Chapter 4d

Bottom Up Parsing

Right Sentential Forms

- Recall the definition of a derivation and a rightmost derivation.
- Each of the lines is a (right) sentential form
- The parsing problem is finding the correct RHS in a right-sentential form to reduce to get the previous right-sentential form in the derivation

```
1 E -> E+T
2 E -> T
3 T -> T*F
4 E -> F
5 F -> (E)
6 F -> id
```

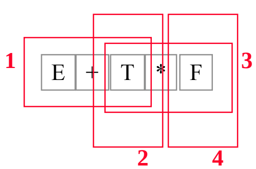
generation ↓

E
E+T
E+T*F
E+T*id
E+F*id
E+id*id
T+id*id
F+id*id
id+id*id

parsing ↑

Bottom up parsing

- A bottom up parser looks at a sentential form and selects a contiguous sequence of symbols that matches the RHS of a grammar rule, and replaces it with the LHS
- There might be several choices, as in the sentential form E+T*F
- Which one should we choose?




```
1 E -> E+T
2 E -> T
3 T -> T*F
4 E -> F
5 F -> (E)
6 F -> id
```

E
E+T
E+T*F
E+T*id
E+F*id
E+id*id
T+id*id
F+id*id
id+id*id

parsing ↑

Bottom up parsing

- If the wrong one is chosen, it leads to failure.
- E.g.: replacing E+T with E in E+T*F yields E+F, which can not be further reduced using the given grammar.
- We'll define the **handle** of a sentential form as the RHS that should be rewritten to yield the next sentential form in the right most derivation.

```
1 E -> E+T
2 E -> T
3 T -> T*F
4 E -> F
5 F -> (E)
6 F -> id
```

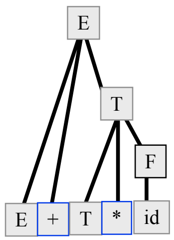
error
E*F
E+T*F
E+T*id
E+F*id
E+id*id
T+id*id
F+id*id
id+id*id

parsing ↑

Sentential forms

```
1 E -> E+T
2 E -> T
3 T -> T*F
4 E -> F
5 F -> (E)
6 F -> id
```

- Think of a sentential form as one of the entries in a derivation that begins with the start symbol and ends with a legal sentence.
- So, it's like a sentence but it may have some "unexpanded" non-terminals.
- We can also think of it as a parse tree where some of the leaves are as yet unexpanded non-terminals.



generation ↓ / parsing ↑

E
E+T
E+T*F
E+T*id
E+F*id
E+id*id
T+id*id
F+id*id
id+id*id

Handles

- A handle of a sentential form is a substring α such that :
  - a matches the RHS of a production A -> α ; and
  - replacing α by the LHS A represents a step in the reverse of a rightmost derivation of s.
- For this grammar, the rightmost derivation for the input abbcde is
  S => aABe => aAde => aAbcde => abbcde

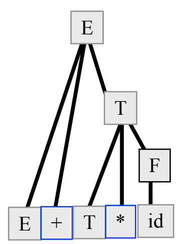
```
1: S -> aABe
2: A -> Abc
3: A -> b
4: B -> d
```

- The string aAbcde can be reduced in two ways:
  (1) aAbcde => aAde *(using rule 2)*
  (2) aAbcde => aAbcBe *(using rule 4)*
- But (2) isn't a rightmost derivation, so Abc is the only handle.
- Note: the string to the right of a handle will only contain terminals (why?)

a A b c d e

Phrases

- A **phrase** is a subsequence of a sentential form that is eventually "reduced" to a single non-terminal.
- A **simple phrase** is a phrase that is reduced in a single step.
- The **handle** is the left-most simple phrase.



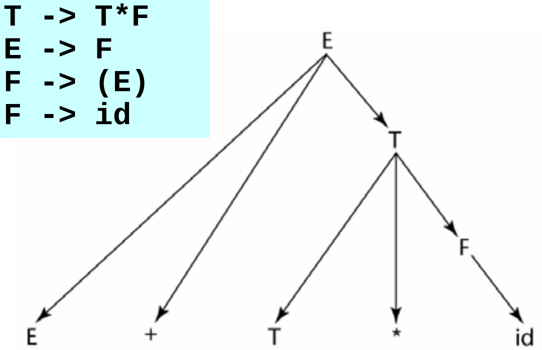
For this sentential form what are the
- phrases
- simple phrases
- handle

Phrases, simple phrases and handles

- **Def:** β is the *handle* of the right sentential form γ = αβw if and only if S =>*rm αAw => αβw
- **Def:** β is a *phrase* of the right sentential form γ if and only if S =>* γ = α1Aα2 =>+ α1βα2
- **Def:** β is a *simple phrase* of the right sentential form γ if and only if S =>* γ = α1Aα2 => α1βα2
- The handle of a right sentential form is its leftmost simple phrase
- Given a parse tree, it is now easy to find the handle
- Parsing can be thought of as handle pruning

Phrases, simple phrases and handles

```
E -> E+T
E -> T
T -> T*F
E -> F
F -> (E)
F -> id
```



E
E+T
E+T*F
E+T*id
E+F*id
E+id*id
T+id*id
F+id*id
id+id*id

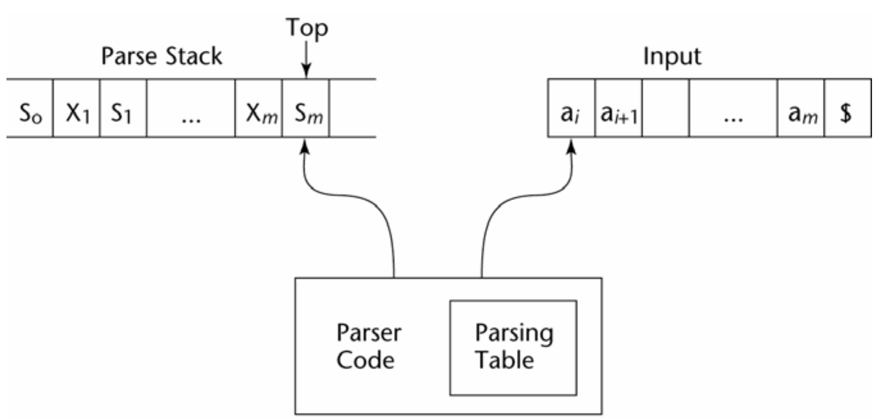
On to parsing

- Ok, so how do we manage when we don’t have the parse tree in front of us?
- We’ll look at a shift-reduce parser, of the kind that yacc uses.
- A shift-reduce parser has a queue of input tokens and an initially empty stack and takes one of four possible actions:
  - **Accept:** if the input queue is empty and the start symbol is the only thing on the stack.
  - **Reduce:** if there is a handle on the top of the stack, pop it off and replace it with the RHS
  - **Shift:** push the next input token onto the stack
  - **Fail:** if the input is empty and we can’t accept.
- In general, we might have a choice of doing a shift or a reduce, or maybe in reducing using one of several rules.
- The algorithm we next describe is deterministic.

Shift-Reduce Algorithms

A shift-reduce parser scans input, at each step, considers whether to:

- **Shift** the next token to the top of the parse stack (along with some state info)
- **Reduce** the stack by POPing several symbols off the stack (& their state info) and PUSHing the corresponding nonterminal (& state info)



Shift-Reduce Algorithms

The stack is always of the form

bottom **top**

$S_1\ X_1\ S_2\ X_2\ \ldots\ S_n\ X_n$

state **terminal or non-terminal**

- A reduction step is triggered when we see the symbols corresponding to a rule’s RHS on the top of the stack

bottom **top**

$S_1\ X_1\ \ldots S_5\ X_5\ S_6\ T\ S_7\ *\ S_8\ F$

T -> T*F

$S_1\ X_1\ \ldots S_5\ X_5\ S_6'\ T$

LR parser table

- LR shift-reduce parsers can be efficiently implemented by precomputing a table to guide the processing

	Action						Goto		
State	id	+	*	(	)	$	E	T	F
0	S5		S4				1	2	3
1		S6				accept			
2		R2	S7		R2	R2			
3		R4	R4		R4	R4			
4	S5			S4			8	2	3
5		R6	R6		R6	R6			
6	S5			S4				9	3
7	S5			S4					10
8		S6			S11				
9		R1	S7		R1	R1			
10		R3	R3		R3	R3			
11		R5	R5		R5	R5			

More on this Later . . .

When to shift, when to reduce

- The key problem in building a shift-reduce parser is deciding whether to shift or to reduce.
  - repeat: reduce if you see a handle on the top of the stack, shift otherwise
  - Succeed if we stop with only S on the stack and no input
- A grammar may not be appropriate for a LR parser because there are conflicts which can not be resolved.
- A conflict occurs when the parser cannot decide whether to:
  - shift or reduce the top of stack (a shift/reduce conflict), or
  - reduce the top of stack using one of two possible productions (a reduce/reduce conflict)
- There are several varieties of LR parsers (LR(0), LR(1), SLR and LALR), with differences depending on amount of lookahead and on construction of the parse table.

Conflicts

Shift-reduce conflict: can't decide whether to shift or to reduce

- Example : "dangling else"

  Stmt -> if Expr then Stmt

  | if Expr then Stmt else Stmt

  | ...
- What to do when else is at the front of the input?

Reduce-reduce conflict: can't decide which of several possible reductions to make

- Example :

  Stmt -> id (params)

  | Expr := Expr

  | ...

  Expr -> id (params)
- Given the input a(i, j) the parser does not know whether it is a procedure call or an array reference.

LR Table

- An LR configuration stores the state of an LR parser

  (S0X1S1X2S2…XmSm, aiai+1…an$)
- LR parsers are table driven, where the table has two components, an ACTION table and a GOTO table
- The ACTION table specifies the action of the parser (e.g., shift or reduce), given the parser state and the next token
  - Rows are state names; columns are terminals
- The GOTO table specifies which state to put on top of the parse stack after a reduce
  - Rows are state names; columns are nonterminals

	Action						Goto		
State	id	+	*	(	)	$	E	T	F
0	S5		S4				1	2	3
1		S6				accept			
2		R2	S7		R2	R2			
3		R4	R4		R4	R4			
4	S5			S4			8	2	3
5		R6	R6		R6	R6			
6	S5			S4				9	3
7	S5			S4					10
8		S6			S11				
9		R1	S7		R1	R1			
10		R3	R3		R3	R3			
11		R5	R5		R5	R5			

Parser actions

Initial configuration: (S0, a1…an$)

Parser actions:

1 If ACTION[Sm, ai] = Shift S, the next configuration is: (S0X1S1X2S2…XmSmaiS, ai+1…an$)

2 If ACTION[Sm, ai] = Reduce $A \rightarrow \beta$ and S = GOTO[Sm-r, A], where r = the length of β, the next configuration is

(S0X1S1X2S2…Xm-rSm-rAS, aiai+1…an$)

3 If ACTION[Sm, ai] = Accept, the parse is complete and no errors were found.

4 If ACTION[Sm, ai] = Error, the parser calls an error-handling routine.

Example

```
1: E -> E+T
2: E -> T
3: T -> T*F
4: E -> F
5: F -> (E)
6: F -> id
```

Stack	Input	action
0	Id + id * id $	Shift 5
0 id 5	+ id * id $	Reduce 6 *goto(0,F)*
0 F 3	+ id * id $	Reduce 4 *goto(0,T)*
0 T 2	+ id * id $	Reduce 2 *goto(0,E)*
0 E 1	+ id * id $	Shift 6
0 E 1 + 6	id * id $	Shift 5
0 E 1 + 6 id 5	* id $	Reduce 6 *goto(6,F)*
0 E 1 + 6 F 3	* id $	Reduce 4 *goto(6,T)*
0 E 1 + 6 T 9	* id $	Shift 7
0 E 1 + 6 T 9 * 7	id $	Shift 5
0 E 1 + 6 T 9 * 7 id 5	$	Reduce 6 *goto(7,E)*
0 E 1 + 6 T 9 * 7 F 10	$	Reduce 3 *goto(6,T)*
0 E 1 + 6 T 9	$	Reduce 1 *goto(0,E)*
0 E 1	$	Accept

	Action						Goto		
State	id	+	*	(	)	$	E	T	F
0	S5		S4				1	2	3
1		S6				accept			
2		R2	S7		R2	R2			
3		R4	R4		R4	R4			
4	S5			S4			8	2	3
5		R6	R6		R6	R6			
6	S5			S4				9	3
7	S5			S4					10
8		S6			S11				
9		R1	S7		R1	R1			
10		R3	R3		R3	R3			
11		R5	R5		R5	R5			

Yacc as a LR parser

- The Unix yacc utility is just such a parser.
- It does the heavy lifting of computing the table.
- To see the table information, use the –v flag when calling yacc, as in

  yacc –v test.y

```
0  $accept : E $end
   1  E : E '+' T
   2    | T
   3  T : T '*' F
   4    | F
   5  F : '(' E ')'
   6    | "id"

state 0
        $accept : . E $end  (0)
        '('  shift 1
        "id"  shift 2
        .  error
        E  goto 3
        T  goto 4
        F  goto 5
state 1
        F : '(' . E ')'  (5)
        '('  shift 1
        "id"  shift 2
        .  error
        E  goto 6
        T  goto 4
        F  goto 5
. . .
```


Chapter 4d

Bottom Up Parsing

Right Sentential Forms

- Recall the definition of a derivation and a rightmost derivation.
- Each of the lines is a (right) sentential form
- The parsing problem is finding the correct RHS in a right-sentential form to reduce to get the previous right-sentential form in the derivation

```
1 E -> E+T
2 E -> T
3 T -> T*F
4 E -> F
5 F -> (E)
6 F -> id
```

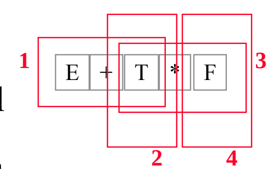
generation ↓

E
E+T
E+T*F
E+T*id
E+F*id
E+id*id
T+id*id
F+id*id
id+id*id

parsing ↑

Bottom up parsing

- A bottom up parser looks at a sentential form and selects a contiguous sequence of symbols that matches the RHS of a grammar rule, and replaces it with the LHS
- There might be several choices, as in the sentential form E+T*F
- Which one should we choose?




```
1 E -> E+T
2 E -> T
3 T -> T*F
4 E -> F
5 F -> (E)
6 F -> id
```

E
E+T
E+T*F
E+T*id
E+F*id
E+id*id
T+id*id
F+id*id
id+id*id

parsing ↑

Bottom up parsing

```
1 E -> E+T
2 E -> T
3 T -> T*F
4 E -> F
5 F -> (E)
6 F -> id
```

- If the wrong one is chosen, it leads to failure.
- E.g.: replacing E+T with E in E+T*F yields E+F, which can not be further reduced using the given grammar.
- We'll define the **handle** of a sentential form as the RHS that should be rewritten to yield the next sentential form in the right most derivation.

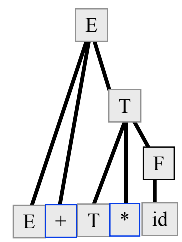
error
E*F
E+T*F
E+T*id
E+F*id
E+id*id
T+id*id
F+id*id
id+id*id

parsing

Sentential forms

```
1 E -> E+T
2 E -> T
3 T -> T*F
4 E -> F
5 F -> (E)
6 F -> id
```

- Think of a sentential form as one of the entries in a derivation that begins with the start symbol and ends with a legal sentence.
- So, it's like a sentence but it may have some "unexpanded" non-terminals.
- We can also think of it as a parse tree where some of the leaves are as yet unexpanded non-terminals.



generation

E
E+T
E+T*F
E+T*id
E+F*id
E+id*id
T+id*id
F+id*id
id+id*id

parsing

Handles

- A handle of a sentential form is a substring α such that :
  - a matches the RHS of a production A -> α ; and
  - replacing α by the LHS A represents a step in the reverse of a rightmost derivation of s.
- For this grammar, the rightmost derivation for the input abbcde is
  S => aABe => aAde => aAbcde => abbcde

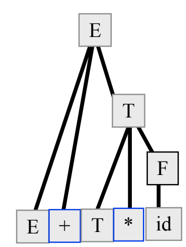
```
1: S -> aABe
2: A -> Abc
3: A -> b
4: B -> d
```

- The string aAbcde can be reduced in two ways:
  (1) aAbcde => aAde *(using rule 2)*
  (2) aAbcde => aAbcBe *(using rule 4)*
- But (2) isn't a rightmost derivation, so Abc is the only handle.
- Note: the string to the right of a handle will only contain terminals (why?)

a $\boxed{\text{A b c}}$ d e

Phrases

- A **phrase** is a subsequence of a sentential form that is eventually "reduced" to a single non-terminal.
- A **simple phrase** is a phrase that is reduced in a single step.
- The **handle** is the left-most simple phrase.



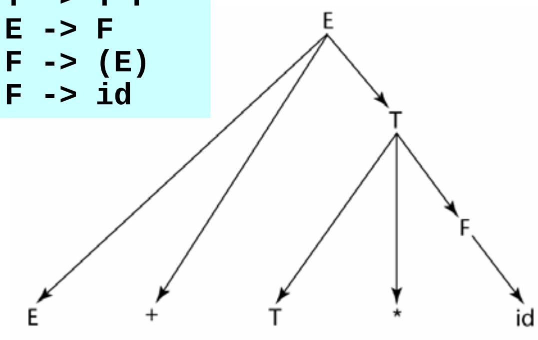
For this sentential form what are the
- phrases
- simple phrases
- handle

Phrases, simple phrases and handles

- **Def:** β is the *handle* of the right sentential form $\gamma = \alpha\beta w$ if and only if $S \Rightarrow^*_{rm} \alpha A w \Rightarrow \alpha\beta w$
- **Def:** β is a *phrase* of the right sentential form γ if and only if $S \Rightarrow^* \gamma = \alpha 1 A \alpha 2 \Rightarrow^+ \alpha 1 \beta \alpha 2$
- **Def:** β is a *simple phrase* of the right sentential form γ if and only if $S \Rightarrow^* \gamma = \alpha 1 A \alpha 2 \Rightarrow \alpha 1 \beta \alpha 2$
- The handle of a right sentential form is its leftmost simple phrase
- Given a parse tree, it is now easy to find the handle
- Parsing can be thought of as handle pruning

Phrases, simple phrases and handles

```
E -> E+T
E -> T
T -> T*F
E -> F
F -> (E)
F -> id
```



E
E+T
E+T*F
E+T*id
E+F*id
E+id*id
T+id*id
F+id*id
id+id*id

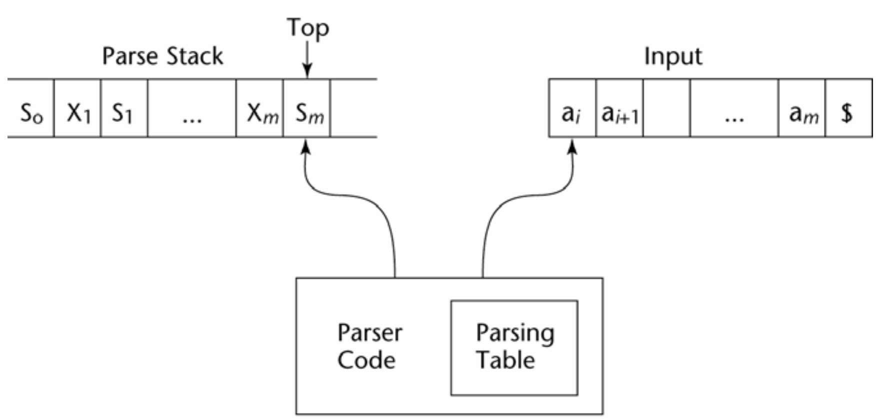
On to parsing

- Ok, so how do we manage when we don't have the parse tree in front of us?
- We'll look at a shift-reduce parser, of the kind that yacc uses.
- A shift-reduce parser has a queue of input tokens and an initially empty stack and takes one of four possible actions:
  - **Accept:** if the input queue is empty and the start symbol is the only thing on the stack.
  - **Reduce:** if there is a handle on the top of the stack, pop it off and replace it with the RHS
  - **Shift:** push the next input token onto the stack
  - **Fail:** if the input is empty and we can't accept.
- In general, we might have a choice of doing a shift or a reduce, or maybe in reducing using one of several rules.
- The algorithm we next describe is deterministic.

Shift-Reduce Algorithms

A shift-reduce parser scans input, at each step, considers whether to:

- **Shift** the next token to the top of the parse stack (along with some state info)
- **Reduce** the stack by POPing several symbols off the stack (& their state info) and PUSHing the corresponding nonterminal (& state info)



Shift-Reduce Algorithms

The stack is always of the form

bottom top

$S_1 \; X_1 \; S_2 \; X_2 \ldots S_n \; X_n$

state terminal or non-terminal

- A reduction step is triggered when we see the symbols corresponding to a rule's RHS on the top of the stack

bottom top

$S_1 \; X_1 \; \ldots S_5 \; X_5 \; S_6 \; T \; S_7 \; * \; S_8 \; F$

T -> T*F

$S_1 \; X_1 \; \ldots S_5 \; X_5 \; S_6' \; T$

LR parser table

- LR shift-reduce parsers can be efficiently implemented by precomputing a table to guide the processing

	Action						Goto		
State	id	+	*	(	)	$	E	T	F
0	S5		S4				1	2	3
1		S6				accept			
2		R2	S7		R2	R2			
3		R4	R4		R4	R4			
4	S5			S4			8	2	3
5		R6	R6		R6	R6			
6	S5			S4				9	3
7	S5			S4					10
8		S6			S11				
9		R1	S7		R1	R1			
10		R3	R3		R3	R3			
11		R5	R5		R5	R5			

More on this Later . . .

When to shift, when to reduce

- The key problem in building a shift-reduce parser is deciding whether to shift or to reduce.
  - repeat: reduce if you see a handle on the top of the stack, shift otherwise
  - Succeed if we stop with only S on the stack and no input
- A grammar may not be appropriate for a LR parser because there are conflicts which can not be resolved.
- A conflict occurs when the parser cannot decide whether to:
  - shift or reduce the top of stack (a shift/reduce conflict), or
  - reduce the top of stack using one of two possible productions (a reduce/reduce conflict)
- There are several varieties of LR parsers (LR(0), LR(1), SLR and LALR), with differences depending on amount of lookahead and on construction of the parse table.

Conflicts

Shift-reduce conflict: can't decide whether to shift or to reduce

- Example : "dangling else"

  Stmt -> if Expr then Stmt
  | if Expr then Stmt else Stmt
  | ...

- What to do when else is at the front of the input?

Reduce-reduce conflict: can't decide which of several possible reductions to make

- Example :

  Stmt -> id (params)
  | Expr := Expr
  | ...
  Expr -> id (params)

- Given the input a(i, j) the parser does not know whether it is a procedure call or an array reference.

LR Table

- An LR configuration stores the state of an LR parser
  (S0X1S1X2S2…XmSm, aiai+1…an$)
- LR parsers are table driven, where the table has two components, an ACTION table and a GOTO table
- The ACTION table specifies the action of the parser (e.g., shift or reduce), given the parser state and the next token
  – Rows are state names; columns are terminals
- The GOTO table specifies which state to put on top of the parse stack after a reduce
  – Rows are state names; columns are nonterminals

	Action						Goto		
State	id	+	*	(	)	$	E	T	F
0	S5		S4				1	2	3
1		S6				accept			
2		R2	S7		R2	R2			
3		R4	R4		R4	R4			
4	S5			S4			8	2	3
5		R6	R6		R6	R6			
6	S5			S4				9	3
7	S5			S4					10
8		S6			S11				
9		R1	S7		R1	R1			
10		R3	R3		R3	R3			
11		R5	R5		R5	R5			

Parser actions

Initial configuration: (S0, a1…an$)

Parser actions:

1 If ACTION[Sm, ai] = Shift S, the next configuration is: (S0X1S1X2S2…XmSmaiS, ai+1…an$)

2 If ACTION[Sm, ai] = Reduce A → β and S = GOTO[Sm-r, A], where r = the length of β, the next configuration is

(S0X1S1X2S2…Xm-rSm-rAS, aiai+1…an$)

3 If ACTION[Sm, ai] = Accept, the parse is complete and no errors were found.

4 If ACTION[Sm, ai] = Error, the parser calls an error-handling routine.

Example

```
1: E -> E+T
2: E -> T
3: T -> T*F
4: E -> F
5: F -> (E)
6: F -> id
```

Stack	Input	action
0	Id + id * id $	Shift 5
0 id 5	+ id * id $	Reduce 6 *goto(0,F)*
0 F 3	+ id * id $	Reduce 4 *goto(0,T)*
0 T 2	+ id * id $	Reduce 2 *goto(0,E)*
0 E 1	+ id * id $	Shift 6
0 E 1 + 6	id * id $	Shift 5
0 E 1 + 6 id 5	* id $	Reduce 6 *goto(6,F)*
0 E 1 + 6 F 3	* id $	Reduce 4 *goto(6,T)*
0 E 1 + 6 T 9	* id $	Shift 7
0 E 1 + 6 T 9 * 7	id $	Shift 5
0 E 1 + 6 T 9 * 7 id 5	$	Reduce 6 *goto(7,E)*
0 E 1 + 6 T 9 * 7 F 10	$	Reduce 3 *goto(6,T)*
0 E 1 + 6 T 9	$	Reduce 1 *goto(0,E)*
0 E 1	$	Accept

	Action						Goto		
State	id	+	*	(	)	$	E	T	F
0	S5		S4				1	2	3
1		S6				accept			
2		R2	S7		R2	R2			
3		R4	R4		R4	R4			
4	S5			S4			8	2	3
5		R6	R6		R6	R6			
6	S5			S4				9	3
7	S5			S4					10
8		S6			S11				
9		R1	S7		R1	R1			
10		R3	R3		R3	R3			
11		R5	R5		R5	R5			

Yacc as a LR parser

- The Unix yacc utility is just such a parser.
- It does the heavy lifting of computing the table.
- To see the table information, use the –v flag when calling yacc, as in

  yacc –v test.y

```
 0  $accept : E $end
   1  E : E '+' T
   2    | T
   3  T : T '*' F
   4    | F
   5  F : '(' E ')'
   6    | "id"

state 0
        $accept : . E $end  (0)
        '('  shift 1
        "id"  shift 2
        .  error
        E  goto 3
        T  goto 4
        F  goto 5
state 1
        F : '(' . E ')'  (5)
        '('  shift 1
        "id"  shift 2
        .  error
        E  goto 6
        T  goto 4
        F  goto 5
. . .
```