Shell

P. J. Denning

Disclaimer:

- The examples following are illustrative and may not match exactly the command interpreters of real systems
- The parsing diagram and actions may contain small errors

Purpose of Shell

- Listen for user to type command line.
- Parse command line to execution script.
- Construct a pipeline of virtual machines that implement the execution script.
- Execute the virtual machine pipeline.
- Clean up and return to the listening state.

Shell Invokes Virtual Machines

- A common template for the structure of all processes.
- Created and deleted by the virtual machines level just below shell
- We use this simplified version of VM template in the examples to follow

input, output info object pointers

name of executing thread

arguments list

parent, sibling, child pointers

pointer to address space image on disk

counter of incomplete children

Example

```
[self=17]: the shell is process 17

date
```

SCRIPT:

```
1  VM1 = CREATE_VM(IN=VM17.IN, OUT=VM17.OUT, thr="date.exe", par=VM17, sib="",  …)
2   VM17.chi=VM1
2   COMPUTE
3   EXIT
```

Example

```
[self=17]

cat A B C
```

"cat" is a shorthand for concatenate. This command takes a list of files and streams them one after the next to the standard OUT.

SCRIPT:

```
1  VM1 = CREATE_VM(IN=VM17.IN, OUT=VM17.OUT, thr="cat.exe", par=VM17, sib="", …)
2  VM17.chi = VM1
3  VM1.args = "A B C"
4  COMPUTE
5  EXIT
```

Example

```
[self=17]

cat < A
```

When cat has no arguments, it reads its standard IN and passes it through to standard OUT.

SCRIPT:

```
1  VM1 = CREATE_VM(IN=VM17.IN, OUT=VM17.OUT, thr=”cat.exe", par=VM17, sib=“”, …)
2  VM17.chi = VM1
3  VM1.IN = OPEN_FILE(“A”)
4  COMPUTE
5  EXIT
```

What is a Valid Command?

Standard format for a command line:

```
[cmd][<file][| cmd]*[> file]

cmd = name [args]
```

Command Language Grammar

```
line ::= cmd [<fn] [|cmd]* [>fn] EOL
cmd ::= cn [ar]*
cn ::= <any alpha string>
fn ::= <any alpha string>
ar ::= <any alpha string>
```

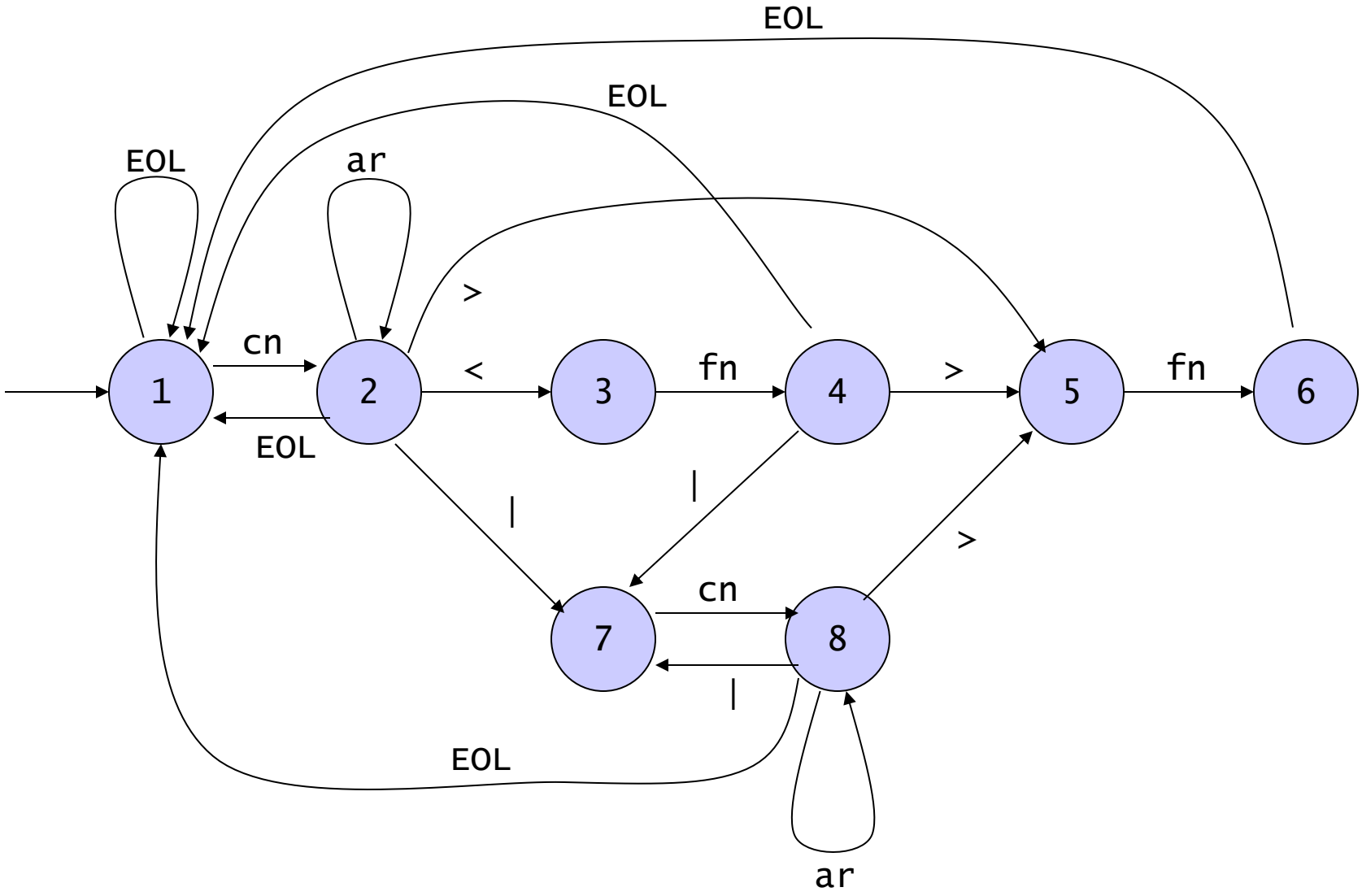
Parser -- “Heart of Shell”

- Analyze an expression, given as a string of characters, into syntactic elements according to a given grammar.
- Build a script of actions that instruct virtual machine manager to implement the meaning of the expression analyzed.
- If parser yields a complete script, then command line syntax is valid.
- If all files mentioned in script exist, then script ready for execution.

Implementation of Parser

- The following slides show the design of a parser as a finite state machine.
- Each transition is labelled with a symbol read from the input line.
- Each transition also generates a line or two of the output script.
- You can skip these slides, if you are not interested in all the detail.

This grammar can be represented as a finite state accepter. It does not need a push-down accepter because its grammar is not self-embedding.



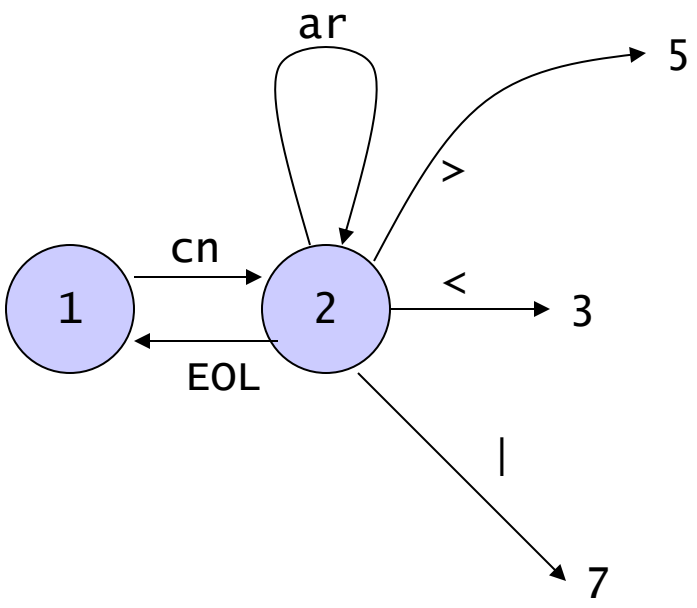
Associate *action code* with each state. Action code examines allowable symbols, specifies next state, and gives next entry in output script. Script tells what OS calls are needed to implement the command. Action code for state 0 initializes counts (v,p) of virtual machines and pipes created.

Subroutine GET scans the input for *tokens* -- substrings terminated by white space, operators, or EOL. GET returns the next token.

Subroutine OUT outputs the given string after prefixing a line number. Double quotes in OUT string mean: insert single quote.

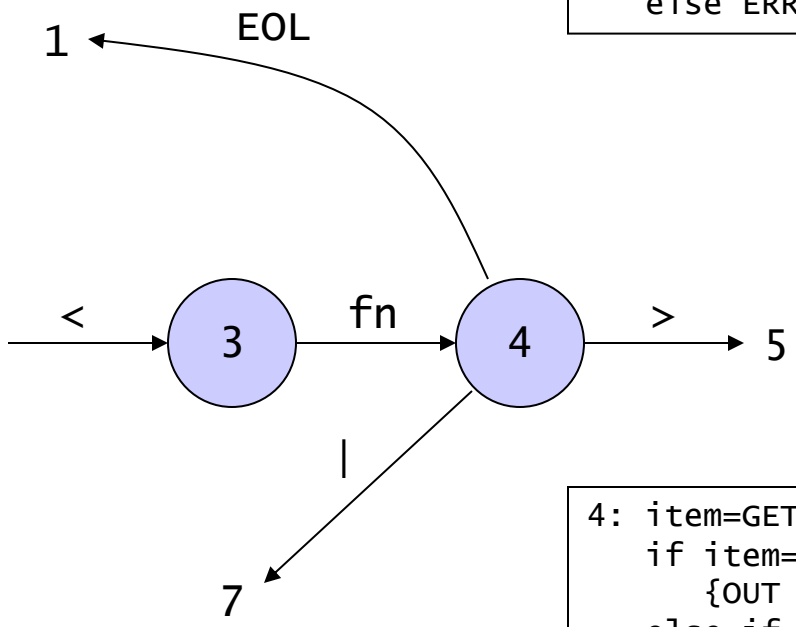
The name "self" refers to the VM in which the parser is embedded. $x means a string obtained by substituting the current value of x.

```
0: v = 0
   p = 0
   goto 1
```




```
1: item=GET
   v++
   if item.type=word then
      {OUT "VM$v = CREATE_VM
                     (IN=VM$self.IN,
                      OUT=VM$self.OUT,
                      thr=""$item"",
                      par=VM$self,
                      sib="", …)"
       OUT "VM$self.chi = VM$v"
       goto 2}
   else ERROR 1
```

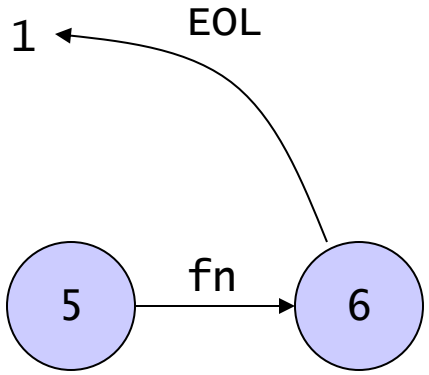
```
2: item=GET
   args=""
   while item.type=word do
      {args=concat(args,item); item=GET}
   OUT "VM$v.args=""$args"""
   if item=EOL then
      {OUT "COMPUTE; EXIT"; goto 1}
   else if item="<" then goto 3
   else if item=">" then goto 5
   else if item="|" then
      {p++;
       OUT "P$p = CREATE_PIPE()";
       OUT "VM$v.OUT = OPEN_PIPE(P$p,w)"
       goto 7}
   else ERROR 2
```



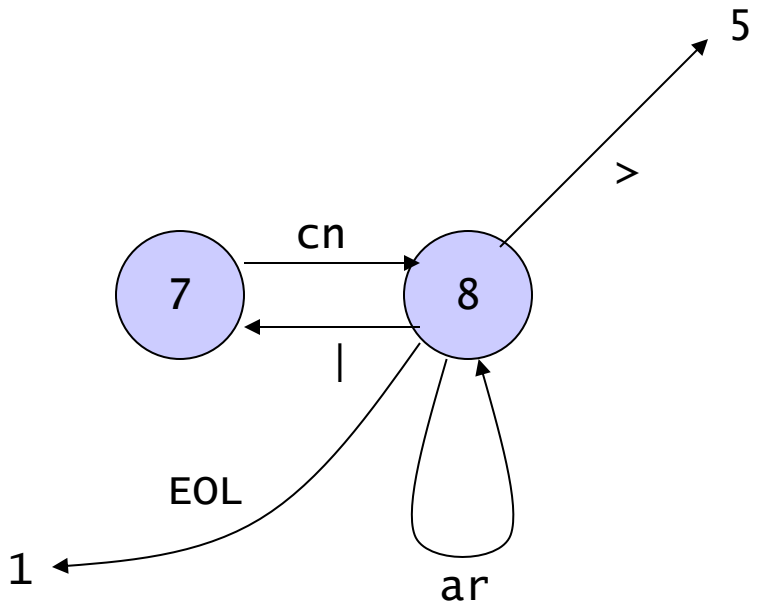

```
3: item=GET
   if item.type=word then
      {OUT "VM$v.IN = OPEN_FILE(""$fn"",r)"
       goto 4}
   else ERROR 3
```

```
4: item=GET
   if item=EOL then
      {OUT "COMPUTE; EXIT"; goto 1}
   else if item=">" then goto 5
   else if item="|" then
      {p++;
       OUT "P$p = CREATE_PIPE()";
       OUT "VM$v.OUT = OPEN_PIPE(P$p,w)"
       goto 7}
   else ERROR 4
```

```
5: item=GET
   if item.type=word then
      {OUT "VM$v.OUT = OPEN_FILE(""$fn"",w)"
       goto 6}
   else ERROR 5
```




```
6: item=GET
   if item=EOL then
      {OUT "COMPUTE; EXIT"; goto 1}
   else ERROR 6
```




```
7: item=GET
   prev=v
   v++
   if item.type=word then
      {OUT "VM$v = CREATE_VM
                     (IN=OPEN_PIPE(P$p,r)
                      OUT=VM$self.OUT,
                      thr=""$item"",
                      par=VM$self,
                      sib=VM$prev, …)"
       OUT “VM$self.chi = VM$v”
       goto 8}
   else ERROR 7
```

```
8: item=GET
   args=""
   while item.type=word do
      {args=concat(args,item); item=GET}
   OUT "VM$v.args=""$args"""
   if item=EOL then
      {OUT ”COMPUTE: EXIT"; goto 1}
   else if item=">" then goto 5
   else if item="|" then
      {p++;
       OUT "P$p = CREATE_PIPE()";
       OUT "VM$v.OUT = OPEN_PIPE(P$p,w)"
       goto 7}
   else ERROR 8
```

Example

```
[self=17]
file -v -u < cat | sort | fill > file
```

SCRIPT:

```
1  VM1 = CREATE_VM(IN=VM17.IN, OUT=VM17.OUT, thr="file", par=VM17, sib="", …)
2  VM1.args = "-v -u"
3  VM17.chi = VM1
4  VM1.IN = OPEN_FILE("cat")
5  P1 = CREATE_PIPE()
6  VM1.OUT = OPEN_PIPE(P1,w)
7  VM2 = CREATE_VM(IN=open_pipe(P1,r), OUT=VM17.OUT, thr="sort", par=VM17, sib="VM1", …)
8  VM17.chi=VM2
9  P2 = CREATE_PIPE()
10 VM2.OUT = OPEN_PIPE(P2,w)
11 VM3 = CREATE_VM(IN=open_pipe(P2,r), OUT=VM17.OUT, thr="fill", par=VM17, sib="VM2", …)
12 VM17.chi=VM3
13 VM3.OUT = OPEN_FILE("file")
14 COMPUTE
15 EXIT
```

Example

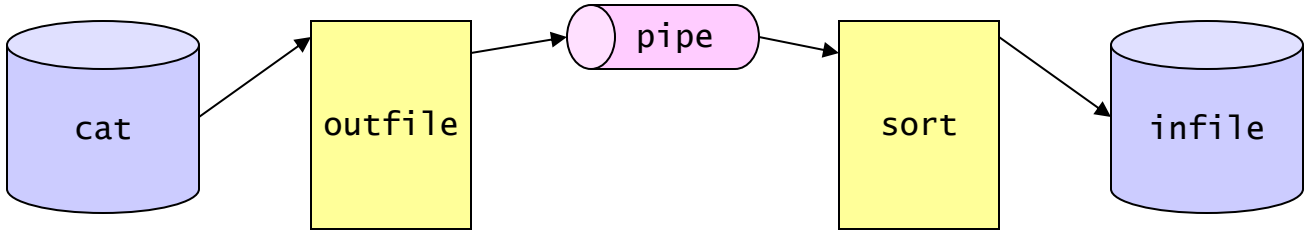
```
outfile < cat | sort > infile
```

DO NOT CONFUSE SYNTAX AND SEMATICS! --

Semantic interpretations would say “outfile” is an output file, “infile” an input file, and “cat” an executable program.

Parser does not know this. Parser knows only that a token in the first position is interpreted as a command name and a token following > or < is interpreted as a file name.

Thus, parser will seek to specify this structure for the example:



Summary

- Shells invokes user-typed commands that are programs in executable library
- Allows command pipelines and file redirection for the first and last commands
- Syntax follows simple grammar
- Parser is finite state machine that generates scripts for the virtual machine manager
- Shell does not “understand” meaning of any command or file name