Computer Science 3 - 2015

Programming Language Translation

Practical 7: Week beginning 12 October 2015

This extended prac is designed to take you the best part of two weeks. Hand in your solutions *before* lunch time on **Wednesday 28 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 g13A1234.** 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 28 October is to free you up to prepare for the main examinations. I shall try to get the marking done as soon as possible after 28 October, 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 on the laser printer 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**.

- Some examples of very short test programs and the corresponding PVM code as generated by your compiler.

- Electronic copies of your 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, so as incorporate void functions with value parameters. This is by nature of an experiment: in past years there has been no practical work using ideas developed in chapter 14 of the text.

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 167) and deals with type compatibility issues (see section 12.6.8). The supplied compiler also incorporates the `char` type, but has left room for improvement in this regard as you will see.

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 128) 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 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,    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 13. 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 14 through 17. 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?

The suggested pragmas would be (included in the source being compiled):

```
          $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 `$HD` pragmas, which will generate debugging code that wiill display the state of the run-time stack area and the runtime heap area at the point where they were encountered in the source code. Modify the system so that these are also dependent on the state of the `$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 140).

Task 3 - Things are not always what they seem

Although not strictly illegal, the appearance of a semicolon in a program immediately following the condition in an *IfStatement* or *WhileStatement*, or immediately preceding a closing brace, may be symptomatic of omitted code. The use of a so-called *EmptyStatement* means that the example below almost certainly will not behave as its author intended:

```
read(i);
while (i > 10);
{
  write(i);
  i = i - 1;
}
```

It should be possible to warn the user when this sort of code is parsed; do so. Here is another example that might warrant a warning

```
while (i > 10) {  }
```

Warnings are all very well, but they can become irritating. Use a `$W-` pragma or a `-w` command line option to allow advanced users to suppress warning messages.

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

Task 4 - You had better do this one or else....

Add the *else* and *elsif* options to the *IfStatement*. Oh, yes, it is trivial to add them 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 eXdetaltension (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 will not matter.

Task 5 Something to do - while you wait for inspiration

Add a *DoWhile* loop to Parva, as exemplified by

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

Task 6 - Repeat discussions with your team mates until you all get the idea

As an alternative to the previous exercise, and a very easy one, add a Pascal-like *Repeat* loop to Parva, as exemplified by

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

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

Task 7 - 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 8 - Your professor is quite a character

Parva is looking closer to C/C++/C# with each successive long hour spent in the Hamilton Labs. Seems a pity to stop now, so I have already extended the system to allow for a character type as well as the integer and Boolean ones, enabling us all 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 = i + 1;
    read(ch);
  }
  while (i > 0) {      // output loop
    i = i - 1;
    write(sentence[i]);
  }
}
```

A major part of this exercise was 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 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 9 - Let's dive in where C programmers fear to go

In an earlier practical we suggested that Parva might be extended to provide a relational operator for determining whether the value of an expression matched one of a set of values in a list. Extend the language and its implementation to allow expressions like these:

```
exp in (a, b, c)
itsASmallPrime = i in (2, 3, 5, 7, 11, 13);
if (list[i] in (i, 2 * i, 3 * i)) write("list[i] is a small multiple of its index");
```

Hint: This is most easily done by adding a new opcode to the PVM, to be used when the list has more than one element. Once again, remember to handle type checking.

Task 10 - 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.

As an alternative to the more familiar Pascal-style *ForStatement*, implement one modelled on a syntax suggested by Python:

```
ForStatement = "for" Ident "in" "(" Expression { "," Expression } ")" Statement .
```

For example

```
int i;  // small primes
for i in (2, 3, 5, 7, 11)
  write(i, " is a small prime");

bool a, b;  // truth tables
for a in (false, true))
  for b in (false, true)
    writeLine(a, b, a || b, a && b);
```

Hint: Once again, you might like to add some more operations to the PVM to assist in this regard. As before, ensure that your loop also supports the *break* statement. And, as always, insist on type compatibility between the control variable and the elements of the list.

Task 11 - 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 (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 expressions**.

Task 12 - If you survive this I'll pass you a reference so that employers will know your value

As supplied, the Parva compiler can only pass parameters "by value". 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
```

Be careful - you must put in checks for compatibility and may have to think carefully what this means in this context.

Task 13 - Time to resolve an awkward LL(1) conflict

You have probably spent considerable time fighting non-LL(1) productions over the last few weeks. Many of these can be resolved by cleverly factoring the grammar, but sometime this is messy. Indeed, in the compiler you have been given there is a wonderful example of this:

Consider the productions which have been written

```
AssignmentOrCall = Designator ( "(" Arguments ")"  |  AssignOp Expression ) ";"  .
Designator       = identifier [ "[" Expression "]" ] .
```

That is too "loose". Syntactically this would accept something like `function[6](a,b,c)` as some sort of function call. The compiler you have been given handles this problem "semantically" (make sure you understand why!)

It would be syntactically far preferable to try to write

```
AssignmentOrCall = ( identifier "(" Arguments ")"     |     Designator AssignOp Expression  ) ";"  .
Designator       = identifier [ "[" Expression "]" ] .
```

but then we have an LL(1) conflict because both options ultimately start with an identifier.

Coco/R has a clever way of being able to use semantic information in so-called "resolvers". At the time when I wrote the book, Coco/R was strictly LL(1) only; the idea of being able to resolve some conflicts, essentially by possibly looking ahead, was only worked into the system just before the book was published. However, the technique is discussed briefly on pages 135 - 136 and in more detail on page 14 - 20 of the Coco/R reference manual which is on the course website. As an interesting exercise - the first of its kind in the courses I have offered - read up on the technique and then use it to deal properly with the *AssignmentOrCall* production.

Task 14 (Bonus) - Do you ever feel threatened by change?

Do you think you should be allowed to change the value of the control variable associated with a *ForStatement* within the loop body - for example, what is your reaction to seeing code like

```
int i;  // small primes
for i in (2, 3, 5, 7, 11) {
  write(i, " is a small prime");
  i = 13;  // should this be allowed?
}
```

or, if you were using a Pascal-like form of the loop - and making the point more strongly:

```
for i = 1 to 10 {
  for i = 1 to 10  // nested loops are allowed, but surely not this one?
    write(i);
  writeLine();
}
```

See if you can find a way to prevent "threatening" (tampering with) the control variable within the body of the loop. And how, if at all, might this facility interact with what you developed for Task 12?

Task 15 (Bonus) - Some staff will go to any length to process a list of marks completely

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 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.

Task 16 (Bonus) - 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 (default) or code generation using a Boolean operator approach (see the textbook, pages 12 and 167). All the opcodes you need are already in the source kit.

Task 17 (Bonus) - Generating tighter PVM code

Way back in Practical 2 we added some specialized opcodes like `LDC_1`, `LDA_2` and so on to the PVM. They are still there in the version supplied with the kit. Seems a shame not to use them, so modify the code generator to achieve this for you.

Making use of `LDL N` and `STL N` in place of `LDA`, `LDV` and `STO` where possible is a bit more challenging, but for even more bonus marks, feel free to experiment if you wish.

Have fun, and good luck.