RHODES UNIVERSITY
Where leaders learn

Computer Science 301

Programming Language Translation

Practical 5: week beginning 5 August 2024

All tasks in this practical should be submitted through the RUC submission link on the Wednesday before your next practical day. For this practical, please submit the file that has been completed at the end of task 2 (call this `PVMChecker2.java`) via the TUTOR marking submission link and then the two separate versions for Task 3 and 4 created for the full parser with one or more test files (other than the provided test files) used to test it via the LECTURER marking submission link.

Objectives

In this practical you will

- develop a recursive descent parser and associated *ad hoc* scanner "from scratch" that will parse a small assembler language similar to the PVM one that you have previously used.

Outcomes

When you have completed this practical you should understand:

- the inner workings of an ad hoc scanner;
- the inner workings of a recursive descent parser;
- how to test that a scanner and parser behave correctly.

To hand in (30 marks)

This week you are required to hand in via the link on RUConnected:

- Electronic copies of the source files for your parser after completing Task 2 (`PVMChecker2.java`), Task 3 (`PVMChecker3.java`) and finally, Task 4 (`PVMChecker4.java`).
- One or more input test files used in the testing.

WARNING. This exercise really requires you to do some real thinking and planning. Please do not just sit at a computer and hack away. I suggest you think carefully about your ideas and discuss these with one another and with the demonstrators. If you don't do this you will probably find that the whole exercise turns into a nightmare, and I don't want that to happen. Remember to acknowledge with whom you have worked.

For this practical, tutors will mark Task 2, the scanner (without comments), and I will mark Task 3, the parser and/or Task 4, handling comments.

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 warning in previous practical handouts regarding “working with another student”.

Task 0 Creating a working directory and unpacking the prac kit

Unpack the prac kit `PRAC5.ZIP`. In it you will find the skeleton of a system adapted for intermediate testing of a scanner (and to which you will later add a parser), and some simple test data files -- but you really need to learn to develop your own test data.

Task 1 Get to grips with the problem

In previous practicals you have used the PVM assembly language to code various problems. However, many of you will have struggled to get the destination addresses correct for operands that require a code memory address. Image how much easier coding would have been if you were allowed to use labels as the destinations of branch instructions, for example:

```
BEGIN
         DSP 2
loop:    LDA 1
         LDA 1
         LDV
         LDC 1
         ADD
         STO
         LDA 0
         LDV
         LDA 1
         LDV
         CEQ
         BZE loop
END .
```

Given below is a grammar for a simple assembler language that can parse code like the above. Note that this grammar does not support all the PVM opcodes and it also requires some additional code at the start and end of a file:

```
COMPILER PVMlike

CHARACTERS
  lf            = CHR(10).
  letter     = "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz" .
  digit      = "0123456789" .

TOKENS
  identifier = letter { letter | digit } .
  number     = [ '+' | '-' ] digit { digit } .
  label      = letter { letter | digit } ":" .
  EOL        = lf .

IGNORE CHR(9) .. CHR(13) - lf

PRODUCTIONS
```

```
  PVMlike   = { EOL } "BEGIN" { EOL } { Statement } "END" { EOL } "." { EOL }
.
  Statement = [label] (OneWord | TwoWord | Branch ) EOL .
  OneWord   = ( "ADD"  | "CEQ"   | "CNE" | "INPI" | "LDV"  | "PRNI" | "STO" )  .
  TwoWord   = ( "DSP" | "LDC" | "LDA" )  number  .
  Branch        = ( "BRN" | "BZE" ) ( number  | identifier ) .
END PVMlike.
```

Tempting as it might be simply to use Coco/R to produce a parser that will parse the PVM input, this week you must produce such a parser more directly, by developing a program in the spirit of the one you will find in Chapter 8 of the notes.

The essence of this program is that it will eventually have a main method that will

- use a command line parameter to retrieve the file name of a data file;
- from this file name derive an output file name with a different extension;
- open these two files;
- initialize the "character handler";
- initialize the "scanner";
- start the "parser" by calling the routine to parse the goal symbol;
- close the output file and report that the system parsed correctly or not.

In this practical you are to develop such a scanner and parser, which you should try in easy stages. So for Task 1, study the grammar above and the skeleton program from the kit (`PVMChecker.java`). In particular, note how the character handler (specifically `getChar()`) has been programmed. Note that each section is clearly demarcated by comments in the skeleton program.

Task 2 First steps towards a scanner [10 marks]

Next, develop the scanner by completing the `getSym` method, whose goal in life is to recognize tokens. Tokens for this application could be defined by an enumeration of

```
noSym, eolSym, labelSym, numSym, identSym, beginSym, endSym,
   addSym, ceqSym, cneSym, brnSym, bzeSym, stoSym, prniSym,
   inpiSym, ldaSym, ldcSym, ldvSym, dspSym, periodSym, EOFSym
```

The scanner must be developed on the understanding that an initial character `ch` has been read. When called, it must (if necessary) read past any "white space" in the input file (although this white space does not include end-of-line characters) until it comes to a character that can form part (or all) of a token. It must then read as many characters as are required to identify a token, and assign the corresponding value from the enumeration to the `kind` field of an object called, say, `sym` -- and then read the next character `ch` (remember that the parsers we are discussing always look one position ahead in the source).

Test the scanner with a program derived from the skeleton, which should be able to scan the test file and simply tell you what tokens it can find, using the simple loop in the `main` method as supplied. At this stage do not construct the parser, or attempt to deal with comments. A simple data file for testing can be found in the files `test.pvm`, a longer one in `test0.pvm`.

You can compile your program by giving the command

```
javac PVMChecker.java
```

and can run it by giving a command like

`java PVMChecker test.pvm` or `java PVMChecker test0.pvm`

You cannot possibly expect to start on Task 3 until such time as the scanner is working properly, so test it thoroughly!

Please submit the code for your GetSym() and any helper routines used in the scanner, i.e. PVMChecker2.java

---

Task 3 At last, a parser! [15 marks]

This task is to develop the associated parser as a set of routines, one for each of the non-terminals suggested in the grammar above. These methods should, where necessary, simply call on the `getSym()` scanner routine to deliver the next token from the input. As discussed in Chapter 8, the system hinges on the premise that each time a parsing routine is called (including the initial call to the goal routine) there will already be a token waiting in the variable `sym`, and whenever a parsing routine returns, it will have obtained the follower token in readiness for the caller to continue parsing (see discussion on page 102 of the notes). It is to make communication between all these routines easy that we declare the lookahead character `ch` and the lookahead token `sym` to be fields "global" to the `PVMChecker` class.

Of course, anyone can write a recognizer for input that is correct. The clever bit is to be able to spot incorrect input, and to react by reporting an appropriate error message. For the purposes of this exercise it will be sufficient first to develop a simple routine along the lines of the `accept` routine that you see on page 105, that simply issues a stern error message, closes the output file, and then abandons parsing altogether.

Something to think about: If you have been following the lectures, you will know that associated with each nonterminal *A* is a set *FIRST(A)* of the terminals that can appear first in any string derived from *A*. So that's why we learned to use the `IntSet` class in practical 1! Library routines especially developed for this course, are available in the `library` folder in the kit.

A note on testing:

To test your parser you might like to make use of the data files supplied. A number of these (`testn.pvm`) have correct syntax. Others (`testn.bad`) have some incorrect syntax. Your parser should, of course, be able to parse `test0.pvm` or `test1.pvm` easily. Parsing the `.bad` files with your system will be a little frustrating, as the parser will simply stop as soon as it finds the first error. You might like to create a number of "one-liner" data files to make this testing stage easier. Feel free to experiment! But, above all, do test your program thoroughly.

Please submit the java code file (PVMChecker3.java) for your parser at this point.

Task 4 Some food for thought [5 marks]

Assume that you were asked to allow comments in your PVM code as defined in the Cocol notation below:

```
COMMENTS FROM  ";" TO lf  /* comments are from the ; symbol to the end of line
*/
```

This would allow code of the form:

```
; something
; something else
BEGIN
 start:   DSP 2
          LDA 0
          LDC 4
          STO         ;comment here
END                         ;comment here
.
```

Make the changes to deal with this in your scanner/parser.

Please submit the java code file (PVMChecker4.java) for your parser at this point.