

*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

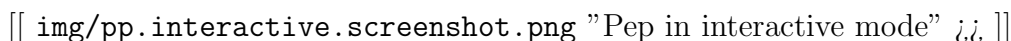
==: The Parsing Virtual Machine and Script Language

Section 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.

[[  "Pep in interactive mode" ]]

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.

Section 2

### ***Documentation***

This file "pars-book.txt" is the principle documentation about the 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.

*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
```

Section 3

### ***Command line usage***

Section 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. `pep -e "r;'\n'{t;}t;d;" /usr/sh`

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

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

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

Section 5

### ***Download***

A ".tar.gz" archive file of the c source code can be downloaded from the <https://sourceforge.net/projects/bumble/> folder.

Section 6

### ***Compiling source code***

Hopefully, in the future, I will package this using "autotools" and "automake" to allow the creation of packages for apt, brew etc. The main folder, source file and booklet are called `pars/` `pars/object/pep.c` and `pars-book.txt`

At the moment (June 2021) the pep executable looks for the file 'asm.pp' in the same folder where it is run or in the folder contained in the environment variable \$ASMPP. I have not yet written "make install" in the Makefile in the `pars/object/` folder to copy `./pep` and `asm.pp` to `/usr/local/bin` and `usr/local/etc`

A very basic "Makefile" is available in the `/books/pars/object/` folder.

The `libmachine.a` file can be used when compiling executables from c source code gen-

*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
```

*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 dictionary 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;
```

*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;
```

*Count the number of lines in the input stream*

```
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;
```

*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;
```

---

*obtain, extract and compile the "pep" source code*

---

```
# extract the tar ball
tar xvzf pep.tar.gz
cd pars/object
# compile the pep tool using the object files in the "
  ⇒ object" folder.
make pep
cd ..
# create a symlink if you like into the bin folder.
ln -s pep /usr/local/bin/pep
```

---

*build the pep tool*

```
cd pars/object/; make pep
```

*manually rebuild the libmachine.a library file*

---

```
cd pars/object
# rebuild all object files
gcc -c *.c
# not all of these files are really necessary.
ar rcs libmachine.a buffer.o charclass.o colours.o
  ⇒ command.o \
  exitcode.o instruction.o labeltable.o machine.o machine
  ⇒ .interp.o \
  machine.methods.o parameter.o program.o tape.o tapecell
  ⇒ .o
```

---

*see how a particular script is compiled to "assembler" format*

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

*load a script and view/execute/step through it interactively*

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

erated by the script "translate.c.pss"

Section 7

## ***Transformations and compilations***

The "pep" virtual machine and language is designed to 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 the syntax of a JSON text data file or convert to another format. convert markdown text into html or LaTeX 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.*

Section 8

## ***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.

The compiled script will be printed to stdout and saved in `sav.pp` `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:

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

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

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

*see the stack when a reduction occurs*

---

```
parse >
  unstack; add "\n"; print; clip; stack;
```

---

*make sure that you are pushing as many times as there are tokens.*

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

## 8.1 Techniques for debugging

The technique used in `eg/mark.latex.pss` is a very powerful method to see how rules are being resolved.

## 8.2 Common bugs

Section 9

*Script examples*

Section 10

*Machine description*

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

Section 11

*Machine elements*

D- a stack: which can contain "parse tokens" if the language is used for parsing or any other text data. - a 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, @indent, @get, @push, @pop etc - a 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 - a 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. - a 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 - a counter : This counter or "accumulator" is an integer variable which can be incremented with the command @plus ,decremented with the command

*make sure there is at least one read command in the script*

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

*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; }
```

---

```
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; }
```

---

*machine instructions relating to the stack*

push, pop

@minus ,and set to zero with the command @zero .

## 11.1 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.

## 11.2 Stack

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 parse script language the "stack" structure is designed to hold and contain the "parse tokens"



*set the parse token delimiter to '/'*

---

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

---

*parse tokens with spaces*

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

parse tokens can be any string ended by the delimiter eg: `add "set*";`

### 11.3 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.

The delimiter should be 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

### 11.4 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. `jumptrue, jumpfalse`

The script writer does not read or write the machine flag register directly.

### 11.5 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 occurrences of miscellaneous 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.

*machine assembler instructions relating to the flag register*

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

*count how many "x" characters occur in the input stream*

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

*machine instructions with an effect on the tape*

```
get, put, ++, --, pop, push, mark, go
```

## 11.6 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'.

## 11.7 Tape

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".

## 11.8 Current cell of tape array

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".

*instructions affecting the current cell of the tape*

```
++, --, push, pop, mark, go
```

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

d;
```

---

*tests on the workspace buffer, followed by a block of commands*

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

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.

Section 12

### ***Syntax of the script language***

The script language, which is implemented in the file `books/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` `pa`

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

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

#### **12.1 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.

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

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

---

*check if the workspace is only alphanumeric characters*

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

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.

## 12.2 Character classes

Character classes are written `[:space:]` `[:alnum:]` etc. Currently (November 2019), 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.

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

## 12.3 Quotes

Both single and double quotes may be used in scripts. `r; ''' { add "<< single quote!\n"; print`

If using the negation operator `!` in an "in-line" script, then enclose the whole thing in single quotes.

The `until` command already understands escaped characters, so it will read past an escaped delimiter (eg `\`).

*read the input stream while the peep register is whitespace*  
`while [:space:];`

*use single quotes in one-liners to avoid special char problems*

```
pep -f compile.pss -i '! [a-z], "a", "b" {nop;}'
```

*single quotes*

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

## 12.4 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.

## 12.5 Begin blocks

Like awk, the parse script 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.

## 12.6 Conditions and tests

### 12.7 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: o- a list of characters eg: [abcxyz.,:] - a range of characters eg: [A-M] - a named character class eg: [:space:]

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

*an example begin block and script*

---

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

---

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

---

*delete all vowels from the input stream using a list class test*

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

error (since it doesn't have any purpose).

## 12.8 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

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

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

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.

## 12.9 Tape test

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

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

## 12.10 Begins with test

Determines if the workspace buffer begins with the given text.

### ***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

*only print certain sequences*

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

*possible formats for the "end-of-stream" test*

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

*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; }
```

*check for a start token*

---

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

---

*check if previous char the same as current*

---

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

---

*only print words beginning with "wh"*

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

*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; }
```

*test if workspace starts with "http:" OR "ftp:"*

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

*print names of animals using OR logic concatenation*

---

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

---

Section 14

### *Concatenating tests*

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

#### 14.1 And logic concatenation

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

"AND" logic can also be achieved by nesting brace blocks. This may have advantages.

#### 14.2 Negated tests

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

Section 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 parse script language), and converts certain patterns into "tokens".

*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] { }
```

In this system, a "token" is just a string (text) terminated in an asterisk (\*) 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 c. 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.

Section 16

### *Commands*

*all commands have a single letter variants for the commands. eg: p, pop, P, push, etc*

*and logic with nested braces*

---

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

---

*examples of negated tests*

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

## 16.1 Command summary

All commands have an abbreviated (one letter) form as well. `++` 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 `parsei` 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 accumulator 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. `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 stdin until the workspace ends in the given text and does not end in the second given text. This maybe used for capturing quoted strings etc. `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

## 16.2 Add

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 a quote after the ':' character*

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

*add a newline to the workspace after a full stop*

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

```
add "text";
```

```
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 is used to create new tokens and to modify the token attributes.

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, but "\*" is the default implementation.

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

### 16.3 Reparse command

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

*Add a space after every character of the input*

---

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

---

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

---

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

---

*parse and print quoted text*

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

If there is no 'parse;' label in the script, then it is an error to use the ".reparse" command.

## 16.4 Clip command

This 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. As in The following script only prints text which is contained within double quote characters, from the input. `"\" { 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;
```

*count the number of dots in the input*  
read; "." { a+; } clear; <eof> { count; print; }

## 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`

The count command only affects the 'workspace' buffer in the virtual machine.

## 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'

## 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 "i"

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.

\*\* See also

'increment' , 'put' , 'get'

## 16.9 Increment command

This command is simple but important. 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 'pop' 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

*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; }
```

---

*basic usage*

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

–; 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)

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.

## 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.

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

*show an error message with a character number*

---

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

---

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.

### 16.15 Chars or cc command

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

I originally named this command "cc" but prefer the longer form "chars" (which is currently implemented as an alias in the script compiler `pars/compile.pss`)

### 16.16 Lines or ll command

The "lines" or "ll" 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.

*pop the stack into the workspace.*

```
pop;
```

*apply a 2 token grammar rule (a shift-reduction).*

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

### 16.17 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.

### 16.18 Stack command

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

### 16.19 Push command

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

### 16.20 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).

The pop command is usually employed in the parsing phase of the script (not the lexing phase); that is, after the "parse<sub>i</sub>" 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;" does not always equivalent to "nop;" (no-operation).

If the stream 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.



*Print each character in the input stream twice:*

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

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

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

## 16.21 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**

**\*\* 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.

## 16.22 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.

## 16.23 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;
```

*Only print text within double quotes:*

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

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

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

*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 essentially 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.

## 16.24 Swap command

Syntax: swap; Abbreviation: 'x'

Swaps the contents of the current tape cell with the workspace buffer.

## 16.25 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" 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.

*prepend the current tape cell to the workspace buffer*

```
swap; get;
```

*replace the letter 'a' with 'A' in the workspace buffer*

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

*indent a block of text*

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

## 16.26 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. The replace command only replaces plain text, not regular expressions.

The replace command is often used for indenting generated code.

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

## 16.27 Until command

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

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 separate 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.

*check if the workspace contains an 'x'*

---

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

---

*example*

```
until 'text';
```

*print any text that occurs between 'j' and 'g' characters*

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

## 16.28 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').

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

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

## 16.29 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.

## 16.30 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

*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;
```

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' ;`

*the List test (not implemented as of july 2020, and I dont think I will implement it)*

A possible future "list" test could be written as `:nouns.txt: { ... }`

The list test determines if the workspace matches any one of a list of strings which are contained within a text file. In the example given above the name of the text file is "nouns.txt". If the "workspace;" was "dog" at the time that the test was executed and the string "dog" was contained in the text file "nouns.txt", then the test would return -true- and the instructions within the braces would be executed. However if the word "dog" was not contained in the text file then the test would return false, and the instructions within the braces would -not- be executed.

## *Using tests in the pep tool*

### 17.1 Test examples

### 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`

*print one word per line*

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

*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;
```

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

Section 18

### ***Accumulator***

Section 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.

Section 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.

*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; }
```

Section 21

### ***Backusnaur form and the pattern parser***

Backus-aur 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`

Section 22

### ***Self referentiality***

The parse script 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.

Section 23

### ***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;`

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.

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

---

***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:

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

***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 precedence. In terms of any concrete application the order of the script statements determines precedence.

**25.1 Shift reductions on the stack**

Imagine we have a 'pep' script as follows:

---

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

---



*using literal tokens with the default token delimiter '\*'*

---

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

---

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 `parsej` label

Section 26

## *Parse tokens*

### 26.1 Literal tokens

One trick in the parse script 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.

### *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 independent and has a level of abstraction.

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

### *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"

<http://bumble.sourceforge.net/books/pars/object.js> An javascript object that can be used with the pep tool and the script "translate.javascript.pss" to translate scripts into javascript.

<http://bumble.sourceforge.net/books/pars/translate.java.pss>

### *Comparison between pep and sed*

The pep tool was largely inspired by the "sed" stream editor. Sed is a program designed to find and replace patterns in text files. The patterns which Sed replaces are "regular expressions"

#### **29.1 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 (pp) *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`

In the case of "pep", the command line could be written `echo abcabcabc pep -e 'b'd;add'B' print;d;` —

and the output, once again will be `aBcaBcaBc`

-

## 29.2 Differences

*Lines vs characters:* Sed (like AWK) is a "line" based system whereas pep is a character based script language. This means that the sed script is executed once for each line of the input text stream, but in the case of pep, the script is executed once for each character of the input text stream. *Strict syntax:*

In some respects, 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`, `parsej` and `.restart` - which affect program flow) and all statements after a test must be enclosed in curly braces. In 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:* 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" for print the workspace, "clear" for clear the workspace, or "pop" to pop the last element of the stack onto 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 in a sense based on "virtual machines", the machine for pep is more extensive. The sed machine essential has 2 buffers, the pattern space, and the hold space. The pep @virtual-machine, however has a workspace, a stack, a tape, an counter variable amongst other things. *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

```
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;
```

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

## BACKUS-NAUR FORM AND THE PATTERN PARSER

Backus-aur 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`

Section 30

### *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
}
```

```
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;
```

---

```
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.

Section 31

### *Token attribute transformations*

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

Section 32

### *Shift reductions on the stack*

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

---

```
read;
";" { clear; add "semicolon*"; push; }
```

```

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

```

---

This script recognises a language which consists of a series of "commands" (which are lower case words) terminated in semicolons. 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, ...

*print only numbers in the input*

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

*single and multiline examples*

---

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

---

Section 33

### ***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 `!!"some-text" {...} # incorrect`

**\*\* Examples**

If the workspace does not begin with the text 'apple' then print the workspace `r; E" " { !B"apple"`

If the workspace does not match any of the lines of the file 'verbs.txt' then add the text 'noun' to the workspace `!=verbs.txt= { add 'noun'; }`

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; }`

Section 34

### ***Comments in the parser language***

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

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

*the rule "nounphrase ::= article noun ;" in a parse script*

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

*check if accumulator is equal to 4*

---

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

---

Section 35

### ***Grammar and script construction***

My knowledge of formal 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 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.

Section 36

### ***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).

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.

#### **36.1 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

*check if the workspace matches the regex /[0-9]{3,}/*

---

```
[0-9] { clip; clip; clip; !" { ## <commands> *# } }
```

---



*see how long a word is*

---

```
read;
![:space:] {
    nochars; whilenot [:space:]; put; clear;
    add "word length = "; chars; print; clear;
}
# if the workspace is not empty, it must be white-space
# just ignore it.
!"" { clear; }
```

---

*print only lines that contain the text 'puma'*

---

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

---

*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;
}
```

---

```
##
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>
42
pop; pop;
"char*star*" { clear; add "pattern*"; .reparse }
"char*char*" { clear; add "pattern*char*"; push; push; .
=> reparse }
"[*]*" { clear; add "error!"; print; quit; }
```

## 36.2 Notes for a regex parser

### *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.

### 37.1 Lookahead

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

Without the empty "text\*" token we would have to write `"text*word*","word*word*" { <command`

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)

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

### 37.2 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 `parsej` label

### 37.3 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.

### 37.4 Line by line tokenisation

See the example below and adapt

### 37.5 Word by word tokenisation

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

```
pop; pop; pop; pop;
# the test below ensures that there is 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
}
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.

#### 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
...
```

#### 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: text = word\*

```
* using an empty start token44
-----
begin { add "text*"; push; }
read;
# ignore whitespace
[:space:] { while [:space:]: clear: }
```

*compiled code for "text\*word\*", "word\*word\*" { ... }*

---

*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) {
}
```

---

## 37.6 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.

## 37.7 Plzero constant declarations

An example of parsing a repeated element occurs in pl/0 constant declarations. The rule in Wirth's ebnf syntax, is: `constdec = "const" ident "=" number {"," ident "=" number} ";"`

Examples of valid constant declarations are

---

```
const g = 300, h=2, height = 200;
const width=3;
```

---

It is necessary to factor the ebnf rule to remove the repetition element (indicated by the braces `{}`). In the script below, the repeated element is factored into the "equality" and "equalityset" parse tokens. The script below may seem very verbose compared to the ebnf rule but it lexes and parses the input stream, recognises keywords and punctuation and provides error messages.

*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;
}
```

---

*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 "
```

---

*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; }
}
```

---

*a technique for building a list token from repeated items*

---

```
# 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;
}
```

---



```
begin {
  add '
  recognising pl/0 constant decs in the form:
  "const g = 300, h=2, height = 200;"
  "const width=3;"
  \n'; print; clear;
}
read;
[:alpha:] {
  while [:alpha:];
  # keywords in pl/0
  "const","var","if","then","while","do","begin","end" {
    put; add "*"; push; .reparse
  }
  put; clear; add "ident*";
  push; .reparse
}
[0-9] {
  while [0-9]; put; clear; add "number*";
  push; .reparse
}
# literal tokens
",", "=", ";" { add "*"; push; }
# ignore whitespace
[:space:] { clear; }
!"" {
  add " << invalid character at position "; chars;
  add ".\n"; print; quit;
}
parse>
```

```
pop; pop; pop;
"ident**number*" {
  clear; add "equality*"; push; .reparse
}
"equality*,*equality*","equalityset*,*equality*" {
  clear; add "equalityset*"; push; .reparse
}
"const*equality*;*", "const*equalityset*;*" {
  clear; add "constdec*"; push;
}
push; push; push;

<eof> {
  pop; pop;
  "constdec*" {
    clear; add " Valid PL/0 constant declaration!\n";
    print; quit;
  }
  push; push;
  add " Invalid PL/0 constant declaration!\n";
```

```
# 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>
```

---

### 37.8 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.

completely 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;
```

---

### 37.9 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.

### 37.10 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 parse-scripts?

In the code below we dont have any separate blist clist tokens

### 37.11 Pl zero

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

In this section, I will explore converting the pl/0 grammar into a form that can be used by the parsing machine and language. I will do this in stages, first parsing "expressions" and then "conditions" etc.

Wirths grammar is designed to be used in a recursive descent parser/compiler, so it will be interesting to see if it can be adapted for the machine which is essentially a LR shift-reduce parser/compiler.

The grammar below seems to be adequate for LR parsing, except for the expression tokens (expression, term, factor etc). Apart from expression we should be able to factor out the various `[] {}` and `()` constructs and create a machine parse script.

```
# 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; }
}
```

---

```
# 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>
# -----
# 1 token
pop;

# -----
# 2 tokens
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*" { add " its a rule!"; print; quit; }
  unstack; add " << parse stack.\n"; print; quit;
}
}
```

---

```
program = block "." .
block = [ "const" ident "=" number {"," ident "=" number}
    => ";" ]
    [ "var" ident {"," ident} ";" ]
    { "procedure" ident ";" block ";" } statement .

statement = [ ident ":=" expression | "call" ident
    | "?" ident | "!" expression
    | "begin" statement { ";" statement } "end"
    | "if" condition "then" statement
    | "while" condition "do" statement ].

condition = "odd" expression |
    expression ("="|"#"|"<"|"<="|">"|">=")
    => expression .
expression = [ "+"|"-" ] term { ("+"|"-" ) term}.
term = factor {("*"|" /") factor}.
factor = ident | number | "(" expression ")" .
```

---

```
program = block "." .
block = [ "const" ident "=" number {"," ident "=" number}
    => ";" ]
    [ "var" ident {"," ident} ";" ]
    { "procedure" ident ";" block ";" } statement .
```

---

### 37.12 Programs and blocks in plzero

I will try to keep wirth's grammatical structure, but factor the 2 rules and introduce new parse tokens for readability, such "constdec\*" for a constant declaration, and "vardec\*" for a variable declaration. I will also try to keep Wirth's rule names for reference

Once this script is written we can check if it is parsing correctly, and then move on to variable declarations, procedure declarations and so forth (but not the forth language, which doesn't require a grammar).

```
expression = [ "+"|"-" ] term { ("+"|"-" ) term}.
term = factor {("*"|" /") factor}.
factor = ident | number | "(" expression ")".
```

---

*example of creating a set of negated classes*

---

```
"!*"quote*", "!*"class*", "!*"begintext*", "!*"endtext*",
"!*"eof*", "!*"tapetest*" {
  replace "!*" "not"; push;
  # now transfer the token value
  get; --; put; ++; clear; .reparse
}
```

---

### 37.13 Plzero expressions

### 37.14 Trigger rules

```
',' orset '{' ::= ',' test '{' ;
```

### 37.15 Negated classes

Often when creating a language or data format, we want to be able to negate operators or tests (so that the test has the opposite effect to what it normally would). In the parse script language I use the prefixed "!" character as the negation operator.

We can create a whole series of negated "classes" or tokens in the following way. So the "negation" logic is actually stored on the stack, not on the tape. This is useful because we can compile all these negated tokens in a similar way .

### 37.16 Lookahead and reverse reductions

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; }

*bnf rules for parsing quotesets*

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

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. `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-womans regular expression pattern matcher. *test if the workspace begins with 'http://', ends with '.txt' and doesnt contain the letter 'x'*

---

```
B"http://", E".txt", ![x] {  
  add " (found text file link!) \n"; print; clear;  
}
```

---

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

### 37.17 Rabbit hops

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.

In my first attempt to parse `quoteset` tokens with the `compile.pss` script compiler, I used a rabbit hop technique. From an efficiency and compiled code size point-of-view this is a very bad way to compile the code, but it seems that it may be useful in other situations. It provides a way to generate functional code when the final jump-target is not know at `shift-reduce` time.

Section 38

#### *Assembly format and files*

The implementation of the 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 file. These files consist of `instructions` on the virtual machine, along with `parameters`, jumps, tests and labels, which make writing assembly files much easier (instead of having to use line numbers). So the `asm.pp` file actually implements the script language



*create rabbit hops for the true jump*

---

```
"quote*,*quote*" {
  clear; add "testis "; get;
  # just jump over the next test
  add "\njumptrue 3 \n"; ++; ++;
  add "testis ";
  get;
  add "\n";
  # add the next jumptrue when the next quote is found
  --; --; put;
  clear;
  add "quoteset*";
  push;
  # always reparse/compile
  .reparse
}

# quoteset ::= quoteset , quote ;
"quoteset*,*quote*" {
  clear; get;
  ++; ++;
  add "jumptrue 4 \n ";
  add "jumptrue 3 \n ";
  add "testis "; get;
  add "\n";
  # add the next jumptrue when/if the next quote is found
  --; --;
  put; clear;
  add "quoteset*";
  push;
  # always reparse/compile
  .reparse
}
```

---

```
# read; "abc" { nop; }
start:
read
testis "abc"
jumpfalse block.end.21
    nop
block.end.21:
jump start
```

---

```
# 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
```

---

The proof is in the pudding: this shows that the parse-machine is capable of implementing "languages" (or at least simple ones).

It should not normally be necessary to write any assembler code, since the script language is much more readable. However, it is useful for debugging scripts to view the assembly listing as it is loaded into the machine (the -I switch allows this).

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.

### *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.

## 39.1 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 parse-machine and language and those tools.

The pep language and machine is (deliberately) a much more limited system than a "lex/yacc" 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 there should be much easier to understand It does not make use of shift-reduce tables. It should be possible to implement it on much smaller systems. 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. It may be simpler and faster to port the system to new platforms/ languages. All that is required is a parse-machine "object" in the target language and a script similar to "translate.c.pss"*

Section 40

### ***Status***

As of July 2020, the interpreter and debugger written in c (i.e `/books/pars/object/pep.c`) works well. However it is not Unicode "aware". A number of interesting and/or useful examples have been written using the "pep" script language and are in the folder `/books/pars/eg/`

The scripts which are designed to compile pep scripts into other languages (java, javascript, python etc) are at varying stages of incompleteness. The 'translate.java.pss' and 'translate.javascript.pss' are currently the most complete, but have not been thoroughly tested. The most comprehensive way to test them is to run them on themselves, with, for example: `pep -f translate.java.pss translate.java.pss > Machine.java`

Section 41

### ***Naming of 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`). Previously I have called this system "chomski" after the linguist Noam Chomsky, but I don't like that name anymore.

'pep' 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"

Section 42

### ***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`)

Section 43

### ***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 ...`

Section 44

### ***Candidates for new commands or syntax for pep***

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. *abbreviations for character classes eg `[:S:]` for `[:space:]`* (already in `translate.java.pss` and some other "transpilers" but not `gh.c`)

\*- replacetape command: would allow unique lists to be constructed (ie replace in workspace text in the current tape cell with a constant string.) a "length" command that sets the accumulator to the length of the current workspace?? some accumulator based tests might be good. eg: `:: i n { j commands j } # check if accumulator greater than n` `:: j n { i commands i } # check if accumulator less than n` `:: setchars # set accumulator = character counter` `:: setlines # set accumulator = line counter` create a java/javascript/python/ruby/forth version of the machine 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` 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#`

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";`

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

Section 45

### *History of idea*

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

Section 46

### *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

## 46.1 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.

Section 47

### *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 recent, but very 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 "off-side" parsing, such as is using in the Python language) june 2021 - nochars and no-lines 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)*

Section 48

### *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/>

## 48.1 Example scripts

The folder `/books/pars/eg/` contains a set of scripts to demonstrate the utility of the parse-script 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'  
- `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.parse.pss` parses and checks json data (but currently only recognises integer numbers).  
- ....

## 48.2 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.

*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
```

*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
```

### 48.3 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.p`

(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 tran`

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

Section 49

#### *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/h`

(The assembly compilation will be printed to stdout)

Given a one line script such as the following `read; "" { until ""; print; } clear;`

*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>"
```

*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| —
```

Typing ":Ppl one'two'three" within the "Vim" text editor, with the cursor positioned on the same line (the line beginning with "i;"), 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, ,
```

*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/Y
```

*run a one line script embedded in a text document with the file "pars-book.txt" as the input stream.*

```
com! Ppl1 .w !sed 's/^ *>>/' > test.pss; /Users/baobab/sf/htdocs/books/
```

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"

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

## 49.1 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 adjusted 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
```

Machine —

Section 50

### **History**

See /books/pars/object/pep.c for detailed development history

13 march 2020 made "chars" and "lines" aliases for cc and ll in compile.pss

*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 —
```

*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
```

2 November 2019

Need to write `tr/translate.c.pss` to create executable code. This can be based on `translate.java.pss`. Also need 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. `translate.swift.pss` `translate.ruby.pss` `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 be negated. There is now an AND concatenation 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 source code is located at <https://bumble.sf.net/books/pars/object/pep.c> This virtual machine can (and 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.

```
parsePalindrome (text) {
  n = text.firstCharacter, m = text.lastCharacter
  if n <> m { return false }
  newText = text - first and last characters
  parsePalindrome(newText)
  return true
}
```

---

Section 51

### ***Palindromes***

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

But the "pep" machine does not use recursive descent parsing. In fact the gh machine was written because recursive descent parsing seemed aesthetically displeasing.

Section 52

### ***Document history***

22 July 2021

13 march 2020 revisiting `eg/mark.html.pss` in order to format this booklet into html, L<sup>A</sup>T<sub>E</sub>X 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