Computer Science 3 - 2014

Programming Language Translation

Practical 7: Week beginning 19 May 2014

This extended prac is designed to take you the best part of two weeks. Hand in your solutions *before* lunch time on **Wednesday 4 June**, 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 4 June is to free you up to prepare for the main June examinations. I shall try to get the marking done as soon as possible after 4 June.

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.

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 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 by following the links at:

`http://www.scifac.ru.ac.za/` or from `http://www.ru.ac.za/`

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.

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

This version of Parva has been restricted so as not to include Parva 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 167) 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 128) 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 `PRAC7.ZIP` (Java) or `PRAC7C.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  prac7
     cd  prac7
     copy  i:\csc301\trans\prac7.zip      (Java)
or   copy  i:\csc301\trans\prac7c.zip     (C#)
     unzip  prac7.zip
```

  This will create several other directories "below" the `prac7` directory:

```
J:\prac7
J:\prac7\library
J:\prac7\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,     Examples\*.PAV
```

- 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 - 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?

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

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 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: These additions are 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 140).

Task 3 - How long is a piece of string?

Why do you suppose languages generally impose a restriction that a literal string must be contained on a single line of code?

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. This provides a mechanism for breaking up strings that are too long to fit on one line of source code. Add this feature to the Parva compiler. It 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, allow this concatenation operator as an option between string literals that are to be concatenated.

Task 4 - 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.

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 - Suppressing some error messages

Identifiers that are undeclared by virtue of mistyped declarations tend to be annoying, for they result in many subsequent errors being reported. Implement a strategy where each undeclared identifier is flagged as undeclared at the point of first reference, and then quietly entered as a variable of the `noType` pseudo-type in the symbol table. Can you extend the idea to recognize an undeclared array reference variable and thereby reduce the plethora of "unexpected subscript" messages that this non-declaration might otherwise generate?

Task 7 - A task to keep you on the right lines

The remaining tasks all involve coming to terms with the code generation process. The first one is very easy.

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.

Task 8 - You had better do this one or else….

Add an *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 this will lead to LL(1) warnings, but if you get the system correct these will not really matter.

Task 9 - 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 10 - Break over - let's continue

The *ContinueStatement* is also syntactically simple, so while you are at it, implement it too - of course the same sorts of constraints apply.

Task 11 - Some ideas are worse than a snake in the grass

If you know any Python you may have been impressed by a Python feature which allows one to write multiple assignments into a single statement, as exemplified by

```
A, B = X + Y, 3 * List[6];
A, B = B, A;                // exchange A and B
A, B = X + Y, 3, 5;         // incorrect
```

which can obviously be described by the context-free production

```
Designator { "," Designator } "=" Expression { "," Expression } ";"
```

Describing the syntax is one thing. Getting the semantics and code generation right should keep you out of mischief for an hour or two.

Task 12 - Your professor is quite a character

Parva is looking 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 13 - Absolutely, yes, ja, yebo …

Add a simple `abs` function to the expression grammar (take care to get the precedence and semantic checking correct, as it is trivial to add it to the grammar. The code generator has and PVM will need some simple extensions too, of course.

```
int
  i = abs(-24),
  j = abs(abs(i) * abs(-24 * i);
```

Task 14 - 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.

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. So, 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
        Temp2 := Expression2
        IF Temp1 > Temp2 THEN GOTO EXIT
        Control := Temp1;
BODY:   Statement
        IF Control = Temp2 THEN GOTO EXIT
        Control := Control + 1
        GOTO BODY
EXIT:
```

```
        Temp1 := Expression1
        Temp2 := Expression2
        IF Temp1 < Temp2 THEN GOTO EXIT
        Control := Temp1;
BODY:   Statement
        IF Control = Temp2 THEN GOTO EXIT
        Control := Control - 1
        GOTO BODY
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 again 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* and *continue* statements. Secondly, insist on type compatibility between the control variable and the expressions defining its initial and final values. And, lastly, see if you can find a way to prevent "threatening" (tampering with) the control variable within the body of the loop, thus forbidding this sort of nonsense:

```
for i = 4 to 10  {
  writeLine(i);
  i = 5;
}
```

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 15 - 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 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. Be careful - only integer and character variables (and array elements) can be handled in this way. Do not bother with the ++ and -- operators within *expressions*.

Have fun, and good luck.