# Shell Basics

P. J. Denning

For CS471/571

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.
- Pass command line to Parser, which analyzes it and prepares an execution script.
- Use the execution script to construct a set of virtual machines that implement the meaning of the command expression.
- Execute the virtual machine structure.
- Clean up and return to the listening state.

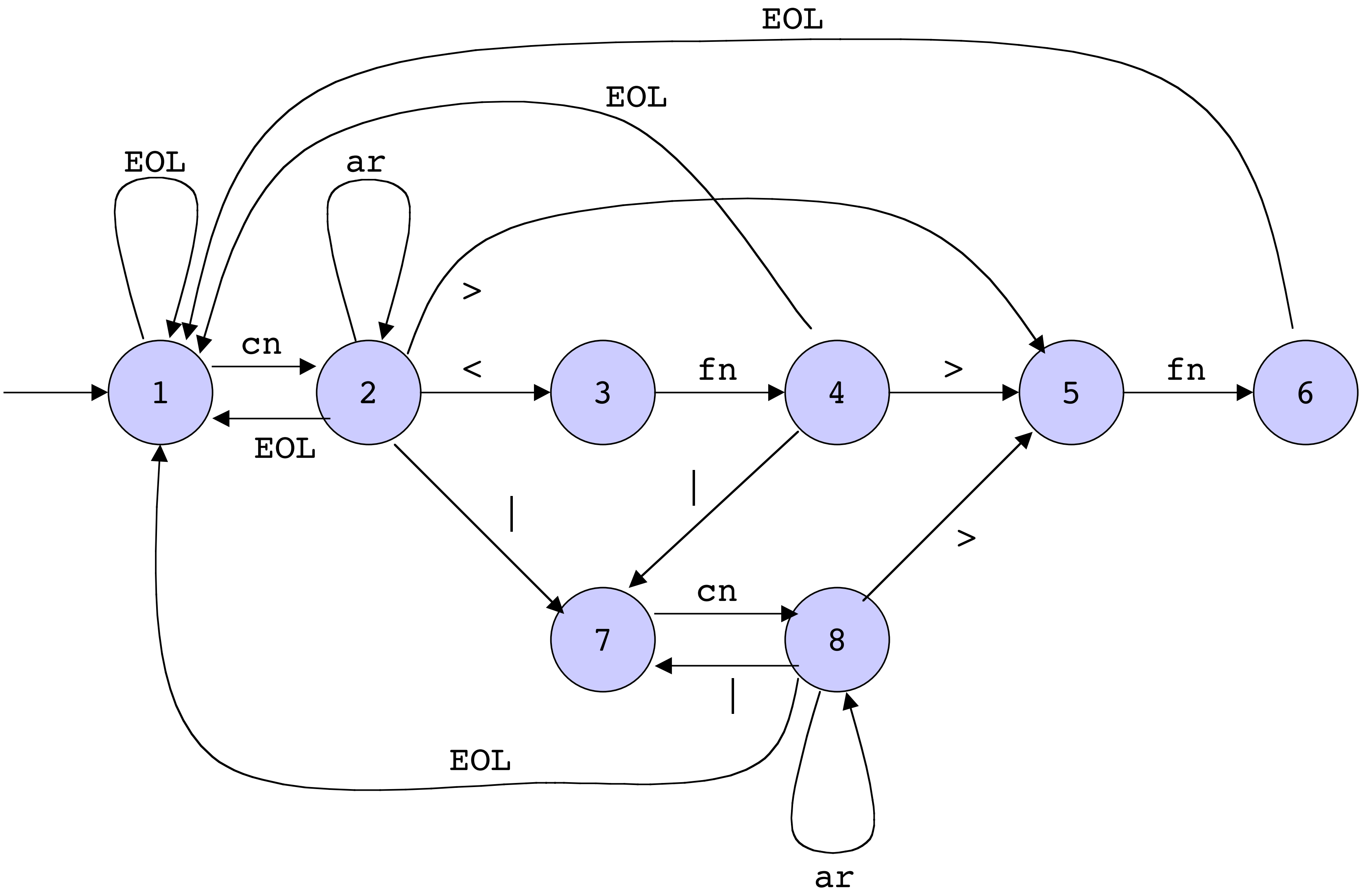
# Parser -- “Heart of Shell”

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

# Example Command Language Grammar

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

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.


EOL
EOL
EOL
ar
>
cn
<
fn
>
fn
1
2
3
4
5
6
EOL
|
|
>
cn
7
8
|
EOL
ar

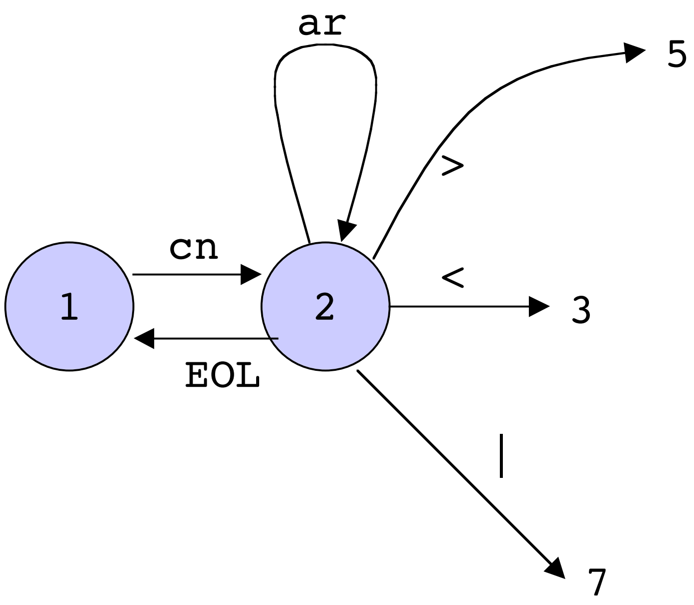
Associate *action code* with each state. Action code examines allowable symbols, specifies next state, and gives next entry in output script. Script tells what OS calls are needed to implement the command. Action code for state 0 initializes counts (v,p) of virtual machines and pipes created.

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.

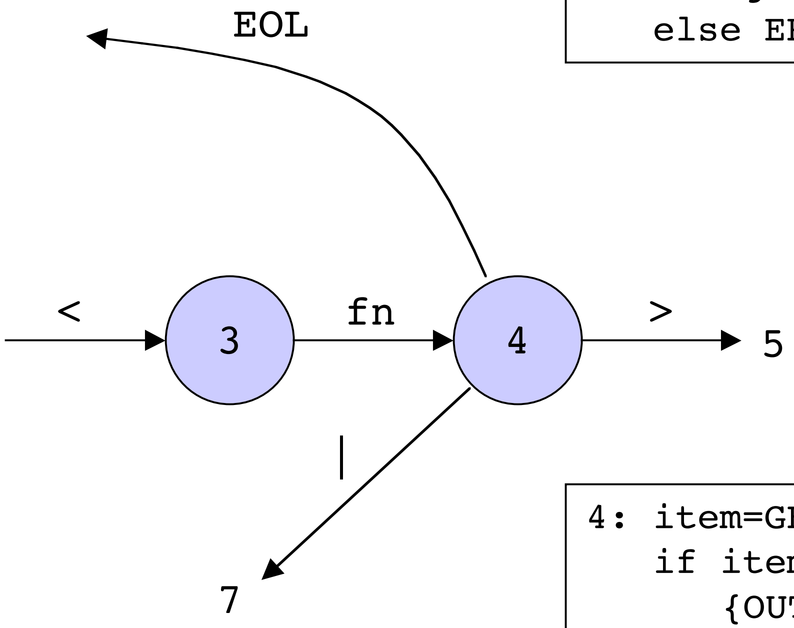
```
0: v = 0
   p = 0
   goto 1
```



```
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, …)"
       goto 2}
   else ERROR 1
```

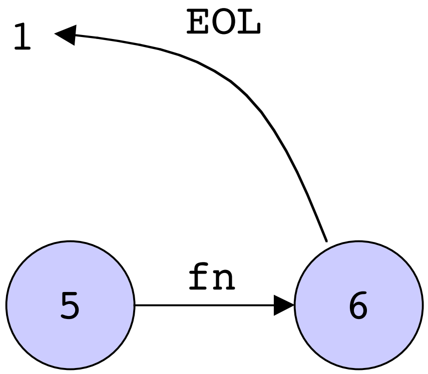
```
2: item=GET
   args=""
   while item.type=word do
      {args=concat(args,item);
       item=GET}
   OUT "VM$v.args=""$args"""
   if item=EOL then
      {OUT "EXIT"; goto 1}
   else if item="<" then goto 3
   else if item=">" then goto 5
   else if item="|" then
      {p++;
       OUT "P$p = create_pipe()";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

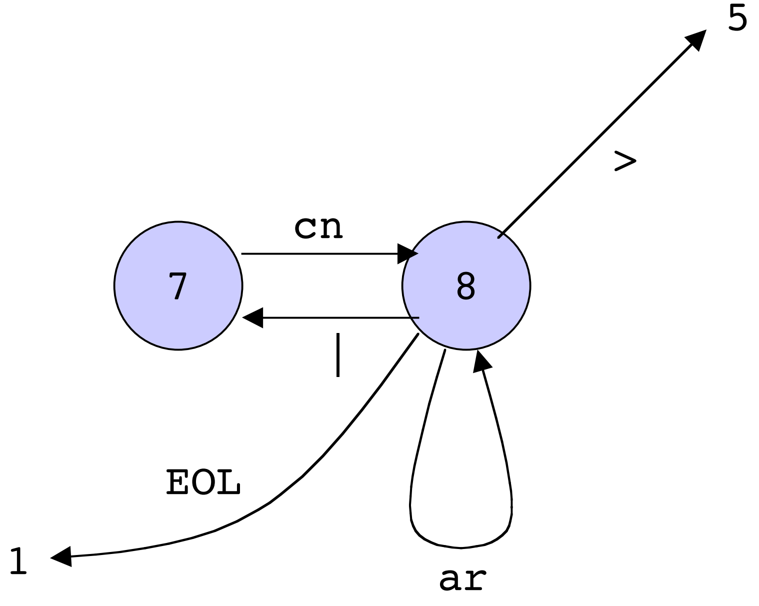


```
4: item=GET
   if item=EOL then
      {OUT "EXIT"; goto 1}
   else if item=">" then goto 5
   else if item="|" then
      {p++;
       OUT "P$p = create_pipe()";
       OUT "VM$v.OUT = open_pipe(P$p,w)"
       goto 7}
   else ERROR 4
```

```
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 "EXIT"; goto 1}
   else ERROR 6
```



```
7: item=GET
   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, …)"
       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 "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:

```
VM1 = create_vm(IN=VM17.IN, OUT=VM17.OUT, thr="file", par=VM17, …)
VM1.args = "-v -u"
VM1.IN = open_file("cat")
P1 = create_pipe()
VM1.OUT = open_pipe(P1,w)
VM2 = create_vm(IN=open_pipe(P1,r), OUT=VM17.OUT, thr="sort", par=VM17, …)
P2 = create_pipe()
VM2.OUT = open_pipe(P2,w)
VM3 = create_vm(IN=open_pipe(P2,r), OUT=VM17.OUT, thr="fill", par=VM17, …)
VM3.OUT = open_file("file")
EXIT
```

# Example

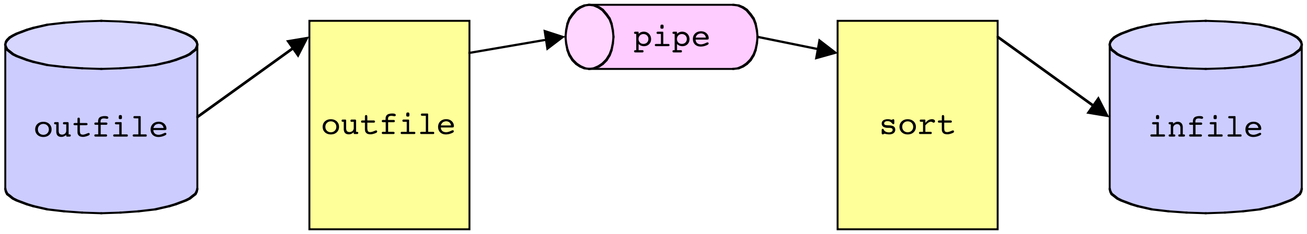
```
outfile < cat | sort > infile
```

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:


outfile
outfile
pipe
sort
infile