, array1.Copy()` and `array1.Equals(array2)` would be preferable to the "function call" syntax suggested in the last examples, and what the implications of this syntax might be. (There is no need to implement it this time, unless you really want to do so.)

Task 11 - 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 variable declarations of a 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 12 - 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, PRNC` 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 13 - Let's untangle that heap of string

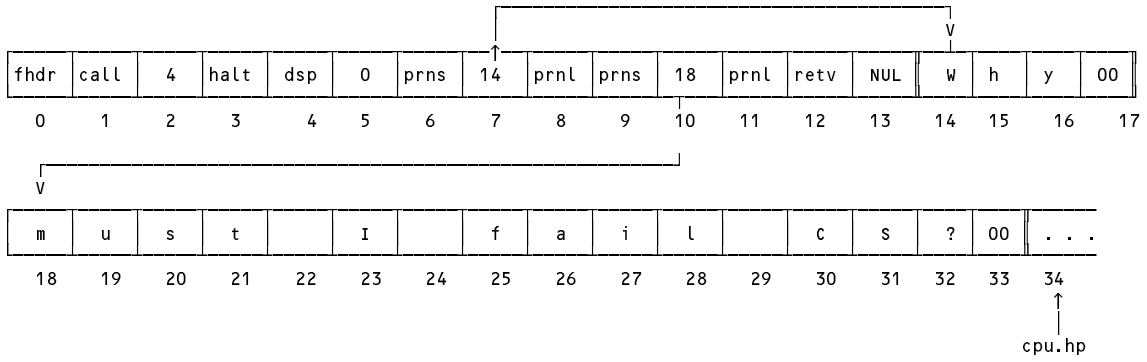
Things are getting more interesting by this stage, and more challenging.

By now you should be well aware that any strings encountered in a Parva program are stacked at the top end of memory, whence they are extracted by the PRNS opcode in the PVM.

There is an alternative model, one that is used in a hardware implementation of the PVM developed by an exceptional former student in this course, called the PAM (Parva Actual Machine). In this system the strings are required to be stored above the program code - effectively in space that the PVM would otherwise have used for the start of its heap. A very simple program like

```
void main () {
// A fatalistic approach to the oncoming examinations?
  writeLine("Why");
  writeLine("must I fail CS?");
}
```

might be stored in memory like this



Modify the code generator and the PVM to use this model.

Hint: You might like to associate a label with each PRNS operation, since at the time you want to generate the opcode you won't know exactly where the string will be stored. If you like developing tight code suited to a small machine like the PAM, you might think of optimising the analysis and code generation so that if the same string appears in the program in two or more places, only one copy is loaded in memory.

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 for its long overdue addition to Parva might be described by the productions:

```
SwitchStatement
=  "switch" "(" Expression ")" "{"
      { CaseLabelList Statement { Statement } }
      [ "default" ":" { Statement } ]
   "}" .

CaseLabelList
= CaseLabel { CaseLabel } .

CaseLabel
= "case" [ "+" | "-" ] Constant ":" .
```

Being a generous fellow, I have added these productions to the `Parva.atg` file already. So your Parva compiler can already recognise a *SwitchStatement*, it just can't generate code to make it work. But I'll leave it to you to handle the static semantic checking and the code generation.

A demonstration (if rather silly) example of a *SwitchStatement* is

```
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 implemented in your 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.

Have fun, and good luck.