The Pep/Nom Parser and Script Language

1 An overview

This booklet is about the pattern-parsing virtual machine and script language "pep". The executable file is /books/pars/pep and is compiled from the `c` source code "`pep.c`" `/books/pars/object/pep.c`

The virtual machine and language allows simple "LR" bottom-up shift reduce parsers to be implemented in a limited script language with a syntax which is very similar to the "`sed`" unix stream editor.

As far as I am aware, this is a new approach to parsing context-free languages and according to the scripts and tests so far written has great potential. The script language is deliberately limited in a number of ways (it does not have regular expressions, for example), but it seems to be an interesting tool for learning about compiler techniques.

2 Documentation

This file "`pars-book.txt`" is the principle documention about the pep/nom pattern-parser machine and language. This should also be available as **html** at `HTTP://BUMBLE.SF.NET/BOOKS/PARS/PARS-BOOK.HTML`

There is also a lot of documentation in the "`pep.c`" file (which implements the virtual machine) as well as in the "`compile.pss`" file, which converts scripts into machine assembler programs which can then be loaded by the `pep` tool.

Also, most example scripts in the `/books/pars/eg/` folder also have notes at the beginning of the script.

3 Download

A "`.tar.gz`" archive file of the `c` source code can be downloaded from the `HTTPS://SOURCEFORGE.NET/PROJECTS/BUMBLE/` folder.

4 One line examples

The `pep` tool may have useful applications in unix "*one-liners*" but its main power is in the implementation of simple context free languages and compilers.

Remove multiple consecutive instances of any character

```
read; !(==) { put; print; } clear;
```

Double space a text file

```
pep -e "r;'\n'{a'\n';}t;d;" /usr/share/dict/words
```

```
pep -e "r;'\n'{t;}t;d;" /usr/share/dict/words # better
```

The same as above with long command names.

```
pep -e "read; '\n' { add '\n'; } print; clear;" someFile
```

Print a string in reverse order

```
read; get; put; clear; <eof> { get; print; }
```

Convert tabs to 2 spaces

```
read; [\t]{d;a '  ';} t;d;
```

Print all words which end with "ess".

```
pep -e 'r; ![:space:] { whilenot [:space:]; [a-z].E"ess"{add "\n";t; }} d;' /usr/sh
```

Convert all whitespace (eg [\r\n\t\f]) to dots

```
read; [:space:] {d;a'.';} t;d;
```

```
read; [:space:] {clear; add '.';} print; clear;
```

```
read; [ \r\b\t\f] {clear; add '.';} print; clear;
```

Double every instance of vowels

```
pep -e "read; [aeiou] { put; get; } print;clear;" -i "a tree"
```

Only print text within single quotes:

```
read; "'" { until "'"; print; } clear;
```

Remove multiple consecutive instances of the character "a":

```
read; print; "a" { while [a]; } clear;
```

Number each line of the input:

```
read; "\n" { lines; add " "; } print; clear;
```

```
read; "\n" { lines; a " "; } t; d;  # the same, with short commands
```

Print the number of lines in the input stream

```
read; (eof) { add "lines="; lines; add "\n"; print; }
```

The same using the accumulator

```
read; "\n" { a+; } clear; (eof) { add "lines: "; count; print; }
```

Delete leading whitespace (spaces, tabs) from the start of each line:

```
read; print; "\n" { while [:space:]; } clear;
```

Print only lines containing "fox"

```
r; E'\n',(eof) {put; replace "fox" ""; !(==) { swap; print; } clear; }
```

Print only lines in input containing "fox" or "dog"

```
 r; E'\n',(eof) {
put; replace "dog" ""; replace "fox" "";
!(==) { swap; print; } clear;
 }
```

Delete whitespace from the input stream

```
r; ![:space:] {print;} d;
```

Insert 5 blank spaces at beginning of each line (make page offset):

```
r; "\n" { add '     '; } print; clear;
```

Print only the first ten lines of the input stream

```
read; print; clear; lines; "10" {quit;} clear;
```

Delete trailing whitespace (spaces, tabs) from end of each line:

```
read;
[ \t] { while [ \t\r]; read; E"\n" { clear; add "\n"; } }
print; clear;
```

5 Command line usage

Specify script and input on the command line

```
pep -e "read; print; print; clear;" -i "abcXYZ"
# prints "aabbccXXYYZZ"
```

Load a script from file and start an interactive session to view/debug

```
pep -If scriptfile somefile.txt
```

Load an "assembly" file into the machines program and view/debug.

```
pep -Ia asmfile somefile.pss
```

Convert a script to compilable java code

```
pep -f translate.java.pss script
```

6 "Transformations" and compilations

The "pep" virtual machine and language is designed to either check the syntax of input or else transform one textual (data) format into another, or compile/transpile one context-free language into another.

Examples might be:

- ⋆ transform a csv (comma-separated-values) file into a **json** data format.
- ⋆ check if a string is a palindrome, or contains palindromes.
- ⋆ check the syntax of a DNS domain file.
- ⋆ Check if brackets and braces are "balanced".
- ⋆ check the syntax of a JSON text data file or convert to another format.
- ⋆ convert "*markdown*" text (or another plain-text format) into **html** or LaTeX See the script `eg/mark.latex.pss` for an example (that is the script that has generated the current document, with the help of "pdflatex".
- ⋆ convert from "*infix*" arithmetic notation to "*postfix*" notation.
- ⋆ compile a (simple?) computer language into an assembly language.
- ⋆ properly indent a computer language source code file.
- ⋆ Analyse genetic sequences (possibly).

7 Debugging

The pep virtual-machine is more complicated than the "sed" (the unix stream editor) virtual machine (sed only has 2 string registers, the "*work-space*" and the "*hold-space*"). So you may find yourself needing to debug a script. There are a number of ways to do this.

See how a particular script is compiled to "assembler" format

```
pep -f compile.pss script
```

The compiled script will be printed to `stdout` and saved in `sav.pp` or an error message will be displayed if the script has a syntax error.

Sometimes the line above is useful for finding errors in a script which are not caught during the script loading process.

Load a script and view/execute/step through it interactively

```
pep -If someScript input.txt
```

If the script did not compile properly there will only be 1 instruction (quit).

Interactively view how some script is being compiled by "asm.pp"

```
pep -Ia asm.pp someScript
```

```
pep -a asm.pp someScript
```

(Now you can step through the compiled program "`asm.pp`" and watch as it parses and compiles "someScript". Generally, use "*rr*" to run the whole script, and "rrw text" to run the script until the workspace is some particular text. This helps to narrow down where the `asm.pp` compiler is not parsing the input script correctly.

Once in an interactive "`pep`" session, there are many commands to run and debug a script. For example:

- ⋆ empty
- ⋆ n - execute the next instruction in the program
- ⋆ m - view the state of the machine (stack/workspace/registers/tape/program)
- ⋆ rrw <text>- run the script until the workspace is exactly some text.
- ⋆ rre <text>- run script until the workspace ends with something
- ⋆ rrc <num>- run script with <num>characters of input.
- ⋆ rr - run the whole script from the current instruction
- ⋆ M.r - reset the virtual machine and input stream (but not the compiled program)

7.1 Grammar and script debugging

Because pep/nom is a "*filter*" style system (which writes output to "`stdout`"), we can `print` the stack and line/character number after each reduction and watch the grammar in action. This is a ***very*** useful technique for debugging grammars and scripts.

The "`print`" statements are placed just after the "*parse>*" label.

Visualise the stack token reductions with line/character numbers

```
 parse>
add "# "; lines; add ":"; chars; add " "; print; clear;
add "\n"; unstack; print; clip; stack;
```

The "`less`" program makes it possible to search for any particular token by name, to watch it being reduced. We can also seach by input line number.

Watch token reduction and search for reductions

```
pep -f eg/script.pss file.txt  less —
```

Get a unique list of tokens used during parsing

```
pep -f eg/mark.latex.simple.pss pars-book.txt  sed '/
```

7.2 Common bugs

Make sure that you are pushing as many times as there are tokens.

```
add "noun*verb*noun*"; push; push; # << error, 3 tokens, 2 pushes
```

In a block, if you "push" the tokens back, you need to .reparse

```
"article*noun*" {
  clear; add "nounphrase*"; push;
  # error! no '.reparse' command
}
```

Make sure there is at least one read command in the script

```
"."{ clear; } print; clear; # << error: no read in script
```

8 More script examples

Remove whitespace only at the beginning of the input stream

```
begin { while [:space:]; clear; } read; print; clear;
```

Print alphanumeric words in input stream one per line

```
read; [:alpha:] { while [:alpha:]; add "\n"; print; clear; }
clear;
```

Print assignments in the form abc:333 or val = 66

```
r;
":","=" { add "*"; push; }
[:alpha:] {
  while [:alpha:]; put; clear; add "id*"; push; .reparse
}
[0-9] {
  while [0-9]; put; clear; add "num*"; push; .reparse
}
!"" {d;}
 parse>
pop; pop; pop;
"id*:*num*", "id*=*num*" {
  clear;
  ++; ++; get; add " assigned to '"; --; --; get;
  add "'\n"; print; clear;
}
push; push; push;
```

Print the number of alphabetical words in the input stream

```
read; [:alpha:] { while [:alpha:]; a+; } clear;
(eof) { add "Words in file: "; count; add "\n"; print; clear;
 ⇒  }
```

The classic fizzbuzz program

```
# untested
lines;
E"5",E"0" { clear; add "fizz "; print; }
clear; count;
"3" { clear; add "buzz "; print; nolines; }
clear; until "\n"; print;
a+
```

Print words beginning with "www." as html links

```
read;
![:space:] {
  whilenot [:space:];
  B"www." {
    put; clear; add "<a href='"; get; add "'>"; get; add "</a
     ⇒ >";
    add "\n"; print; a+;
  }
} clear;
(eof) { add "Http urls in file: "; count; add "\n"; print;
 ⇒ clear; }
```

9 Machine description

The parsing virtual machine consists of a number of parts or registers which I will describe in the following subsections.

10 Machine elements

stack
: which can contain "parse tokens" if the language is used for parsing or any other text data.

workspace buffer
: This is where all "text change" operations within the machine are carried out. It is similar in concept to a register within a "*cpu*" or to the "`sed`" stream editor pattern space. The workspace is affected by various commands, such as clear, add, get, push, `pop` etc

tape
: which is an array of text data which is synchronized with the machine stack using a tape pointer. The tape is manipulated with the "`get;`" and "`put;`" commands

tape-pointer or the current tape cell
: This variable determines the current tape element which will be used by "`get;`" and "`put;`" commands. The tape pointer is incremented with the ++ command and decremented with the "`--;`" command.

peep character
: this character is not directly manipulable, but it constitutes a very simple "look ahead" mechanism and is used by the "*while*" and "*whilenot*" commands

counter
: This counter or "*accumulator*" is an integer variable which can be incremented with the command @plus ,decremented with the command @minus ,and set to zero with the command @zero .

10.1 Workspace buffer

The workspace buffer is the heart of the virtual machine and is analogous to an "*accumulator*" register in a cpu chip. All incoming and outgoing text data is processed through the workspace. All machine instructions either affect or are affected by the state of the workspace.

The workspace buffer is a buffer within the virtual machine of the stream parsing language. In this buffer all of the text transformation processed take place. For example, the commands clear, add, indent, newline all affect the text in the workspace buffer.

The workspace buffer is analogeous to a processor register in a non-virtual cpu. In order to manipulate a value, it is generally necessary to first load that value into a cpu register. In the same way, in order to manipulate some text in the `pep` machine, it is necessary to first load that text into the workspace buffer. This can be achieve with the @get, @pop, @read commands.

10.2 Stack

Machine instructions relating to the stack
`push, pop, stack, unstack`

The stack is used to store and access the parse tokens that are constructed by the virtual machine while parsing input.

** parse tokens

Within the virtual machine of the pep/nom language the "*stack*" structure is designed to hold and contain the "parse tokens"

parse tokens can be any string ended by the delimiter eg:

```
add "set*";
```

10.3 Delimiter register

The delimiter register determines what character will be used for delimiting parse tokens on the stack when using the "`push`" and "`pop`" commands. This can be set with the "*delim*" command. The default parse-token delimiter is the '*' asterix character.

Set the parse token delimiter to '/'

```
begin { delim '/'; }
read; "(",")" { put; d; add "bracket/"; push; }
```

The delimiter character will often set in a 'begin' block. I normally use the "*" character, but "/" or ";" might be good options. Apart from the parse-token delimiter character, any character (including spaces) may be used in parse tokens.

Parse tokens with spaces

```
r; [A-Z] { while [A-Z]; put; add "\n"; print; d; add "cap word*"; push; }
```

10.4 Flag register

The flag register affects the operation of the conditional jump instructions "*jumpfalse*" and "*jumptrue*" and it is affected by the test instructions such as "testis", "testtape", "*testeof*" etc. It is analogous to a "*flags*" register in a cpu, but (currently) it only contains one boolean (true/false) value.

Machine assembler instructions relating to the flag register

```
testis, testbegins, testends, testtape, testeof

jumptrue, jumpfalse
```

The script writer does not read or write the machine flag register directly. It is set automatically by the testing instructions

10.5 Accumulator register

The machine contains 1 integer accumulator register that can be incremented (with "a+"), decremented (with "a-") and set to zero (with "zero"). This register is useful for counting occurances of miscelaneous elements that occur during parsing and translating.

An example of the use of the accumulator register is given during the parsing of "*quote-sets*" in old versions of the "`compile.pss`" script. The accumulator in this case, keeps track of the target for true-jumps.

Count how many "x" characters occur in the input stream

```
r; 'x' {a+;} d; <eof> { add ' # of Xs == '; count; print; }
```

10.6 Peep register

The peep buffer is a single character buffer which stores the next character in the input stream. When a 'read' command is performed the current value of the peep buffer is appended to the 'workspace' buffer and the next character from the input stream is placed into the peep buffer.

The "*end-of-stream*" tests <eof><EOF>(eof) (EOF) check to see if the peep buffer contains the end of input stream marker.

The 'while' command reads from the input stream while the peep buffer is, or is not, some set of characters For example

```
while [abc];
```

reads the input stream while the peep buffer is any one of the characters 'abc'.

10.7 Tape

Machine instructions with an effect on the tape
`get, put, ++, --, pop, push, mark, go`

The tape is an array of string cells (with memory allocated dynamically), each of which can be read or written, using the workspace buffer. The tape cell array also includes a tape cell pointer, which is usually called the "current cell".

10.8 Current cell of tape array

Instructions affecting the current cell of the tape
`++, --, push, pop, mark, go`

The current cell of the tape is a very important mechanism for storing attributes of parse tokens, and then manipulating those attributes. The `pep` virtual machine has the ability to "*compile*" or translate one text format into another. (I call this compilation because because the target text format may be an assembly language - for either a real or virtual machine)

In the parsing phase of a script, the attributes of different parse tokens are accessed from the tape and combined and manipulated in the workspace buffer, and then stored again in the current cell of the tape. This means that a script which is transforming some input stream may finish with the entire transformed input in the 1st cell of the tape structure. The script can then `print` out the cell to `stdout` (with "get; print;", which allows further processing by other tools in the pipe chain) or else write the contents of the cell to the file 'sav.pp' (with "get; write;").

The current cell is also affected by the stack machine commands "`pop`" and "push". The `push` command increments the tape pointer (current cell) by 1 and the `pop` command decrements the tape pointer by one.

This is a simple but powerful mechanism that allows the tape pointer to stay in sync with the stack. After a "`push`" or "`pop`" command the tape pointer will be pointing at the correct tape cell for that item on the stack. In some cases, it may be easiest to see how these mechanisms work by running the machine engine in interactive mode (pep -If script input) and stepping through a script or executing commands at the prompt.

The tape pointer is also incremented and decremented by the ++ and – commands, and these commands are mainly used during the compilation phase to access and combine attributes to transform the input into the desired output.

11 Syntax of the pep or nom script language

The script language, which is implemented in the file `/pars/compile.pss` has a syntax very similar to the "`sed`" stream editor. Unlike sed, it also allows long names for commands (eg "*clear*" instead of "d", "`add`" instead of "a"). Each command has a long and a short form.

All commands must be terminated with a semicolon except for the following:

```
.reparse .restart parse>
```

White-space is not significant in the syntax of the parse-script language, except within ' and " quote characters and square brackets []

Random whitespace

```
read; !
[a-e]{t;}

d;
```

Braces "{" and "}" are used to define blocks of commands (as in sed, `awk` and c).

11.1 Language features

The script language (and its syntax) is implemented in the file `compile.pss` . Some commands, such as "`.reparse`" and "`.restart`" affect the flow of the program, but not the virtual machine.

Tests on the workspace buffer, followed by a block of commands

```
[a-z] { print; clear; }
```

Most scripts start with "`read;`" or "`r;`" (which is the abbreviated equivalent). This reads one character from the input stream. Whereas `sed` and `awk` are line oriented (they process the input stream one line at a time), `pep` is character orientated (the input stream is processed one character at a time).

As with `sed` and awk, `pep` scripts have an implicit loop. When the interpreter reaches the end of the script, it jumps back to the first command (usually "read") and continues looping until the input stream is finished.

11.2 Character classes

Character classes are written [:space:] [:alnum:] etc. Currently (July 2022), the `c` language implementation of the parse machine and language uses plain `ctype.h` character classes as a way of grouping characters.

These classes are important in a unicode setting because they allow specifying types of characters in a locale-neutral way. The "*while*" and "*whilenot*" commands can use character classes as their argument.

Check if the workspace is only alphanumeric characters

```
r; ![:alnum:] { add " not alpha-numeric! \n"; print; } clear;
```

Read the input stream while the peep register is whitespace

```
while [:space:];
```

11.3 Quotes

Both single and double quotes may be used in scripts and they have exactly the same role in the syntax of the pep/nom languages. But quoted text which starts with a double quote (") must end with a double quote, and the same rules applies to single quotes.

Use single or double quotes in pep/nom scripts

```
r; '"' { add "<< single quote!\n"; print; } d;
```

If using the negation operator "*!*" in an "*in-line*" script, then enclose the whole thing in single quotes, so that the shell does not substitute the "*!*" character (or other special characters).

Use single quotes in one-line pep scripts to avoid bash substitution

```
pep -e '![u-z],"/",";"{print;} clear;' -i "axbycz/;"
```

Prints: abc/;

The pep/nom language syntax allows quoted text to span more than one line. No special syntax is required and all special characters have the same meaning. This is mainly only usefull as the argument to the "`add`" command, in order to improve the readability of scrips

A multiline quoted text example

```
 begin { add '
A multiline
argument for "add".
';
 } print; clear; quit;
```

It is not necessary to have any space between a command and any quoted argument. This is because the pep/nom language knows how to parse itself (see `compile.pss` for the gory details). In fact, "*white-space*" (\n\t\s\r\f) is not really significant anywhere in a pep/nom script - except within quotes.

No space is required before quotes

```
read; add"."; # this is ok!
replace""..""; # this should be ok too.
print;clear;
# if the input is 'abc' this should print 'a..b..c..'
```

12 "Escaping" within quotes

The until command automatically recognises "*escaped*" quote characters, so it will not stop reading the input stream when it encounters \\" or \\'

Using single quotes, and some abbreviated commands

```
r; 'a' { add 'A'; print; } d;
```

Within quoted text, certain "*escaped*" characters have a special meaning. These charactes have the same meaning in double and single quotes.

n - represents a 'newline' character
empty

r - a return character
empty

f - a form feed (rarely used).

empty

t - a tab
empty

Convert all tabs to 2 spaces in the input stream

```
read; "\t" { clear; add "  "; } print; clear;
```

An abbreviated version of the above in-line pep/nom script

```
r; "\t" { d; a "  "; } t;d;
```

Create a "go" program, equivalent of the above script, compile and run

```
pep -f tr/translate.go.pss -i 'r;"\t"{d;a"  ";}t;d;' >
  ⇒ notabs.go
go build -o notabs notabs.go
echo -e "no\ttabs\tplease!" | notabs
# should print 'no  tabs  please!'
```

12.1 Comments

Both single line comments (line starting with "#"), or multiline comments (hash+asterix .... asterix+hash) are available.

Multiline comments are useful for disabling blocks of code during script development, as well as for long comments at the beginning of a script.

12.2 Begin blocks

Like *awk*, the pep/nom language allows 'begin' blocks. These blocks are only executed once, whereas the rest of the script is executed in a loop for every character in the input stream.

An example begin block and script

```
begin {
  add "Starting script ...\n"; print; clear;
  # set the token delimiter to /
  delim "/";
}
read; print; clear;
```

Create a basic html document

```
begin { add "<html><body><pre>\n"; print; clear; }
read; replace ">" "&gt;"; replace "<" "&lt;";
print; clear;
(eof) { add "\n</pre></body></pre>\n"; print; clear;}
```

12.3 Conditions and tests

12.4 Class tests

The class test checks whether the workspace buffer matches any one of the characters or character classes listing between the square braces. A class test is written.

```
[character-class] { <commands>  }
```

There are 3 forms of the character class test:

1. a list of characters eg: [abcxyz.,;]
2. a range of characters eg: [A-M]
3. a named character class eg: [:space:]

Delete all vowels from the input stream using a list class test

```
read; ![aeiou] { print; } clear;
```

All characters in the workspace must match given class, so that a class test is equivalent to the regular expression "*^[abcd]+$*"

As in the previous example, class tests, like all other type of tests, can be negated with a prefixed "*!*" character. Double and multiple negation, such as "*!!*" or "*!!!*" is a syntax error (since it doesn't have any purpose).

Only print certain sequences

```
r; [abc-,] { while [abc-,]; print; } clear;
```

12.5 Eof end of stream test

This test returns true if the 'peep' look-ahead register currently contains the <EOF>end of stream marker for the input stream. This test is equivalent to the "END { ... }" block syntax in the AWK script language. The eof test is written as follows

Possible formats for the "end-of-stream" test

```
<eof> <EOF> (eof) (EOF)
```

```
r; print; (eof) { add " << end of stream!"; } clear;
```

This test can be combined with other tests either with AND logic or with OR logic

Test if the input ends with "horse" when the end-of-stream is reached

```
r; (eof).E"horse" { add ' and cart'; print; }
```

Test if workspace ends with "horse" or end-of-file has been reached

```
r; (eof),E"horse" { add " <horse OR end-of-file> "; print; }
```

The <eof>test is important for checking if the script has successfully parsed the input stream when the end of stream is reached. Usually this means checking for the "start token" or tokens of the given grammar.

Check for a start token

```
read;
# ... more code
(eof) {
  pop;
  "statement*" {
     # successful parse
     quit;
  }
  # unsuccessful pase
}
```

12.6 Tape test

```
(==) { ...}
```

This test determines if the current tape cell is equal to the contents of the workspace buffer.

Check if previous char the same as current

```
read;
(==) {
  put; add ".same.";
}
print; clear;
```

12.7 Begins with test

Determines if the workspace buffer begins with the given text.

Only print words beginning with "wh"

```
read; E" ",E"\n",(eof) { B"wh" { print; } clear; }
```

13 Ends with test

Tests if the workspace ends with the given text. The 'E' (ends-with modifier) can only be used with quoted text but not with class tests

A syntax error, using e with a class

```
r; E[abcd] { print; } clear;
```

Correct using the e modifier with quoted text

```
r; E"less" { print; } clear;
```

Only print words ending with "ess"

```
read; E" ",E"\n" { clip; E"ess" { add " "; print; } clear; }
```

14 Concatenating tests

Conditional tests can be chained together with OR (,) or AND (.)

Test if workspace starts with "http:" or "ftp:"

```
B"http:",B"ftp:" { print; }
```

"AND" and "OR" test cannot be combined simply, but a similar thing can be achieved by nesting tests in braces. It may be useful to `add` logic grouping for tests, eg (B"a",B"b").E"z" { ... } however this is not currently supported.

An incorrect test

```
B"a".E"z",E"x" { print; }   # wrong!
```

Using nesting to combine "and" and "or" tests

```
B"a",B"b" { E"z" { ... }}
```

This line above is equivalent to the logic:

```
((workspace BEGINS WITH "a") OR (workspace BEGINS WITH "b"))
AND (workspace ENDS WITH "z")
```

Print only urls

```
r; B"http://",B"https://",B"www.",B"ftp:" {
  E" ",E"\n",(eof) { add "\n"; print; clear; }
}
E" ",E"\n",(eof) { clear; }
```

Print only names of animals using or logic concatenation

```
read; [:space:] {d;} whilenot [:space:];
"dog","cat","lion","puma","bear","emu" { add "\n"; print; }
clear;
```

14.1 And logic concatenation

It is also possible to do AND logic concatenation with the "." operator

Test if the workspace starts with 'a' and ends with 'z'

```
r; B"a".E"z" { print; clear; }
```

Check if workspace is a url but not a ".txt" text file

```
B"http://".!E".txt" { add "<< non-text url"; print; clear; }
```

Workspace begins with # and only contains digits and #

```
B"#".[#0123456789] { add " (timestamp?)\n"; print; }
```

"AND" logic can also be achieved by nesting brace blocks. In some circumstances this may may have advantages.

And logic with nested braces

```
B"/" { E".txt" { ... } }
```

14.2 Negated tests

The "*!*" not operator is used for negating tests.

Examples of negated tests

```
!B"abc", !E"xyz", ![a-z], !"abc" ...
```

15 Structure of a script

15.1 Lexical phase

Like most systems which are designed to parse and compile context-free languages, parse scripts normally have 2 distinct phases: A "*lexing*" phase and a parse/compile/translate phase. This is shown by the separation of the unix tools "*lex*" and "*yacc*" where `lex` performs the lexical analysis (which consists of the recognition and creation of lexical tokens), and `yacc` performs parsing and compilation of tokens.

In the `compile.pss` program, as with many scripts that can be written in the parse language, the lexical and compilation phases are combined into the same script.

In the first phase, the program performs lexical analysis of the input (which is an uncompiled script in the pep/nom language), and converts certain patterns into "tokens". In this system, a "*token*" is just a string (text) terminated in an asterix (*) character. `Asm.pp` constructs these tokens by using the "`add`" machine instruction, which appends text to the workspace buffer.

Next the parse token is "*pushed*" onto the "stack". I have quoted these words because the stack buffer is implemented as a string (char *) buffer and "*pushing*" and "*popping*" the stack really just involves moving the workspace pointer back and forth between asterix characters.

I used this implementation because I thought that it would be fast and simple to implement in the "`c`" language. It also means that we dont have to worry about how much memory is allocated for the stack buffer or each of its items. As long as there is enough memory allocated for the workspace buffer (which is actually just the end of the stack buffer) there will be enough room for the stack.

The lexical phase of "`asm.pp`" also involves preserving the "*attribute*" of the parse token. For example if we have some text such as "*hannah*" then our parse token may be "*quoted.text**" and the attribute is the actual text between the quotes 'hannah'. The attribute is preserved on the machine tape data structure, which is array of string cells (with no fixed size), in which data can be inserted at any point by using the tape pointer.

15.2 Parsing phase

The parsing phase of the `asm.pp` compiler involves recognising and shift-reducing the token sequences that are on the machine "stack". These tokens are just strings post-delimited with the '*' character. Because the tokens are text, they popped onto the workspace buffer and then manipulated using the workspace text commands.

16 Commands

View all commands supported by the pep/nom interpreter

```
pep -C
```

16.1 Command summary

All pep/nom commands have an abbreviated (one letter) form as well.

```
eg: p=pop, P=push, g=get, G=put;
```

++
: increments the tape pointer by one (see 'increment')

– decrements the tape pointer by one. (see "decrement;")

mark "text"
: adds a marker to the current tape cell (used with 'go')

go "text"
: sets the current tape cell to the marked cell

add "text" (or add 'text')
: adds text to the end of the workspace

.reparse
: jumps back or forward to the parse>label. This is used to ensure that all shift reductions take place.

clip
: removes the last character from the workspace

clop
: removes the first character from the workspace

quit
: exits the script without reading anything more from standard input. (like the `sed` command 'q')

clear
: sets the workspace to a zero length string. Equivalent to the `sed` command

```
s/^.*$//;
```

put
: puts the contents of the workspace into the current item of the tape (as indicated by the tape-pointer)

get gets the current item of the tape and adds it to the ***end*** of the workspace with ***no*** separator character

swap
: swaps the contents of the current tape cell and the workspace buffer.

count
: appends the integer counter to the ***end*** of the workspace

a+
: increments the accumulatorr variable/register in the virtual machine by 1.

a- this command decrements a counter variable by one

zero;
: sets the counter to zero

lines (or 'll')
: appends the line number register to the workspace buffer.

nolines
: sets the automatic line counter to zero.

chars (or 'cc')
: appends the character number register to the workspace buffer

nochars
sets the automatic character counter to zero.

print
prints the contents of the workspace buffer to the standard output stream (stdout). Equivalent to the `sed` command 'p'

pop
pops one token from the stack and adds it to the -beginning- of the stack

push
pushes one token from the workspace onto the stack, or reads upto the first star "*" character in the @workspace buffer and pushes that section of the buffer onto the stack.

unstack
pops the entire stack as a prefix onto the workspace buffer

stack
pushes the entire workspace (regardless of any token delimiters

in the workspace) onto the stack.
empty

replace
replaces a string in the workspace with another string.

read
reads one more character from the stdin.

state
prints the current state of the virtual machine to the standard output stream. maybe useful for debugging I may remove this command.

until 'text'
reads characters from the input stream until the "*workspace*" ends in the given text. This can be used for "*lexing*" quoted strings etc.

until;
reads characters from the input-stream until the "*workspace*" ends with the text contained in the current tape-cell. (this version of the until command is still being implemented)

cap
converts the workspace to 'capital case' (first upper, then all lower)

lower
converts all characters in the workspace to lowercase

upper
converts all characters in the workspace to upper case.

while class/text;
reads characters from the input stream while the peep character is the given class

whilenot class/text;
reads characters from the input stream while the peep register is ***not*** the given character or class

write
saves the current contents of the workspace in the file "`sav.pp`"

16.2 "Add" command

The "`add`" command appends some text to the end of the 'workspace' buffer. No other register or buffer within the virtual machine is affected. The command is written:

```
add 'text'; #* or *#

add "text";
```

Add a quote after the ':' character

```
r; [\:] { add "\""; } print; clear;  # non java fix!!

r; [:] { add "\""; } print; clear; # java version
```

But it shouldn't be necessary to escape ':'

The quoted argument may span more than one line. for example:

```
  begin { add '
   A multiline
   argument for "add".
 '; } read; print; clear;
```

It is possible to use escaped characters such as \n \r \t \f or \" in the quoted argument.

Add a newline to the workspace after a full stop

```
r; E"." { add "\n"; } print; clear;
```

Add a space after every character of the input

```
read; add ' '; print; clear;
# or the same using abbreviated commands
# r; a ' ';p;d;
```

"*Add*" is used to create new tokens and to modify the token attributes.

Create and push (shift) a token using the "add" command

```
"style*type*" {
  clear;
  add "command*";
  push;
}
```

The script above does a shift-reduce operation while parsing some hypothetical language. The "`add`" command is used to `add` a new token name to the 'workspace' buffer which is then pushed onto the stack (using the 'push' operation, naturally). In the above script the text added can be seen to be a token name (as opposed to some other arbitrary piece of text) because it ends in the "*" character. Actually any character could be used to delimit the token names, `put` "*" is the default implementation.

The `add` command takes -one- and only one parameter, or argument.

16.3 "Reparse" command

The "`.reparse`" command makes the interpreter jump to the "*parse>*" label. The `.reparse` command takes no arguments, and is ***not*** terminated with a semi-colon. The `.reparse` command is important for ensuring that all shift-reductions occur at a particular phase of a script.

If there is no 'parse>' label in the script, then it is a (compile-time) error to use the "`.reparse`" command.

16.4 "Clip" command

The clip command removes one character from the end of the 'workspace' buffer and sends it into the void. It deletes it. The character is removed from the ***end*** of the workspace, and so, it represents the last character which would have been added to the workspace from a previous 'read' operation.

The command is useful, for example, when parsing quoted text, used in conjunction with the 'until' command. The following script only prints text which is contained within double quote characters, from the input.

Parse and print quoted text

```
read; '"' { clear; until '"'; clip; print; }
"\"" { clear; until "\""; clip; print; }
```

If the above script receives the text 'this "*big*" and "*small*" things' as its input, then the output will be 'big small' .That is, only that which is within double quotes will be printed.

The script above can be translated into plain english as follows

a- If the workspace is a " character then - clear the workspace - read the input stream until the workspace -ends- in " - remove the last character from the workspace (the ") - `print` the workspace to the console - end if -

The script should `print` the contents of the quoted text without the quote characters because the 'clear' and the clip commands got rid of them.

16.5 "Clop" command

The "*clop*" command removes one character from the front of the workspace buffer. The clop command is the counterpart of the 'clip' command.

Print only quoted words without the quotes

```
r; '"' { until '"'; clip; clop; print; } clear;
```

16.6 "Count" command

The count command adds the value of the counter variable to the end of the workspace buffer . For example, if the counter variable is 12 and the workspace contains the text 'line:', then after the count command the workspace will contain the text

```
line:12
```

Count the number of dots in the input

```
read; "." { a+; } clear; <eof> { count; print; }
```

The count command only affects the 'workspace' buffer in the virtual machine. For counting lines and characters in the input-stream, the commands "lines", "chars", "*nolines*" and "*nochars*" exist.

Count blank lines in input

```
read;
"\n" {
  while [ \t\r]; clear;
  !(eof) { read; "\n" { a+; }}
}
clear;
(eof) {
  add "Blank lines: "; count; add "\n";
  print; quit;
}
```

16.7 "Quit" command

This command immediately exits out of the script without processing any more script commands or input stream characters.

This command is similar to the "`sed`" command 'q' It would be useful for this command to provide an exit (or error) code. eg "quit 1;"

16.8 "Decrement" command

The decrement command reduces the tape pointer by one and thereby moves the current 'tape' element one to the 'left'

The decrement command is written "–" or "<"

If the current tape element is the first element of the tape then the tape pointer is not changed and the command has no effect. The tape pointer is also decremented by the "`pop`" command (if it is greater than zero).

16.9 "Increment" command

The command adds ***one*** to the "*tape*" pointer. The command is written

```
++ or >
```

Notice that the only part of the machine state which has changed is that the 'tape-pointer' (the arrow in the 'tape' structure) has been incremented by one cell.

This command allows the tape to be accessed as a 'tape' . This is tortological but true. Being able to increment and 'decrement' the tape pointer allows the script writer and the virtual machine to access any value on the tape using the 'get' and 'put' commands.

It should be remembered that the 'push' command also automatically increments the tape pointer, in order to keep the tape pointer and the stack in synchronization.

Since the "`get`" command is the only way to retrieve values from the tape the "`++;`" and "`--;`" commands are necessary. The tape cannot be accessed using some kind of array index because the `get` and 'put' commands to not have any arguments.

An example, the following script will `print` the string associated with the "*value*" token.

```
pop;pop; "field*value*" { ++; get; --; print; }
```

16.10 "Mark" command

The mark command adds a text "*tag*" to the current tapecell. This allows the tapecell to be accessed later in the script with the "*go*" command. The mark and go commands should allow "*offside*" or indent parsing (such as for the ***`python`*** language)

Use mark and go to use the 1st tape cell as a buffer.

```
begin { mark "topcell"; ++; }
read; [:space:] { d; }
whilenot [:space:]; put;
# create a list of urls in the 1st tapecell
B"http:" {
  mark "here"; go "topcell"; add " "; get; put; go "here";
}
clear; add "word*"; push;
<eof> { go "topcell"; get; print; quit; }
```

See the script `pars/eg/markdown.toc.pss` for an example of using the "*mark*" and "*go*" commands to create a table of contents for a document from markdown-style underline headings.

16.11 "Go" command

The "*go*" command set the current tape cell pointer to a cell which has previously been marked with a "*mark*" command (using a text tag. The go/mark commands may be useful for offside parsing as well as for assembling, for example, a table of contents for a document, while parsing the document structure.

Basic usage

```
read; mark "a"; ++; go "a"; put; clear;
```

16.12 "Minus" counter command

The minus command decreases the counter variable by one. This command takes no arguments and is written

```
a-;
```

The 'testeof' (eof) checks to see if the peep buffer contains the end of input stream marker.

The 'while' command reads from the input stream while the peep buffer is or is not some set of characters For example

```
while 'abc';
```

reads the input stream while the peep buffer is any one of the characters 'abc'.

16.13 "Plus" counter command

This command increments the machine counter variable by one. It is written

```
a+
```

The plus command takes no arguments. Its counter part is the 'minus' command.

16.14 "Zero" command

The "*zero*" command sets the internal counter to zero. This counter may be used to keep track of nesting during parsing processes, or used by other mundane purposes such as numbering lines or instances of a particular string or pattern.

```
read; "x" { zero; }
```

16.15 "Chars" command

The "*chars*" (or "cc") command appends the value of the current character counter to the workspace register.

Show an error message with a character number

```
read;
[@#$%^&] {
  put; clear;
  add "illegal character ("; get; ") ";
  add "at character number "; chars; add ".\n";
  print; clear;
}
```

I originally named this command "*cc*" but prefer the longer form "chars".

16.16 "Nochars" command

The "*nochars*" command sets the automatic character counter to zero. This is useful for making the character number relative to the line number. For example at the beginning of a script, we can `put`

Show the line and character number of 'z's in input

```
read;
"\n" { nochars; }
"z" {
  add "\n[Found a 'z' at line: "; lines; add ", char: ";
    ⇒ chars;
  add "]\n";
}
print; clear;
```

16.17 "Lines" command

The "*lines*" (or "ll", it's original, cryptic name) command appends to the workspace the current value of the line counter register. This is very useful when writing compilers in order to produce an error message with a line number when there is a syntax error in the input stream.

Using "lines" in error messages

```
(eof) {
  clear;
  add "Unexpected end-of-file at line: "; lines;
  add " of input.";
  print; quit;
}
```

16.18 "Nolines" command

Sets the line number character counter to zero.

Make line numbers relative to current paragraph

```
read;
# find empty lines
"\n" { while [ \t\n]; E"\n".!"\n" { nolines; } }
```

16.19 "Unstack" command

Pop the entire stack as a prefix onto the workspace. This may be useful for displaying the state of the stack at the end of parsing or when an error has occurred. Currently (Aug 2019) the tape pointer is not affected by this command.

16.20 "Stack" command

Push the entire workspace onto the stack regardless of token delimiters.

16.21 "Push" command

The "`push`" command pushes one token from (the beginning of) the workspace onto the stack.

16.22 "Pop" command

The "`pop`" command pops the last item from the virtual-machine stack and places its contents, without modification, at the ***beginning*** of the workspace buffer, and decrements the tape pointer. If the stack is empty, then the "`pop`" command does nothing (and the tape pointer is unchanged).

Pop the stack onto the beginning of the workspace.

```
pop;
```

Apply a 2 token grammar rule (a shift-reduction).

```
pop;pop; "word*colon*" { clear; add 'command*'; push; }
```

The `pop` command is usually employed in the parsing phase of the script (not the lexing phase); that is, after the "*parse>*" label. The "`pop;`" command is almost the inverse machine operation of the "`push;`" command, but it is important to realise that a command sequence of "`pop;push;`" is not equivalent to "`nop;`" (no-operation).

If the pep/nom parser language is being used to parse and translate a language then the script writer needs to ensure that each token ends with a delimiter character (by default "*") in order for the `push` and `pop` commands to work correctly.

The `pop` command also affects the "*tape*" of the virtual machine in that the tape-pointer is automatically decremented by one once for each `pop` command. This is convenient because it means that the tape pointer will be pointing to the corresponding element for the token. In other words in the context of parsing and compiling a "formal language" the tape will be pointing to the "*value*" or "*attribute*" for the the token which is currently in the workspace.

16.23 "Print" command

The `print` command prints the contents of the workspace to the standard output stream (the console, normally). This is analogous to the the `sed` 'p' command (but its abbreviated form is 't' not 'p' because 'p' means "pop".

** Examples

Print each character in the input stream twice:

```
read; print; print; clear;
```

Replace all 'a' chars with 'a's;

```
read; "a" { clear; add "A"; } print; clear;
```

Only print text within double quotes:

```
read; '"' { until '"'; print; } clear;
```

** Details

The `print` command does not take any arguments or parameters. The `print` command is basically the way in which the parse-language communicates with the outside world and the way in which it generates an output stream. The `print` command does not change the state of the `pep` virtual machine in any way.

Unlike sed, the parse-language does not do any "*default*" printing. That is, if the `print` command is not explicitly specified the script will not `print` anything and will silently exit as if it had no purpose. This should be compared with the default behavior of `sed` which will `print` each line of the input stream if the script writer does not specify otherwise (using the -n switch).

Actually the `print` command is not to be so extensively used as in sed. This is because if an input stream is successfully parsed by a given script then only -one- `print` statement will be necessary, at the end of the input stream. However the `print` command could be used to output progress or error messages or other things.

16.24 "Get" command

This command obtains the value in the current 'tape' cell and adds it (appends it) to the end of the 'workspace' buffer. The "`get`" command only affects the 'workspace' buffer of the virtual machine.

If the workspace has the "noun" token, get its value and print it.

```
"noun*" { clear; get; print; clear; }
```

16.25 "Put" command

The `put` command places the entire contents of the workspace into the current tape cell, overwriting any previous value which that cell might have had. The command is written

```
put;
```

Put the text "one" into the current tape cell and the next one.

```
clear; add "one"; put; ++; put;
```

The `put` command only affects the current tape cell of the virtual machine.

After a `put` command the workspace buffer is -unchanged- . This contrasts with the machine stack 'push' command which pushes a certain amount of text (one token) from the workspace onto the stack and deletes the same amount of text from the workspace buffer.

The `put` command is the counterpart of the 'get' command which retrieves or gets the contents of the current item of the tape data structure. Since the tape is generally designed for storing the values or the attributes of parse tokens, the `put` command is essentiallly designed to store values of attributes. However, the `put` command overwrites the contents of the current tape cell, whereas the "`get`" command appends the contents of the current tape cell to the work space.

The `put` command can be used in conjunction with the 'increment' ++ and 'decrement' – commands to store values in the tape which are not the current tape item.

16.26 "Swap" command

Syntax: swap; Abbreviation: 'x'

Swaps the contents of the current tape-cell with the "*workspace*" buffer. The swap command allows the tape-cell to be added to the beginning of the workspace, not the end.

Prepend the current tape cell to the workspace buffer

```
swap; get;
```

16.27 "Read" command

The read command reads one character from the input stream and places that character in the 'peep' buffer. The character which was in the peep buffer is added to the end of the 'workspace' buffer.

The read command is the fundamental mechanism by which the input stream is "tokenized", which is also known as "lexical analysis". The commands which also perform tokenization are "until", "*while*" and "whilenot". These commands perform implicit

"*read*" operations.

There is no implicit read command at the beginning of a script (unlike "`sed`"), so all scripts will probably need at least one read command.

16.28 "Replace" command

This command replaces one piece of text with another in the workspace. The "*replace*" command is useful for indenting blocks of text during formatting operations, among other things. The replace command only replaces plain text, not regular expression patterns.

Replace the letter 'a' with 'a' in the workspace buffer

```
replace "a" "A";
```

The replace command is often used for indenting generated code.

Indent a block of text

```
clear; get; replace "\n" "\n  "; put; clear;
```

Replace can also be used to test if the workspace contains a particular character, in conjuntion with the "*(==)*" tape test.

Check if the workspace contains an 'x'

```
# fragment
put; replace 'x' '';
(==) {
  clear; add "no 'x'"; print; clear;
}
```

16.29 Until command

Example

```
until 'text';
```

Reads the input stream until the workspace ends with the given text.

Print any text that occurs between '<' and '>' characters

```
/</ { until ">"; print; } clear;
```

Print only text between quote characters (excluding the quotes)

```
r; '"' { until '"'; clip; clop; print; } clear;
```

Create a parse token 'quoted' from quoted text

```
r; '/' { until '"'; clip; clop; put; add 'quoted*'; push; } clear;
```

Print quoted text, reading past escaped quotes (\")

```
/"/ { until '"'; print; } clear;
```

The 'while' and 'whilenot' commands are similar to the until command but they depend on the value of the 'peep' virtual machine buffer (which is a single-character buffer) rather than on the contents of the 'workspace' buffer like the until command.

** notes

The 'until' command usually will form part of the 'lexing' phase of a script. That is, the until command permits the script to turn text patterns into 'tokens'. While in traditional

parsing tools (such as Lex and Yacc) the lexing and parsing phases are carried out by seperate tools, with the 'pep' tool the two functions are combined.

The until command essentially performs multiple read operations or commands and after each read checks to see whether the workspace meets the criteria specified in the argument.

16.30 Whilenot command

It reads into the workspace characters from the input stream -while- the 'peep' buffer is -not- a certain character. This is a "*tokenizing*" command and allows the input stream to be parsed up to a certain character without reading that character.

The whilenot command does not exit if it reaches the end of the input-stream (unlike 'read').

Print one word per line

```
r; [:space:] { d; } whilenot [:space:]; add "\n"; print; clear;
```

(there seems to be a bug in `pep` with whilenot "*x*" syntax)

Whilenot can also take a single character quote argument

```
r; whilenot [z]; add "\n"; print; clear;
```

Another way to print one word per line

```
r; [ ] { while [ ]; clear; add "\n"; } print; clear;
```

The advantage of the first example is that it allows the script to tokenise the input stream into words

16.31 While command

The 'while' command in the pattern-parse language reads the input stream while the 'peep' buffer is any one of the characters or character sets mentioned in the argument. The command is written

```
while [cdef];
```

The command takes one argument. This argument may also include character classes as well as literal characters. From example,

```
while [:space:];
```

reads the input stream while the peep buffer is a digit. The read characters are appended to the 'workspace' buffer. The while command cannot take a quoted argument ("xxx").

Negation for the while command is currently supported using the "*whilenot*" command.

16.32 Endswith test

The 'ends with' test checks whether the workspace ends with a given string. This test is written

```
E"xyz" { ... }
```

The script language contains a structure to perform a test based on the content of the workspace and to execute commands depending on the result of that test. An example of the syntax is

```
"ocean" { add " blue"; print; }
```

In the script above, if the workspace buffer is the text "*ocean*" then the commands within the braces are executed and if not, then not. The test structure is a simple string equivalence test, there are -no- regular expressions and the workspace buffer must be -exactly- the text which is written between the // characters or else the test will fail, or return false, and the commands within the braces will not be executed.

This command is clearly influenced by the "sed" HTTP://SED.SF.NET stream editor command which has a virtually identical syntax except for some key elements. In sed regular expressions are supported and in sed the first opening brace must be on the same line as the test structure.

There is also another test structure in the script language which checks to see if the workspace buffer -begins- with the given text and the syntax looks like this

```
B"ocean" { add ' blue'; print; }
```

17 Using tests in the pep tool

17.1 Test examples

If the workspace is not empty, add a dot to the end of the workspace

```
!"" { add '.'; }
```

If the end of the input stream is reached print the message "end of file"

```
<eof> { add "end of file"; print; }
```

If the workspace begins with 't' trim a character from the end

```
B"t" { clip; }
```

17.2 Tape test

The tape test determines if the current element of the tape structure is the same as the workspace buffer.

The tape test is written

```
<==>
```

This test was included originally in order to parse the sed structure

```
s@old@new@g  or s/old/new/g or s%old%new%g
```

In other words, in sed, any character can be used to delimite the text in a substitue command.

18 Accumulator

19 Stack structure in the virtual machine

The 'virtual'-machine of the pep language contains a stack structure which is primarily intended to hold the parse tokens during the parsing and transformation of a text pattern or language. However, the stack could hold any other string data. Each element of the stack structure is a string buffer of unlimited size.

The stack is manipulated using the `pop` and `push` commands. When a value is popped off the stack, that value is appended to the -front- of the workspace buffer. If the stack is empty, then the `pop` command has no effect.

20 Tape in the pep machine

The tape structure in the virtual machine is an infinite array of elements. Each of these elements is a string buffer of infinite size. The elements of the tape structure may be accessed using the @increment , @decrement , @get and @put commands.

20.1 Tape and the stack

The tape structure in the virtual machine and the @stack structure and designed to be used in tandem, and several mechanisms have been provided to enable this. For example, when a "`pop`" operation is performed, the @tape-pointer is automatically decremented, and when a @push operation is performed then the tape pointer is automatically incremented.

Since the parsing language and machine have been designed to carry out parsing and transformation operations on text streams, the tape and stack are intended to hold the values and tokens of the parsing process.

21 Backusnaur form and the pattern parser

Backus-naur form is a way of expressing grammar rules for formal languages. A variation of BNF is "EBNF" which modifies slightly the syntax of bnf. Sometimes on this site I use (yet another) syntax for **bnf** or **ebnf** rules but the idea is the same.

There is close relationship between the syntax of the 'pep' language and a **bnf** grammar. For example

```
"word*colon*" { clear; add 'command*'; push }
```

corresponds to the grammar rule

```
command := word colon
```

Bnf rule written in pep/nom

```
# lines: lines line
#        | line
#        ;
# or in "ebnf" format
# lines: line+ ;
until "\n"; put; clear; add "line*"; push;
parse>
pop; pop;
"line*line*","lines*line*"
  { clear; add "lines*"; push; .reparse }
```

22 Brainf

Pep/nom can compile the bf language

```
> = increases memory pointer, or moves the pointer to the
  ⇒ right 1 block.
< = decreases memory pointer, or moves the pointer to the
  ⇒ left 1 block.
+ = increases value stored at the block pointed to by the
  ⇒ memory pointer
- = decreases value stored at the block pointed to by the
  ⇒ memory pointer
[ = like c while(cur_block_value != 0) loop.
] = if block currently pointed to's value is not zero, jump
  ⇒ back to [
, = like c getchar(). input 1 character.
. = like c putchar(). print 1 character to the console
```

23 Self referentiality

The pep/nom language is a language which is designed to parse/compile/translate languages. This means that it can recognise/parse/compile, and translate itself. The script `books/pars/translate.c.pss` is an example of this.

Another interesting application of this self-referentiality is creating a new compiling system in a different target language.

24 Reflection self hosting and self parsing

The script "`compile.pss`" is a parse-script which implements the parse-script language. This was achieved by first writing a hand-coded "*assembler*" program for the machine (contained in the file "asm.pp"). Once a working `asm.pp` program implemented a basic syntax for the language, the `compile.pss` script was written. This makes it possible to maintain and `add` new syntax to the language using the language itself.

A new "`asm.pp`" file is generated by running

```
pep -f compile.pss compile.pss >> asm.new.pp; cp asm.new.pp asm.pp
```

Finally it is necessary to comment out the 2 "`print`" commands near the end of the "`asm.pp`" file which are labelled "*:remove:*"

The command above runs the script `compile.pss` and also uses `compile.pss` as its text input stream. In this sense the system is "*self-hosting*" and "self-parsing". It is also a good idea to preserve the old copy of `asm.pp` in case there are errors in the new compiler.

25 Sed the stream editor

The concept of "*cycles*" is drawn directly from the `sed` language or tool (sed is a unix utility). In the `sed` language each statement in a `sed` script is executed once for each -line- in a given input stream. In other words there is a kind of implicit "*loop*" which

goes around the `sed` script. This loop in some fictional programming language might look like:

Pseudo

```
 while more input lines
 do
sed script
 loop
```

In the current parse-language the cycles are executed for each -character- in the input stream (as opposed to line).

26 Shifting and reducing

There is one complicating factor which is the concept of multiple shift-reduces during the "shift-reduce parsing one cycle or the interpreter. This concept has already been treated within the @flag command documentation. Another tricky concept is grammar rule precedence, in other words, which grammar rule shift-reduction should be applied first or with greater predence. In terms of any concrete application the order of the script statements determines precedence.

26.1 Shift reductions on the stack

See the following script fragment

```
pop;pop;
"command*command*",
"commandset*command*" {
  clear; add 'commandset*'; push;
}
"word*colon*" {
  clear; add 'command*'; push;
}
push; push;
```

This script corresponds directly to the (e)bnf grammar rules

```
commandset := command , command;
commandset := commandset , command;
command := word , colon;
```

But in the script above there is a problem; that the first rule needs to be applied after the second rule.

But it seems now that another reduction should occur namely

```
if-block --> if-statement block
```

which can be simply implemented in the language using the statements:

```
pop; pop;
"if-statements*block*" { clear; add "if-block*"; push; }
```

But the crucial question is, what happens if the statements just written come before the statements which were presented earlier on? The problem is that the second reduction will not occur because the script has already passed the relevant statements. This problem is solved by the `.reparse` command and the parse>label

27 Parse tokens

27.1 Literal tokens

One trick in the pep/nom language is to use a "*terminal*" character as its own parse-token. This simplifies the lexing phase of the script. The procedure is just to read the terminal symbol (one or more characters) and then `add` the token delimiter character on the end.

Using literal tokens with the default token delimiter ''*

```
read;
"{","}","(",")","=" {
  put; add "*"; push; .reparse
}
```

28 Purpose of the language and virtual machine

The language is designed to allow the simple creation of parsers and translators, without the necessity to become involved in the complexities of something like Lex or Yacc. Since the interpreter for the language is based on a virtual machine, the language is platform independant and has a level of abstraction.

The language combines the features of a tokenizer and parser and translator.

29 Available implementations

`HTTP://BUMBLE.SOURCEFORGE.NET/BOOKS/PARS/OBJECT/` This folder contains a working implementation in the `c` language. The code can be compiled with the `bash` functions in the file "`helpers.pars.sh`"

30 Comparison between pep and sed

The `pep` tool was largely inspired by the "`sed`" stream editor, both its capabilities and limitations. Sed is a program designed to find and replace patterns in text files. The patterns which Sed replaces are "regular expressions". Pep has many similarities with sed, both in the syntax of its scripts and also the underlying concepts. To better understand pep, it may be useful to analyse these similarities and differences.

30.1 Similarities

The workspace:

Both `sed` and the `pep` machine have a 'workspace' buffer (which in `sed` is called the "pattern space"). This workspace is the area where manipulations of the text input stream are carried out.

The script cycle:

The languages are based on an implicit cycle. That is to say that each command in a `sed` or a `pep` script is executed once for each line (sed) or once for each character (pep/nom)

Syntax:

The syntax of `sed` and `pep` are similar. Statement blocks are surrounded by curly braces {} and statements are terminated with a semi-colon ; .Also the `sed` idea of "*tests*" based on the contents of the "pattern space" (or @workspace) is used in the `pep` language.

Text streams:

Both `sed` and `pep` are text stream based utilities, like many other unix tools. This means that both `sed` and `pep` consume an input text stream and produce as output an output text stream. These streams are directed to the programs using "*pipes*" — in a console window on both unix and windows systems. For example, for `sed` we could write in a console window

`echo abcabcabc` sed "s/b/B/g" —

and the output would be

```
aBcaBcaBc
```

However the `pep` implementation in `c` cannot receive input from stdin. This is because ***stdin*** is used for receiving interactive commands in debug mode. Nevertheless, once a script has been translated to another language (eg: python, ruby, java, go, tcl) using the translation scripts in the `tr/` folder, then the translated script can be used in a pipe.

For example, we can translate a simple script into ***ruby*** and then run it as a "*filter*" program in a pipe

```
pep -f tr/translate.ruby.pss -i "r;t;t;d;" > test.rb
chmod o+x test.rb; echo "abcd" | ./test.rb
```

In the case of "pep", the command line could be written

```
pep -e "r;'b'{d;add'B'} print;d;" -i "abcabcabc"
```

and the output, once again will be

```
aBcaBcaBc
```

30.2 Differences

Lines vs characters:

Sed (like AWK) is a "*line*" based text manipulation tool (text is processed one line at a time), whereas `pep` is character based (text is normally processed one character at a time). This means that all the instructions in a `sed` script are executed once for each line of the input text stream, but in the case of pep, the instructions are executed once for each character of the input text stream.

Strict syntax:

The syntax of the `pep` language is stricter than that of sed. For example, in a `pep` script all commands must end with a semi-colon (except .reparse, parse>and `.restart` - which affect program flow) and all statements after a test must be enclosed in curly braces. In many versions of sed, it is not always necessary to terminate commands with a semi-colon. For example, both of the following are valid `sed` statements.

```
s/b/B/g s/b/B/g;
```

White space and formatting in scripts:

Pep is more flexible with the placement of whitespace in scripts. For example, in `pep` one can write

```
"text"
{
  print; }
```

That is, the opening brace is on a different line to the test. This would not be a legal syntax in most versions of sed.

Command names:

Sed uses single character "*memnonics*" for its commands. For example, "*p*" is print, "*s*" is substitute, "*d*" is delete. In the `pep` language, in contrast, commands also have a long name, such as "`print`" (which prints the workspace), "*clear*" (to clear or delete the workspace), or "`pop`" (to `pop` the last element off the stack onto the beginning of the workspace). While the `sed` approach is useful for writing very short, terse scripts, the readability of the scripts is not good. Pep allows for improved readability, as well as terseness if required.

The virtual machines:

While both `sed` and `pep` are based on simple "virtual machines" (which consist of string registers and commands to manipulate those registers), the `pep` machine is more extensive. The `sed` machine essentially has 2 buffers, the "pattern space", and the "hold space". The `pep` virtual machine however, has a workspace buffer, a stack, a tape array, several counting registers and others.

Regular expressions:

The tests or "*ranges*" in sed, as well as the substitutions are based on regular expressions. In pep, however, no regular expressions are used. The reason for this difference is that `pep` is designed to parse and transform a different set of patterns than sed. The patterns that `pep` is designed to deal with are referred to formally as "context free languages"

Negation of tests:

The negation operator ! in the "`pep`" language is placed before the test to which is applies, where as in `sed` the negation operator comes after the test or range. So, in `pep` it is correct to write

```
!"tree" { ... }
```

But in `sed` the correct syntax is

```
/tree/! { ... }
```

Implicit read and print

Sed implicitly reads one line of the input stream for each cycle of the script. Pep does not do this, so most scripts need an explicit "*read*" command at the beginning of the script. For example

A sed script with an implicit line read

```
s/a/A/g;
```

Pep has no implicit character read or print

```
r; replace "a" "A"; print; clear;
```

BACKUS-NAUR FORM AND THE PATTERN PARSER

Backus-naur form is a way of expressing grammar rules for formal languages. A variation of BNF is "EBNF" which modifies slightly the syntax of bnf. There are many versions of **bnf** and **ebnf**

There is close relationship between the syntax of the 'pep' language and a **bnf** grammar. For example:

```
"word*colon*" { clear; add "command*"; push }
```

corresponds to the backus-naur form grammar rule

```
command := word colon
```

31 Pep language parsing itself

One interesting challenge for the `pep` language is to "generate itself" from a set of **bnf** rules. In other words given rules such as

```
command := word semicolon;
command := word quotedtext semicolon;
```

then it should be possible to write a script in the language which generates the output as follows

```
pop;pop;
"word*semicolon*" {
  clear; add "command*"; push; .reparse
}
pop;
"word*quotedtext*semicolon*" {
  clear;  add "command*"; push; .reparse
}
push;push;push;
```

When I first thought of a virtual machine for parsing languages, I thought that it would be interesting and important to build a more expressive language "*on-top-of*" the `pep` commands. However, that now seems `less` important.

32 Token attribute transformations

We can write complete translators/compilers with the script language. However it may be nice to have a more expressive format like the one shown below.

A more expressive compiling language built on top of the parse language

```
command := word semicolon {
  $0 = "<em>" $1 "</em"> $2;
};
 ...

 The $n structure will fetch the tape-cell for the
  ⇒ corresponding
 identifier in the bnf rule at the start of the brace block.
 This would be "compiled" to pep syntax as

  ----
pop;pop;
"word*semicolon*" {
  clear;
  # assembling: $0 = "<em>" $1 "</em"> $2;
  add "<em>"; get; add "</em>"; ++; get; --; put; clear;
  # resolve new token
  add "command*"; push; .reparse
}

push;push;
```

33 Shift reductions on the stack

Imagine we have a "*recogniser*" `pep` script as follows:

```
read;
'"' { until '"'; clear; add "quote*"; push; }
";" { clear; add "semicolon*"; push; }
[a-z] {
  while [a-z]; clear; add "word*"; push;
}
[:space:] { clear; }
 parse>
unstack; add "\n"; print; clip; stack;
pop;pop;
"command*command*","commandset*command*"
  { clear; add 'commandset*'; push; .reparse }
"word*semicolon*"
  { clear; add 'command*'; push; .reparse }
pop;
"word*quote*semicolon*"
  { clear; add 'command*'; push; .reparse }
push; push; push;
(eof) {
   pop; pop;
   "command*", "commandset*" {
     clear; add "Correct syntax! \n"; print; quit;
   }
   clear; add "incorrect syntax! \n"; print; quit;
}
```

This script recognises a simple language which consists of a series of "*commands*" (which are lower case words) terminated in semicolons. The commands can have an optional quoted argument. At the end of the script, there should only be one token on the stack (either command* or commandset*).

This script corresponds reasonable directly to the **ebnf** rules

```
word := [a-z]+;
semicolon := ';';
commandset := command command;
command := word semicolon;
```

But in the script above there is a problem; that the first rule needs to be applied after the second rule.

The above statements in the `pep` language execute a reduction according to the grammar rule written above. Examining the state of the machine before and after the script statements above ....

==:: virtual machine .. stack, if-statement, open-brace*, statements*, close-brace* .. tape, if(a==b), {, a=1; b=2*a; ..., } .. workspace, ..,

34 Negation of tests

A test such as

```
"some-text" { #* commands *# }
```

can be modified with the logic operator "*!*" (not) as in

```
!"some-text" { #* commands *# }
```

Note that the negation operator '!' must come before the test which it modifies, instead of afterwards as in "`sed`" .

Multiple negations are not allowed, because any even number of negations is equivalent to the positive test.

```
!!"a" {...} # incorrect
```

34.1 Negation examples

Print only numbers in the input

```
r; ![0-9] { clip; add " "; print; clear; }
```

Print only words not beginning with 'a'

```
r; [ \n] { d; } !B"a".!"" { E" ",E"\n",(eof) { print; d; }}
```

If the end of the input stream has not been reached then `push` the contents of the workspace onto the stack

```
!(eof) { push; }
```

If the value of the workspace is exactly equal to the value of the current element of the tape, then exit the script immediately

```
!(==) { quit; }
```

35 Comments in the pep nom language

Both single line, and multiline comments are available in the parser script language as implemented in `books/pars/asm.pp` and `compile.pss`

Single and muliline examples

```
 #* check if the workspace
is the text "drib" *#
 "drib" {
clear; # clear the workspace buffer
 }
```

The script "`compile.pss`" attempts to preserve comments in the output "*assembler*" code to make that code more readable.

36 Grammar and script construction

My knowledge of formal language grammar theory is quite limited. I am more interested in practical techniques. But there is a reasonably close correlation between bnf-type

grammar rules and pep/nom script construction.

The right-hand-side of a (E)BNF grammar rule is represented by the quoted text before a brace block, and the left-hand-side correlates to the new token pushed onto the stack.

The rule "<nounphrase>::= <article><noun>;" in a parse script

```
"article*noun*" { clear; add "nounphrase*"; push; }
```

37 Tricks

This section contains tips about how to perform specific tasks within the limitations of the parse machine (which does not have regular expressions, nor any kind of arithmetic).

See the example `eg/plzero.pss` for an example of reducing high token rules before low token rules, in order to resolve precedence issues.

Check if accumulator is equal to 4

```
read; a+; put; clear; count;
"4" {  clear; add "4th char is '"; get; "'\n"; print; clear;
  ⇒  }
```

Print only if number 3 digits or greater

```
# check if the input matches the regex /[0-9]{3,}/
r;
(eof) {
  [0-9] { put; clip; clip; clip; !"" { clear; get; print; }
    ⇒ }
}
```

Print the length of each word in input

```
read;
![:space:] {
  nochars; whilenot [:space:];
  add " ("; chars; add ") "; print; clear;
}
# ignore whitespace
!"" { clear; }
```

Another way to print the length of each word in input

```
read;
E" ",E"\n",(eof) {
  add "("; chars; add ") "; print; clear; nochars;
}
```

The script below uses a trick of using the replace command with the "tape equals workspace" test (==) to check if the workspace contains a particular string.

Print only lines that contain the text 'puma'

```
whilenot [\n];
put; replace "puma" ""; !(==) { clear; get; print; }
(eof) { quit; }
```

37.1 Multiplexing token sequences

Sometimes it is useful to have a long list of token sequences before a brace block. One way to reduce this list is as follows

Using nested tests to reduce token sequence lists

```
pop; pop;
B"aa*","bb*","cc*" {
  E"xx*","yy*","zz*" {
    # process tokens here.
    nop;
  }
}
# equivalent long token sequence list
"aa*xx*","aa*yy*","aa*zz*","bb*xx*","bb*yy*","bb*zz*"
"cc*xx*","cc*yy*","cc*zz*" {
  nop;
}
```

37.2 Notes for a regex parser

Parse a regex between / and /

```
#*
tokens for the regex parser
  class: [^-a\][bc1-5+*()]
  spec: the list and ranges in [] classes
  char: one character
*#

begin { while [:space:]; clear; }
read;
!"/" {
  clear; add "error"; print; quit;
}
# special characters for regex can be literal tokens
[-/\](){+*$^.?] {
  "*" { put; clear; add "star*"; push; .reparse }
  add '*'; push; .reparse
}

# the start of class tests [^ ... ]
"[" {
  read;
  # empty class [] is an error
  "]" {
    clear; add "Empty class test [] at char "; chars;
    print; quit;
  }
  # negated class test
  "^" { clear; add "[neg*"; push; .reparse }
  # not negated
  clear; add "[*char*"; push; push; .reparse
}

# just get the next char after the escape char
"\ " {
  (eof) { clear; add "error!"; print; quit; }
  clear; read;
  [ntfr] {
    "n" { clear; add "\n"; }
    "t" { clear; add "\t"; }
    "f" { clear; add "\f"; }
    "r" { clear; add "\r"; }
    put; clear; add "char*"; push;
  }
}
!"" { put; add "char*"; push; .reparse }
parse>
pop; pop;
"char*star*" { clear; add "pattern*"; .reparse }
"char*char*" { clear; add "pattern*char*"; push; push; .
  → reparse }
```

38 Compilation techniques

One of the enjoyable aspects of this parsing/compiling machine is discovering interesting practical "*heuristic*" techniques for compiling syntactical structures, within the limitations of the machine capabilities.

This section details some of these techniques as I discover them.

38.1 Lookahead

the script `eg/mark.latex.pss` contains a clumsy token lookahead. But I may try to convert it to a new technique.

A technique for lookahead parsing (json)

```
pop; pop; pop; pop;
# the test below ensures that there are 4 tokens in the
 ⇒ workspace
# the final token will normally be ]* or ,* (this is a json
 ⇒ array)
# but we dont have to worry about it
B"items*,item*.!"items*item*" {
  replace "items*,item*" "items*";
  push; push; .reparse
}
```

38.2 Rule order

In a script, after the parse>label we can parse rules in the order of the number of tokens. Or we can group the rules by token. There are some traps, for example: "pop; pop;" doesnt guarantee that there are 2 tokens in the workspace.

38.3 Recognisers and checkers

A recogniser is a parser that only determines if a given string is a valid "*word*" in the given language. We can extend a recogniser to be an error checker for a given string, so that it determines at what point (character or line number) in the string, the error occurs. The error-checker can also give a probably reason for the error (such as the missing or excessive syntactic element) This is much more practically useful than a recogniser

Examples with error messages

...

38.4 Empty start token

When the start symbol is an array of another token, it may often simplify parsing to create an empty start token in a "*begin*" block **ebnf** equivalent: text = word*

Using an empty start token

```
begin { add "text*"; push; }
read;
# ignore whitespace
[:space:] { while [:space:]; clear; }
!"" { whilenot [:space:]; put; clear; add "word*"; push; }
 parse>
pop; pop;
"text*word*" { clear; add "text*"; push; .reparse }
(eof) {
  # check for start symbol 'text* here
}
push; push;
```

Without the empty "*text**" token we would have to write

```
"text*word*","word*word*" { <commands> }
```

This is not such a great disadvantage, but it does lead to inefficient compiled code, because the "*word*word**" token sequence only occurs once when running the script (at the beginning of the input stream)

*Compiled code for "text*word*","word*word*"* { ... }

Secondly, in other circumstances, there are other advantages of the empty start symbol. See the `pars/eg/history.pss` script for an example.

38.5 End of stream token

This is an analogous technique to the "empty start symbol". In many cases it may simplify parsing to create a "*dummy*" end token when the end-of-stream is reached. This token should be created immediately after the parse>label

Example of use of "end" token to parse dates in text

```
 # tokens: day month year word
 # rules:
 #    date = day month year
 #    date = day month word
 #    date = day month end
 read;
 ![:space:] {
   whilenot [:space:]; put; clear;
   [0-9] {
     # matches regex: /[0-9][0-9]*/
     clip; clip; !"" {
     }
   add "word*"; push;
 }
parse>
 (eof) {
 }
```

38.6 Palindromes

See `eg/palindrome.pss` for a complete pep/nom palindrome example that uses token-stack reductions to recognise palindromes.

Palindromes are an interesting exercise for the machine because they may be the simplest "*context-free*" language.

The script below is working but also prints single letters as palindromes. See the note below for a solution.

Print only words that are palindromes

```
read;
# the code in this block builds 2 buffers. One with
# the original word, and the other with the word in reverse
# Later, the code checks whether the 2 buffers contain the
# same text (a palindrome).
![:space:] {
  # save the current character
  ++; ++; put; --; --;
  get; put; clear;
  # restore the current character
  ++; ++; get; --; --;
  ++; swap; get; put; clear; --;
}

# check for palindromes when a space or eof found
[:space:],<eof> {
  # clear white space
  [:space:] { while [:space:]; clear; }
  # check if the previous word was a palindrome
  get; ++;
  # if the word is the same as its reverse and not empty
  # then its a palindrome.
  (==) {
    # make sure that palindrome has > 2 characters
    clip; clip;
    !"" { clear; get; add "\n"; print; }
  }
  # clear the workspace and 1st two cells
  clear; put; --; put;
}
```

38.7 Line by line tokenisation

See the example below and adapt

A simple line tokenisation example

```
read;
[\n] { clear; }
whilenot [\n]; put; clear;
add "line*"; push;
 parse>
pop; pop;
"line*line*", "lines*line*" {
  clear; get; add "\n"; ++; get; --; put; clear;
  add "words*"; push; .reparse
}
push; push;
(eof) {
   pop; "lines*" { clear; get; print; }
}
```

Remove all lines that contain in a particular text

```
until "
```

38.8 Word by word tokenisation

A common task is to treat the input stream as a series of space delimited words.

A simple word tokenisation example, print one word per line

```
read; [:space:] { clear; }
whilenot [:space:]; put; clear;
add "word*"; push;
 parse>
pop; pop;
"word*word*", "words*word*" {
  clear; get; add "\n"; ++; get; --; put; clear;
  add "words*"; push; .reparse
}
push; push;
(eof) {
   pop; "words*" { clear; get; print; }
}
```

38.9 Repetitions

The parse machine cannot directly encode rules which contain the **ebnf** repetition construct {}. The trick below only creates a new list token if the preceding token is not a list of the same type.

```
  # ebnf rules:
  #   alist := a {a}
  #   blist := b {b}

  read;
  # terminal symbols
  "a","b" { add "*"; push; }
  !"" {
    put; clear;
    add "incorrect character '"; get; add "'";
    add " at position "; chars; add "\n";
    add " only a's and b's allowed. \n"; print; quit;
  }
parse>
  # 1 token (with extra token)
  pop;
  "a*" {
    pop; !"a*".!"alist*a*" { push; }
    clear; add "alist*"; push; .reparse
  }
  "b*" {
    pop; !"b*".!"blist*b*" { push; }
    clear; add "blist*"; push; .reparse
  }
  push;
  <eof> {
    unstack; put; clear; add "parse stack is: "; get;
    print; quit;
  }
```

38.10 Offside or indent parsing

Some languages use indentation to indicate blocks of code, or compound statements. Python is an important example. These languages are parsed using "*indent*" and "*outdent*" or "*dedent*" tokens.

The mark/go commands should allow parsing of indented languages.

Untested and incomplete... the idea is to issue "*outdent*" or "*indent*" tokens by comparing the current leading space to a previous space token. But the code below is a mess. The tricky thing is that we can have multiple "*outdent**" tokens from one space* token eg

```
  if g==x:
    while g<100:
      g++
  g:=0;
```

A basic indent parsing procedure

```
 # incomplete!!
 read;
 begin { mark "b"; add ""; ++; }
 [\n] {
   clear; while [ ]; put; mark "here"; go "b";
   # indentation is equal so, do nothing
   (==) { clear; go "here"; .reparse }
   add "  ";
   (==) { clear; add "indent*"; push; go "here"; .reparse }
   clip; clip;
   clip; clip;
   (==) { clear; add "outdent*"; push; go "here"; .reparse }
   put; clear; add "lspace*"; push;
   mark "b";
 }
parse>
```

38.11 Optionality

The parse machine cannot directly encode the idea of an optional "*[...]*" element in a **bnf** grammar.

A rule with an optional element

```
r := 'a' ['b'] .
```

In some cases we can just factor out the optional into alternation "—"

```
r := 'a'  'a' 'b' . —
```

However once we have more than 2 or 3 optional elements in a rule, this becomes impractical, for example

```
r := ['a'] ['b'] ['c'] ['d'] .
```

In order to factor out the optionality above we would end up with a large number of rules which would make the parse script very verbose. Another approach is to encode some state into a parse token.

```
  # parse the ebnf rule
  # rule := ['a'] ['b'] ['c'] ['d'] ';' .
  begin { add "0.rule*"; push; }
  read;
  [:space:] { clear; }
  "a","b","c","d",";" { add "*"; push; .reparse  }
  !"" { add " unrecognised character."; print; quit; }
parse>
  pop; pop;
  E"rule*a*" {
    B"0" { clear; add "1.rule*"; push; .reparse }
    clear; add "misplaced 'a' \n"; print; quit;
  }
  E"rule*b*" {
    B"0",B"1" { clear; add "2.rule*"; push; .reparse }
    clear; add "misplaced 'b' \n"; print; quit;
  }
  E"rule*c*" {
    B"0",B"1",B"2" { clear; add "3.rule*"; push; .reparse }
    clear; add "misplaced 'c' \n"; print; quit;
  }
  E"rule*d*" {
    B"0",B"1",B"2",B"3" { clear; add "4.rule*"; push; .
     ⇒ reparse }
    clear; add "misplaced 'd' \n"; print; quit;
  }

  E"rule*;*" { clear; add "rule*"; push; }

 push; push;
 (eof) {
   pop;
   "rule*" { add " its a rule!"; print; quit; }
 }
```

38.12 Repetition parsing

Similar to the notes above about parsing grammar rules containing optional elements, we have a difficulty when parsing elements or tokens which are enclosed in a "*repetition*" structure. In **ebnf** syntax this is usually represented with either braces "{...}" or with a kleene star "*".

We can use a similar technique to the one above to parse repeated elements within a rule.

The rule parsed below is equivalent to the regular expression

```
/a?b*c*d?;/
```

So the script below acts as a recogniser for the above regular expression. I wonder if it would be possible to write a script that turns simple regular expressions into `pep` scripts?

In the code below we don't have any separate "*blist*" or "*clist*" tokens. The code below appears very verbose for a simple task.

```
  # parse the ebnf rule
  # rule := ['a'] {'b'} {'c'} ['d'] ';' .
  # equivalent regular expression: /a?b*c*d?;/

  begin { add "0/rule*"; push; }
  read;
  [:space:] { clear; }
  "a","b","c","d",";" { add "*"; push; .reparse  }
  !"" { add " unrecognised character."; print; quit; }
parse>
# ------------
# 2 tokens
  pop; pop;

  E"rule*a*" {
    B"0" { clear; add "a/rule*"; push; .reparse }
    clear; add "misplaced 'a' \n"; print; quit;
  }
  E"rule*b*" {
    B"0",B"a",B"b" { clear; add "b/rule*"; push; .reparse }
    unstack; add " << parse stack.\n";
    add "misplaced 'b' \n"; print; quit;
  }
  E"rule*c*" {
    B"0",B"a",B"b",B"c" { clear; add "c/rule*"; push; .
     ⇒ reparse }
    clear; add "misplaced 'c' \n";
    unstack; add " << parse stack.\n"; print; quit;
  }
  E"rule*d*" {
    B"0",B"a",B"b",B"c" { clear; add "d/rule*"; push; .
     ⇒ reparse }
    clear; add "misplaced 'd' \n"; print; quit;
  }

  E"rule*;*" { clear; add "rule*"; push; }

 push; push;
 (eof) {
   pop;
   "rule*" {
     clear; add "text is in regular language /a?b*c*d?;/ \n";
      ⇒  print; quit;
   }
   push;
   add "text is not in regular language /a?b*c*d?;/ \n";
   add "parse stack was:"; print; clear;
   unstack; print; quit;
 }
```

38.13 Pl zero

Pl/0 is a minimalistic language created by Niklaus Wirth, for teaching compiler construction.

38.14 Lookahead and reverse reductions

The so called "*quotesets*" have been replaced in the current (aug 2019) implementation of `compile.pss` with 'ortestset' and 'andtestset'. But the compilation techniques are similar to those shown below.

The old implementation of the "*quotesets*" token in old versions of `compile.pss` seems quite interesting. It waits until the stack contains a brace token "{ *" until it starts reducing the quoteset list.

A "quoteset", or a set of tests with or logic

```
'a','b','c','d' { nop; }
```

so the `compile.pss` script actually parses "'c','d' { " first, and then resolves the other quotes ('a','b'). This is good because the script can work out the jump-target for the forward true jump. (the accumulator is used to keep track of the forward true jump).

It also uses the brace as a lookahead, and then just pushes it back on the stack, to be used later when parsing the whole brace block.

Bnf rules for parsing quotesets

```
quoteset '{' := quote ',' quote '{' ;

quoteset '{' := quote ',' quoteset '{' ;
```

But this has 2 elements on the left-hand side. This works but is not considered good grammar (?)

Multiple testset sequences may be used as a poor-persons regular expression pattern matcher.

Print words beginning with 'z', ending with '.txt' and

Not containing the letter 'o'

```
r;
E" ",E"\n",(eof) {
  !(eof) { clip; }
  B"z".E".txt".![o] { add " (yes!) \n"; print; }
  clear;
}
```

It doesnt really make sense to combine a text-equals test with any other test, but the other combinations are useful.

The "*set*" token syntax parses a string such as

```
'a','b','c','d' { nop; }
```

The comma is the equivalent of the alternation operator (—) in **bnf** syntax.

39 Assembly format and files

The implementation of the `pep` script language uses an intermediary "*assembly*" phase when loading scripts. `asm.pp` is responsible for converting the script into an assembly (text) format. "`asm.pp`" is itself an "*assembly-format*" file. I refer to this format as "*assembly*" or "*assembler*" because it is similar to other assembly languages: It has one instruction per line. These files consist of "*instructions*" on the virtual machine, along with "parameters", jumps, tests and jump labels, (which make writing assembly files much easier since line numbers do not have to be used). So the `asm.pp` file actually implements the `pep` script language.

It is not necessary for the pep/nom script writer to know anything about the assembly format or process. However, for scripts which contain hard-to-find bugs it is useful to load a script into the `pep` interpreter in "*interactive*" mode and step through the script. This interactive mode displays the compiled script in assembly format.

Load the script 'eg/palindromes.pss' in interactive mode

```
pep -If eg/palindrome.pss -i 'hannah'
```

"The proof is in the pudding": The implementation of the `pep` script language shows that the `pep` system (and the `pep` virtual-machine) is capable of implementing code languages and data languages (or at least simple ones).

The assembler file syntax is similar to other machine assemblers: 1 command per line, leading space is insignificant. Labels are permitted and end in a "*:*" character.

Convert the script 'brackets.pss' to its compiled form

```
pep -f compile.pss brackets.pss
```

An example compilation of a simple script

```
 # pep script source
 # read; "abc" { nop; }
 # 'assembly' equivalent of the above script
start:
read
testis "abc"
jumpfalse block.end.21
  nop
block.end.21:
jump start
```

Another example showing begin-block compilation

```
# pep script source
# begin { whilenot [:space:]; clear; } read; [:space:] { d;
  ⇒  }
# compilation:

whilenot [:space:]
clear
start:
read
testclass [:space:]
jumpfalse block.end.60
  clear
block.end.60:
jump start
```

40 Comparison with other compiler compilers

As far as I am aware, all other compiler compiler systems take some kind of a grammar as input, and produce source code as output. The produced source code acts as a "*recogniser*" for strings which conform to the given grammar.

40.1 Yacc and lex comparison

The tools "*yacc*" and "*lex*" and the very numerous clones, rewrites and implementations of those tools are very popular in the implementation of parsers and compilers. This section discusses some of the important differences between the `pep` machine and language and those tools.

The `pep` language and machine is (deliberately) a much more limited system than a "*lex/yacc*" style combination. A lex/yacc-type system often produces "`c`" language code or some other language code which is then compiled and run to implement the parser/-compiler.

The `pep` system, on the other hand, is a "text stream filter"; it simply transforms one text format into another. For this reason, it cannot perform the complex programmatic "*actions*" that tools such as lex/yacc `bison` or **antlr** can achieve.

While clearly more limited than a lex-yacc style system, in my opinion the current machine has some advantages:

- ⋆ It may be simpler and therefore should be easier to understand
- ⋆ It does not make use of shift-reduce tables.
- ⋆ It should be possible to implement it on computer environments with modest resources (data/code memory).
- ⋆ Because it is a text-filter, it should be more accessible for "playing around" or experimentation. Perhaps it lacks the psychological barrier that a lex-yacc system has for a non-specialist programmer.
- ⋆ Using translation "`pep`" scripts (in the `tr/` folder), we can translate any pep/nom

script into a number of other languages, such as python, ruby, java, go and plain c. These translation scripts (eg tr/translate.python.pss) can also translate themselves, thus creating a complete stand-alone `pep` system in the target language.

⋆ It is relatively easy to create a `pep` translation script for a new language: a class/object/data-structure is created representing the `pep` virtual machine, and then an existing translation script is adapted for the new language.

41 Status

As of june 2022, the interpreter and debugger written in `c` (i.e `/books/pars/object/pep.c`) works well. This implementation is not unicode-aware and has a "*tape*" array of fixed size, but these problems are somewhat obviated by the existance of the translation scripts in the `tr/` folder.

A number of interesting and/or useful examples have been written using the "`pep`" script language and are in the "*eg/*" folder

Several translation scripts have been written and are largely bug-free such as for the languages java, go, ruby, python, tcl, and c. These scripts can be tested with the "*pep.tt*" helper function in the `helpers.pars.sh` `bash` script. I would like to finish the translation scripts for swift, c++, javascript and rust.

42 Naming of the pep system

The executable is called 'pep' standing for "Parse Engine for Patterns" The folder is called `/books/pars/` The source file is called 'pep.c' for no particularly good reason. Pep scripts are given a "`.pss`" file extension, and files in the "assembler format have a "`.pp`" file extension.

The source files are split into `.c` files where each one corresponds to a particular "*object*" (data structure) within the machine (eg tapecell, tape, buffer with stack and workspace).

'pep' is is not an "*evocative*" name (unlike, for example, "lisp"), but it fits with standard short unix tool naming.

Another possible name for the system could be "`nom`" which is a slight reference to "*noam*" and also an indo-european (?) root for "*name*"

43 Limitations and bugs

- The main interpreter 'pep' (source `/books/pars/object/pep.c)` is written using plain `c` byte characters. This seemed a big limitation, but the scripts `translate.xxx.pss` may be a simple way to accomodate unicode characters without rewriting the code in `pep.c` - loadScript() does not look for the "`asm.pp`" in the PPASM folder, which means that all scripts have to be run from the 'pars' folder. This is a bug. - some segmentation faults may remain in `pep.c` - the whilenot command may not be well implemented in `pep.c` - the `pep` tool cannot receive the input-stream from stdin. This is very un-unix-like but is unavoidable because the "`pep`" executable allows interactive debugging. The solution is to separate `pep` into 2 tools, one which contains a debugger and the other which dedicate "`stdin`" to the input. But I dont think it is worthwhile to do this work until `pep` can deal directly with wide characters (eg wchar)

44 Ideas

- ⋆ a simple language which can generate xcode *swift* and android *java* for writing apps. A json-like layout language to replace android xml layouts.
- ⋆ parsing regular expressions shouldn't be that difficult rules:
  `[0-9abcd]n*a+`
- ⋆ A vim command to compile and run a fragment with `translate.java.pss`
- ⋆ A script to turn a `bash` history file with comments into a *python* or *perl* array of objects (so that we can easily eliminate duplicated commands). And eliminate simple commands immediately with `pep`
- ⋆ An indent parser like this tokens: space newline word words leading.space = nl space ....

45 "Future changes to the "pepnom" language and machine

In the future, scripts may be able to define new character classes in the following way.

Define new named character classes

```
begin {
  class "brackets" [{}()];
  class "idchar" [a-z],[.$];
}
# now use the new character class
[:brackets:] {
  while [:brackets:]; clear;
} print;
```

The commands and new syntax below have not been implemented but might solve a range of problems.

Here are some possible future changes to the machine.

1. abbreviations for character classes eg [:S:] for [:space:] (already in `translate.java.pss` and some other "*transpilers*" but not "`pep.c`")
2. replacetape command: would allow unique lists to be constructed (ie replace in workspace text in the current tape cell with a constant string.)
3. a "*length*" command that sets the accumulator to the length of the current workspace??
4. some accumulator based tests might be good. eg: :: >n { <commands>} # check if accumulator greater than n :: <n { <commands>} # check if accumulator `less` than n :: setchars # set accumulator = character counter :: setlines # set accumulator = line counter
5. create a java/javascript/python/ruby/forth version of the machine
6. I may separate the error checking code which is currently in `compile.pss` into a separate script `error.pss` . This will allow the same code to be used in other scripts such as `translate.java.pss`
7. `add` a new command "*untiltape*" which has no arguments, which reads the input stream until the workspace ends with the text contained in the current cell of the tape. eg: untiltape; One application of this command would be parsing `gnu`

`sed` syntax, where the pattern delimiter is what ever character follows the "*s*" for example:

```
s/a*b/c/
s@a*b@c@
s#a*b#c#
```

8. a new command "*replacetape*" which replaces text in the workspace with the contents of the current tape cell. eg: replace all newlines in the workspace with current cell contents

   ```
   replacetape "\n";
   ```

9. remove "*bail*" the command. Instead allow the "*quit*" command to return an exit code.

46 History of idea

The file `object/pep.c` contains detailed information about the development of this idea.

47 Design philosophy for the machine

When designing the parse machine, I wanted to make its capabilities as limited as possible, while still being able to properly parse and translate "*most*" context-free languages and some context-sensitive languages. Related to this idea, was the aim to make the machine implementable in the smallest possible way.

Also, I deliberately excluded the use of regular expressions, so that the script writer would not be tempted to try to "*parse*" context-free patterns with them.

The general design of the syntax and command-line usage is inspired by some old unix tools, such as sed, `grep` and `awk`

47.1 Regular expressions or lack thereof

As oft-repeated in this document, the parse machine and language does not support regular expressions. This may seem a strange decision, considering that all existing "*lexers*" (tools that perform the lexing phase of compilation) support regexes (as far as I know).

I omitted regular expressions from the machine so that the machine could be implemented in a minimal size and also, so that it would run quickly. I am still hopeful that it is possible to implement the machine on embedded architectures, with very limited resources.

48 Evolution of the machine and language

The a+ and a- commands were initially called "plus" and "minus"

The "lines" and "chars" (line number and character number) registers and commands are new-ish, but important additions, because they allow script error messages to pinpoint the line and character number of the error.

The "mark" and "go" commands are also new additions, and were at

first added to try to allow "*indent*" parsing, (also called "*offside*" parsing, such as is using in the Python language)

- nochars and nolines added to `object/pep.c` (object/machine.interp.c) although they - have been in `pars/tr/translate.java.pss` for a while upper, lower, and cap (capital case)

49 Important files and folders

This section describes some of the key files and folders within the parse-machine implementation at HTTP://BUMBLE.SOURCEFORGE.NET/BOOKS/PARS/

49.1 Example scripts

The folder `eg/` contains a set of scripts to demonstrate the utility of the pep/nom language and machine. Here is a description of some of these scripts.

- ⋆ `mark.html.pss` This converts a particular plain text document format into **html** An example of this format is the current file 'pars-book.txt'
- ⋆ `sed.tojava.pss` converts a `gnu` "`sed`" script into ***java*** (branching not implemented).
- ⋆ `mark.latex.pss` Converts a plain text format into LaTeX (and hence pdf).
- ⋆ `palindrome.pss` Finds almost all palindromes (and sub-palindromes) in the input.
- ⋆ `exp.tolisp.pss` formats simple arithmetic expressions, of the form "*a+b*c+(d/e)*" into a lisp-style syntax.
- ⋆ `history.pss` parses a `bash` history file which may contain comments for a particular command as well as the timestamp (either before or after the timestamp)
- ⋆ `json.check.pss` parses and checks **json** data (but currently only recognises integer numbers).
- ⋆ ....
- ⋆ empty

49.2 Compile dot pss

This is the script compiler and also the compiler compiler. It has replaced the handcoded `/books/pars/asm.pp` file because it is easier to write and maintain.

49.3 Asm dot pp

This file implements the "`pep`" scripting language. It is a text file which consists of a series of "*instructions*" or commands for the `pep` virtual machine. These instructions include instructions which alter the registers of the virtual machine; tests, which set the flag register of the machine to true if the test returns true, or else false; and conditional and unconditional jumps which change the instruction pointer for the machine if the flag register is true.

'Asm.pp' also contains labels (lines ending in ":"). These labels make it much easier to write code containing jumps (a label can be used instead of an instruction number.

Because of the similarity of this format to many "*assembly*" languages I refer to this as assembly language for the `pep` virtual machine.

"`asm.pp`" is now generated from `/books/pars/compile.pss` with

```
pep -f compile.pss compile.pss > asm.new.pp; cp asm.new.pp asm.pp;
```

(and then delete the final `print` statement at the end of asm.pp)

This is a good example of the utility of scripts compiling themselves. In fact, all the "`translate.xxx.pss`" scripts could be used in this way. For example:

```
pep -f translate.java.pss translate.java.pss > Machine.java
```

This creates a ***java*** source file which, when compiled with "*javac*" is able to compile scripts into java.

50 Vim and pep

I usually edit with the "*vim*" text editor (although "*sam*" or "*acme*" might be worthwhile alternatives)). Here are some techniques for using vim with the `pep` tool. The vim mappings and commands below are useful for checking that `pep` "*one-liners*" and `pep` scripts or script fragments contained within a text document, actually compile and run. This may be a way of approximating Knuth's "literate programming" idea.

The multiline snippets are contained in a plain text document within "—" and "*,,,*" tags, which are both on an otherwise empty line.

Create a vim command to run a script embedded in a text document

With input provided as an argument to the vim command

```
com! -nargs=1 Ppm ?^ *---?+1,/^ *,,,/-1w !sed 's/^//' > test.pss; /Users/baobab/sf/
```

Create a vim command to compile to "assembly" format, an embedded script

```
com! Ppcc ?^ *---?+1,/^ *,,,/-1w !sed 's/^//' > test.pss; /Users/baobab/sf/htdocs/b
```

(The assembly compilation will be printed to `stdout`)

Compile a one line script to assembly format and save as test.asm

```
com! -nargs=1 Pplcc .w !sed 's/^ *>>//' > test.pss; /Users/baobab/sf/htdocs/books/p
```

Run a one line script embedded in a text document, input stream as arg

```
com! -nargs=1 Ppl .w !sed 's/^ *>>//' > test.pss; ./pep -f test.pss -i "<args>"
```

Given a one line script such as the following

```
read; "'" { until "'"; print; } clear;
```

Typing ":Ppl one'two'three" within the "*Vim*" text editor, with the cursor positioned on the same line (the line beginning with ">>"), will execute the script with the text as input.

There will be quoting problems if the input contains " characters.

Run a multiline script embedded in a text document

With the input given as an argument

```
com! -nargs=1 Ppm ?^ *---?+1,/^ *,,,/-1w !sed 's/^//' > test.pss; /home/rowantree/s
```

Run a multiline script embedded in a text document

With the file pars-book.txt as the input stream

```
com! Ppf ?^ *---?+1,/^ *,,,/-1w !sed 's/^//' > test.pss; /Users/baobab/sf/htdocs/bo
```

Run a one line script embedded in a text document

With the file "pars-book.txt" as the input stream.

```
com! Ppll .w !sed 's/^ *>>//' > test.pss; /Users/baobab/sf/htdocs/books/pars/pep -f
```

The mapping below can only run the script with a static input "*abc*" which is not very useful, but at least it tests if the script compiles properly. The compiled script will be saved in "`sav.pp`"

Create a vim mapping to run a script embedded in a text document
`map ,pp :?^ *---?+1,/^ *,,,/-1w! test.pss \` !pp -f test.pss -i "abc"¡cr¿ —

Create a vim mapping to execute the current line as a bash "one-liner"
`map ,pl :.w !sed 's/^ *>>//' \` bash —

Create a vim command to execute the current line as a pep "one-liner"
`command! Ppl .w !sed 's/^ *>>//'` bash —

The mappings and commands are for putting in the vimrc file. To create them within the editor prepend a "*:*" to each mapping etc.

`:command! Ppl .w !sed 's/^ *>>//'` bash —

50.1 Convert and run with java

The vim commands below work because 'translate.java.pss' and 'pep' and `pars-book.txt` (this document) are all in the same folder. The paths below would have to be ajusted if that were not the case.

The commands below are very useful for testing the soundness of the 'translate.java.pss' script.

Convert to java and run a multiline script embedded in a text document

With the input given as an argument

```
com! -nargs=1 Ppmj ?^ *---?+1,/^ *,,,/-1w !sed 's/^//' > test.pss; echo "[translati
```

Convert to java a script embedded in a text document, input stream as arg

```
com! -nargs=1 Pplj .w !sed 's/^ *>>//' > test.pss; echo "[translating to java and c
```

51 History

See `/books/pars/object/pep.c` for detailed development history of the script interpreter (written in c).

22 june 2022 Continued work on translator scripts (perl, js) and on examples 13 march 2020 made "*chars*" and "*lines*" aliases for cc and ll in `compile.pss` 2 November 2019

Need to write `tr/translate.c.pss` to create executable code. This is based on `translate.java.pss` . Also would like to write `translate.php.pss` so that scripts can easily be run on a web-server. Also, `translate.python.pss` since ***python*** is an important modern language (done). `translate.swift.pss` `translate.ruby.pss` (done) `translate.forth.pss`

Need to fix `mark.html.pss` to produce acceptable **html** output from this booklet file. Also need to write `mark.latex.pss` , based on `mark.html.pss` so that I can create a decent **pdf** booklet. Then need to `print` the booklet with some images and send to

people who may be interested in this.

27 september 2019

`translate.javascript.pss` is nearing completion... seems to be able to compile many scripts to javascript.

25 august 2019

Great progress has been made. `compile.pss` has all sorts of nice new syntax like negated text= tests !"abc"{ ... } Almost all tests can now be negated. There is now an AND concatentation operator (.).

begin blocks, begintests in ortestsets. `compile.pss` has replaced `asm.handcode.pp` for compiling scripts.

2019

For a number of years I have been working on a project to write a virtual machine for pattern parsing. The code is located at `HTTPS://BUMBLE.SF.NET/BOOKS/PARS/` is) used to implement a script language for parsing and compiling some context-free languages. (The implementation is in 'asm.pp')

The project is now at a stage where useful scripts can be written in the parse-script language.

The purpose of the virtual machine is to be able to parse and transform patterns which cannot normally be dealt with through "regular expressions". I.e patterns which are not "regular languages". Possibly the simplest example of one of these would be palindromes (eg "aba", "hannah", "anna"). The machine also allows a script language to describe patterns and transformations, and this language has similarities to `sed` and to awk.

In fact the whole idea was inspired by `sed` and its limitations for context free languages.

Palindromes are interesting because they can be parsed with the simplest possible recursive-descent parser.

The "`pep`" machine does not use recursive-descent parsing. In fact, the `pep` machine was written to avoid recursive-descent parsing.

52 Document history

10 july 2022 Editing to reduce unnecessary material, which has been left in the file pars-notes.txt. The "`eg/mark.latex.pss`" is now in a good enough state to produce an acceptable **pdf** file. The 3 list types are now suported. I will work to produce a printed version of this booklet, with `mark.latex.pss` and "*pdflatex*" and use pdfpages latex module to produce signatures and folios for binding.

13 march 2020 revisiting `eg/mark.html.pss` in order to format this booklet into html, LaTeX and **pdf** for printing. Also, some revisions of the booklet.

1 november 2019 Revising this book file and attempting to make the examples work and more useful.

13 sept 2019 some editing.

4 September 2019 Adding some ideas about parsing optional elements.

23 August 2019 trying to organise this document.

131517 a