RHODES UNIVERSITY
Where leaders learn

Computer Science 301

Programming Language Translation

Practical 7: week beginning 9 September 2024

NB: Task 1 must be submitted via the Task1 submission link by 5pm on Thursday 12 September 2024.

Tasks 2+3 must be submitted through the Tutor RUC submission link and all remaining tasks through the Lecturer link by Wednesday 18 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 the front-end of a compiler described in Chapter 12 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:

- several aspects of semantic constraint analysis in an incremental compiler.

To hand in (23 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 Tasks 2 and 3, 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.

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.

Please resist the temptation simply to copy code from model answers issued in previous years. If you do, you may find that they do not solve the problems posed in this practical, and you will then be given zero for the entire practical as plagiarism will be clearly apparent.

This version of Parva has been developed from the system described in Chapters 12 and 13 of the notes. The operator precedence in Parva as supplied uses a precedence structure based on that in C++, C# or Java, rather than the "Pascal-like" one used in the notes. Study these carefully and note how the compiler provides "short-circuit" semantics correctly (see Section 13.5.2) and deals with type compatibility issues (see Section 12.6.8).

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 small basic 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 notes, page 156) 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 0 Creating a working directory and unpacking the prac kit

Unpack the prac kit `PRAC7.ZIP`.

- You will find a `Parva` directory "below" the `prac7` directory, containing the Java source for the code generator, symbol table handler and PVM interpreter.
- 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 to run it, a command like

```
crun Parva testfile.PAV
```

Task 1 Use of the debugging and other pragmas [4 marks]

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

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 can be provided by the `$ST` pragma, which will print out the current symbol table. Two more are the `$SD` and `$HD` pragmas, which will generate debugging code that will display the state of the runtime stack area and the runtime heap area at the point where they were encountered in the source code. Modify the system so that these three pragmas are available depending on the state of the `$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.

Hint: These additions are almost trivially easy. However, you will also need to look at (and modify) the `Parva.frame` file (see Section 10.6 of the course notes).

Task 2 Things are not always what they seem (4 marks)

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; implement this. Here is another example that you should also implement by providing a warning:

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

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

Task 3 Improving Reading, Writing and Halting (6 marks)

a) 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).

e.g. `halt("program terminated because … ");`

Make sure that your parser still accepts the normal `halt;` statement as well

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

c) Extend the *ReadStatement* to allow a variation introduced by a new keyword `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.

Note that for this task you should also make any code generation changes needed to produce the correct code for the new statements. This should be fairly trivial after considering what is done in the code generation for the `read` *and* `write` *statements.*

Task 4 Repeat discussions with your team mates until you get the idea (4 marks)

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

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

For added practice, add a d*o-while* loop as well, as exemplified by

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

Note: You do not need to add in any code generation annotations at this stage, but you should add in any semantic checks needed.

Task 5 This may make you square-eyed (5 marks)

Extend the compiler to allow support for the `sqr()` and `sqrt()` functions. Here `sqr(a)` means `a * a` and `sqrt()` denotes the square root function.

Ensure that your addition allows syntax such as that given below, as well as supporting all other valid expressions:

```
write(sqr(9));
a = sqrt(b*(1+c));
c = sqr(sqrt(c));
```

Note: You do not need to add in any code generation annotations at this stage, but you should add in all semantic checks needed!
