A New Look at LR(k)

Bill McKeeman,

MathWorks Fellow

for

Worcester Polytechnic

Computer Science Colloquium

February 20, 2004

Abstract

Knuth offered LR(k) as an algorithm for parsing computer languages (*Information and Control*, Nov. 1965). Most work since has concentrated on parser usability and space/time optimization.

This talk presents LR(k) from first principles, teasing apart some complex formalisms applied by Knuth into simpler components.

Assumptions

The audience is comfortable with mathematical notation and computer programming.

Points to Ponder

- A context-free grammar can describe meaning in addition to syntax.
- The set of reducible parse stacks is a regular language.
- Any finite automata can be described by a context-free grammar.
- The LR(0) machine can be constructed in two stages, first an NFA, then a DFA.
- In most cases, the LALR(1) lookahead can be teased out of the LR(0) machine.

A context-free grammar <**N**, **T**, **G**, **R**> is a four-tuple describing a language **L**.

N is a set of non-terminal symbols (naming syntactic structure like *expression*).

T is a set of terminal symbols (naming the written symbols like `+`**,** `for`**,** `int`…)

V = **N** $\cup$ **T**,   **N** $\cap$ **T** = { }.

G $\subseteq$ **N**, is a set of symbols describing the language (usually just *sentence* or *translation-unit*).

R is a set of rewriting rules, **R** $\subseteq$ **N** $\times$ **V***.

Given an input text from language **L**, we repeatedly substitute the LHS of a rewriting rule for its RHS, leading eventually to something in **G**. The sequence of substitutions is called the *parse*.

Why Use Context-free Grammars?

N = {P D C B}, **T** = {⊥ | & t f}, **G** = {P}.

7 rewriting rules:

1. $P \to D\perp$
2. $D \to D \mid C$
3. $D \to C$
4. $C \to C\&B$
5. $C \to B$
6. $B \to t$
7. $B \to f$

Grammar for boolean expressions

rewritings	*rule*	*consumed terminals*
$\underline{t}\&t \mid f\&f\perp$	6	t
$\underline{B}\&t \mid f\&f\perp$	5	
$C\&\underline{t} \mid f\&f\perp$	6	t
$\underline{C\&B} \mid f\&f\perp$	4	&
$\underline{C} \mid f\&f\perp$	3	
$D \mid \underline{f}\&f\perp$	7	f
$D \mid \underline{B}\&f\perp$	5	
$D \mid C\&\underline{f}\perp$	7	f
$D \mid \underline{C\&B}\perp$	4	&
$\underline{D \mid C}\perp$	2	$\mid$
$\underline{D\perp}$	1	⊥
P		

pcode

tt&ff&|⊥

How to Get a Parse?

- Recursive Descent (top down)
  - Write a C function for each nonterminal
  - Report rule applications
  - Pro: no tools needed, Con: can be buggy
- YACC (bottom up)
  - Write a CFG
  - Implement rule applications
  - Pro: reliable, Con: deal with YACC diagnostics

Parse Stack	*Input*	*S/R sequence*
	t&t|f&f⊥	*start*
t	&t|f&f⊥	*shift* t
B	&t|f&f⊥	*rule* 6
C	&t|f&f⊥	*rule* 5
C&	t|f&f⊥	*shift* &
C&t	|f&f⊥	*shift* t
C&B	|f&f⊥	*rule* 6
C	|f&f⊥	*rule* 4
D	|f&f⊥	*rule* 3
D|	f&f⊥	*shift* |
D|f	&f⊥	*shift* f
D|B	&f⊥	*rule* 7
D|C	&f⊥	*rule* 5
D|C&	f⊥	*shift* &
D|C&f	⊥	*shift* f
D|C&B	⊥	*rule* 5
D|C	⊥	*rule* 4
D	⊥	*rule* 2
D⊥		*shift* ⊥
P		*rule* 1 (*quit*)

The red parse stacks are *reducible*

S/R is useful for languages where a lexer is needed.

The set **L** of all reducible parse stacks for **L** is a regular language (recognizable with a finite automaton). **L** has a grammar that can be derived from the CFG of the original language **L**. The terminal symbols of **L** are the things than can appear on parse stacks (that is, all of **T** and **N**). The nonterminal symbols of **L** are all the partial rules of **L** (that is, leave off zero or more symbols on the end of a rule of **L** and get a nonterminal of **L**). The resulting rules **R** are all of the forms

A→Bb

A→B

A→

Using finite automata terminology, rules A→Bb are read as "in state B if you see b, eat it and go to state A". Rules A→B are the same but no b. Rule A→ says "create A from nothing", which is how we get started.

This definition leaves undefined the situation if an unacceptable symbol appears in the input. One can make the NFA 1-1 by adding an error state E and rules of the form E→Bb to fill out the tables. In practice the error state triggers a syntax diagnostic. It is too verbose for use here.

LR(0) NFA Construction

notation: $\alpha,\beta \in \mathbf{V}^*$

L *defined by* $\langle \mathbf{N}, \mathbf{T}, \mathbf{G}, \mathbf{R} \rangle$

L *defined by* $\langle \mathbf{N}, \mathbf{T}, \mathbf{G}, \mathbf{R} \rangle$

$\mathbf{T} = \mathbf{V}$ *// anything on the parse stack*

$\mathbf{N} = \{A\to\alpha \mid A\to\alpha\beta \in \mathbf{R}\}$ *// any partial rule*

$\mathbf{G} = \mathbf{R}$ *// any completed rule*

$\mathbf{R} = \{A\to\alpha a \to A\to\alpha\ a \mid A\to\alpha a\beta \in \mathbf{R}\}$

$\quad \cup \{B\to\ \to A\to\alpha \mid A\to\alpha B\beta \in \mathbf{R} \wedge A \in \mathbf{N}\}$

$\quad \cup \{A\to\ \to \mid A \in \mathbf{G}\}$

R for **L**

$P \to D \perp$
$D \to D \mid C$
$D \to C$
$C \to C \,\&\, B$
$C \to B$
$B \to t$
$B \to f$

*elements of **N** renamed for readability*

N for **L**

0	$P \to$
1	$P \to D$
2	$P \to D\perp$
3	$D \to$
4	$D \to D$
5	$D \to D \mid$
6	$D \to D \mid C$
7	$D \to C$
8	$C \to$
9	$C \to C$
10	$C \to C\&$
11	$C \to C\&B$
12	$C \to B$
13	$B \to$
14	$B \to t$
15	$B \to f$

R for **L**

$0 \to$
$1 \to 0D$
$2 \to 1\perp$
$4 \to 3D$
$5 \to 4\mid$
$6 \to 5C$
$7 \to 3C$
$9 \to 8C$
$10 \to 9\&$
$11 \to 10B$
$12 \to 8B$
$14 \to 13t$
$15 \to 13f$
$3 \to 0$
$8 \to 3$
$8 \to 5$
$13 \to 8$
$13 \to 10$

G for **L**

2 6 7 11
12 14 15

Apply LR(0) NFA

$D|C\&B$
$0D\mid C\&B$
$3D\mid C\&B$
$4\mid C\&B$
$5C\&B$
$8C\&B$
$9\&B$
$10B$
11

For rule 11 we rewrite $C \to C\&B$ and start over:

$D\mid C$

NFA to DFA

The NFA is not efficient. So we apply the (textbook) NFA-to-DFA transformation, getting yet another context-free grammar.

In this case the nonterminals are *sets* of partial rules, and the terminals are the same as the for the NFA (since they describe the same language).

LR(0) DFA Construction

$E(M) = M \cup \{X \mid Y \in E(M) \wedge X \to Y \in \mathbf{R}\}$ // tx on empty

$S(M, b) = E(\{X \mid Y \in M \wedge X \to Yb \in \mathbf{R}\})$ // tx on b from M

$0 = E(\{0\})$ // start state

$\mathbf{N} = \{0\} \cup \{S(M,b) \mid M \in \mathbf{N} \wedge b \in \mathbf{T}\}$

$\mathbf{G} = \left\{M \,\middle|\, M \in \mathbf{N} \wedge M \cap \mathbf{G} \neq \{\}\right\}$ // contains a rule

$\mathbf{P} = \{0 \to\} \cup \{N \to Mb \mid M \in \mathbf{N} \wedge b \in \mathbf{T} \wedge N = S(M,b)\}$

Note: this construction gives just the states reachable from 0.
The textbook formalism is simpler but computes useless states

The LR(0) DFA

T for **L**

P D C B ⊥
| & t f

N for **L**

0 1 2 3 4 5 6
7 8 9 10 11
12 13 14 15

N renamed for readability

T for **L**

P D C B ⊥
| & t f

N for **L**

0	0 3 8 13
1	1 4
2	2
3	5 8 13
4	6 9
5	10 13
6	11
7	14
8	15
9	7 9
10	12

R for **L**

0→
1→0D
2→1⊥
3→1|
4→3C
5→4&
6→5B
7→5t
8→5f
10→3B
7→3t
8→3f
9→0C
10→0B
7→0t
8→0f

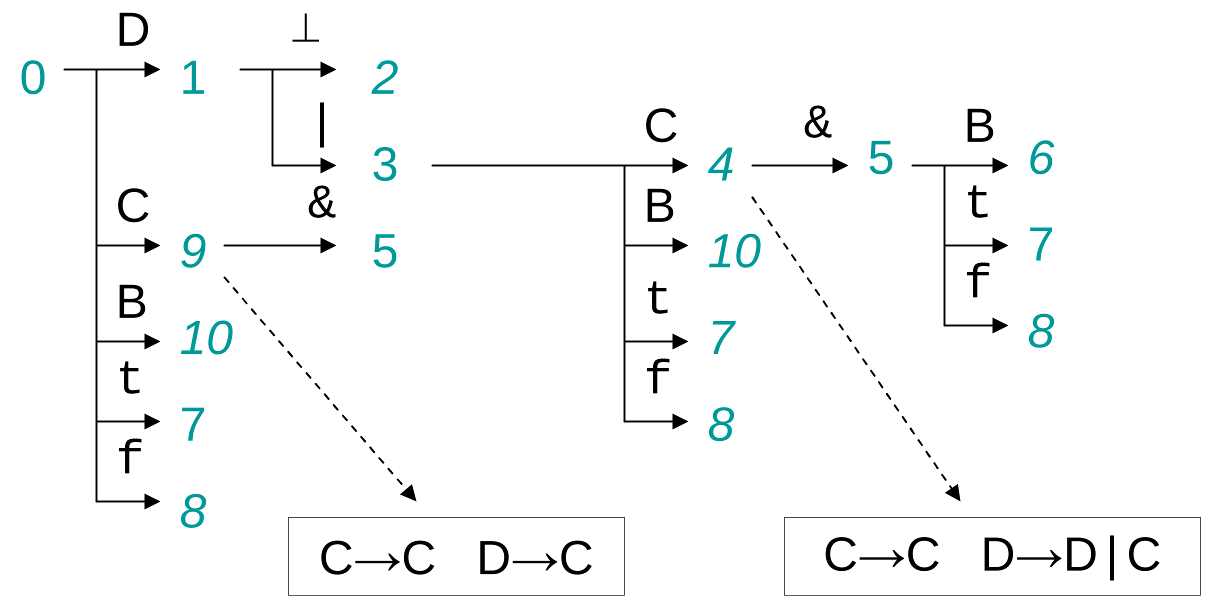
G for **L**

2 4 6 7
8 9 10

Apply LR(0) DFA

D|C&B
0D|C&B
1|C&B
3C&B
4&B
5B
6

The LR(0) DFA



Apply LR(0) DFA

D|C&B
0D|C&B
1|C&B
3C&B
4&B
5B
6

States 2, 6, 7, 8, 10 are obviously in **G** (no place to go).

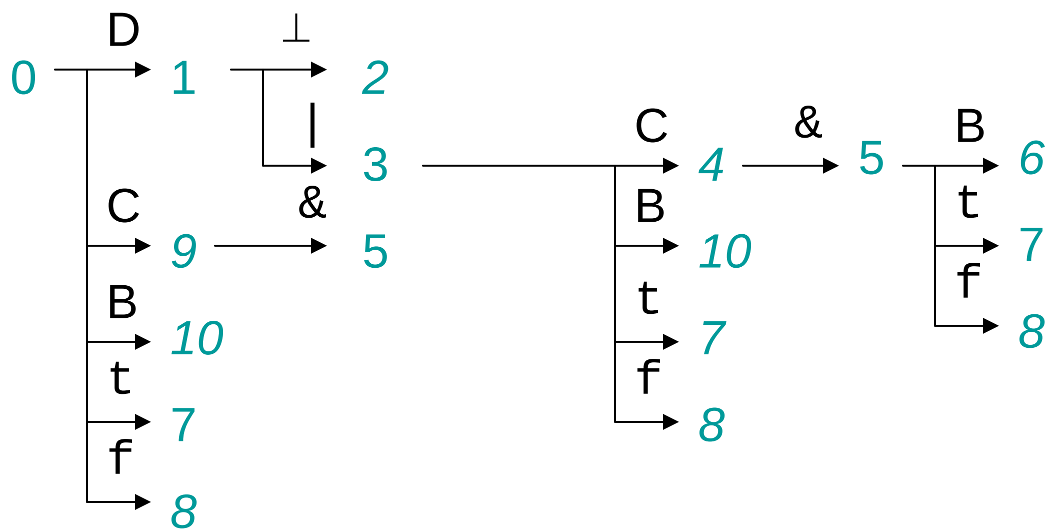
States 4 and 9 are in **G** but also can shift.

States 4 and 9 show LR(0) shift/reduce conflicts.

For this CFG, *shift if you can* resolves the conflicts.

Note that is what we did for state *4* above.

The LALR(1) Lookahead



The LALR(1) lookahead for D→D | C in state *4* is found by stepping back from *4*, over the RHS of the rule (D | C), then stepping forward (from 0 in this case) over the LHS of the rule (D) and collecting the transition symbols (⊥ |). Since the transition symbols out of 1 are different from the transition symbols out of *4* (&), there is no LALR(1) shift/reduce conflict.

If we erroneously did the reduction D→D | C while looking at something other than ⊥ or | , we would immediately fail on the next step.

Conventional
LALR(1)
parsing table

see:

in state:

	P	D	C	B	⊥	|	&	t	f
0		1	9	10				7	8
1					2	3			
2	R1	R1	R1	R1	R1	R1	R1	R1	R1
3			4	10				7	8
4					R2	R2	5		
5				6				7	8
6					R4	R4	R4		
7					R6	R6	R6		
8					R7	R7	R7		
9					R3	R3	5		
10					R5	R5	R5		

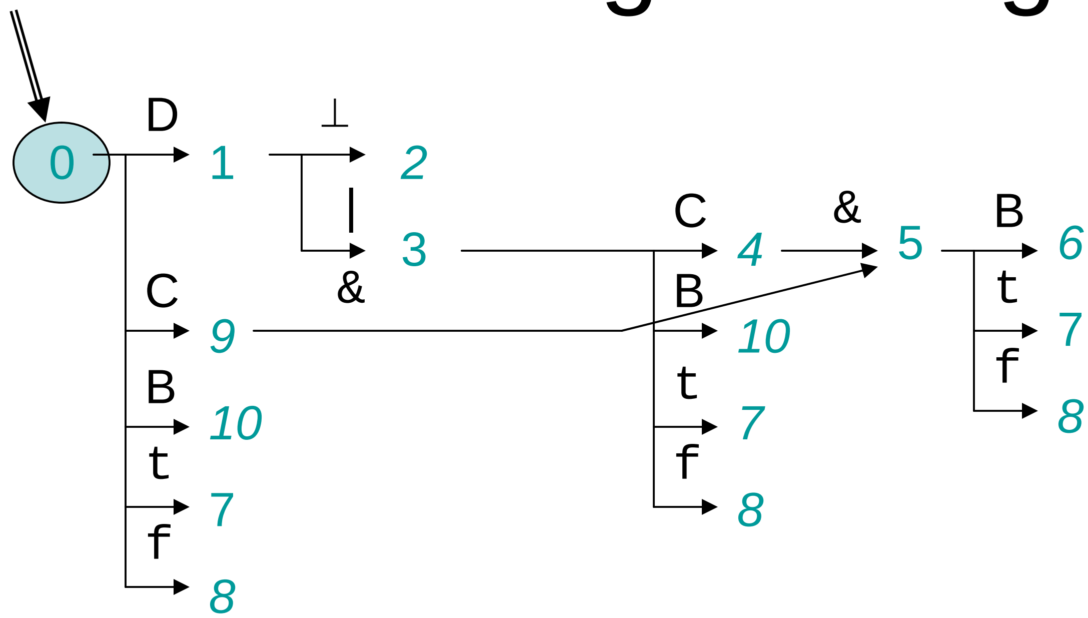
NQLALR vs LALR

The simple trick for LALR(1) lookahead worked because the grammar for **L** had no erasing rules (rules with empty RHS). One way to insure the no-erasing property is to transform the user's grammar, removing the erasing rules.

The trick is to remove each erasing rule, say F→, and everywhere in the grammar that F is used, create a new rule with F erased.

F→ F →i H→hFj	⟹	F→i H→hFj H→hj

Putting it all together (1)

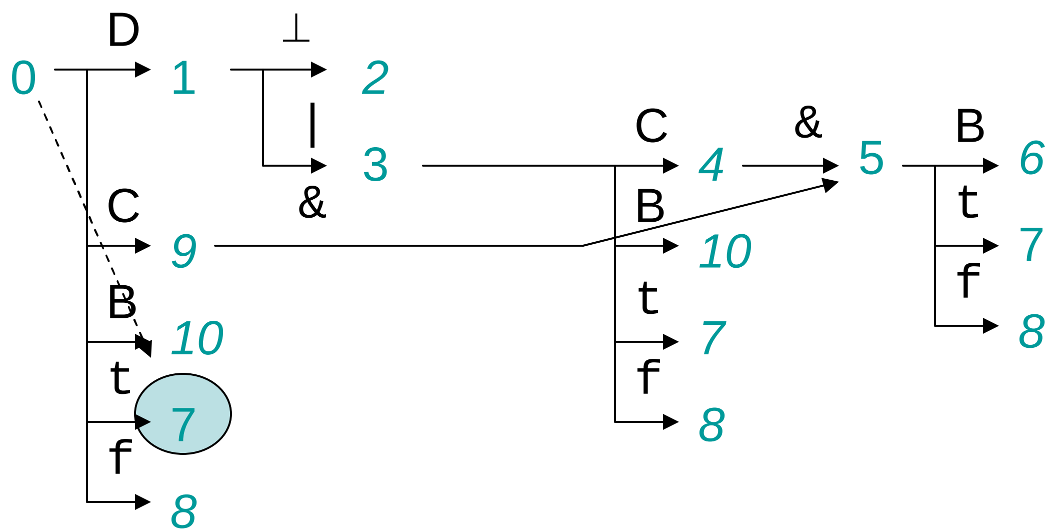


reduce states

2	P→D⊥
4	D→D|C
6	C→C&B
7	B→t
8	B→f
9	D→C
10	C→B

parse stack	*input*	*action*
0	`t&t|f&f⊥`	*start*

Putting it all together (2)

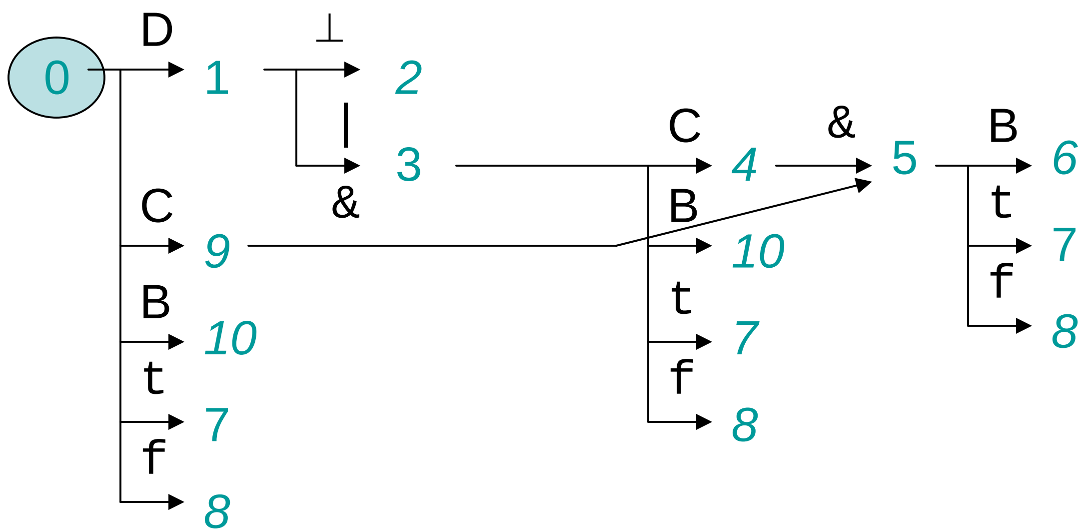


reduce states

2	P→D⊥
4	D→D|C
6	C→C&B
7	B→`t`
8	B→`f`
9	D→C
10	C→B

parse stack	*input*	*action*
7 `t` 0	`&t|f&f⊥`	*shift* `t`

Putting it all together (3)

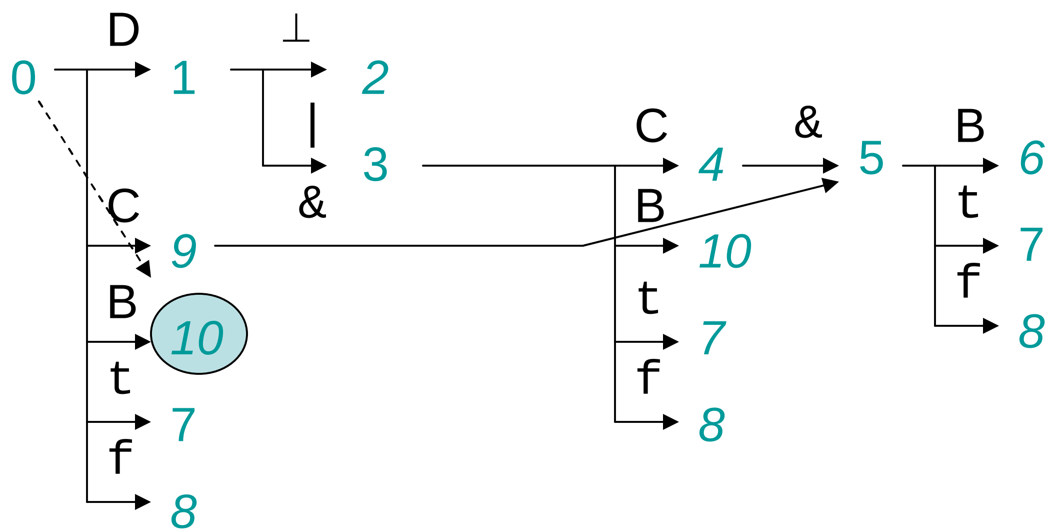


reduce states

2	P→D⊥
4	D→D|C
6	C→C&B
7	B→t
8	B→f
9	D→C
10	C→B

parse stack	*input*	*action*
B 0	`&t|f&f⊥`	*reduce* B→t

Putting it all together (4)



reduce states

2 P→D⊥

4 D→D|C

6 C→C&B

7 B→t

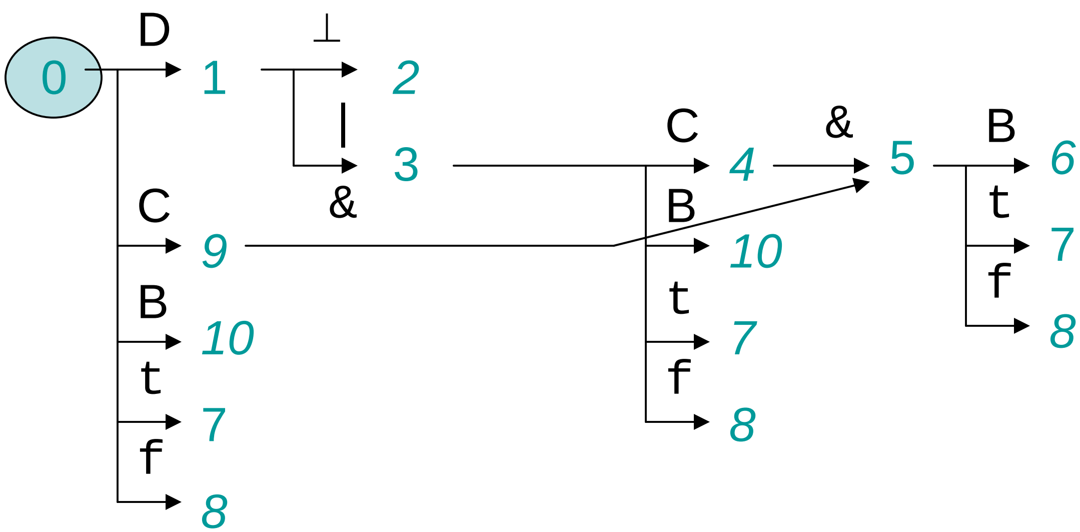
8 B→f

9 D→C

10 C→B

parse stack	*input*	*action*
10 B 0	`&t|f&f⊥`	*shift* B

Putting it all together (5)



reduce states

2 P→D⊥

4 D→D|C

6 C→C&B

7 B→t

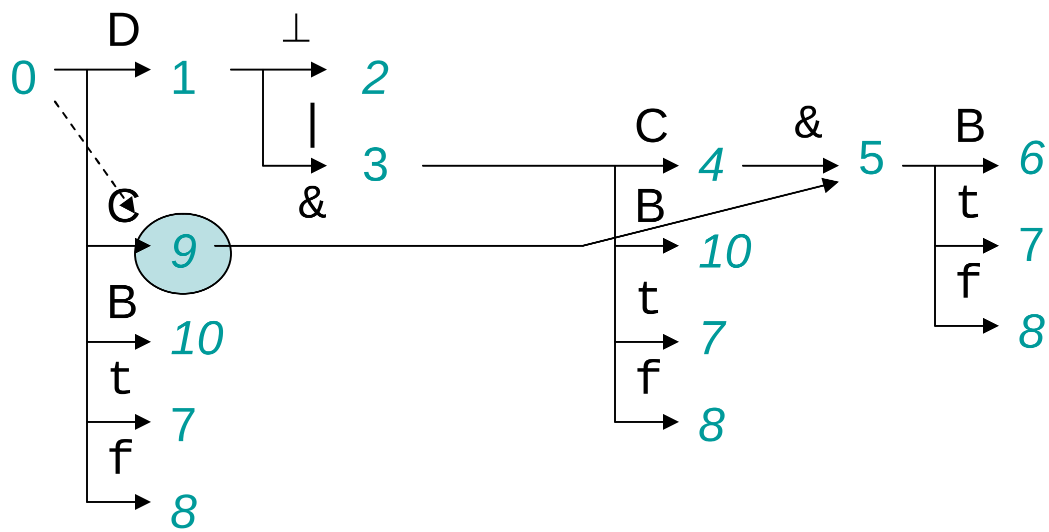
8 B→f

9 D→C

10 C→B

parse stack	*input*	*action*
0 C	&t|f&f⊥	*reduce* C→B

Putting it all together (6)



reduce states

2 P→D⊥

4 D→D|C

6 C→C&B

7 B→t

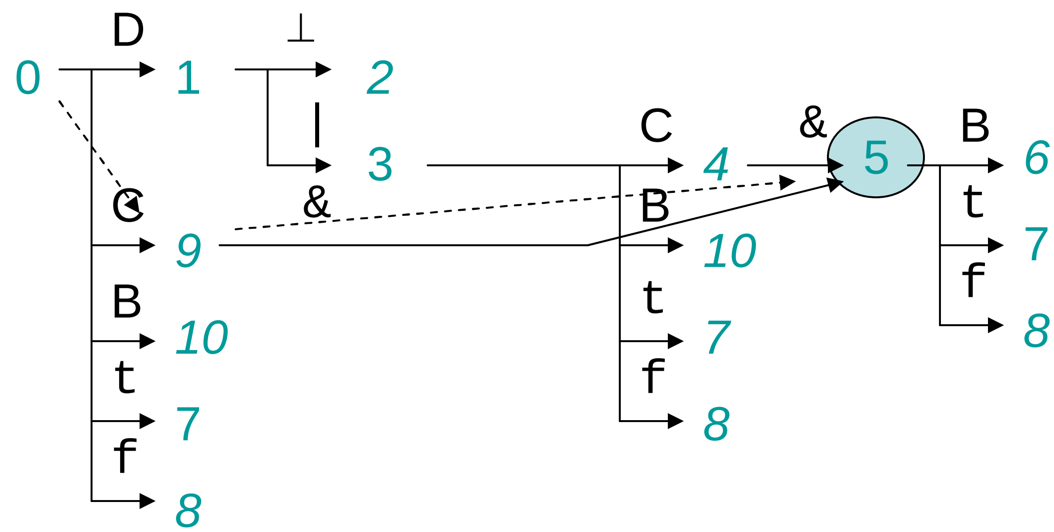
8 B→f

9 D→C

10 C→B

parse stack	*input*	*action*
9 C 0	&t|f&f⊥	*shift* C

Putting it all together (7)



reduce states

2 P→D⊥

4 D→D|C

6 C→C&B

7 B→t

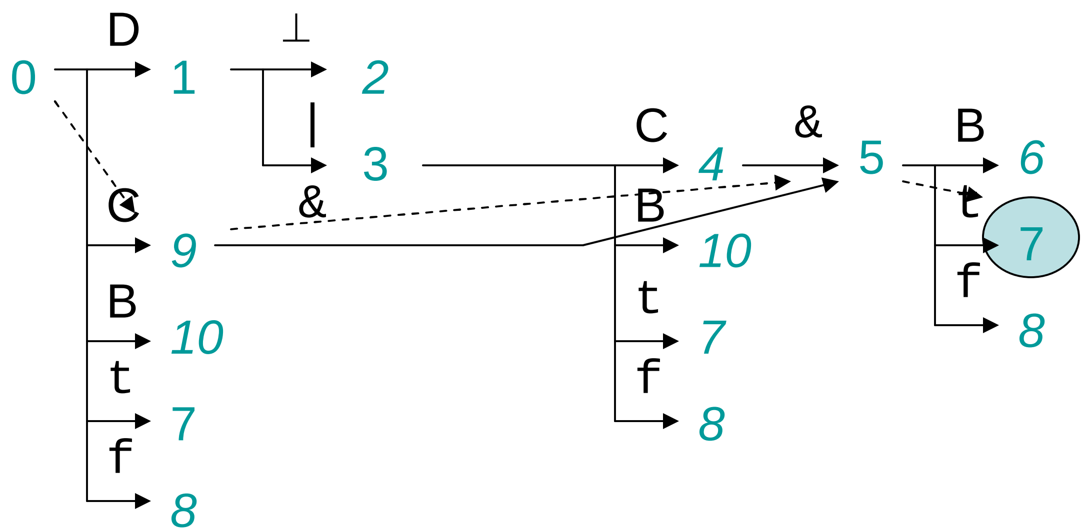
8 B→f

9 D→C

10 C→B

parse stack	*input*	*action*
5	`t|f&f⊥`	*shift* &
&		
9		
C		
0		

Putting it all together (8)

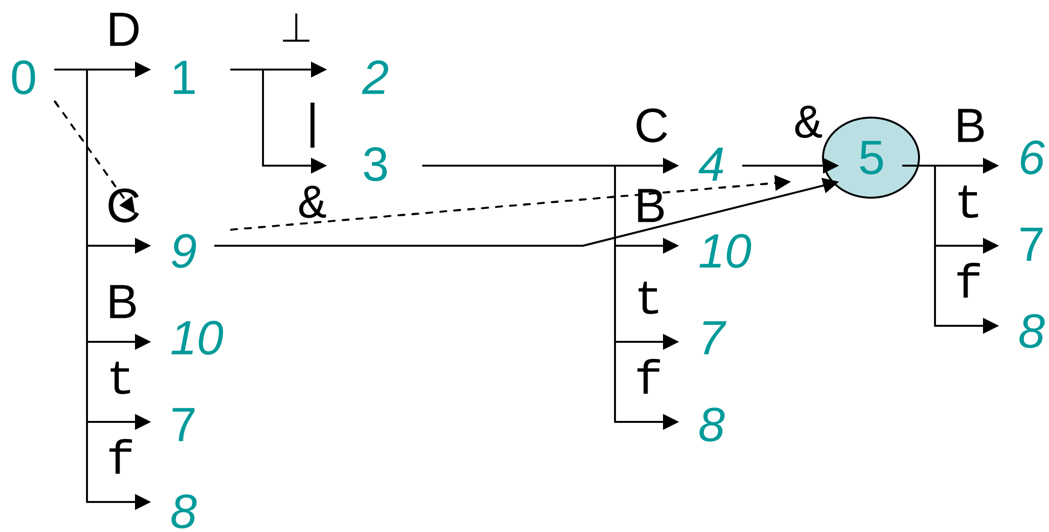


reduce states

2	P→D⊥
4	D→D|C
6	C→C&B
7	B→`t`
8	B→`f`
9	D→C
10	C→B

parse stack	*input*	*action*
7 `t` 5 & 9 C 0	`|f&f⊥`	*shift* `t`

Putting it all together (9)



reduce states

2 P→D⊥

4 D→D|C

6 C→C&B

7 B→t

8 B→f

9 D→C

10 C→B

parse stack

B
5
&
9
C
0

input

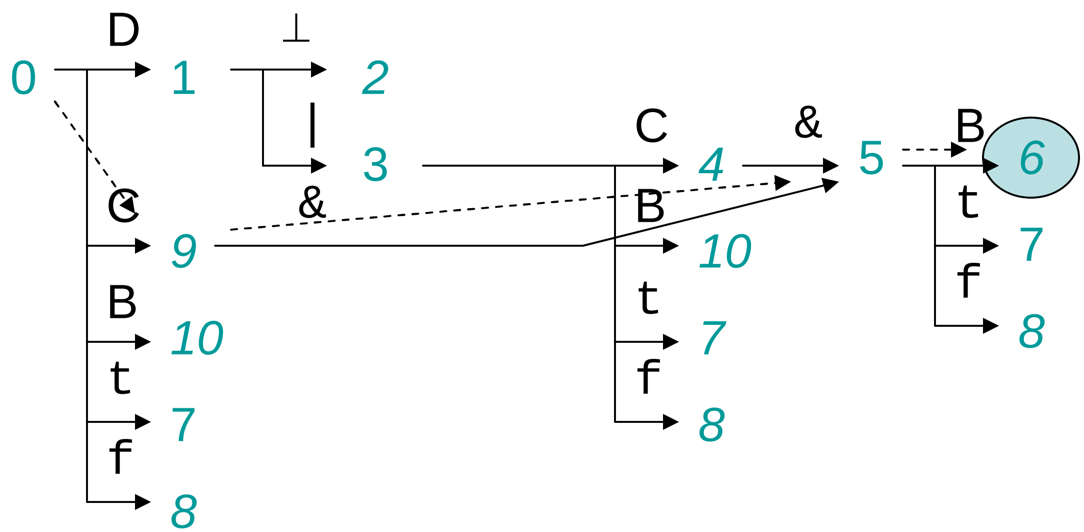
`|f&f⊥`

action

reduce B→t

Note: we do not have to go all the way back to state 0.

Putting it all together (10)



reduce states

2 P→D⊥

4 D→D|C

6 C→C&B

7 B→t

8 B→f

9 D→C

10 C→B

parse stack

6
B
5
&
9
C
0

input

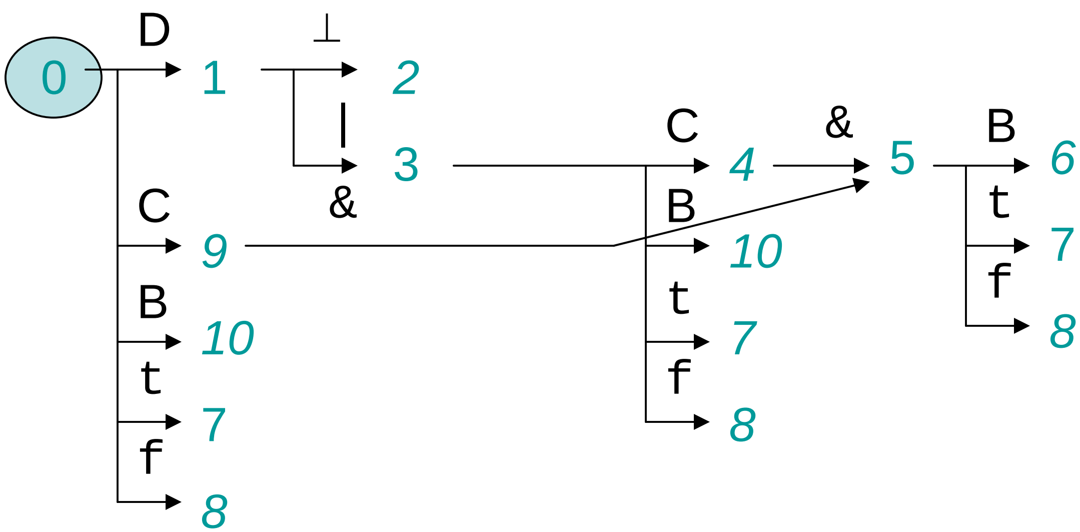
`|f&f⊥`

action

shift B

Note: we do not have to go all the way back to state 0.

Putting it all together (11)

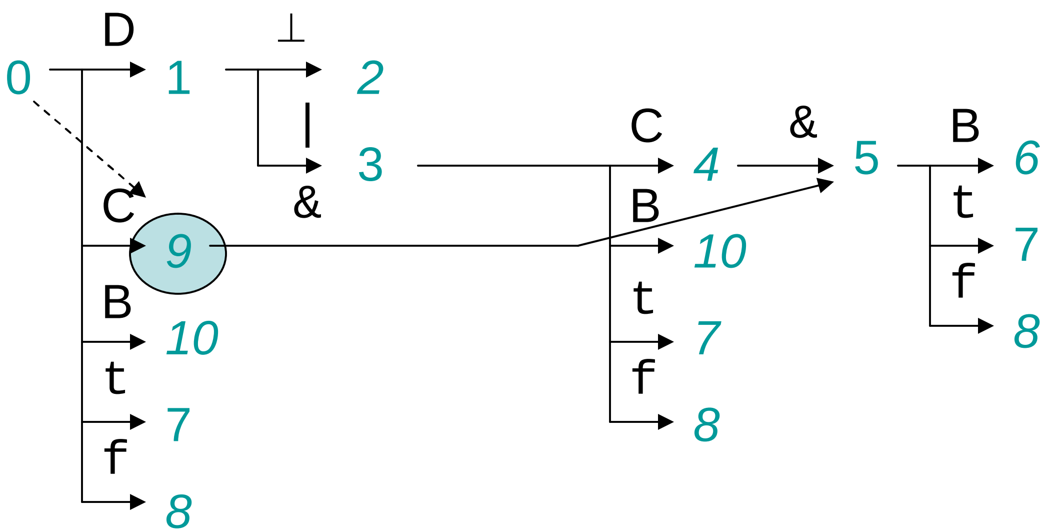


reduce states

2	P→D⊥
4	D→D|C
6	C→C&B
7	B→t
8	B→f
9	D→C
10	C→B

parse stack	*input*	*action*
C 0	`|f&f⊥`	*reduce* C→C&B

Putting it all together (12)



reduce states

2 P→D⊥

4 D→D|C

6 C→C&B

7 B→t

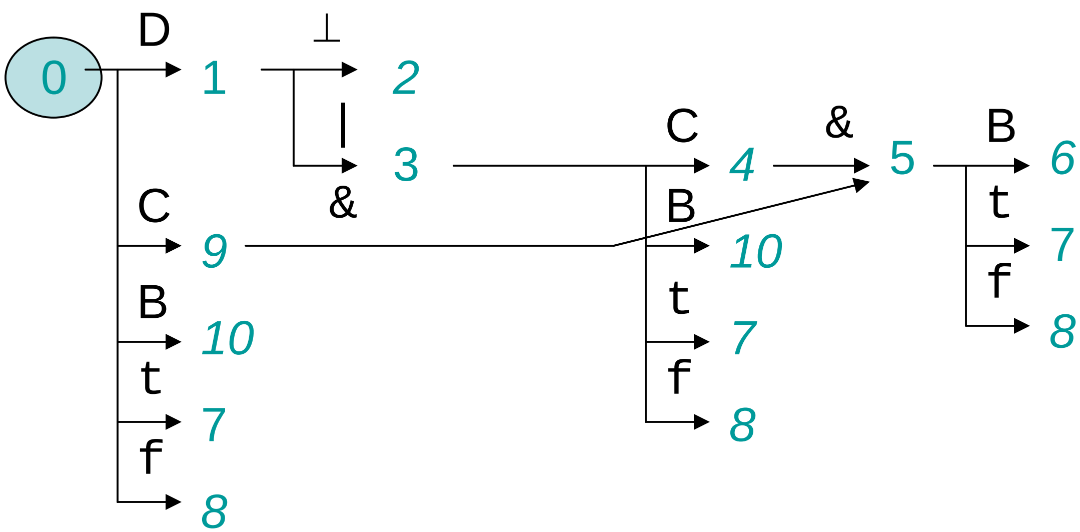
8 B→f

9 D→C

10 C→B

parse stack	*input*	*action*
9 C 0	|f&f⊥	*shift* C

Putting it all together (13)



reduce states

2 P→D⊥

4 D→D|C

6 C→C&B

7 B→t

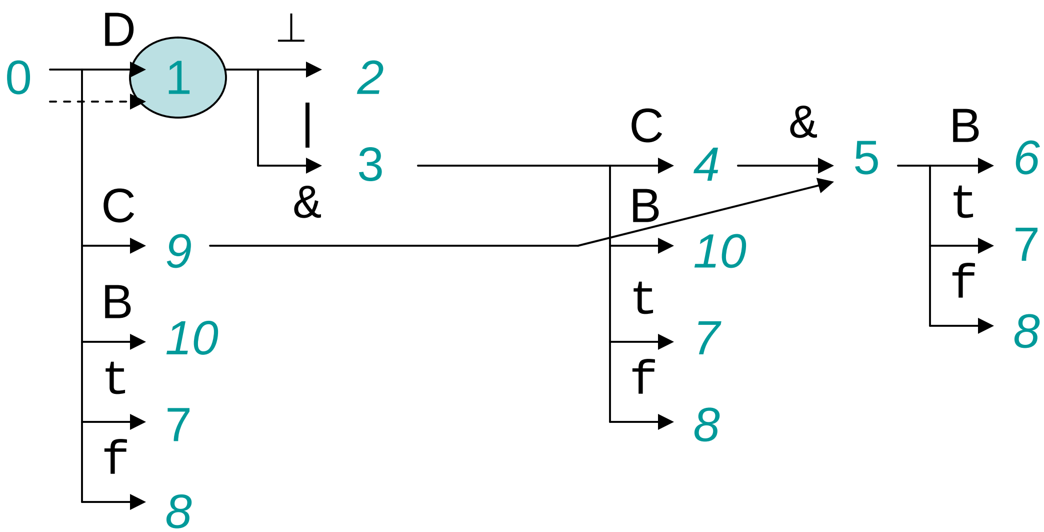
8 B→f

9 D→C

10 C→B

parse stack	*input*	*action*
D 0	|f&f⊥	*reduce* D→C

Putting it all together (14)

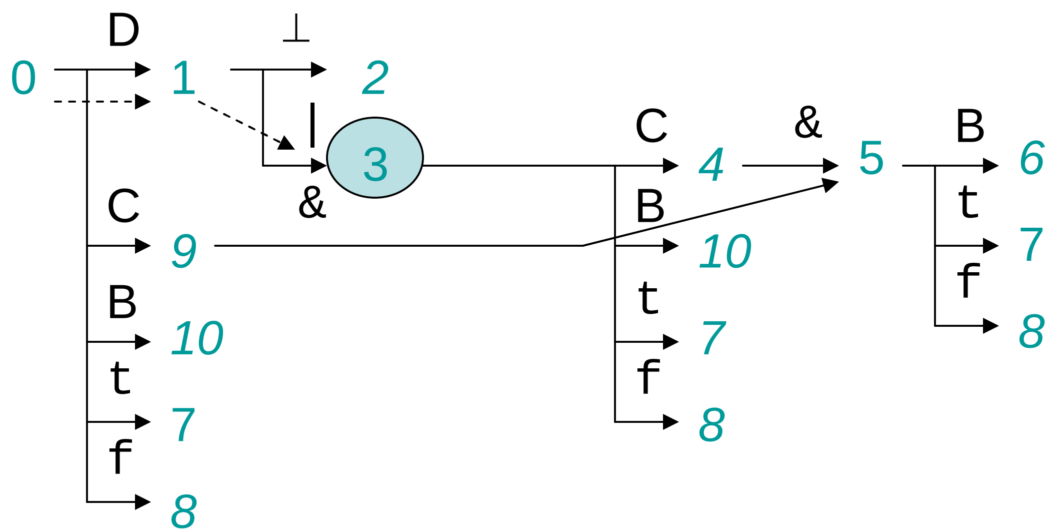


reduce states

2	P→D⊥
4	D→D|C
6	C→C&B
7	B→t
8	B→f
9	D→C
10	C→B

parse stack	*input*	*action*
1 D 0	|f&f⊥	*shift* D

Putting it all together (15)

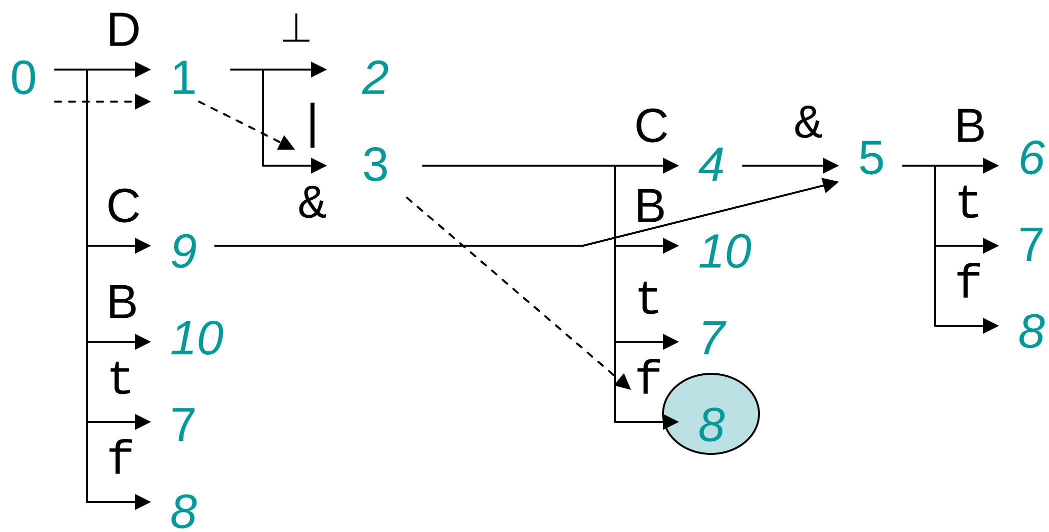


reduce states

2	P→D⊥
4	D→D|C
6	C→C&B
7	B→t
8	B→f
9	D→C
10	C→B

parse stack	*input*	*action*
3 | 1 D 0	`f&f⊥`	*shift* |

Putting it all together (16)

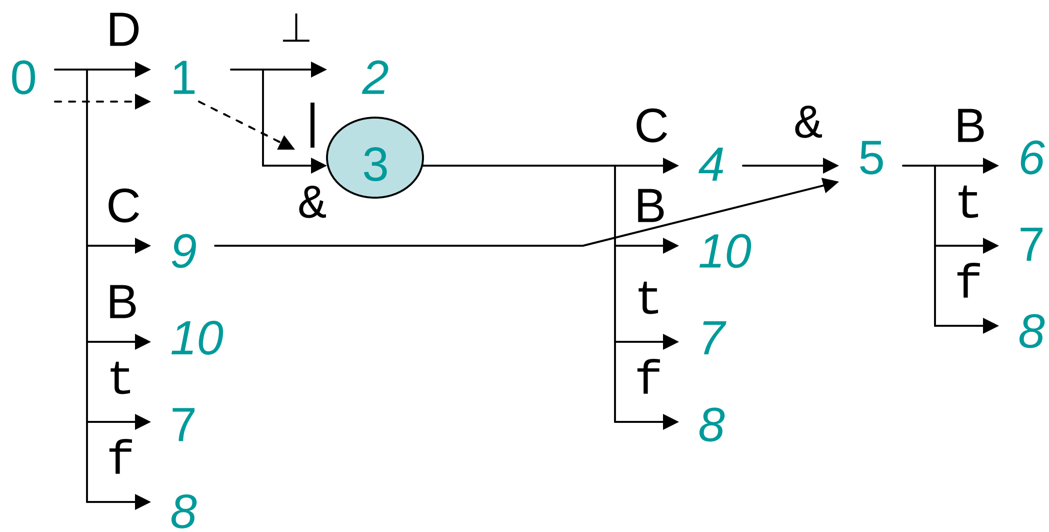


reduce states

2	P→D⊥
4	D→D|C
6	C→C&B
7	B→t
8	B→f
9	D→C
10	C→B

parse stack	*input*	*action*
8 f 3 | 1 D 0	&f⊥	*shift* f

Putting it all together (17)



reduce states

2 P→D⊥

4 D→D|C

6 C→C&B

7 B→t

8 B→f

9 D→C

10 C→B

parse stack

B
3
|
1
D
0

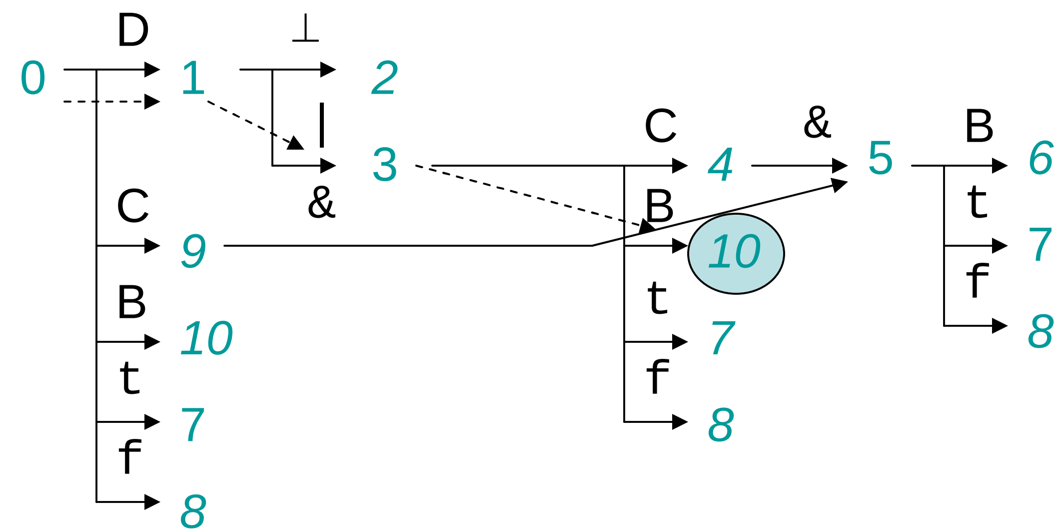
input

&f⊥

action

reduce B→f

Putting it all together (18)



reduce states

2 P→D⊥

4 D→D|C

6 C→C&B

7 B→t

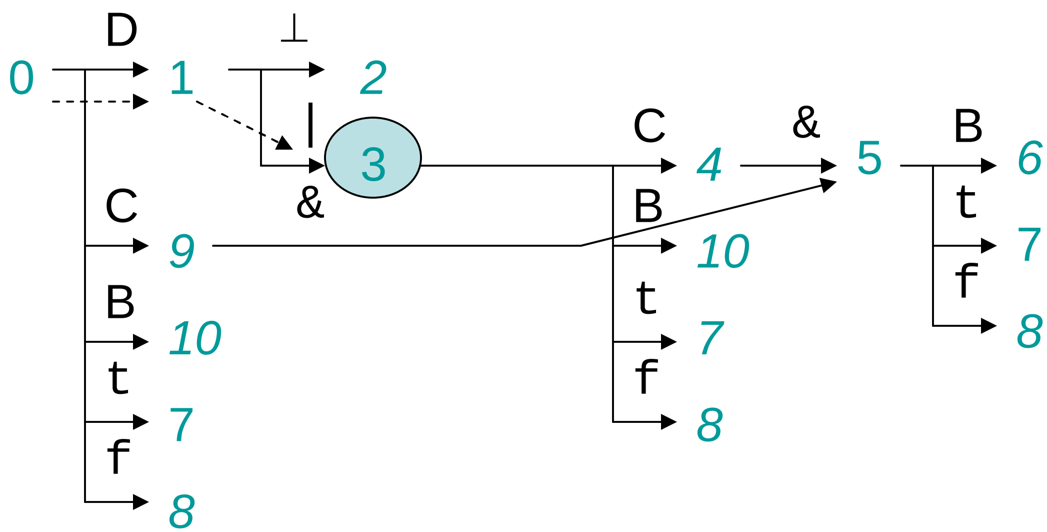
8 B→f

9 D→C

10 C→B

parse stack	*input*	*action*
10 B 3 | 1 D 0	&f⊥	*shift* B

Putting it all together (19)



reduce states

2 P→D⊥

4 D→D|C

6 C→C&B

7 B→t

8 B→f

9 D→C

10 C→B

parse stack

C
3
|
1
D
0

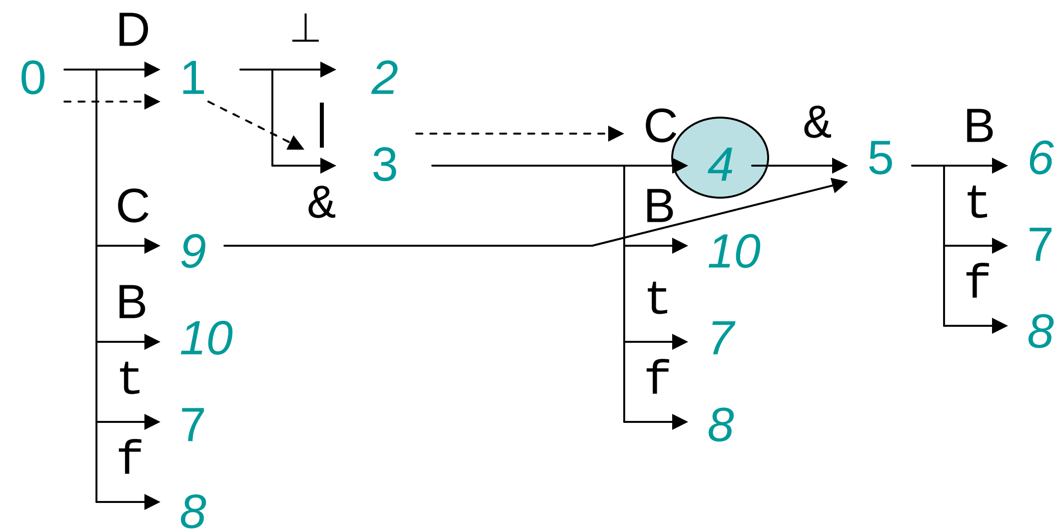
input

&f⊥

action

reduce C→B

Putting it all together (20)



reduce states

2 P→D⊥

4 D→D|C

6 C→C&B

7 B→t

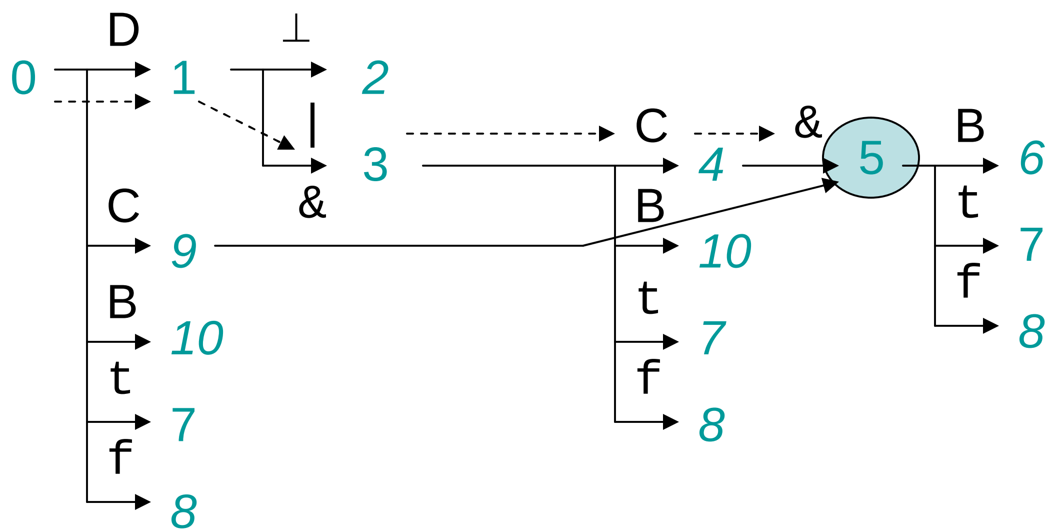
8 B→f

9 D→C

10 C→B

parse stack	*input*	*action*
4	&f⊥	*shift* C
C		
3		
|		
1		
D		
0		

Putting it all together (21)



reduce states

2 P→D⊥

4 D→D|C

6 C→C&B

7 B→t

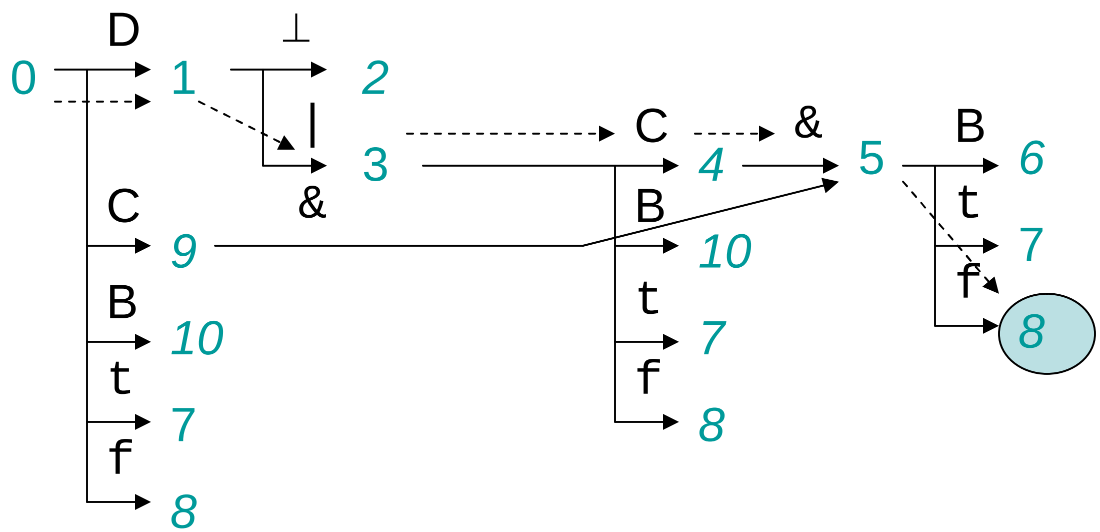
8 B→f

9 D→C

10 C→B

parse stack	*input*	*action*
5 & *4* C 3 | 1	f⊥	*shift* &

Putting it all together (22)



reduce states

2 P→D⊥

4 D→D|C

6 C→C&B

7 B→t

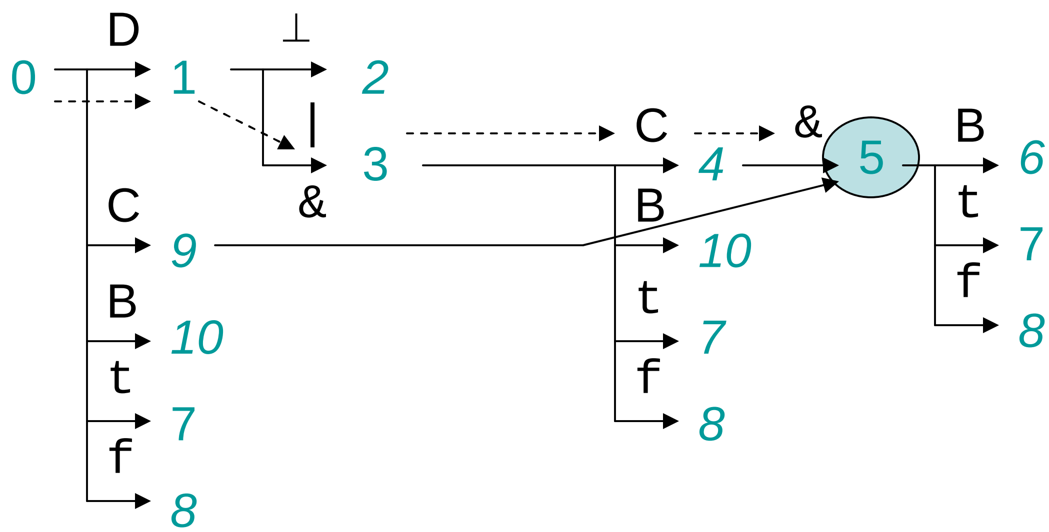
8 B→f

9 D→C

10 C→B

parse stack	*input*	*action*
8	⊥	*shift* f
f		
5		
&		
4		
C		
3		

Putting it all together (23)

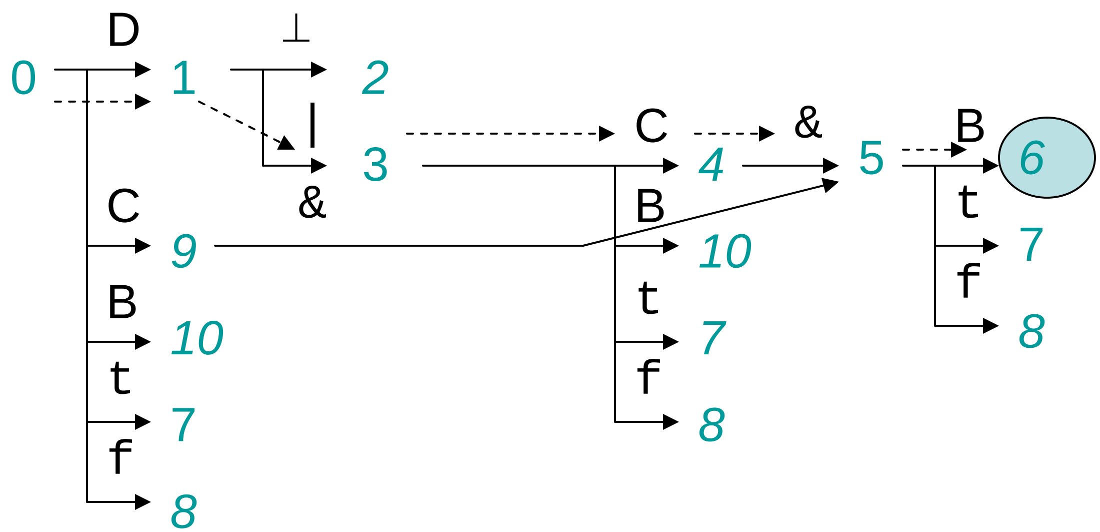


reduce states

- *2* P→D⊥
- *4* D→D|C
- *6* C→C&B
- *7* B→t
- *8* B→f
- *9* D→C
- *10* C→B

parse stack	*input*	*action*
B	⊥	*reduce* B→f
5		
&		
4		
C		
3		
|		

Putting it all together (24)

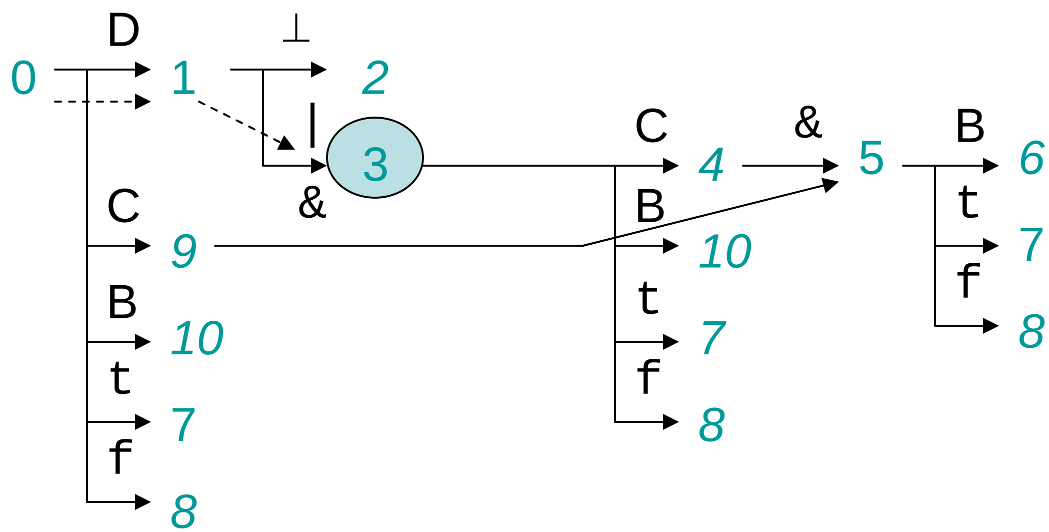


reduce states

2	P→D⊥
4	D→D | C
6	C→C&B
7	B→t
8	B→f
9	D→C
10	C→B

parse stack	*input*	*action*
6 B 5 & 4 C 3	⊥	*shift* B

Putting it all together (25)



reduce states

2	P→D⊥
4	D→D | C
6	C→C&B
7	B→t
8	B→f
9	D→C
10	C→B

parse stack

C
3
|
1
D
0

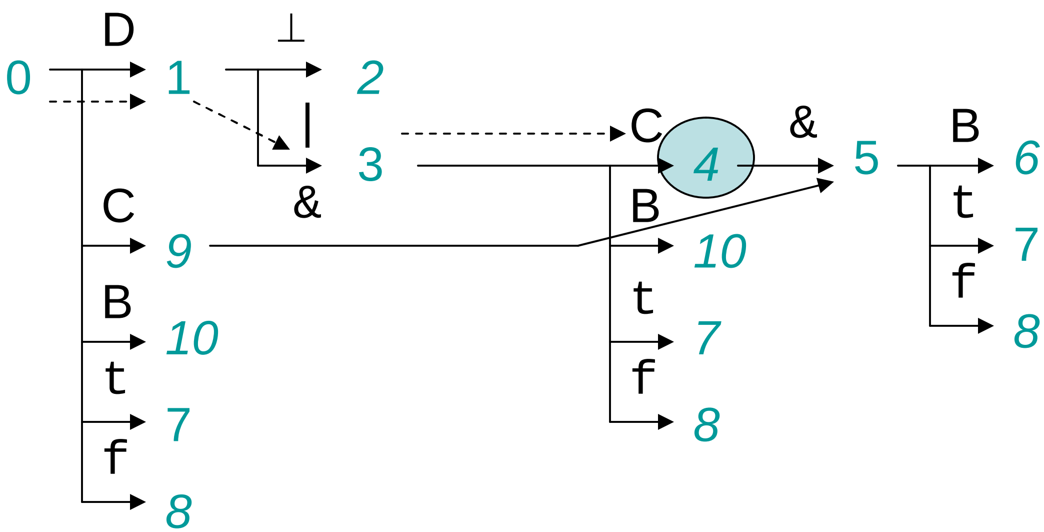
input

⊥

action

reduce C→C&B

Putting it all together (26)

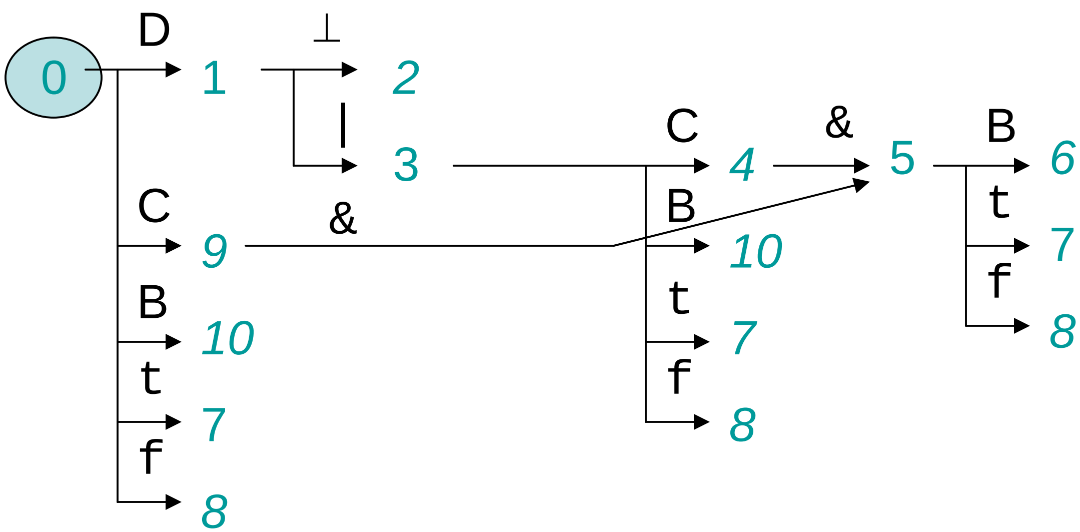


reduce states

2	P→D⊥
4	D→D | C
6	C→C&B
7	B→t
8	B→f
9	D→C
10	C→B

parse stack	*input*	*action*
4 C 3 | 1 D 0	⊥	*shift* C

Putting it all together (27)



reduce states

2	P→D⊥
4	D→D | C
6	C→C&B
7	B→t
8	B→f
9	D→C
10	C→B

parse stack

D

0

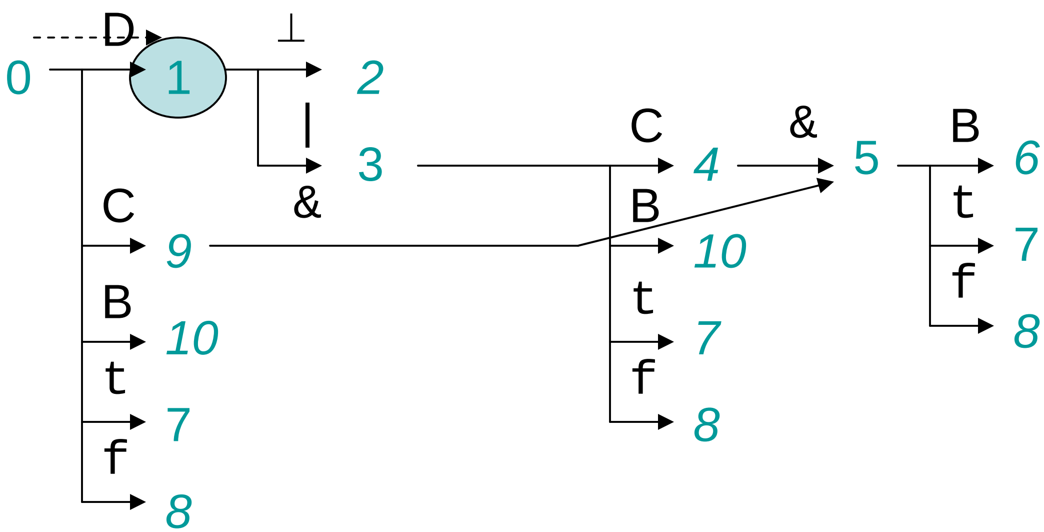
input

⊥

action

reduce D→D | C

Putting it all together (28)

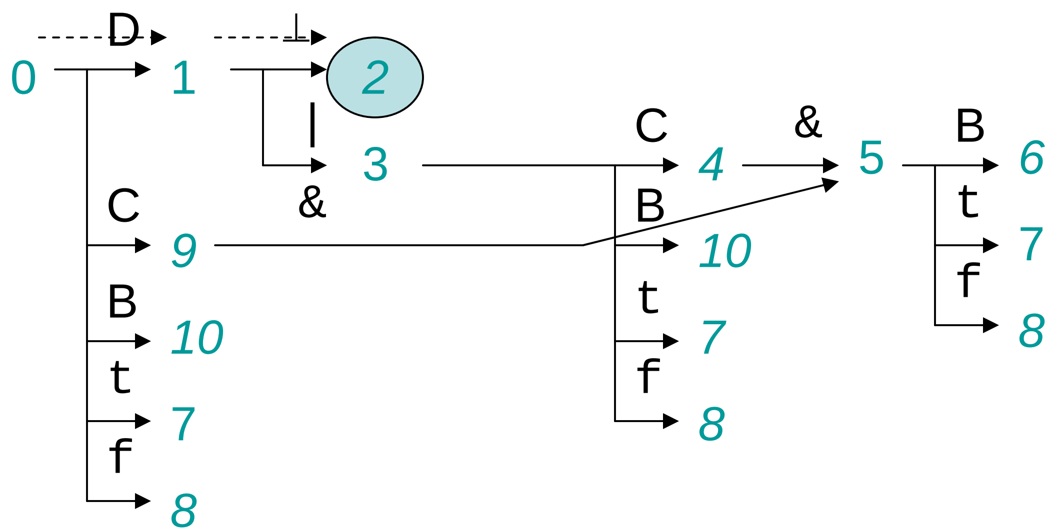


reduce states

2	P→D⊥
4	D→D | C
6	C→C&B
7	B→t
8	B→f
9	D→C
10	C→B

parse stack	*input*	*action*
1 D 0	⊥	*shift* D

Putting it all together (29)



reduce states

2 P→D⊥

4 D→D | C

6 C→C&B

7 B→t

8 B→f

9 D→C

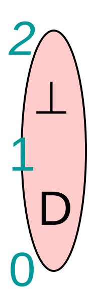
10 C→B

parse stack

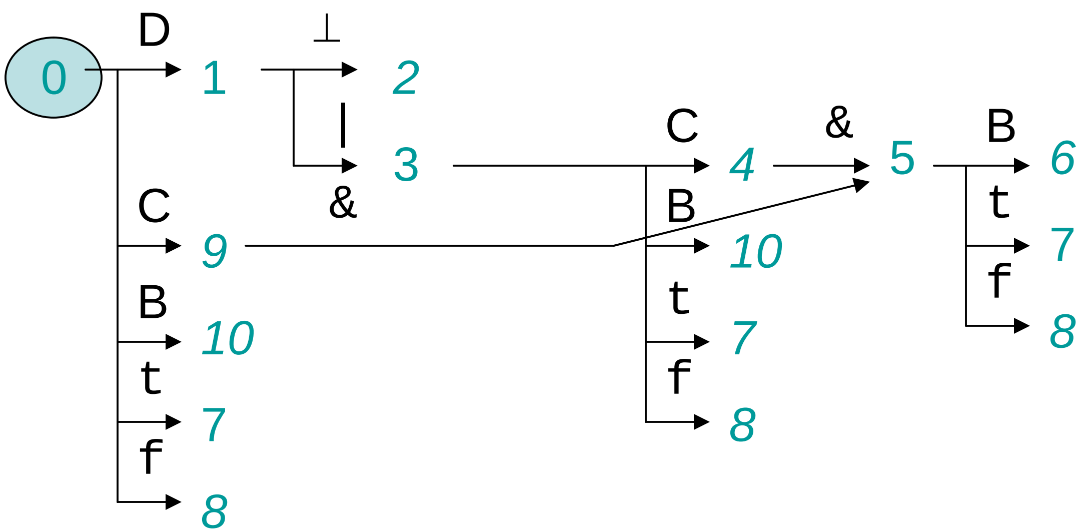
input

action

shift ⊥



Putting it all together (30)



reduce states

2 P→D⊥

4 D→D|C

6 C→C&B

7 B→t

8 B→f

9 D→C

10 C→B

parse stack

P

0

input

action

reduce P→D⊥

quit