Notices

- Please submit your lab hash to the `compiler_labs.xlsx` spreadsheet in the Files tab of the General channel on Teams (first lab is due **tomorrow**).

- Coursework spec now available at `https://github.com/LangProc/langproc-cw`

- Please use the `compiler_pairs.xlsx` spreadsheet to record your pair, or to help you find a partner.

Lecture 5:
More parsing

John Wickerson

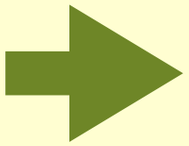
Compilers

What we know so far

- Languages can be defined using a **grammar** made up of **production rules** featuring **non-terminals** and **terminals**.
- Grammars should be written to avoid **ambiguity**.
- **Regular grammars** are equivalent to regexes.
- **Context-free grammars** are more expressive than regular grammars.
- And **context-sensitive grammars** are more expressive still.
- Most programming languages can be defined using a grammar that is (more or less) context-free.

Outline

- A lesson in **grammar**



How to build a **recursive descent** parser

- How to build a **shift/reduce** parser

- How to use **Yacc** to generate a parser automatically

Recursive descent

expr ::= expr + term | expr – term | term

term ::= term * factor | factor

factor ::= (expr) | **N**

Recursive descent

expr ::= term + expr | term − expr | term

term ::= factor * term | factor

factor ::= (expr) | **N**

Recursive descent

E ::= T + E | T − E | T

T ::= F * T | F

F ::= (E) | **N**

NB: this is only the rough idea

```
int pE() {
  return pT() && p('+') && pE()
      || pT() && p('-') && pE()
      || pT();
}
...
```

What's the problem here?

E ::= E + T | E − T | T

T ::= T * F | F

F ::= (E) | **N**

```
int pE() {
  return pE() && p('+') && pT()
      || pE() && p('-') && pT()
      || pT();
}
...
```

Recursive descent

- **First problem.** Recursive descent cannot parse **left-recursive** grammars (those that contain a production of the form "A ⟶ A...").

- **Solution.** Left-recursion can be removed by rewriting:

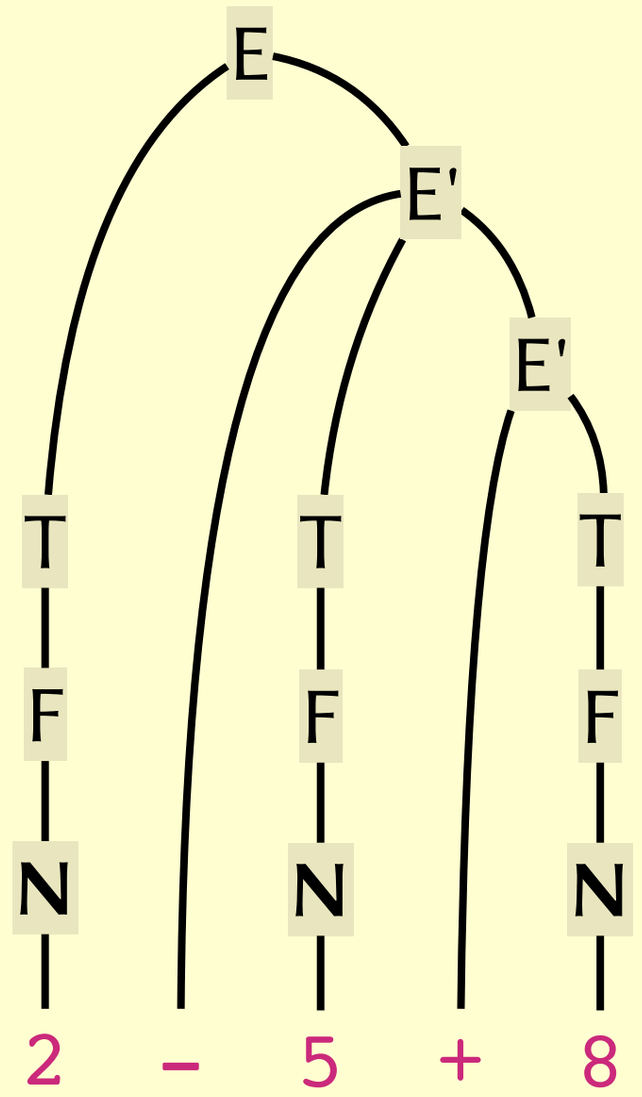
E ::= T E'

E' ::= + T E' | − T E' | ε

T ::= F T'

T' ::= * F T' | ε

F ::= (E) | **N**



Recursive descent

- **Second problem.** Simple recursive descent involves **backtracking**.

- **Solution.** Can eliminate this using **lookahead**.

- This gives a **predictive** parser.

FIRST and FOLLOW

- NULLABLE(X) = can X generate the empty word?
- FIRST(X) = which tokens can start words generated by X?
- FOLLOW(X) = which tokens can follow words generated by X?

	NULLABLE	FIRST	FOLLOW
E ::= T E'	✘	(N	$)
E' ::= + T E' | − T E' | ε	✔	+ −	$)
T ::= F T'	✘	(N	+ − $)
T' ::= * F T' | ε	✔	*	+ − $)
F ::= (E) | N	✘	(N	* + − $)

FIRST and FOLLOW

- NULLABLE(X) = can X generate the empty word?
- FIRST(X) = which tokens can start words generated by X?
- FOLLOW(X) = which tokens can follow words generated by X?

	NULLABLE	FIRST	FOLLOW
E ::= T E'	✘	(N	$)
E' ::= + T E' | − T E' | ε	✔	+ −	$)
T ::= F T'	✘	(N	+ − $)
T' ::= * F T' | ε	✔	*	+ − $)
F ::= (E) | N	✘	(N	* + − $)

	NULLABLE	FIRST	FOLLOW
E ::= T E'	✘	(N	$)
E' ::= + T E' | − T E' | ε	✔	+ −	$)
T ::= F T'	✘	(N	+ − $)
T' ::= * F T' | ε	✔	*	+ − $)
F ::= (E) | N	✘	(N	* + − $)

	*	+	−	(	)	N	$
E				E → T E'		E → T E'	
E'		E' → + T E'	E' → − T E'		E' → ε		E' → ε
T				T → F T'		T → F T'	
T'	T' → * F T'	T' → ε	T' → ε		T' → ε		T' → ε
F				F → (E)		F → N	

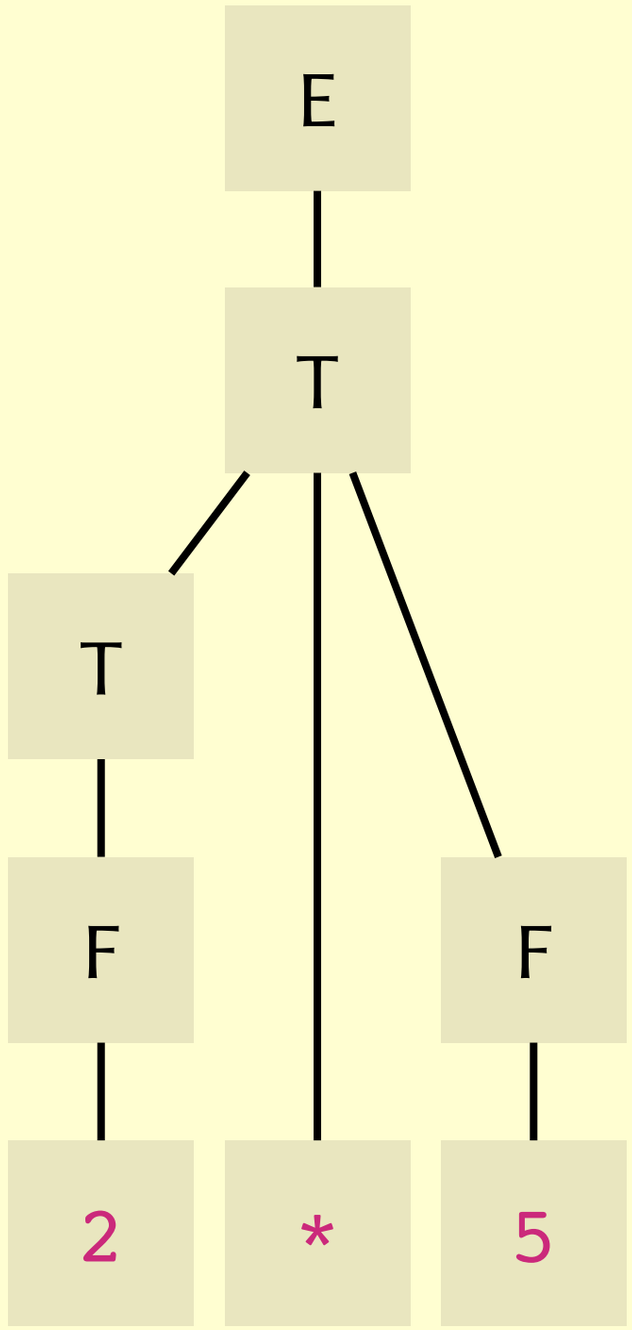
	NULLABLE	FIRST	FOLLOW
E ::= T E'	✘	(N	$)
E' ::= + T E' | – T E' | ε	✔	+ –	$)
T ::= F T'	✘	(N	+ – $)
T' ::= * F T' | ε	✔	*	+ – $)
F ::= (E) | N	✘	(N	* + – $)

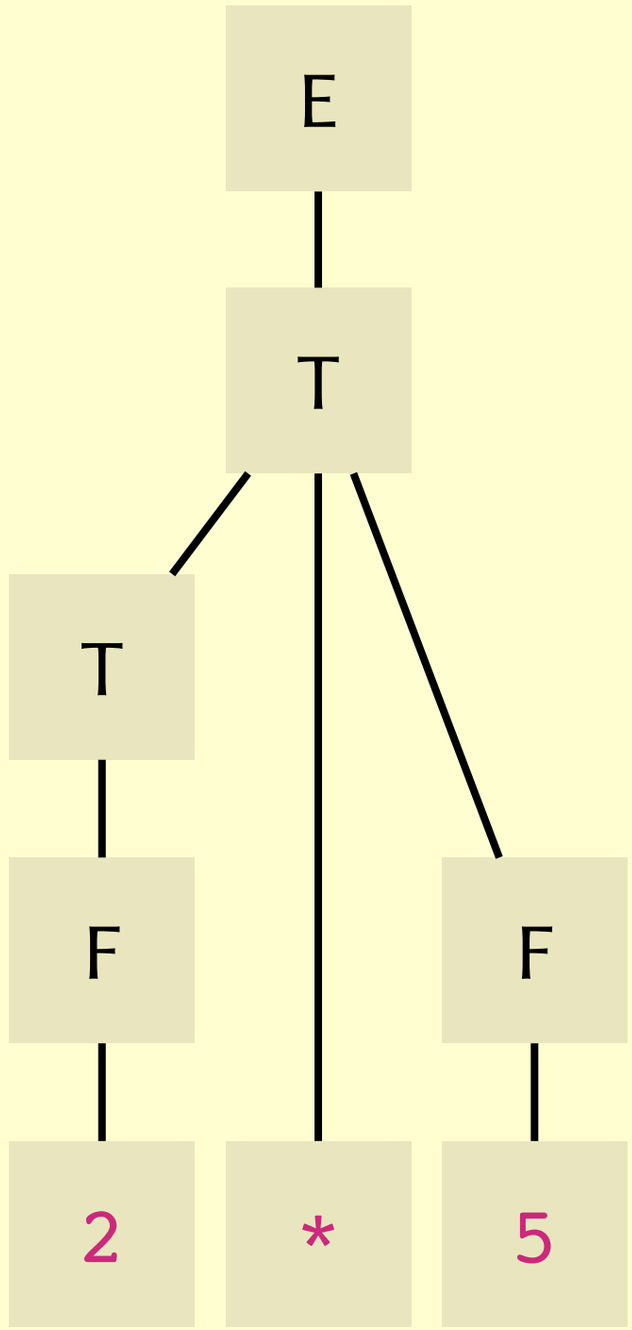
	*	+	–	(	)	N	$
E	✘	✘	✘	E → T E'	✘	E → T E'	✘
E'	✘	E' → + T E'	E' → – T E'	✘	E' → ε	✘	E' → ε
T	✘	✘	✘	T → F T'	✘	T → F T'	✘
T'	T' → * F T'	T' → ε	T' → ε	✘	T' → ε	✘	T' → ε
F	✘	✘	✘	F → (E)	✘	F → N	✘

Can we do better?

- Recursive descent and predictive parsers are **top-down**.
- Top down parsers can't handle left-recursive grammars.
- Can rewrite, but then grammar becomes hard to read!
- Alternative: **bottom-up** parsing.

Top-down vs. Bottom-up





Shift/reduce parsing

E ::= E + T | E − T | T

T ::= T * F | F

F ::= (E) | **N**

Shift/reduce parsing

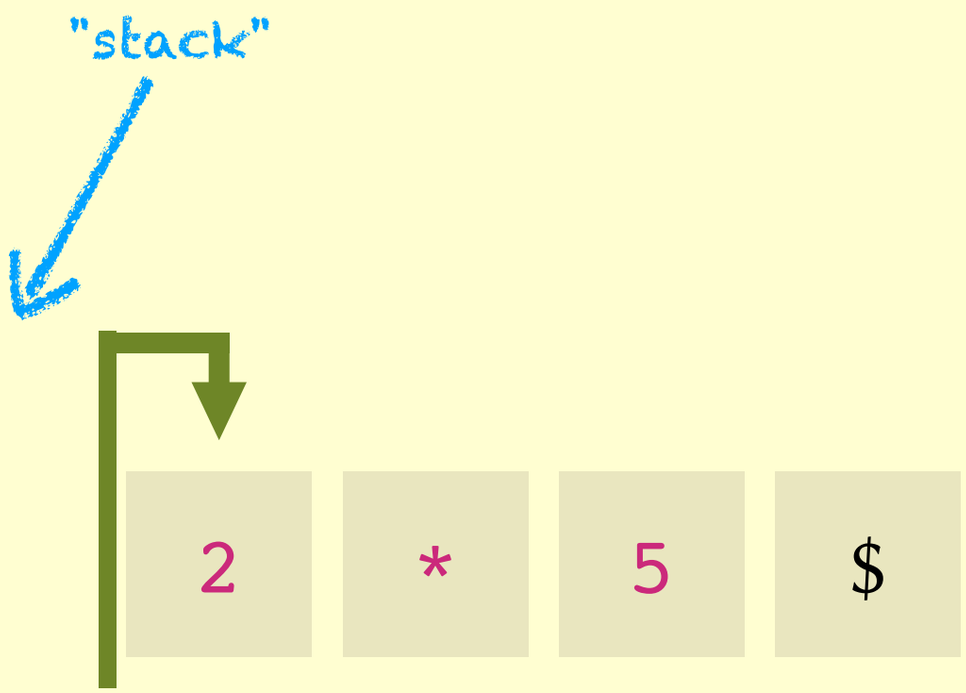
E ::= E + T | T

T ::= T * F | F

F ::= (E) | **N**

ACTION LOG

shift



Shift/reduce parsing

E ::= E + T | T

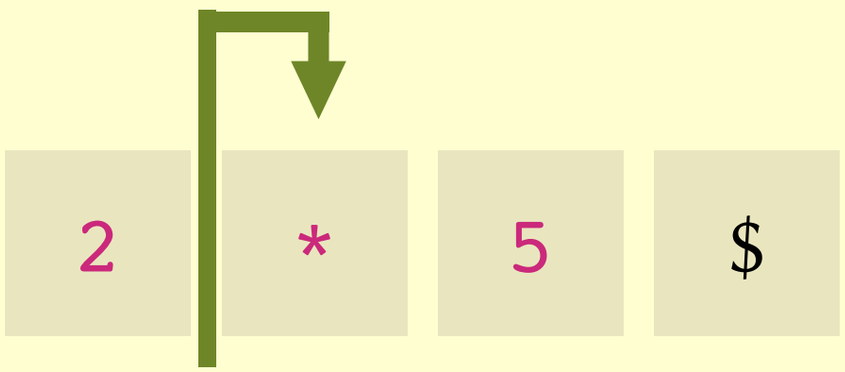
T ::= T * F | F

F ::= (E) | **N**

ACTION LOG

shift

reduce (F ⟶ **N**)



Shift/reduce parsing

E ::= E + T | T

T ::= T * F | F

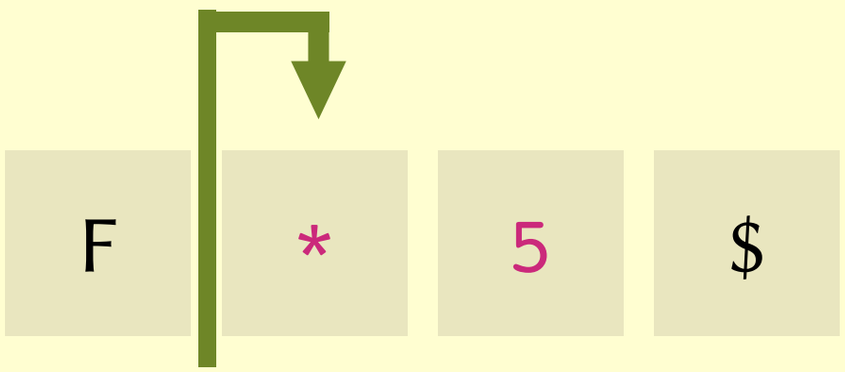
F ::= (E) | **N**

ACTION LOG

shift

reduce (F ⟶ **N**)

reduce (T ⟶ F)



Shift/reduce parsing

E ::= E + T | T

T ::= T * F | F

F ::= (E) | **N**

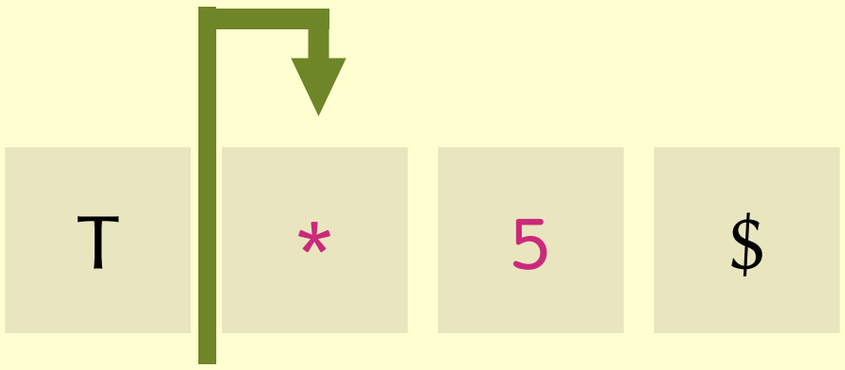
ACTION LOG

shift

reduce (F ⟶ **N**)

reduce (T ⟶ F)

shift



Shift/reduce parsing

E ::= E + T | T

T ::= T * F | F

F ::= (E) | **N**

ACTION LOG

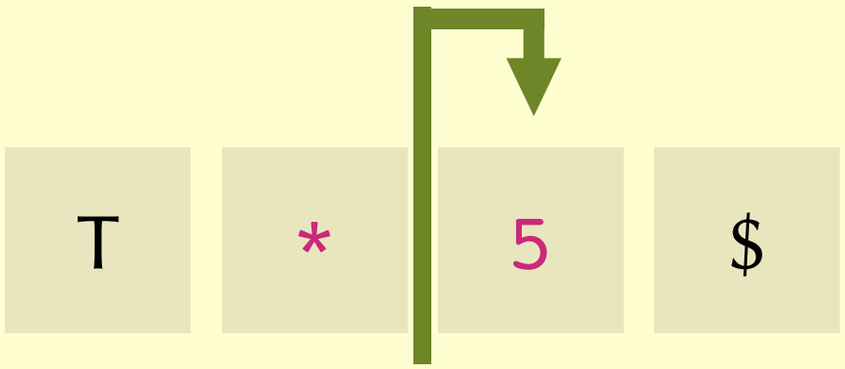
shift

reduce (F ⟶ **N**)

reduce (T ⟶ F)

shift

shift



Shift/reduce parsing

E ::= E + T | T

T ::= T * F | F

F ::= (E) | **N**

ACTION LOG

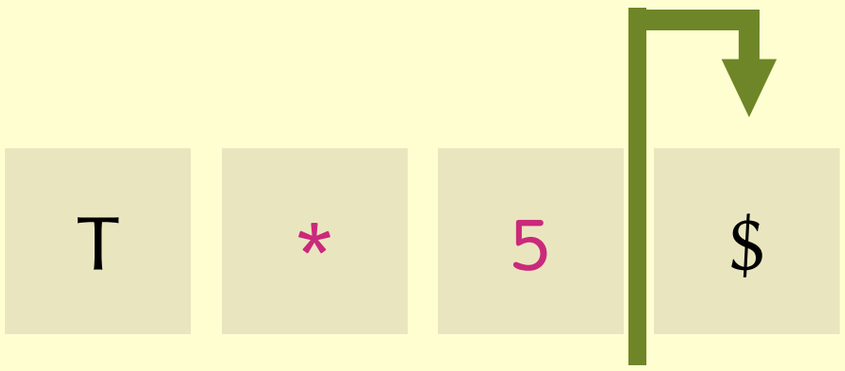
shift

reduce (F ⟶ **N**)

reduce (T ⟶ F)

shift

reduce (F ⟶ **N**)



Shift/reduce parsing

E ::= E + T | T

T ::= T * F | F

F ::= (E) | **N**

ACTION LOG

shift

reduce (F ⟶ **N**)

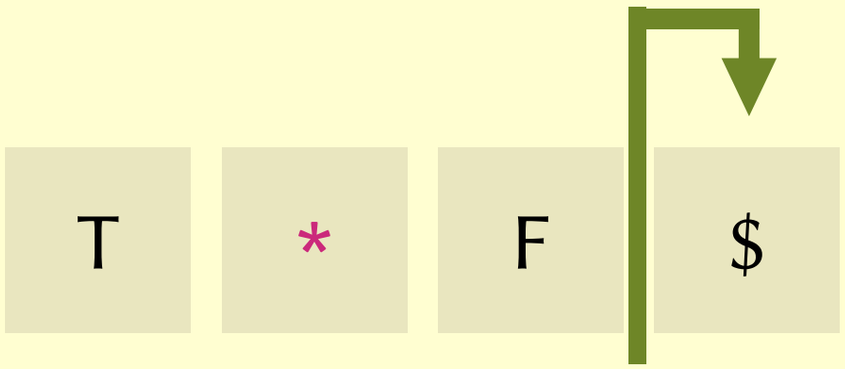
reduce (T ⟶ F)

shift

shift

reduce (F ⟶ **N**)

reduce (T ⟶ T * F)



Shift/reduce parsing

E ::= E + T | T

T ::= T * F | F

F ::= (E) | **N**

ACTION LOG

shift

reduce (F ⟶ **N**)

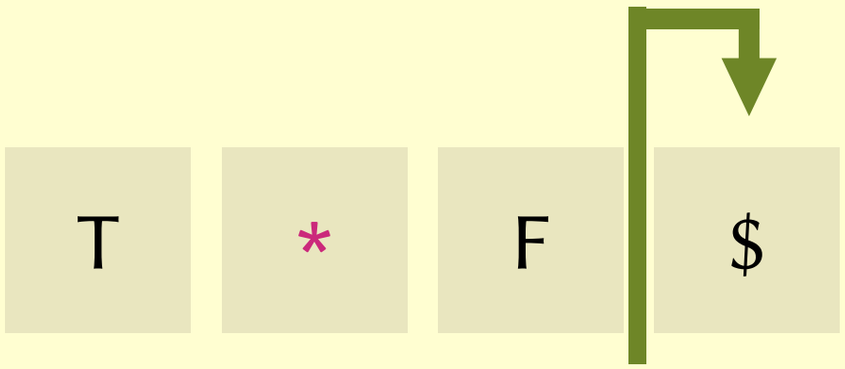
reduce (T ⟶ F)

shift

shift

reduce (F ⟶ **N**)

reduce (T ⟶ T * F)



Shift/reduce parsing

E ::= E + T | T

T ::= T * F | F

F ::= (E) | **N**

ACTION LOG

shift

reduce (F ⟶ **N**)

reduce (T ⟶ F)

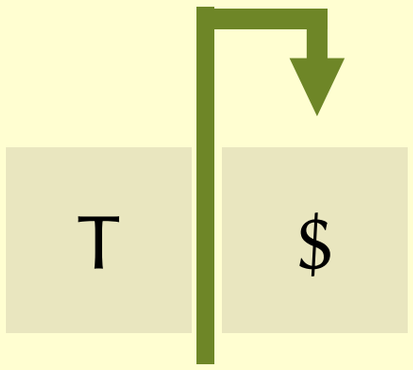
shift

shift

reduce (F ⟶ **N**)

reduce (T ⟶ T * F)

reduce (E ⟶ T)



Shift/reduce parsing

E ::= E + T | T

T ::= T * F | F

F ::= (E) | **N**

ACTION LOG

shift

reduce (F ⟶ **N**)

reduce (T ⟶ F)

shift

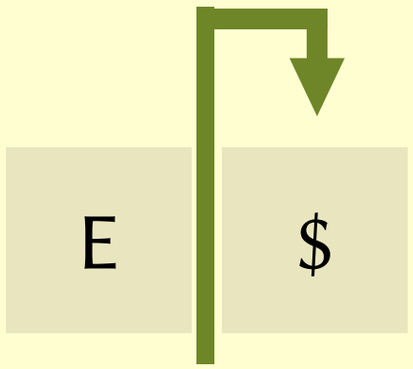
shift

reduce (F ⟶ **N**)

reduce (T ⟶ T * F)

reduce (E ⟶ T)

accept

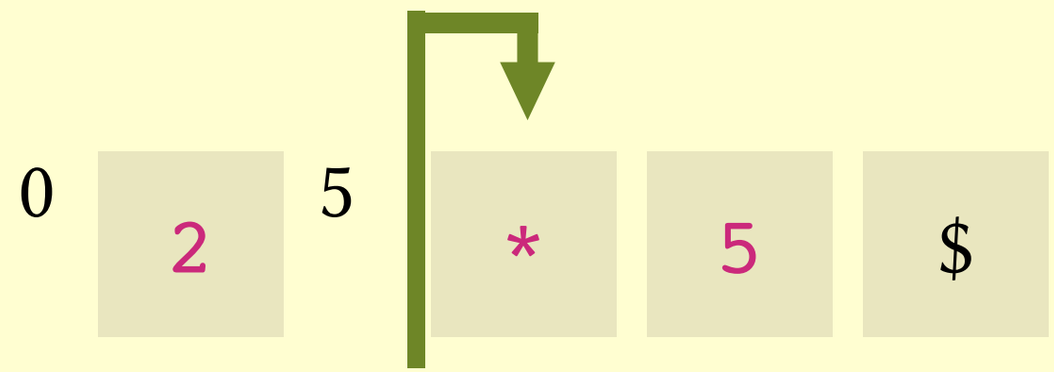


Shift/reduce parsing

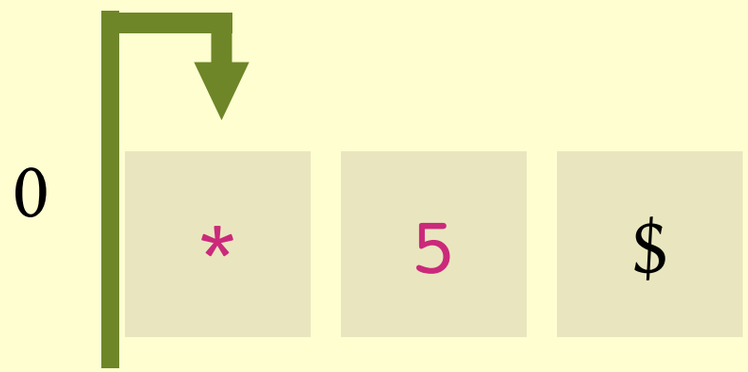
- **Problem.** How do we know whether to shift or reduce?
- **Solution.** Consult the **parsing table**.

STATE	ACTION						GOTO		
	*	+	(	)	N	$	E	T	F
0			s4		s5		1	2	3
1		s6				acc			
2	s7	r(E → T)		r(E → T)		r(E → T)			
3	r(T → F)	r(T → F)		r(T → F)		r(T → F)			
4			s4		s5		8	2	3
5	r(F → **N**)	r(F → **N**)		r(F → **N**)		r(F → **N**)			
6			s4		s5			9	3
7			s4		s5				10
8		s6		s11					
9	s7	r(E → E+T)		r(E → E+T)		r(E → E+T)			
10	r(T → T*F)	r(T → T*F)		r(T → T*F)		r(T → T*F)			
11	r(F → (E))	r(F → (E))		r(F → (E))		r(F → (E))			

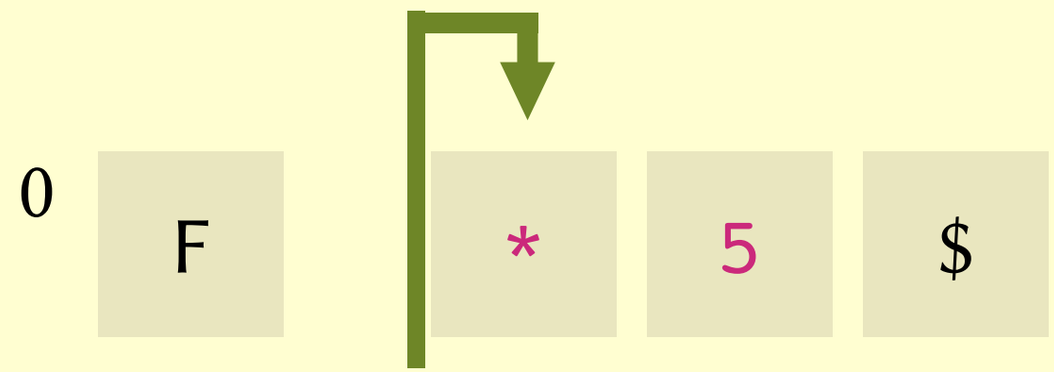
STATE	ACTION						GOTO		
	*	+	(	)	N	$	E	T	F
0			s4		s5		1	2	3
1		s6				acc			
2	s7	r(E → T)		r(E → T)		r(E → T)			
3	r(T → F)	r(T → F)		r(T → F)		r(T → F)			
4			s4		s5		8	2	3
5	r(F → **N**)	r(F → **N**)		r(F → **N**)		r(F → **N**)			
6			s4		s5			9	3
7			s4		s5				10
8		s6		s11					
9	s7	r(E → E+T)		r(E → E+T)		r(E → E+T)			
10	r(T → T*F)	r(T → T*F)		r(T → T*F)		r(T → T*F)			
11	r(F → (E))	r(F → (E))		r(F → (E))		r(F → (E))			



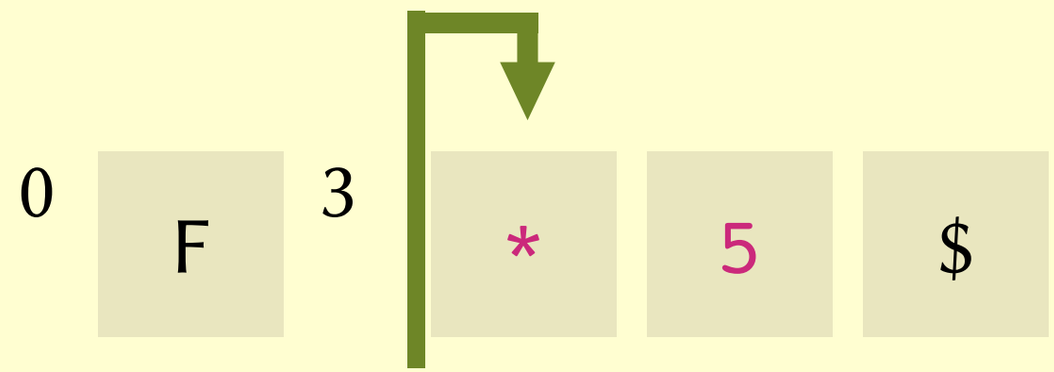
STATE	ACTION						GOTO		
	*	+	(	)	N	$	E	T	F
0			s4		s5		1	2	3
1		s6				acc			
2	s7	r(E → T)		r(E → T)		r(E → T)			
3	r(T → F)	r(T → F)		r(T → F)		r(T → F)			
4			s4		s5		8	2	3
5	r(F → **N**)	r(F → **N**)		r(F → **N**)		r(F → **N**)			
6			s4		s5			9	3
7			s4		s5				10
8		s6		s11					
9	s7	r(E → E+T)		r(E → E+T)		r(E → E+T)			
10	r(T → T*F)	r(T → T*F)		r(T → T*F)		r(T → T*F)			
11	r(F → (E))	r(F → (E))		r(F → (E))		r(F → (E))			



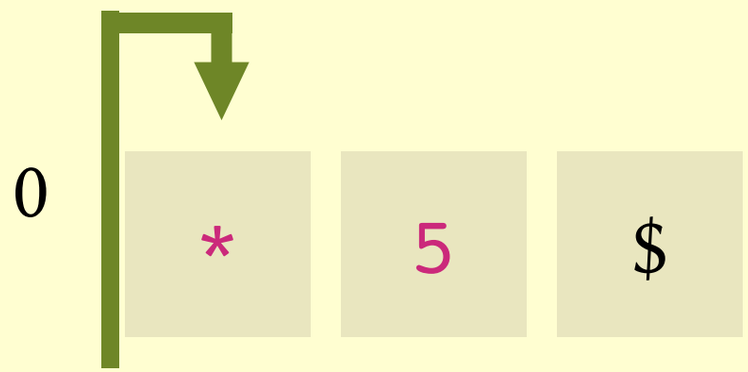
STATE	ACTION						GOTO		
	*	+	(	)	N	$	E	T	F
0			s4		s5		1	2	3
1		s6				acc			
2	s7	r(E → T)		r(E → T)		r(E → T)			
3	r(T → F)	r(T → F)		r(T → F)		r(T → F)			
4			s4		s5		8	2	3
5	r(F → **N**)	r(F → **N**)		r(F → **N**)		r(F → **N**)			
6			s4		s5			9	3
7			s4		s5				10
8		s6		s11					
9	s7	r(E → E+T)		r(E → E+T)		r(E → E+T)			
10	r(T → T*F)	r(T → T*F)		r(T → T*F)		r(T → T*F)			
11	r(F → (E))	r(F → (E))		r(F → (E))		r(F → (E))			



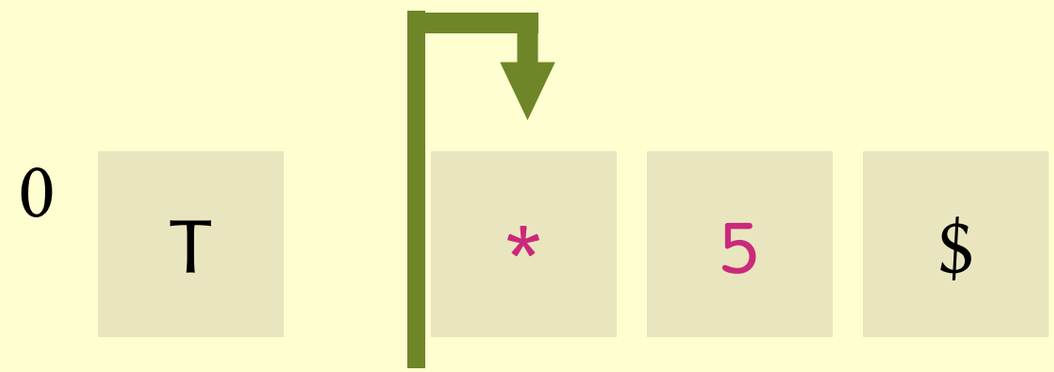
STATE	ACTION						GOTO		
	*	+	(	)	N	$	E	T	F
0			s4		s5		1	2	3
1		s6				acc			
2	s7	r(E → T)		r(E → T)		r(E → T)			
3	r(T → F)	r(T → F)		r(T → F)		r(T → F)			
4			s4		s5		8	2	3
5	r(F → **N**)	r(F → **N**)		r(F → **N**)		r(F → **N**)			
6			s4		s5			9	3
7			s4		s5				10
8		s6		s11					
9	s7	r(E → E+T)		r(E → E+T)		r(E → E+T)			
10	r(T → T*F)	r(T → T*F)		r(T → T*F)		r(T → T*F)			
11	r(F → (E))	r(F → (E))		r(F → (E))		r(F → (E))			



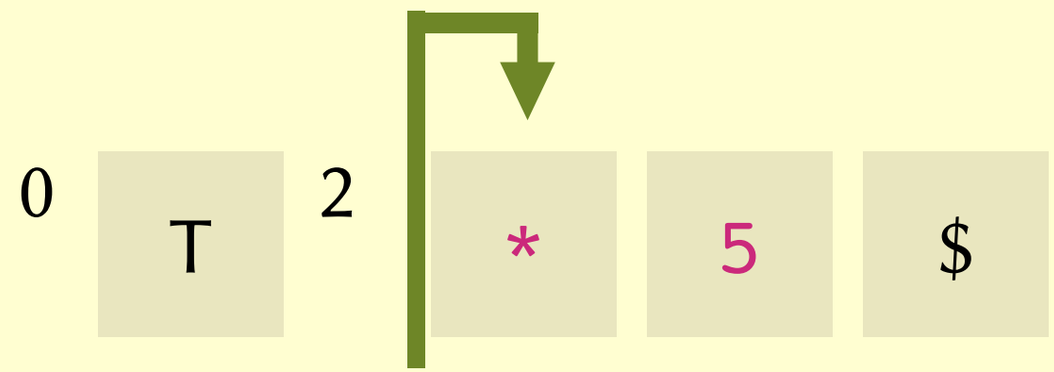
STATE	ACTION						GOTO		
	*	+	(	)	N	$	E	T	F
0			s4		s5		1	2	3
1		s6				acc			
2	s7	r(E → T)		r(E → T)		r(E → T)			
3	r(T → F)	r(T → F)		r(T → F)		r(T → F)			
4			s4		s5		8	2	3
5	r(F → **N**)	r(F → **N**)		r(F → **N**)		r(F → **N**)			
6			s4		s5			9	3
7			s4		s5				10
8		s6		s11					
9	s7	r(E → E+T)		r(E → E+T)		r(E → E+T)			
10	r(T → T*F)	r(T → T*F)		r(T → T*F)		r(T → T*F)			
11	r(F → (E))	r(F → (E))		r(F → (E))		r(F → (E))			



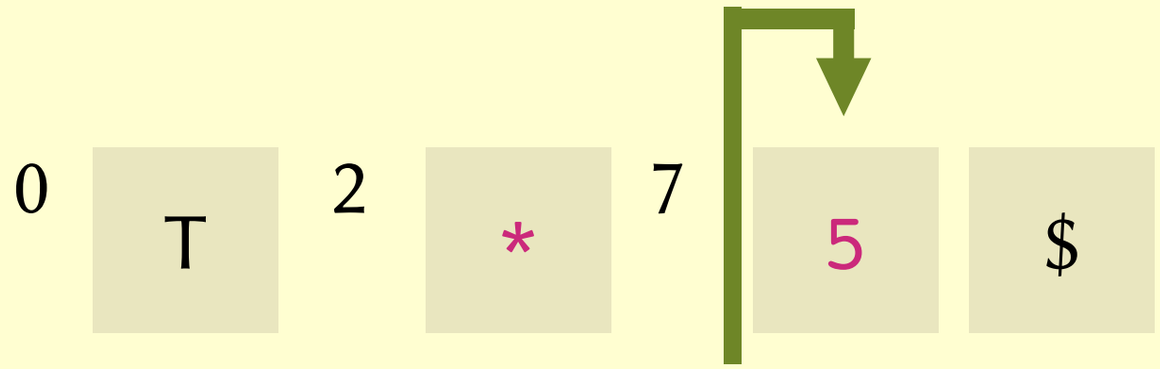
STATE	ACTION						GOTO		
	*	+	(	)	N	$	E	T	F
0			s4		s5		1	2	3
1		s6				acc			
2	s7	r(E → T)		r(E → T)		r(E → T)			
3	r(T → F)	r(T → F)		r(T → F)		r(T → F)			
4			s4		s5		8	2	3
5	r(F → **N**)	r(F → **N**)		r(F → **N**)		r(F → **N**)			
6			s4		s5			9	3
7			s4		s5				10
8		s6		s11					
9	s7	r(E → E+T)		r(E → E+T)		r(E → E+T)			
10	r(T → T*F)	r(T → T*F)		r(T → T*F)		r(T → T*F)			
11	r(F → (E))	r(F → (E))		r(F → (E))		r(F → (E))			



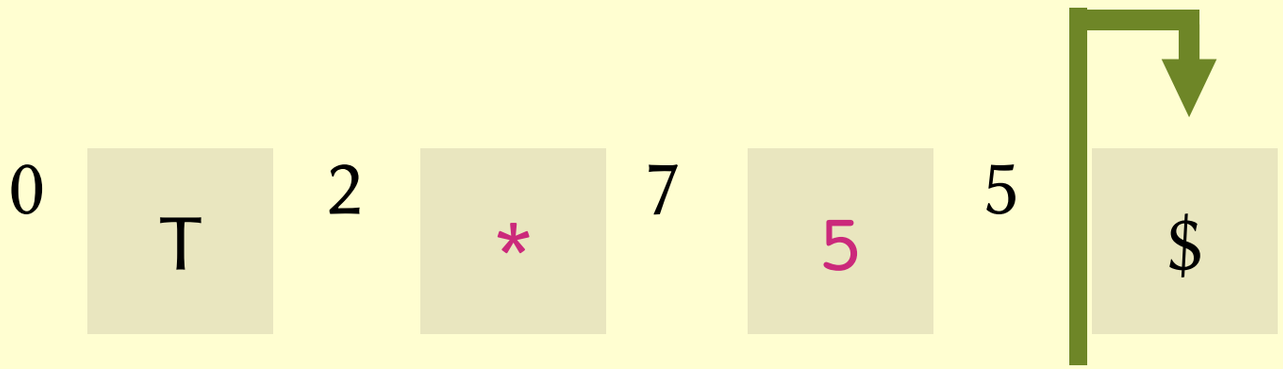
STATE	ACTION						GOTO		
	*	+	(	)	N	$	E	T	F
0			s4		s5		1	2	3
1		s6				acc			
2	s7	r(E → T)		r(E → T)		r(E → T)			
3	r(T → F)	r(T → F)		r(T → F)		r(T → F)			
4			s4		s5		8	2	3
5	r(F → **N**)	r(F → **N**)		r(F → **N**)		r(F → **N**)			
6			s4		s5			9	3
7			s4		s5				10
8		s6		s11					
9	s7	r(E → E+T)		r(E → E+T)		r(E → E+T)			
10	r(T → T*F)	r(T → T*F)		r(T → T*F)		r(T → T*F)			
11	r(F → (E))	r(F → (E))		r(F → (E))		r(F → (E))			



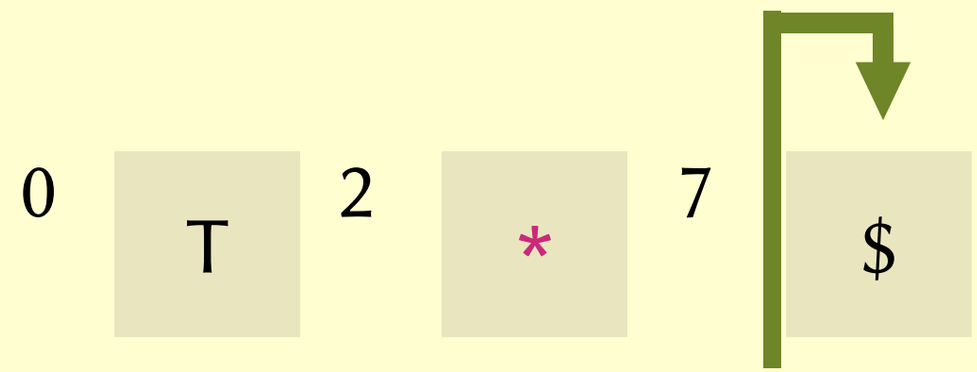
STATE	ACTION						GOTO		
	*	+	(	)	N	$	E	T	F
0			s4		s5		1	2	3
1		s6				acc			
2	s7	r(E → T)		r(E → T)		r(E → T)			
3	r(T → F)	r(T → F)		r(T → F)		r(T → F)			
4			s4		s5		8	2	3
5	r(F → **N**)	r(F → **N**)		r(F → **N**)		r(F → **N**)			
6			s4		s5			9	3
7			s4		s5				10
8		s6		s11					
9	s7	r(E → E+T)		r(E → E+T)		r(E → E+T)			
10	r(T → T*F)	r(T → T*F)		r(T → T*F)		r(T → T*F)			
11	r(F → (E))	r(F → (E))		r(F → (E))		r(F → (E))			



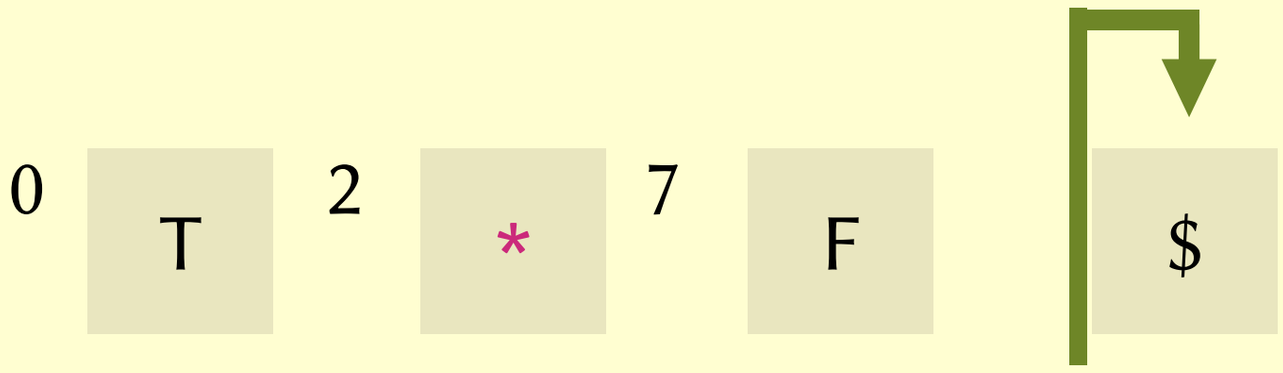
STATE	ACTION						GOTO		
	*	+	(	)	N	$	E	T	F
0			s4		s5		1	2	3
1		s6				acc			
2	s7	r(E → T)		r(E → T)		r(E → T)			
3	r(T → F)	r(T → F)		r(T → F)		r(T → F)			
4			s4		s5		8	2	3
5	r(F → **N**)	r(F → **N**)		r(F → **N**)		r(F → **N**)			
6			s4		s5			9	3
7			s4		s5				10
8		s6		s11					
9	s7	r(E → E+T)		r(E → E+T)		r(E → E+T)			
10	r(T → T*F)	r(T → T*F)		r(T → T*F)		r(T → T*F)			
11	r(F → (E))	r(F → (E))		r(F → (E))		r(F → (E))			



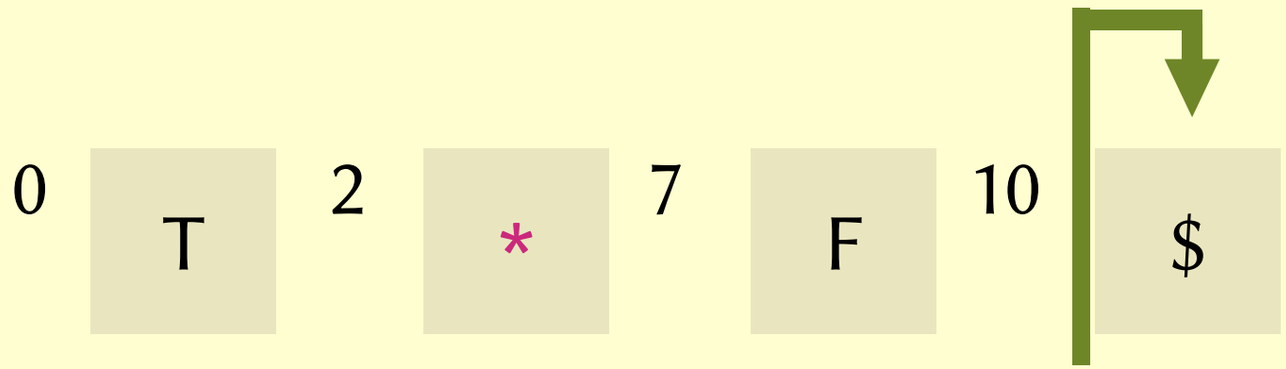
STATE	ACTION						GOTO		
	*	+	(	)	N	$	E	T	F
0			s4		s5		1	2	3
1		s6				acc			
2	s7	r(E → T)		r(E → T)		r(E → T)			
3	r(T → F)	r(T → F)		r(T → F)		r(T → F)			
4			s4		s5		8	2	3
5	r(F → **N**)	r(F → **N**)		r(F → **N**)		r(F → **N**)			
6			s4		s5			9	3
7			s4		s5				10
8		s6		s11					
9	s7	r(E → E+T)		r(E → E+T)		r(E → E+T)			
10	r(T → T*F)	r(T → T*F)		r(T → T*F)		r(T → T*F)			
11	r(F → (E))	r(F → (E))		r(F → (E))		r(F → (E))			



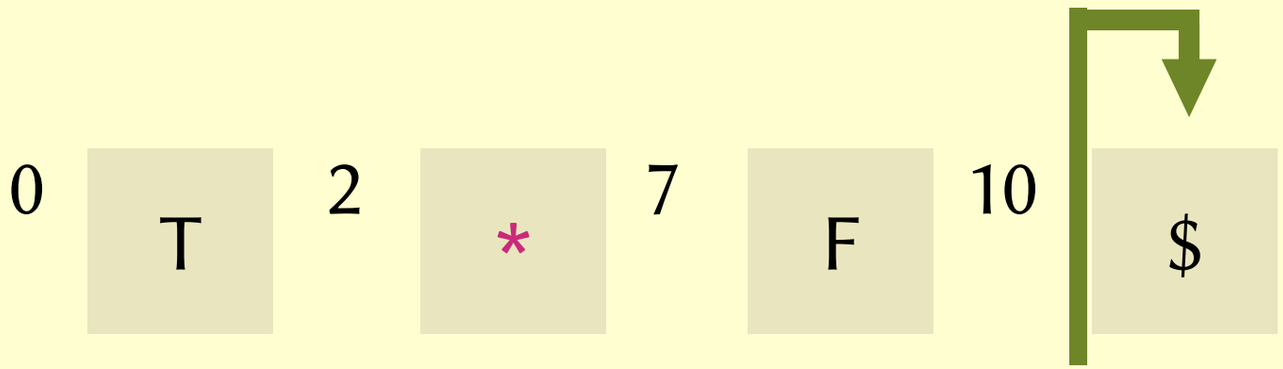
STATE	ACTION						GOTO		
	*	+	(	)	N	$	E	T	F
0			s4		s5		1	2	3
1		s6				acc			
2	s7	r(E → T)		r(E → T)		r(E → T)			
3	r(T → F)	r(T → F)		r(T → F)		r(T → F)			
4			s4		s5		8	2	3
5	r(F → **N**)	r(F → **N**)		r(F → **N**)		r(F → **N**)			
6			s4		s5			9	3
7			s4		s5				10
8		s6		s11					
9	s7	r(E → E+T)		r(E → E+T)		r(E → E+T)			
10	r(T → T*F)	r(T → T*F)		r(T → T*F)		r(T → T*F)			
11	r(F → (E))	r(F → (E))		r(F → (E))		r(F → (E))			



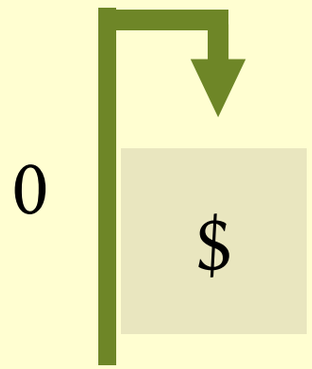
STATE	ACTION						GOTO		
	*	+	(	)	N	$	E	T	F
0			s4		s5		1	2	3
1		s6				acc			
2	s7	r(E → T)		r(E → T)		r(E → T)			
3	r(T → F)	r(T → F)		r(T → F)		r(T → F)			
4			s4		s5		8	2	3
5	r(F → **N**)	r(F → **N**)		r(F → **N**)		r(F → **N**)			
6			s4		s5			9	3
7			s4		s5				10
8		s6		s11					
9	s7	r(E → E+T)		r(E → E+T)		r(E → E+T)			
10	r(T → T*F)	r(T → T*F)		r(T → T*F)		r(T → T*F)			
11	r(F → (E))	r(F → (E))		r(F → (E))		r(F → (E))			



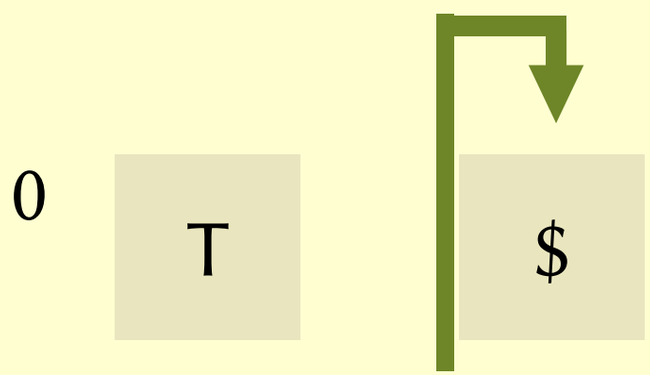
STATE	ACTION						GOTO		
	*	+	(	)	N	$	E	T	F
0			s4		s5		1	2	3
1		s6				acc			
2	s7	r(E → T)		r(E → T)		r(E → T)			
3	r(T → F)	r(T → F)		r(T → F)		r(T → F)			
4			s4		s5		8	2	3
5	r(F → **N**)	r(F → **N**)		r(F → **N**)		r(F → **N**)			
6			s4		s5			9	3
7			s4		s5				10
8		s6		s11					
9	s7	r(E → E+T)		r(E → E+T)		r(E → E+T)			
10	r(T → T*F)	r(T → T*F)		r(T → T*F)		r(T → T*F)			
11	r(F → (E))	r(F → (E))		r(F → (E))		r(F → (E))			



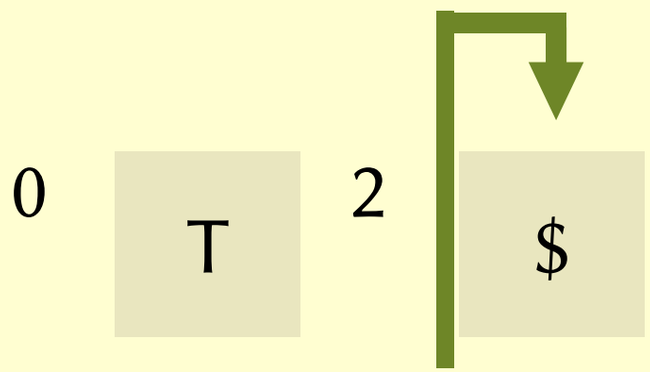
STATE	ACTION						GOTO		
	*	+	(	)	N	$	E	T	F
0			s4		s5		1	2	3
1		s6				acc			
2	s7	r(E → T)		r(E → T)		r(E → T)			
3	r(T → F)	r(T → F)		r(T → F)		r(T → F)			
4			s4		s5		8	2	3
5	r(F → **N**)	r(F → **N**)		r(F → **N**)		r(F → **N**)			
6			s4		s5			9	3
7			s4		s5				10
8		s6		s11					
9	s7	r(E → E+T)		r(E → E+T)		r(E → E+T)			
10	r(T → T*F)	r(T → T*F)		r(T → T*F)		r(T → T*F)			
11	r(F → (E))	r(F → (E))		r(F → (E))		r(F → (E))			



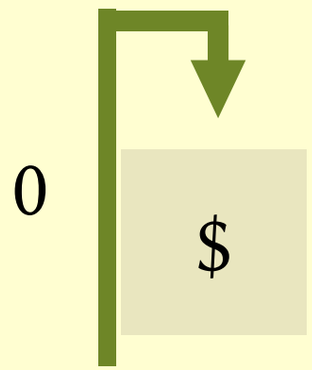
STATE	ACTION						GOTO		
	*	+	(	)	N	$	E	T	F
0			s4		s5		1	2	3
1		s6				acc			
2	s7	r(E → T)		r(E → T)		r(E → T)			
3	r(T → F)	r(T → F)		r(T → F)		r(T → F)			
4			s4		s5		8	2	3
5	r(F → **N**)	r(F → **N**)		r(F → **N**)		r(F → **N**)			
6			s4		s5			9	3
7			s4		s5				10
8		s6		s11					
9	s7	r(E → E+T)		r(E → E+T)		r(E → E+T)			
10	r(T → T*F)	r(T → T*F)		r(T → T*F)		r(T → T*F)			
11	r(F → (E))	r(F → (E))		r(F → (E))		r(F → (E))			



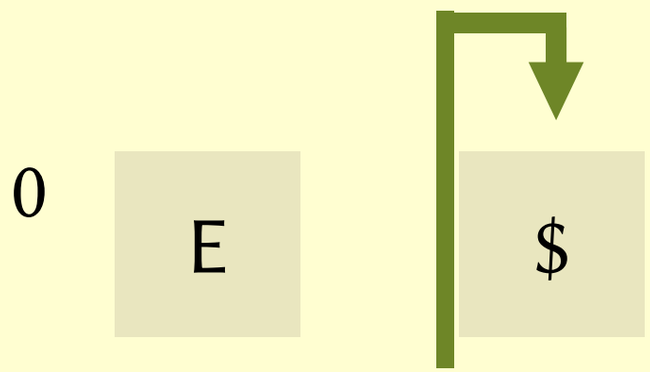
STATE	ACTION						GOTO		
	*	+	(	)	N	$	E	T	F
0			s4		s5		1	2	3
1		s6				acc			
2	s7	r(E → T)		r(E → T)		r(E → T)			
3	r(T → F)	r(T → F)		r(T → F)		r(T → F)			
4			s4		s5		8	2	3
5	r(F → **N**)	r(F → **N**)		r(F → **N**)		r(F → **N**)			
6			s4		s5			9	3
7			s4		s5				10
8		s6		s11					
9	s7	r(E → E+T)		r(E → E+T)		r(E → E+T)			
10	r(T → T*F)	r(T → T*F)		r(T → T*F)		r(T → T*F)			
11	r(F → (E))	r(F → (E))		r(F → (E))		r(F → (E))			



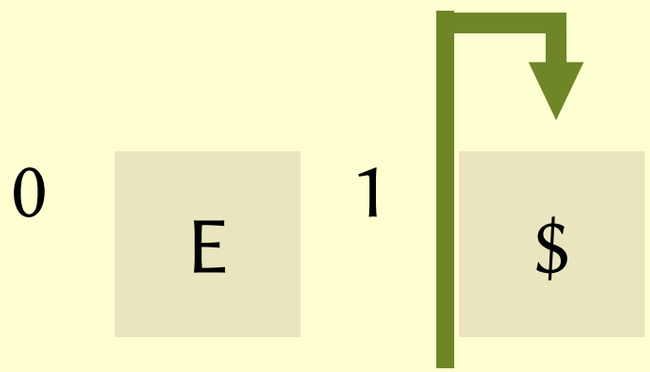
STATE	ACTION						GOTO		
	*	+	(	)	N	$	E	T	F
0			s4		s5		1	2	3
1		s6				acc			
2	s7	r(E → T)		r(E → T)		r(E → T)			
3	r(T → F)	r(T → F)		r(T → F)		r(T → F)			
4			s4		s5		8	2	3
5	r(F → **N**)	r(F → **N**)		r(F → **N**)		r(F → **N**)			
6			s4		s5			9	3
7			s4		s5				10
8		s6		s11					
9	s7	r(E → E+T)		r(E → E+T)		r(E → E+T)			
10	r(T → T*F)	r(T → T*F)		r(T → T*F)		r(T → T*F)			
11	r(F → (E))	r(F → (E))		r(F → (E))		r(F → (E))			



STATE	ACTION						GOTO		
	*	+	(	)	N	$	E	T	F
0			s4		s5		1	2	3
1		s6				acc			
2	s7	r(E → T)		r(E → T)		r(E → T)			
3	r(T → F)	r(T → F)		r(T → F)		r(T → F)			
4			s4		s5		8	2	3
5	r(F → **N**)	r(F → **N**)		r(F → **N**)		r(F → **N**)			
6			s4		s5			9	3
7			s4		s5				10
8		s6		s11					
9	s7	r(E → E+T)		r(E → E+T)		r(E → E+T)			
10	r(T → T*F)	r(T → T*F)		r(T → T*F)		r(T → T*F)			
11	r(F → (E))	r(F → (E))		r(F → (E))		r(F → (E))			

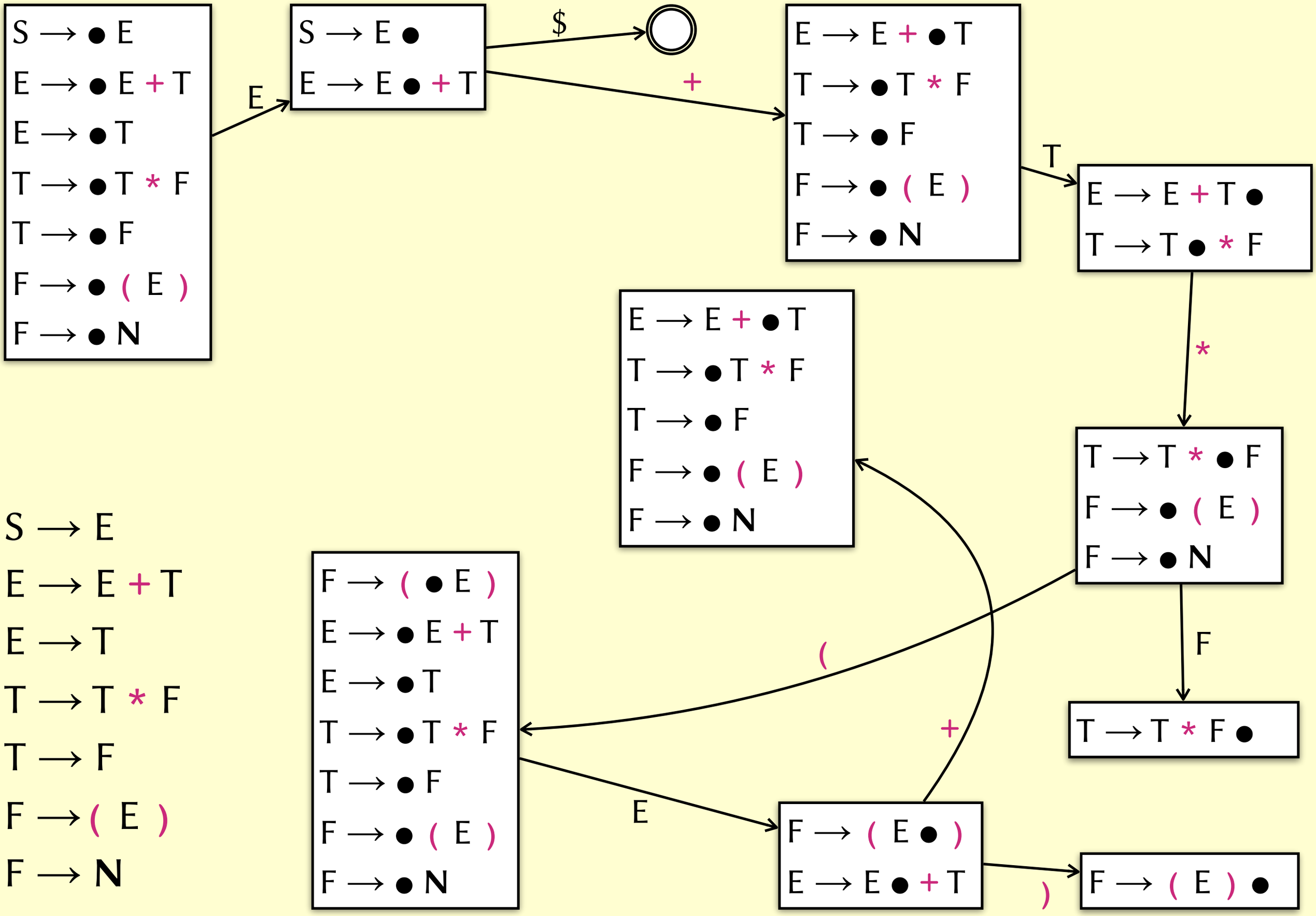


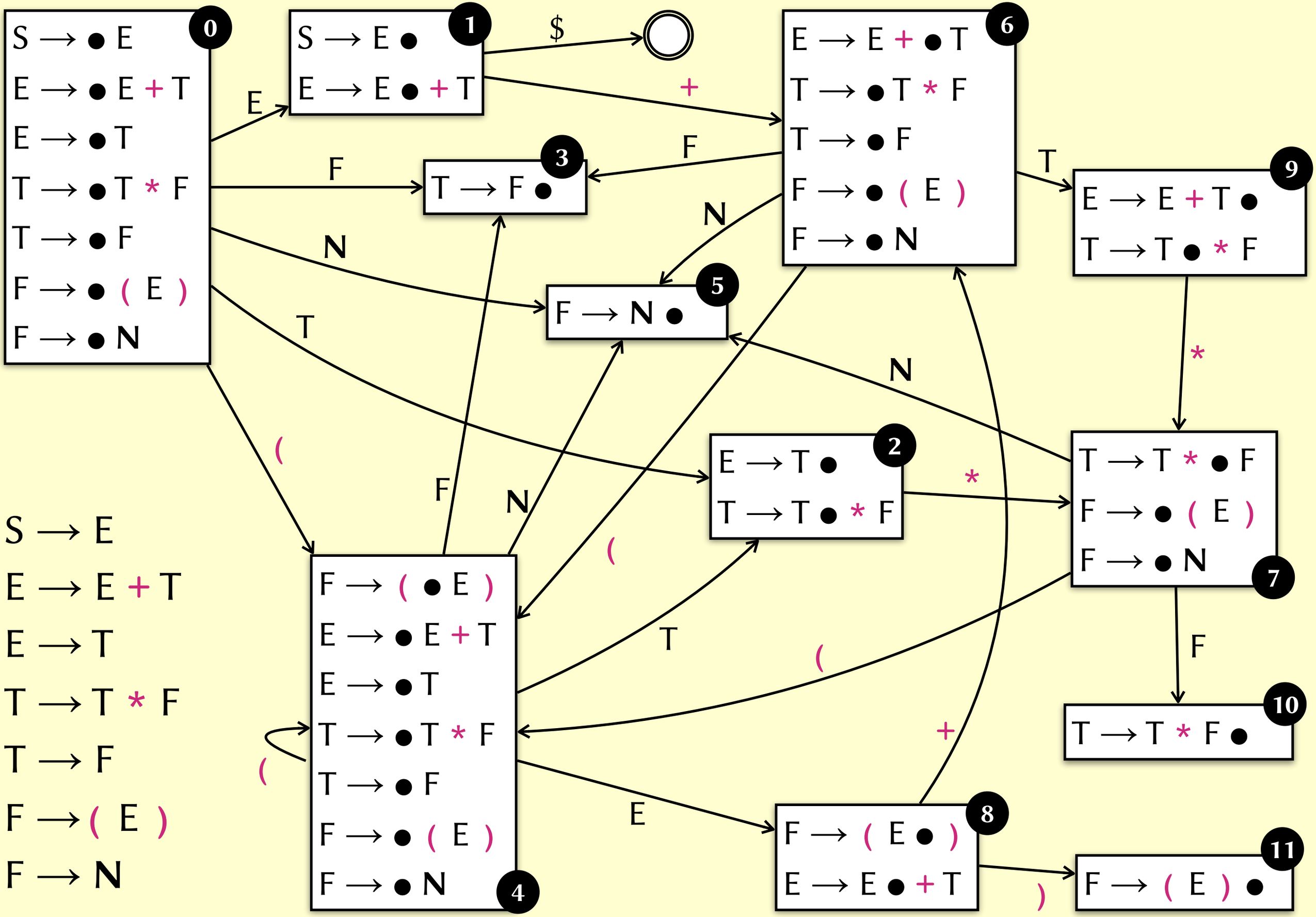
STATE	ACTION						GOTO		
	*	+	(	)	N	$	E	T	F
0			s4		s5		1	2	3
1		s6				acc			
2	s7	r(E → T)		r(E → T)		r(E → T)			
3	r(T → F)	r(T → F)		r(T → F)		r(T → F)			
4			s4		s5		8	2	3
5	r(F → **N**)	r(F → **N**)		r(F → **N**)		r(F → **N**)			
6			s4		s5			9	3
7			s4		s5				10
8		s6		s11					
9	s7	r(E → E+T)		r(E → E+T)		r(E → E+T)			
10	r(T → T*F)	r(T → T*F)		r(T → T*F)		r(T → T*F)			
11	r(F → (E))	r(F → (E))		r(F → (E))		r(F → (E))			



Shift/reduce parsing

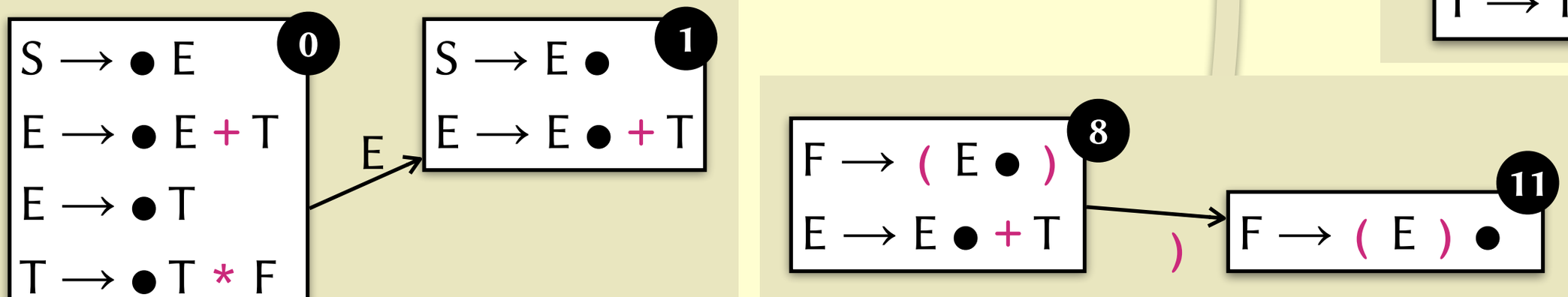
- **Problem.** How do we know whether to shift or reduce?
- **Solution.** Consult the **parsing table**.
- **Problem.** But how did we build the parsing table?
- **Solution.** Like this...


S → ● E
E → ● E + T
E → ● T
T → ● T * F
T → ● F
F → ● (E)
F → ● N
E
S → E ●
E → E ● + T
$
+
E → E + ● T
T → ● T * F
T → ● F
F → ● (E)
F → ● N
T
E → E + T ●
T → T ● * F
*
T → T * ● F
F → ● (E)
F → ● N
F
T → T * F ●
+
F → (● E)
E → ● E + T
E → ● T
T → ● T * F
T → ● F
F → ● (E)
F → ● N
(
E
F → (E ●)
E → E ● + T
)
F → (E) ●
S → E
E → E + T
E → T
T → T * F
T → F
F → (E)
F → N

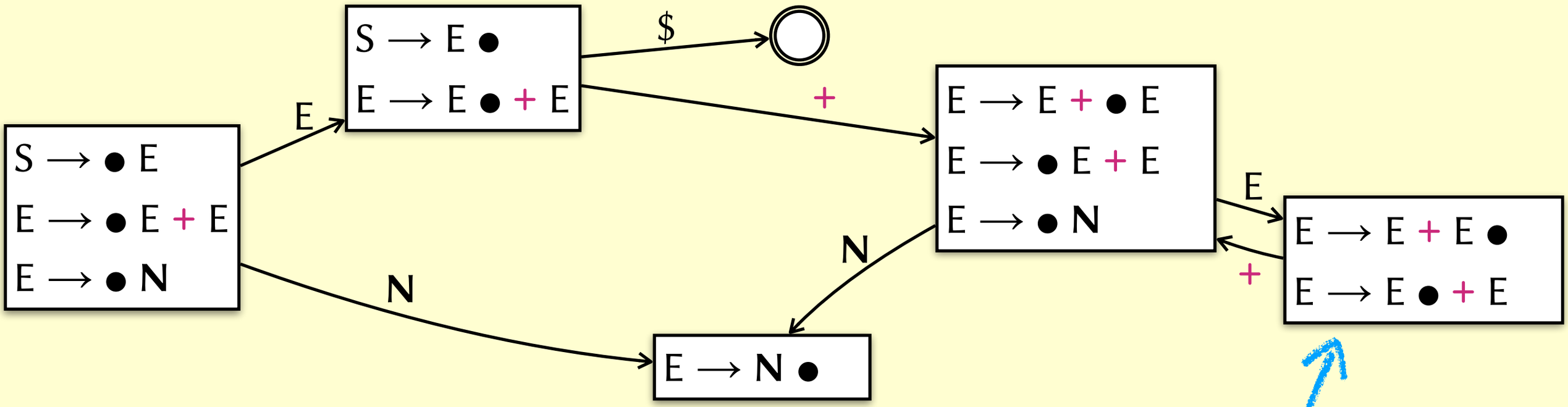


S → E
E → E + T
E → T
T → T * F
T → F
F → (E)
F → N

STATE	ACTION						GOTO		
	*	+	(	)	N	$	E	T	F
0			s4		s5		1	2	3
1		s6				acc			
2	s7	r(E → T)		r(E → T)		r(E → T)			
3	r(T → F)	r(T → F)		r(T → F)		r(T → F)			
4			s4		s5		8	2	3
5	r(F → **N**)	r(F → **N**)		r(F → **N**)		r(F → **N**)			
6			s4		s5			9	3
7			s4		s5				10
8		s6		s11					
9	s7	r(E → E+T)		r(E → E+T)		r(E → E+T)			
10	r(T → T*F)	r(T → T*F)		r(T → T*F)		r(T → T*F)			
11	r(F → (E))	r(F → (E))		r(F → (E))		r(F → (E))			



What can go wrong?



S → E

E → E + E

E → N

What to do when we read a + in this state?

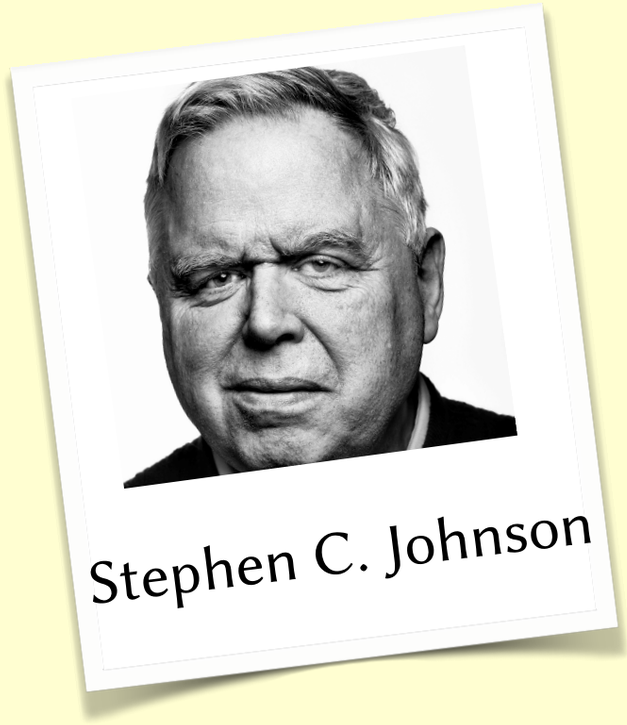
Shift/reduce parsing

- **Problem.** How do we know whether to shift or reduce?
- **Solution.** Consult the **parsing table**.
- **Problem.** But how did we build the parsing table?
- **Solution.** Like this...
- **Problem.** Seems like a lot of work!
- **Solution.** Use a tool to build the parsing table for you!

Yacc

- Yacc (yet another compiler compiler) was written in the 1970s at AT&T Labs, and first published in 1975.

- There is an open-source equivalent called Bison.



Stephen C. Johnson

```
expr   : expr '+' term
       | expr '-' term
       | term
       ;
term   : term '*' factor
       | factor
       ;
factor : '(' expr ')'
       | N
       ;
```

```
%token N

%start line

%%

line   : expr '\n'          { return $1; }
       ;

expr   : expr '+' term      { $$ = $1 + $3; }
       | expr '-' term      { $$ = $1 - $3; }
       | term               { $$ = $1; }
       ;

term   : term '*' factor    { $$ = $1 * $3; }
       | factor             { $$ = $1; }
       ;

factor : '(' expr ')'       { $$ = $2; }
       | N                  { $$ = $1; }
       ;

%%

int main() {...}
```

Summary

- **Recursive descent parsers** use a set of mutually recursive functions.

Summary

- **Recursive descent parsers** use a set of mutually recursive functions. **Predictive parsers** use a **lookahead table** (built with the help of the **FIRST** and **FOLLOW** functions) to avoid **backtracking**.

Summary

- **Recursive descent parsers** use a set of mutually recursive functions. **Predictive parsers** use a **lookahead table** (built with the help of the **FIRST** and **FOLLOW** functions) to avoid **backtracking**. Both build the parse-tree **top-down**.

Summary

- **Recursive descent parsers** use a set of mutually recursive functions. **Predictive parsers** use a **lookahead table** (built with the help of the **FIRST** and **FOLLOW** functions) to avoid **backtracking**. Both build the parse-tree **top-down**. Neither can cope with **left-recursive** grammars.

Summary

- **Recursive descent parsers** use a set of mutually recursive functions. **Predictive parsers** use a **lookahead table** (built with the help of the **FIRST** and **FOLLOW** functions) to avoid **backtracking**. Both build the parse-tree **top-down**. Neither can cope with **left-recursive** grammars.

- **Shift/reduce parsers** build the parse-tree **bottom-up**.

Summary

- **Recursive descent parsers** use a set of mutually recursive functions. **Predictive parsers** use a **lookahead table** (built with the help of the **FIRST** and **FOLLOW** functions) to avoid **backtracking**. Both build the parse-tree **top-down**. Neither can cope with **left-recursive** grammars.

- **Shift/reduce parsers** build the parse-tree **bottom-up**. A state machine instructs the parser whether to shift or reduce each time it receives a token.

Summary

- **Recursive descent parsers** use a set of mutually recursive functions. **Predictive parsers** use a **lookahead table** (built with the help of the **FIRST** and **FOLLOW** functions) to avoid **backtracking**. Both build the parse-tree **top-down**. Neither can cope with **left-recursive** grammars.

- **Shift/reduce parsers** build the parse-tree **bottom-up**. A state machine instructs the parser whether to shift or reduce each time it receives a token.

- **Yacc** can automatically generate a shift/reduce parser from a grammar.