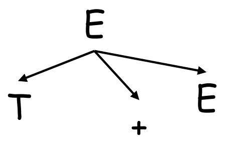
Introduction to Bottom-Up Parsing

Outline

- Review LL parsing
- Shift-reduce parsing
- The LR parsing algorithm
- Constructing LR parsing tables

Top-Down Parsing: Review

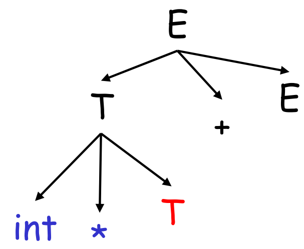
- Top-down parsing expands a parse tree from the start symbol to the leaves
  - Always expand the leftmost non-terminal



int * int + int

Top-Down Parsing: Review

- Top-down parsing expands a parse tree from the start symbol to the leaves
  - Always expand the leftmost non-terminal

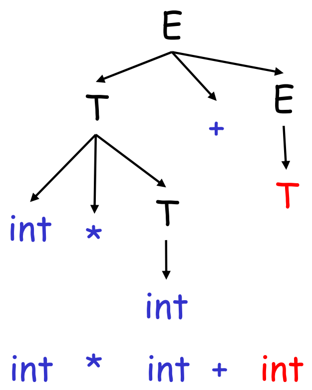


- The leaves at any point form a string $\beta A \gamma$
  - β contains only terminals
  - The input string is $\beta b \delta$
  - The prefix β matches
  - The next token is b

int * int + int

Top-Down Parsing: Review

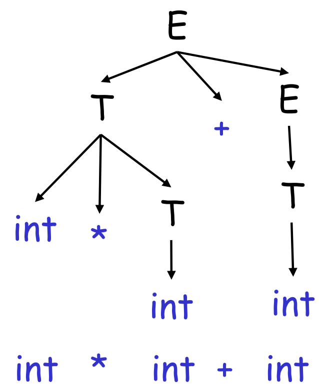
- Top-down parsing expands a parse tree from the start symbol to the leaves
  - Always expand the leftmost non-terminal



- The leaves at any point form a string $\beta A \gamma$
  - β contains only terminals
  - The input string is $\beta b \delta$
  - The prefix β matches
  - The next token is b

Top-Down Parsing: Review

- Top-down parsing expands a parse tree from the start symbol to the leaves
  - Always expand the leftmost non-terminal



- The leaves at any point form a string $\beta A \gamma$
  - β contains only terminals
  - The input string is $\beta b \delta$
  - The prefix β matches
  - The next token is b

Predictive Parsing: Review

- A predictive parser is described by a table
  - For each non-terminal A and for each token b we specify a production $A \to \alpha$
  - When trying to expand A we use $A \to \alpha$ if b follows next

- Once we have the table
  - The parsing algorithm is simple and fast
  - No backtracking is necessary

Constructing Predictive Parsing Tables

1. Consider the state $S \to^* \beta A \gamma$
   - With b the next token
   - Trying to match $\beta b \delta$

There are two possibilities:

- b belongs to an expansion of A
  - Any $A \to \alpha$ can be used if b can start a string derived from α
  - We say that $b \in First(\alpha)$

Or…

Constructing Predictive Parsing Tables (Cont.)

2. b does not belong to an expansion of A
   - The expansion of A is empty and b belongs to an expansion of γ
   - Means that b can appear after A in a derivation of the form $S \to^* \beta A b \omega$
   - We say that $b \in$ Follow(A) in this case
   - What productions can we use in this case?
     - Any $A \to \alpha$ can be used if α can expand to ε
     - We say that $\varepsilon \in$ First(A) in this case

Computing First Sets

Definition

$$\text{First}(X) = \{ b \mid X \to^* b\alpha \} \cup \{ \varepsilon \mid X \to^* \varepsilon \}$$

Algorithm sketch

1. First(b) = { b }
2. $\varepsilon \in$ First(X) if $X \to \varepsilon$ is a production
3. $\varepsilon \in$ First(X) if $X \to A_1 \ldots A_n$ and $\varepsilon \in$ First(A_i) for $1 \le i \le n$
4. First(α) $\subseteq$ First(X) if $X \to A_1 \ldots A_n\ \alpha$ and $\varepsilon \in$ First(A_i) for $1 \le i \le n$

First Sets: Example

- Recall the grammar

$E \to T X \qquad\qquad X \to + E \mid \varepsilon$

$T \to (E) \mid \text{int } Y \qquad Y \to * T \mid \varepsilon$

- First sets

First(( ) = { (} — First(T) = {int, (}

First( )) = {) } — First(E) = {int, (}

First(int) = { int } — First(X) = {+, ε }

First(+) = { + } — First(Y) = {*, ε }

First(*) = { * }

Computing Follow Sets

- Definition

$$\text{Follow}(X) = \{ b \mid S \to^* \beta\ X\ b\ \delta \}$$

- Intuition
  - If $X \to A\ B$ then First(B) $\subseteq$ Follow(A) and Follow(X) $\subseteq$ Follow(B)
  - Also if $B \to^* \varepsilon$ then Follow(X) $\subseteq$ Follow(A)
  - If S is the start symbol then $\$ \in$ Follow(S)

Computing Follow Sets (Cont.)

Algorithm sketch

1. $\$ \in \text{Follow}(S)$
2. $\text{First}(\beta) - \{\varepsilon\} \subseteq \text{Follow}(X)$
   - For each production $A \to \alpha\ X\ \beta$
3. $\text{Follow}(A) \subseteq \text{Follow}(X)$
   - For each production $A \to \alpha\ X\ \beta$ where $\varepsilon \in \text{First}(\beta)$

Follow Sets: Example

- Recall the grammar

  $E \to T\ X \qquad X \to +\ E \mid \varepsilon$

  $T \to (\ E\ ) \mid \text{int}\ Y \qquad Y \to *\ T \mid \varepsilon$

- Follow sets

  Follow(+) = { int, (} Follow(*) = { int, (}

  Follow(( ) = { int, (} Follow(E) = {), $ }

  Follow(X) = { $,) } Follow(T) = { +,) , $ }

  Follow( )) = { +,) , $ } Follow(Y) = { +,) , $ }

  Follow(int) = { *, +,) , $ }

Constructing LL(1) Parsing Tables

- Construct a parsing table T for CFG G

- For each production $A \to \alpha$ in G do:
  - For each terminal $b \in \text{First}(\alpha)$ do
    - $T[A, b] = \alpha$
  - If $\varepsilon \in \text{First}(\alpha)$, for each $b \in \text{Follow}(A)$ do
    - $T[A, b] = \alpha$
  - If $\varepsilon \in \text{First}(\alpha)$ and $\$ \in \text{Follow}(A)$ do
    - $T[A, \$] = \alpha$

Constructing LL(1) Tables: Example

- Recall the grammar

  $E \to T\ X \qquad X \to +\ E \mid \varepsilon$

  $T \to (\ E\ ) \mid \text{int}\ Y \qquad Y \to *\ T \mid \varepsilon$

- Where in the line of Y we put $Y \to *\ T$?
  - In the lines of First(*T) = { * }

- Where in the line of Y we put $Y \to \varepsilon$?
  - In the lines of Follow(Y) = { $, +,) }

Notes on LL(1) Parsing Tables

- If any entry is multiply defined then G is not LL(1)
  - If G is ambiguous
  - If G is left recursive
  - If G is not left-factored
  - <u>And in other cases as well</u>

- For some grammars there is a simple parsing strategy: *Predictive parsing*
- Most programming language grammars are not LL(1)
- Thus, we need more powerful parsing strategies

Bottom Up Parsing

Bottom-Up Parsing

- Bottom-up parsing is more general than top-down parsing
  - And just as efficient
  - Builds on ideas in top-down parsing
  - Preferred method in practice

- Also called LR parsing
  - L means that tokens are read left to right
  - R means that it constructs a rightmost derivation !

An Introductory Example

- LR parsers don't need left-factored grammars and can also handle left-recursive grammars

- Consider the following grammar:

$$E \rightarrow E + (E) \mid int$$

  - Why is this not LL(1)?

- Consider the string: int + (int) + (int)

The Idea

- LR parsing *reduces* a string to the start symbol by inverting productions:

str w input string of terminals

repeat

- Identify β in str such that $A \rightarrow \beta$ is a production (i.e., str = $\alpha\ \beta\ \gamma$)
- Replace β by A in str (i.e., str w = $\alpha\ A\ \gamma$)

until str = S (the start symbol)

OR all possibilities are exhausted

A Bottom-up Parse in Detail (1)

int + (int) + (int)

int + (int) + (int)

A Bottom-up Parse in Detail (2)

int + (int) + (int)

E + (int) + (int)

E

int + (int) + (int)

A Bottom-up Parse in Detail (3)

int + (int) + (int)

E + (int) + (int)

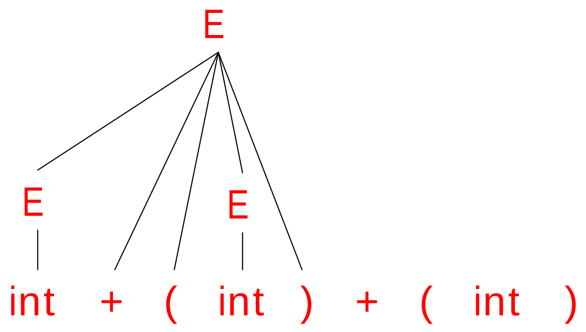
E + (E) + (int)

E E

int + (int) + (int)

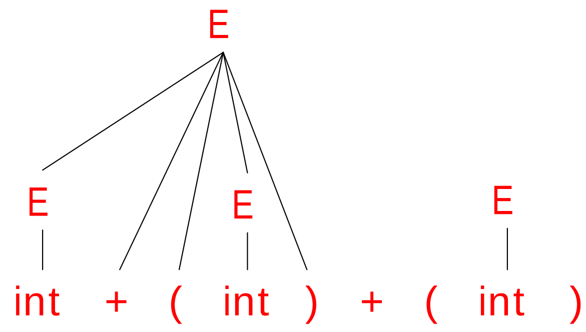
A Bottom-up Parse in Detail (4)

int + (int) + (int)
E + (int) + (int)
E + (E) + (int)
E + (int)



A Bottom-up Parse in Detail (5)

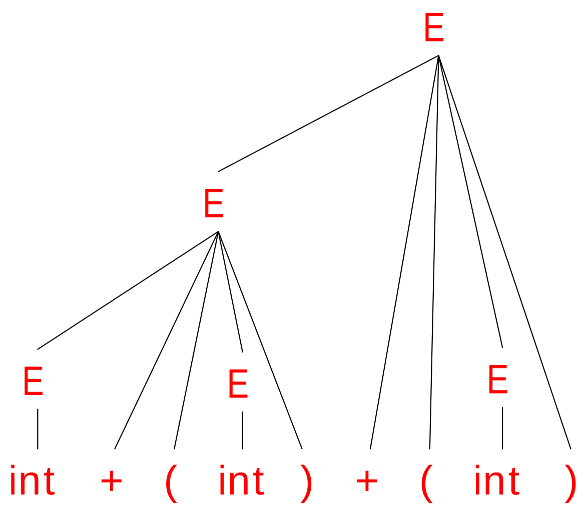
int + (int) + (int)
E + (int) + (int)
E + (E) + (int)
E + (int)
E + (E)



A Bottom-up Parse in Detail (6)

int + (int) + (int)
E + (int) + (int)
E + (E) + (int)
E + (int)
E + (E)
E

A rightmost derivation in reverse



Important Fact #1

Important Fact #1 about bottom-up parsing:

An LR parser traces a rightmost derivation in reverse

Where Do Reductions Happen

Important Fact #1 has an interesting consequence:

- Let $\alpha\beta\gamma$ be a step of a bottom-up parse
- Assume the next reduction is by using $A \to \beta$
- Then γ is a string of terminals

Why? Because $\alpha A\gamma \to \alpha\beta\gamma$ is a step in a right-most derivation

Notation

- Idea: Split string into two substrings
  - Right substring is as yet unexamined by parsing (a string of terminals)
  - Left substring has terminals and non-terminals
- The dividing point is marked by a ।
  - The । is not part of the string
- Initially, all input is unexamined: $|x_1 x_2 \ldots x_n$

Shift-Reduce Parsing

Bottom-up parsing uses only two kinds of actions:

Shift

Reduce

Shift

Shift: Move । one place to the right

- Shifts a terminal to the left string

$$E + (\,|\, \text{int}\,) \Rightarrow E + (\text{int}\,|\,)$$

In general:

$$ABC\,|\,xyz \Rightarrow ABCx\,|\,yz$$

Reduce

Reduce: Apply an inverse production at the right end of the left string

- If $E \to E + (E)$ is a production, then

$$E + (\underline{E + (E)} \mid) \Rightarrow E + (\underline{E} \mid)$$

In general, given $A \to xy$, then:

$$Cbxy \mid ijk \Rightarrow CbA \mid ijk$$

Shift-Reduce Example

$E \to E + (E) \mid int$

| int + (int) + (int)$ shift

int   +   (int   ) +   (int   )

Shift-Reduce Example

$E \to E + (E) \mid int$

| int + (int) + (int)$ shift
int | + (int) + (int)$ reduce $E \to int$

int   +   (int   ) +   (int   )

Shift-Reduce Example

$E \to E + (E) \mid int$

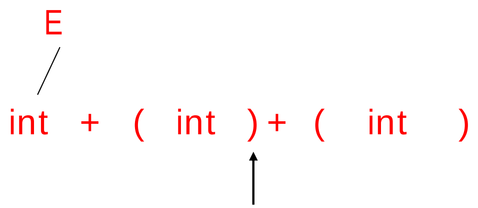
| int + (int) + (int)$ shift
int | + (int) + (int)$ reduce $E \to int$
E | + (int) + (int)$ shift 3 times

E
int   +   (int   ) +   (int   )

Shift-Reduce Example

$E \to E + (E) \mid int$

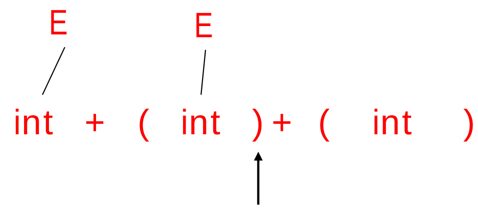
| int + (int) + (int)$ shift
int | + (int) + (int)$ reduce E → int
E | + (int) + (int)$ shift 3 times
E + (int |) + (int)$ reduce E → int



Shift-Reduce Example

$E \to E + (E) \mid int$

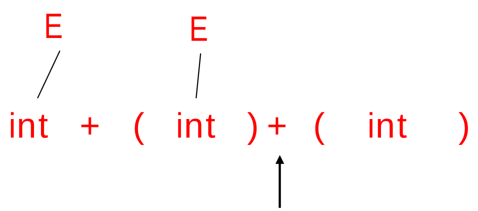
| int + (int) + (int)$ shift
int | + (int) + (int)$ reduce E → int
E | + (int) + (int)$ shift 3 times
E + (int |) + (int)$ reduce E → int
E + (E |) + (int)$ shift



Shift-Reduce Example

$E \to E + (E) \mid int$

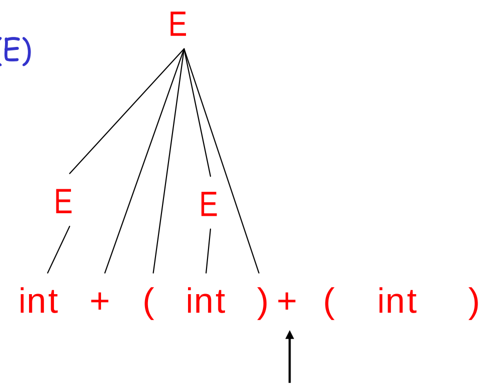
| int + (int) + (int)$ shift
int | + (int) + (int)$ reduce E → int
E | + (int) + (int)$ shift 3 times
E + (int |) + (int)$ reduce E → int
E + (E |) + (int)$ shift
E + (E) | + (int)$ reduce E → E + (E)



Shift-Reduce Example

$E \to E + (E) \mid int$

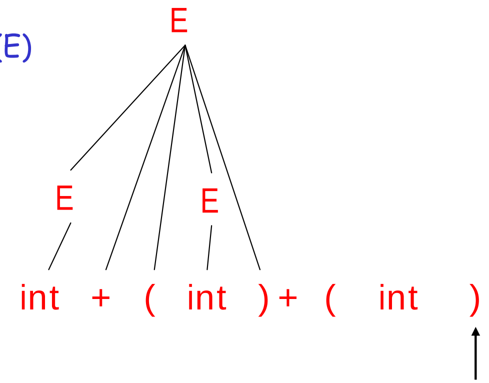
| int + (int) + (int)$ shift
int | + (int) + (int)$ reduce E → int
E | + (int) + (int)$ shift 3 times
E + (int |) + (int)$ reduce E → int
E + (E |) + (int)$ shift
E + (E) | + (int)$ reduce E → E + (E)
E | + (int)$ shift 3 times



Shift-Reduce Example

$E \to E + (E) \mid \text{int}$

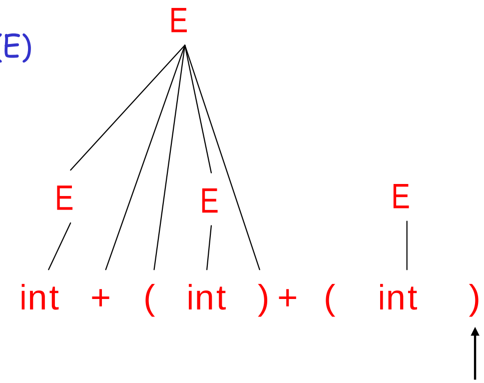
∣ int + (int) + (int)$	shift
int ∣ + (int) + (int)$	reduce $E \to$ int
E ∣ + (int) + (int)$	shift 3 times
E + (int ∣) + (int)$	reduce $E \to$ int
E + (E ∣) + (int)$	shift
E + (E) ∣ + (int)$	reduce $E \to E + (E)$
E ∣ + (int)$	shift 3 times
E + (int ∣)$	reduce $E \to$ int



Shift-Reduce Example

$E \to E + (E) \mid \text{int}$

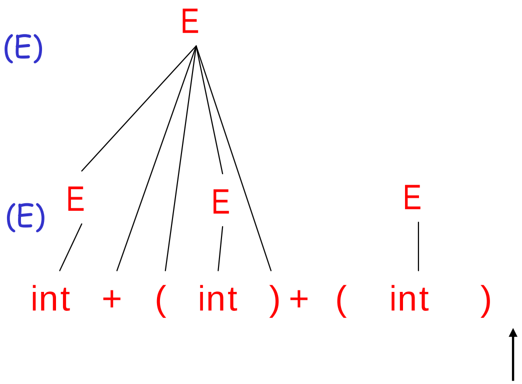
∣ int + (int) + (int)$	shift
int ∣ + (int) + (int)$	reduce $E \to$ int
E ∣ + (int) + (int)$	shift 3 times
E + (int ∣) + (int)$	reduce $E \to$ int
E + (E ∣) + (int)$	shift
E + (E) ∣ + (int)$	reduce $E \to E + (E)$
E ∣ + (int)$	shift 3 times
E + (int ∣)$	reduce $E \to$ int
E + (E ∣)$	shift



Shift-Reduce Example

$E \to E + (E) \mid \text{int}$

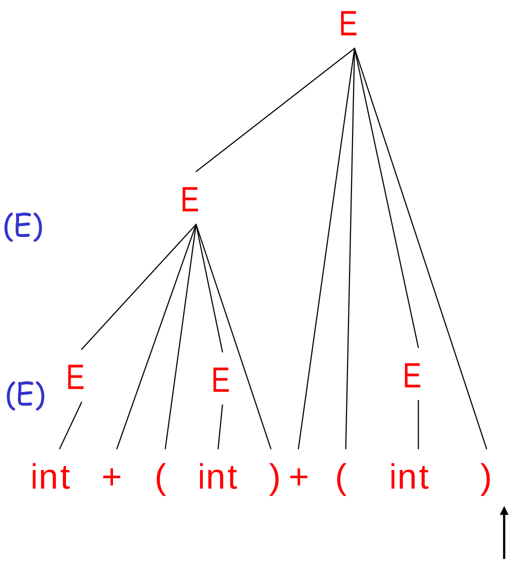
∣ int + (int) + (int)$	shift
int ∣ + (int) + (int)$	reduce $E \to$ int
E ∣ + (int) + (int)$	shift 3 times
E + (int ∣) + (int)$	reduce $E \to$ int
E + (E ∣) + (int)$	shift
E + (E) ∣ + (int)$	reduce $E \to E + (E)$
E ∣ + (int)$	shift 3 times
E + (int ∣)$	reduce $E \to$ int
E + (E ∣)$	shift
E + (E) ∣ $	reduce $E \to E + (E)$



Shift-Reduce Example

$E \to E + (E) \mid \text{int}$

∣ int + (int) + (int)$	shift
int ∣ + (int) + (int)$	reduce $E \to$ int
E ∣ + (int) + (int)$	shift 3 times
E + (int ∣) + (int)$	reduce $E \to$ int
E + (E ∣) + (int)$	shift
E + (E) ∣ + (int)$	reduce $E \to E + (E)$
E ∣ + (int)$	shift 3 times
E + (int ∣)$	reduce $E \to$ int
E + (E ∣)$	shift
E + (E) ∣ $	reduce $E \to E + (E)$
E ∣ $	accept



The Stack

- Left string can be implemented by a stack
  - Top of the stack is the ǀ
- Shift pushes a terminal on the stack
- Reduce pops 0 or more symbols off of the stack (production RHS) and pushes a non-terminal on the stack (production LHS)

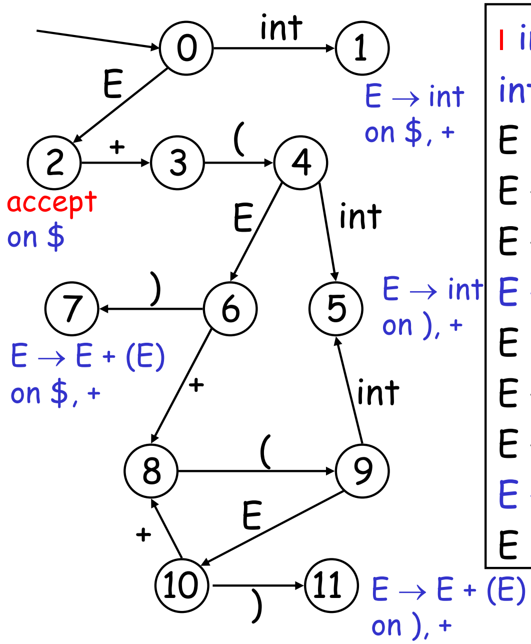
Key Question: To Shift or to Reduce?

Idea: use a finite automaton (DFA) to decide when to shift or reduce

- The input is the stack
- The language consists of terminals and non-terminals

- We run the DFA on the stack and we examine the resulting state X and the token tok after ǀ
  - If X has a transition labeled tok then shift
  - If X is labeled with "*A* → β on tok" then reduce

LR(1) Parsing: An Example



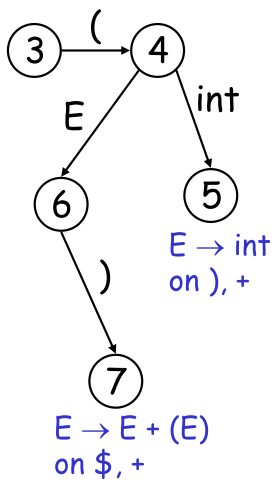
ǀ int + (int) + (int)$	shift
int ǀ + (int) + (int)$	E → int
E ǀ + (int) + (int)$	shift(x3)
E + (int ǀ) + (int)$	E → int
E + (E ǀ) + (int)$	shift
E + (E) ǀ + (int)$	E → E+(E)
E ǀ + (int)$	shift (x3)
E + (int ǀ)$	E → int
E + (E ǀ)$	shift
E + (E) ǀ $	E → E+(E)
E ǀ $	accept

Representing the DFA

- Parsers represent the DFA as a 2D table
  - Recall table-driven lexical analysis
- Lines correspond to DFA states
- Columns correspond to terminals and non-terminals
- Typically columns are split into:
  - Those for terminals: action table
  - Those for non-terminals: goto table

Representing the DFA: Example

- The table for a fragment of our DFA:



	int	+	(	)	$	E
…						
3			s4			
4	s5					g6
5		$r_{E \to int}$		$r_{E \to int}$		
6	s8		s7			
7		$r_{E \to E+(E)}$			$r_{E \to E+(E)}$	
…						

The LR Parsing Algorithm

- After a shift or reduce action we rerun the DFA on the entire stack
  - This is wasteful, since most of the work is repeated
- Remember for each stack element on which state it brings the DFA
- LR parser maintains a stack

$$\langle \text{sym}_1, \text{state}_1 \rangle \ldots \langle \text{sym}_n, \text{state}_n \rangle$$

state_k is the final state of the DFA on $\text{sym}_1 \ldots \text{sym}_k$

The LR Parsing Algorithm

```
Let I = w$ be initial input
Let j = 0
Let DFA state 0 be the start state
Let stack = ⟨ dummy, 0 ⟩
   repeat
        case action[top_state(stack), I[j]] of
                shift k: push ⟨ I[j++], k ⟩
                reduce X → A:
                      pop |A| pairs,
                      push ⟨X, Goto[top_state(stack), X]⟩
                accept: halt normally
                error: halt and report error
```


LR Parsers

- Can be used to parse more grammars than LL
- Most programming languages grammars are LR
- LR Parsers can be described as a simple table
- There are tools for building the table
- How is the table constructed?