RHODES UNIVERSITY
Where leaders learn

Computer Science 301

Programming Language Translation

Practical 8: week beginning 16 September 2024

NB: Task 6 must be submitted via the Task6 submission link by 5pm on Thursday 19 September 2024.

Task 7 must be submitted through the Tutor RUC submission link and all remaining tasks through the Lecturer link by Wednesday 25 September 2024.

Reminder: There will be a written exam-like test on the 19th September 2024 starting at 2pm sharp in the lab

---

Objectives

In this practical you will

- familiarize yourself with a compiler largely 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.

Outcomes

When you have completed this practical you should understand:

- code generation for a simple stack machine.

To hand in (36 marks)

- An electronic copy of your grammar file (Parva.atg).
- Electronic copies of the auxiliary source files (i.e., files in the Parva subdirectory) if these have been changed.

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

For this practical, tutors will mark Task 7, and I will mark one or more of the other tasks.

Feedback for the tasks marked will be provided via RUConnected. Check carefully that your mark has been entered into the Departmental Records.

You are expected to be familiar with the University Policy on Plagiarism and to heed the warnings in previous practical handouts.

Task 0 Unpacking the prac kit

Note that the starting kit for this prac is where you left off in Practical 7. For those who did not manage to create a working parser at the end of Practical 7, you might like to start fresh with the solution to the previous practical provided as `prac8.zip`.

Note that the tasks in this practical all involve coming to terms with the code generation process. Ensure that for each of these tasks, you include all the parsing, semantic constraint checking, and code generation annotations in the grammar and any required extra code in the auxilliary files, like e.g. new opcodes.

Task 6 You had better do this one or else … (5 marks)

NOTE: This task must be handed in by 5 pm on the prac afternoon

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) c = d; else b = 0;
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 7 Fixing what couldn't be executed last week (8 marks)

Extend the solution to Tasks 4 and 5 from Prac 7 to include the correct code generation for the new statements and expression functions introduced, namely, repeat, do while, sqr() and sqrt(). Note that you **might** need to introduce new opcodes into the PVM to deal with handling sqr() and sqrt().

Task 8 Break, then continue with the good work (10 marks)

The *BreakStatement* and *ContinueStatement* are syntactically simple, but take a lot of thought. Give it some! Be careful – break can only appear in loops or the switch statement (which we don't yet support) while continue can currently only appear within loops. Be aware that there might be several break/continue statements inside a single loop, and loops can be nested inside one another.

Also ensure that you include support for using break/continue in all the looping statements that currently exist in this grammar.

Task 9 What are we doing this for? (13 marks)

Many languages provide a *ForStatement* in one or other form. Although most people are familiar with these, their semantics and implementation can actually be quite tricky. In a previous practical, you introduced the Pascal for loop into a Parva grammar, so the basic syntax should be familiar:

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

Some examples:

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

Optional: See if you can find a way to prevent "threatening" (tampering with) the control variable within the body of the loop, thus forbidding this kind of nonsense:

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