Compilers

Bottom-up Parsing

Yannis Smaragdakis, U. Athens
(original slides by Sam Guyer@Tufts)

Bottom-Up Parsing

- More general than top-down parsing
  - And just as efficient
  - Builds on ideas in top-down parsing
  - Preferred method in many instances
- Specific algorithm: ***LR parsing***
  - L means that tokens are read left to right
  - R means that it constructs a rightmost derivation
  - Donald Knuth (1965)
    "On the Translation of Languages from Left to Right"

The Idea

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

  str ℑ 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 becomes $\alpha\ A\ \gamma$)

  until str = G

A simple example

- LR parsers:
  - Can handle left-recursion
  - Don’t need left factoring

- Consider the following grammar:

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

  - Is this grammar LL(1) *(as shown)*?

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  )
```

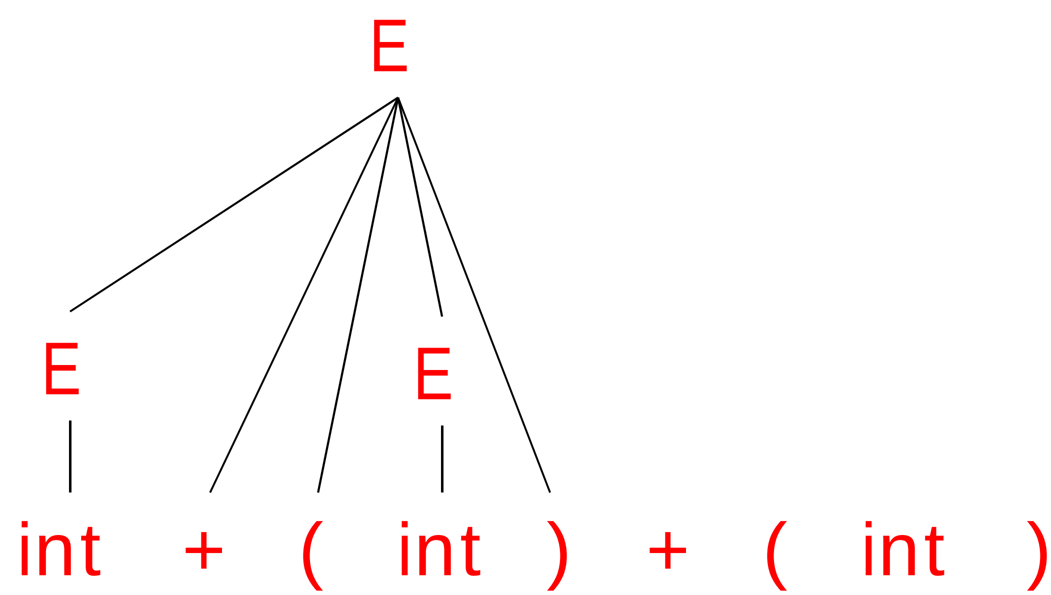
A Bottom-up Parse in Detail (4)

int + (int) + (int)

E + (int) + (int)

E + (E) + (int)

E + (int)



A Bottom-up Parse in Detail (5)

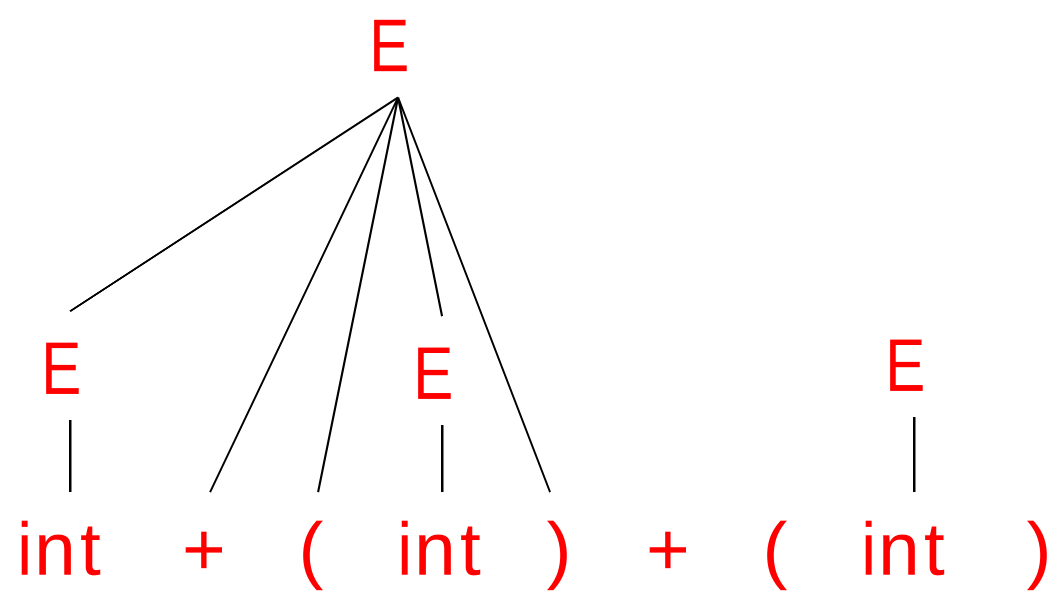
int + (int) + (int)

E + (int) + (int)

E + (E) + (int)

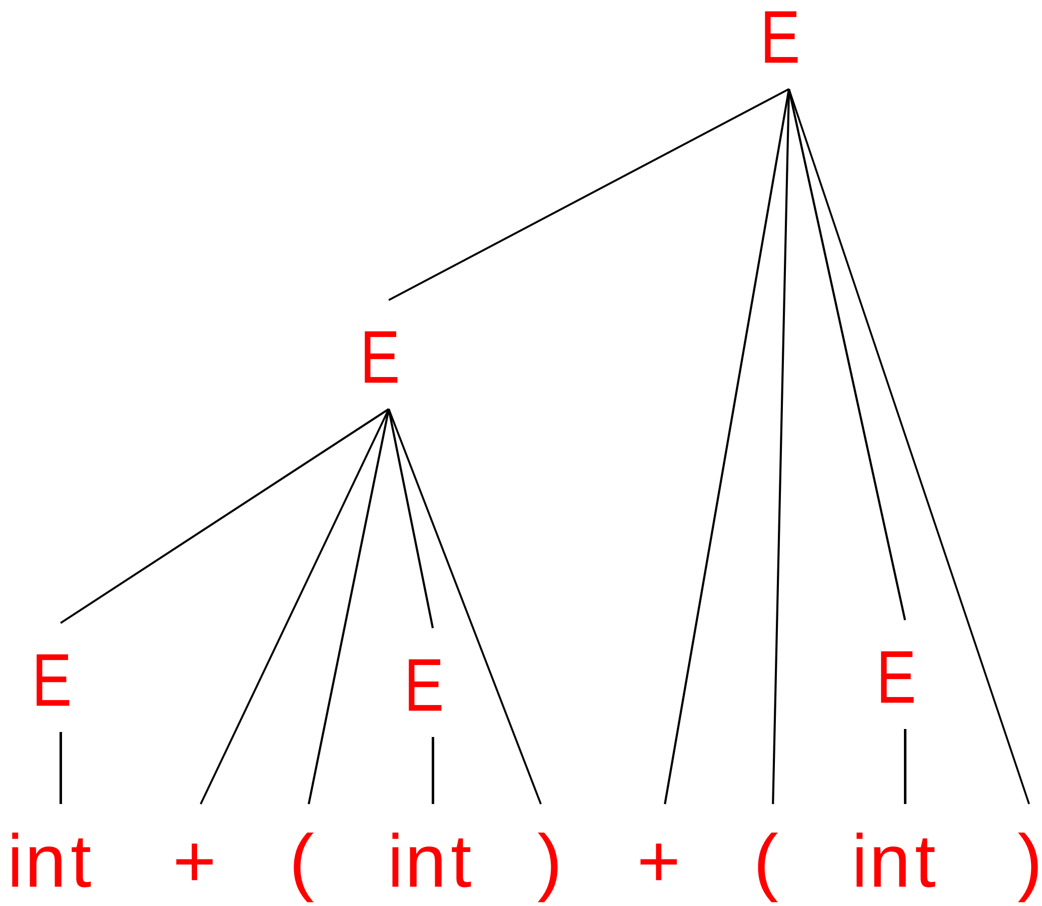
E + (int)

E + (E)



A Bottom-up Parse in Detail (6)

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

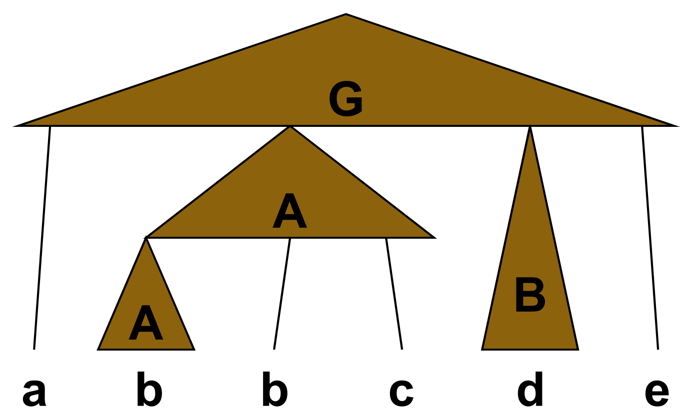


Another example

- Start with input stream
  - "Leaves" of parse tree
- Build up towards goal symbol
  - Called "***reducing***"
  - Construct the reverse derivation

#	*Production rule*
1	*G* → *a A B e*
2	*A* → A b c
3	| b
4	B → *d*

Rule	*Sentential form*
-	*abbcde*
3	*aAbcde*
2	*aAde*
4	*aABe*
1	*G*



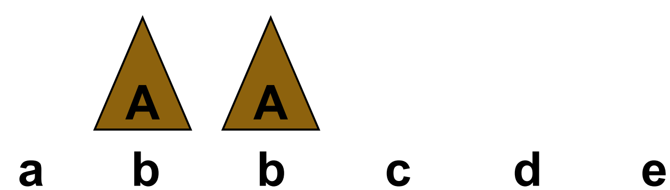
Easy?

- **Choosing a reduction**:
  - Not good enough to simply find production right-hand sides and reduce
  - *Example*:

Rule	*Sentential form*
-	***abbcde***
3	***aAbcde***
2	***aAAcde***
?	***…now what?***

#	*Production rule*
1	*G* → *a A B e*
2	*A* → A b c
3	| b
4	B → *d*

  - *"**aAAcde**" is not part of any sentential form*



Key problems

- How do we make this work?
  - How do we know we won't get stuck?
  - How do we find the next reduction?
  - Also: how do we find it *efficiently*?

- **Key**:
  - We are constructing the right-most derivation
  - Grammar is unambiguous
    - Unique right-most derivation for every string
    - Unique production applied at each forward step
    - Unique correct reduction at each backward step

Right-most derivation

Rule	*Sentential form*
-	*expr*
1	*expr op expr*
3	*expr op* <id,y>
6	*expr* * <id,y>
1	*expr op expr* * <id,y>
2	*expr op* <num,2> * <id,y>
5	*expr* - <num,2> * <id,y>
3	<id,x> - <num,2> * <id,y>

Rule	*Sentential form*
-	*expr*
1	*expr op expr*
3	*expr op* <id,y>
6	*expr* * <id,y>
1	*expr op expr* * <id,y>
2	*expr op* <num,2> * <id,y>
5	*expr* - <num,2> * <id,y>
3	<id,x> - <num,2> * <id,y>

- *Forward derivation:*
  - Always expand right-most non-terminal
- *Reverse derivation (parse):*
  - Correct reduction always occurs immediately to the left of some point in a left-to-right reading of the tokens

LR parsing

- State of the parser:

  $$\alpha \mid \gamma$$

  - α is a stack of terminals and non-terminals
  - γ is string of unexamined terminals

#	*Production rule*
1	*E* → *E* `+ (` *E* `)`
2	| `int`

- Two operations:
  - ***Shift*** – read next terminal, push on stack

    E + (| int) → **E + (int |)**
  - ***Reduce*** – pop RHS symbols off stack, push LHS

    E + (E + (E) |) → **E + (E |)**

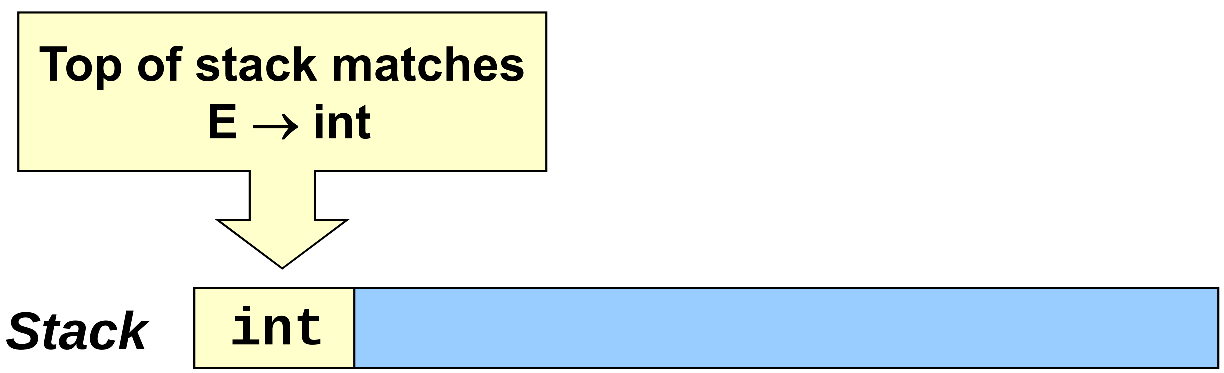
Example

1. | `int + ( int ) + ( int )` Nothing on stack, get next token

Stack

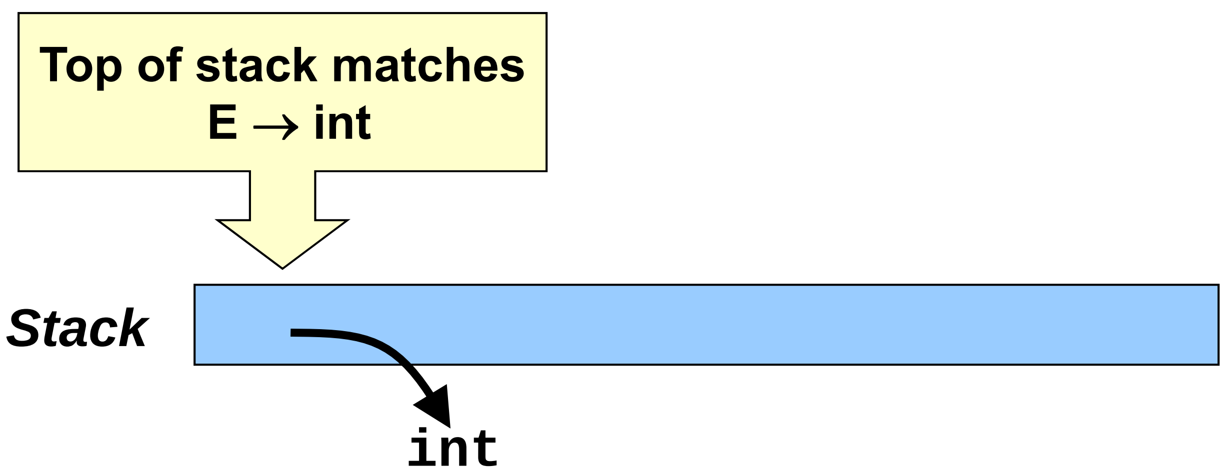
Example

1. | `int + ( int ) + ( int )` Nothing on stack, get next token
2. `int | + ( int ) + ( int )` *Shift*: push **`int`**



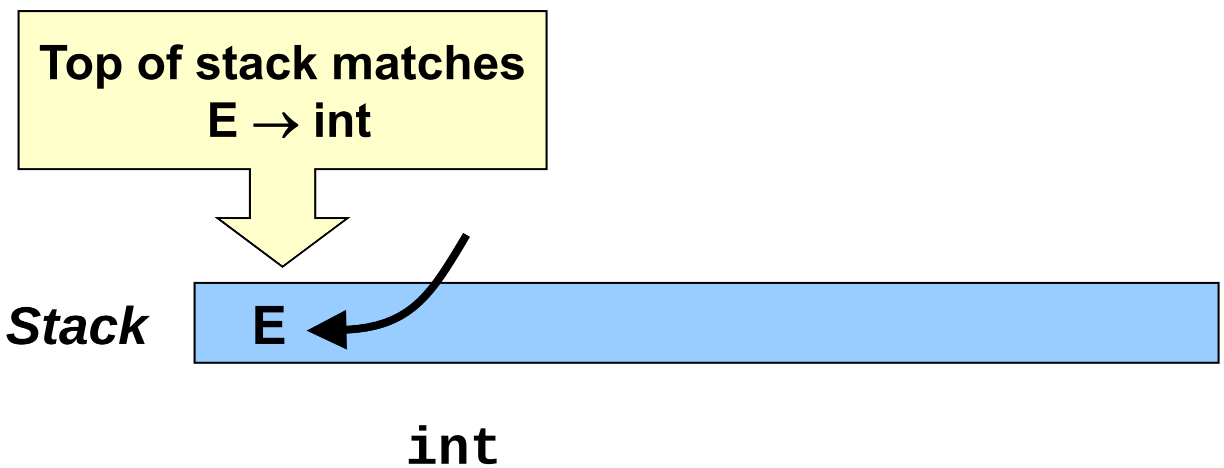
Example

1. `| int + ( int ) + ( int )` Nothing on stack, get next token
2. `int | + ( int ) + ( int )` ***Shift***: push **`int`**
3. `int | + ( int ) + ( int )` ***Reduce***: pop **`int`**



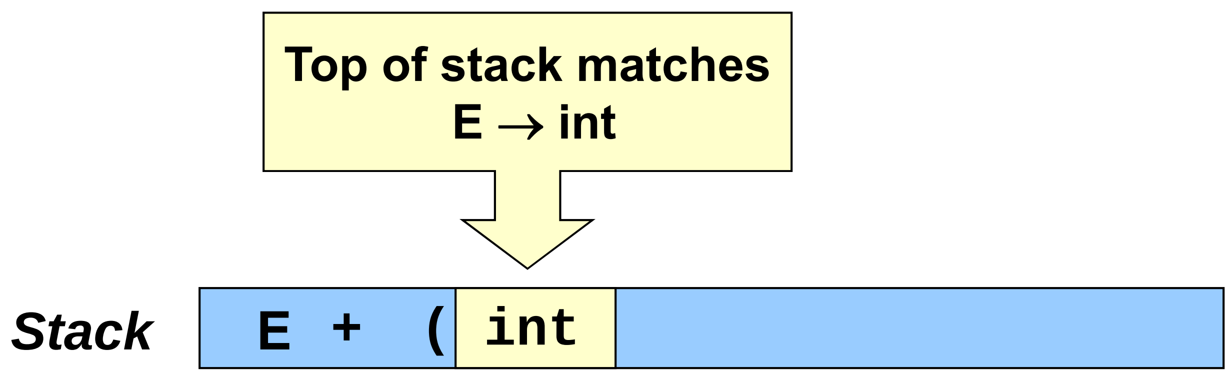
Example

1. `| int + ( int ) + ( int )` Nothing on stack, get next token
2. `int | + ( int ) + ( int )` ***Shift***: push **`int`**
3. `int | + ( int ) + ( int )` ***Reduce***: pop **`int`**, push E



Example

1. `| int + ( int ) + ( int )` Nothing on stack, get next token
2. `int | + ( int ) + ( int )` *Shift*: push **`int`**
3. `int | + ( int ) + ( int )` *Reduce*: pop **`int`**, push E
4. `int + | ( int ) + ( int )` *Shift*: push **+**
5. `int + ( | int ) + ( int )` *Shift*: push **(**
6. `int + (int | ) + ( int )` *Shift*: push **`int`**



Example

1. `| int + ( int ) + ( int )` — Nothing on stack, get next token
2. `int | + ( int ) + ( int )` — *Shift*: push **`int`**
3. `int | + ( int ) + ( int )` — *Reduce*: pop **`int`**, push E
4. `int + | ( int ) + ( int )` — *Shift*: push **`+`**
5. `int + ( | int ) + ( int )` — *Shift*: push **`(`**
6. `int + ( int | ) + ( int )` — *Shift*: push **`int`**
7. `int + ( int | ) + ( int )` — *Reduce*: pop **`int`**, push E

Stack `E + ( E`

Example

1.	`| int + ( int ) + ( int )`	Nothing on stack, get next token
2.	`int	+ (int) + (int)`	*Shift*: push **`int`**
3.	`int	+ (int) + (int)`	*Reduce*: pop **`int`**, push E
4.	`int +	(int) + (int)`	*Shift*: push **+**
5.	`int + ( | int ) + ( int )`	*Shift*: push **(**
6.	`int + ( int | ) + ( int )`	*Shift*: push **`int`**
7.	`int + ( int | ) + ( int )`	*Reduce*: pop **`int`**, push E
8.	`int + (int)	+ (int)`	*Shift*: push **)**

Top of stack matches E --> E + (E)

Stack **E + (E)**

Example

1.	`| int + ( int ) + ( int )`	Nothing on stack, get next token
2.	`int	+ (int) + (int)`	*Shift*: push **int**
3.	`int	+ (int) + (int)`	*Reduce*: pop **int**, push E
4.	`int +	(int) + (int)`	*Shift*: push **+**
5.	`int + ( | int ) + ( int )`	*Shift*: push **(**
6.	`int + ( int | ) + ( int )`	*Shift*: push **int**
7.	`int + ( int | ) + ( int )`	*Reduce*: pop **int**, push E
8.	`int + (int)	+ (int)`	*Shift*: push **)**
9.	`int + (int)	+ (int)`	*Reduce*: pop x 5, push E

Stack **E**

Example

….

9. `int + ( int ) | + ( int )` ***Reduce***: pop x 5, push E

10. `int + ( int ) + | ( int )` ***Shift***: push **+**

11. `int + ( int ) + ( | int )` ***Shift***: push **(**

12. `int + ( int ) + ( int | )` ***Shift***: push **`int`**

Stack `E + ( int`

Example

....

9. `int + ( int ) | + ( int )` *Reduce*: pop x 5, push E
10. `int + ( int ) + | ( int )` *Shift*: push **+**
11. `int + ( int ) + ( | int )` *Shift*: push **(**
12. `int + ( int ) + ( int | )` *Shift*: push **`int`**
13. `int + ( int ) + ( int | )` *Reduce*: pop **`int`**, push E
14. `int + ( int ) + ( int ) |` *Shift*: push **)**

Stack `E + ( E )`

Example

....

9. `int + ( int ) | + ( int )` *Reduce*: pop x 5, push E
10. `int + ( int ) + | ( int )` *Shift*: push **+**
11. `int + ( int ) + ( | int )` *Shift*: push **(**
12. `int + ( int ) + ( int | )` *Shift*: push **`int`**
13. `int + ( int ) + ( int | )` *Reduce*: pop **`int`**, push E
14. `int + ( int ) + ( int ) |` *Shift*: push **)**
15. `int + ( int ) + ( int ) |` *Reduce*: pop x 5, push E

DONE!

Stack **E**

Key problems

- (1) Will this work?

  How do we know that shifting and reducing using a stack is sufficient to compute the reverse derivation?

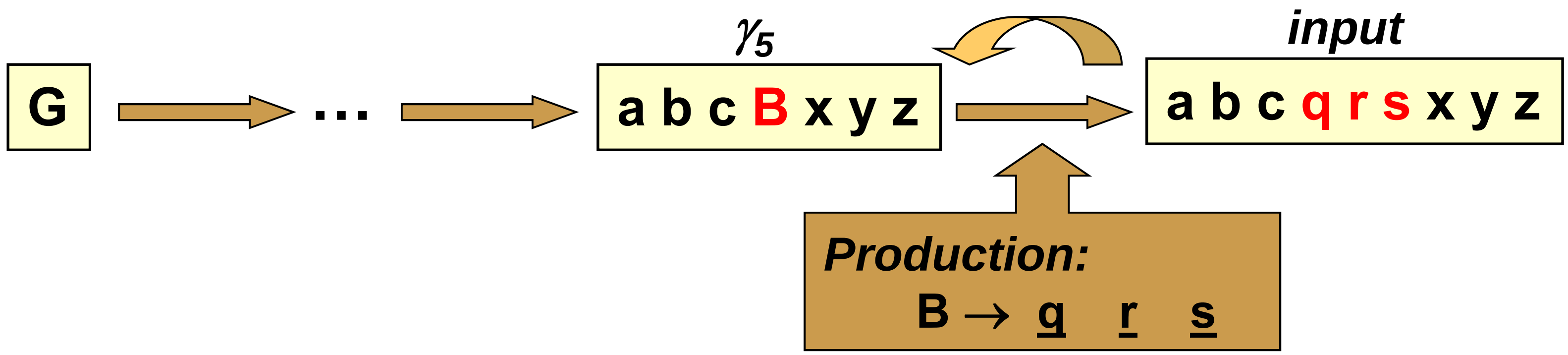
- (2) How do we know when to **shift** and **reduce**?
  - Can we efficiently match top symbols on the stack against productions?
    - *Right-hand sides of productions may have parts in common*
  - Will shifting a token move us closer to a reduction?
    - *Are we making progress?*
    - *How do we know when an error occurs?*

Why does it work?

- Right-most derivation

$$G \rightarrow \gamma_1 \rightarrow \gamma_2 \rightarrow \gamma_3 \rightarrow \gamma_4 \rightarrow \gamma_5 \rightarrow \textbf{input}$$

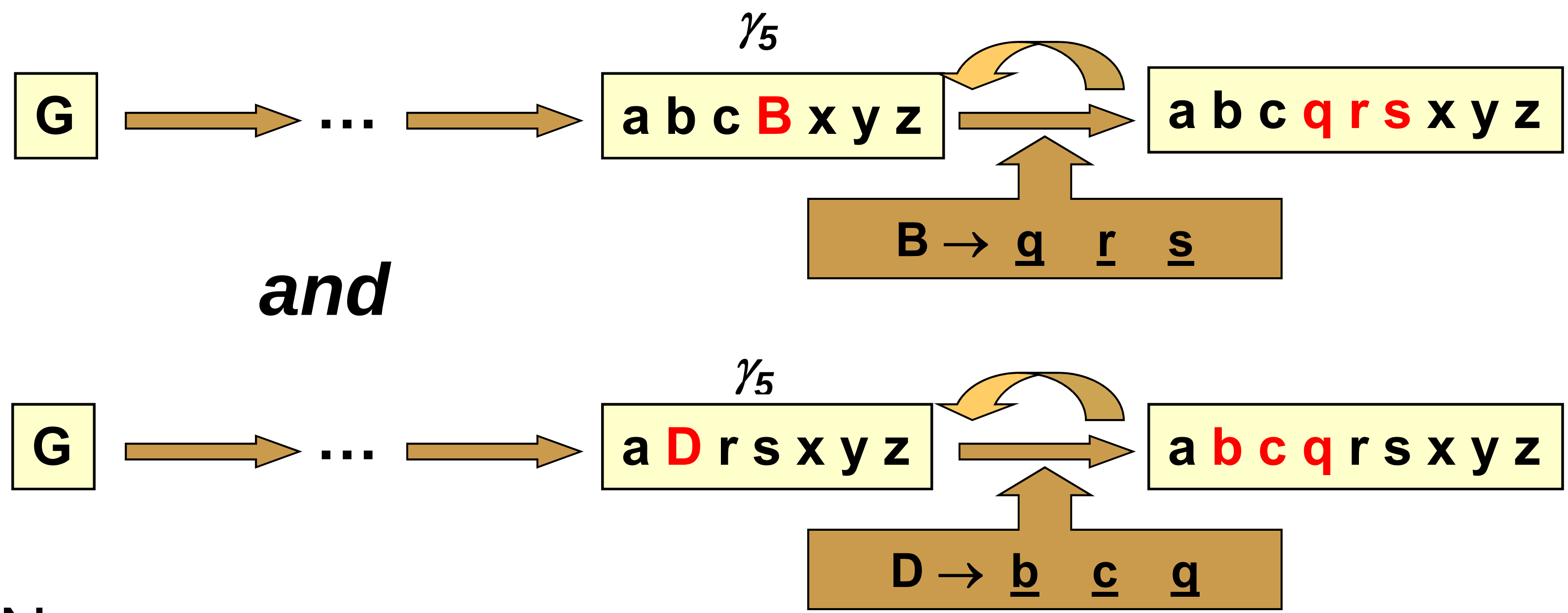
- Consider last step:



- To reverse this step:
  - Read input until **q**, **r**, **s** on top of stack
  - **Reduce** **q**, **r**, **s** to **B**

Right-most derivation

- Could there be an alternative reduction?



- No
  - Two right-most derivations for the same string
  - I.e., the grammar would be ambiguous

Reductions

- Where is the next reduction?
  - Includes **B:**

γ_5

a b c B x y z

… γ_4 a b F y z ← γ_5 a b c B x y z ←

  - Later in the input stream:

… γ_4 a b c B H z ← γ_5 a b c B x y z ←

  - Could it be earlier?

… γ_4 a J B x y z ← γ_5 a b c B x y z ←

    - No – this is not the right-most derivation!

Implications

- Cases:

γ_4: a b **F** y z ← γ_5: a b **c B x** y z

a b c B **H** z ← a b c B **x y** z

- Parsing state:
  - *Input:* a b c q r s | x y z
  - *Stack:* a b c B
- ***Key***: next reduction must consume top of stack

  Possibly after shifting some terminal symbols
- How does this help?
  - Can consume terminal symbols in order
  - Never need to search inside the stack

We can perform LR parsing using only stack operations

LR parsing

repeat

if top symbols on stack match β for some $A \rightarrow \beta$

Reduce: "*found an A*"

Pop those symbols off

Push A on stack

else Get next token from scanner

if token is useful

Shift: "*still working on something*"

Push on stack

else **error**

until stack contains goal **and**

no more input

Key problems

- (2) How do we know when to shift or reduce?
  - **Shifts**
    - Default behavior: shift when there's no reduction
    - Still need to handle errors
  - **Reductions**
    - Good news:
      - At any given step, reduction is unique
      - Matching production occurs at top of stack
    - ***Problem***:
      - How to efficiently find the right production

Identifying reductions

- Where is the next reduction?
  - Includes **B:**

γ_5

a b c B | x y z

… a b F y z (γ_4) a b c B x y z (γ_5)

  - Later in the input stream:

… a b c B H z (γ_4) a b c B x y | z (γ_5)

- What is on the stack?
  - Sequence of terminals and non-terminals
  - All applicable reductions, except last, already applied
  - Called a ***viable prefix***

Identifying reductions

- Do viable prefixes have any special properties?
- **Key**: viable prefixes are a *regular language*
- **Idea**: a DFA that recognizes viable prefixes
  - Input: stack contents
    (a mix of terminals, non-terminals)
  - Each state represents either
    - A right sentential form – labeled with the reduction to apply
    - A viable prefix – labeled with tokens to expect next

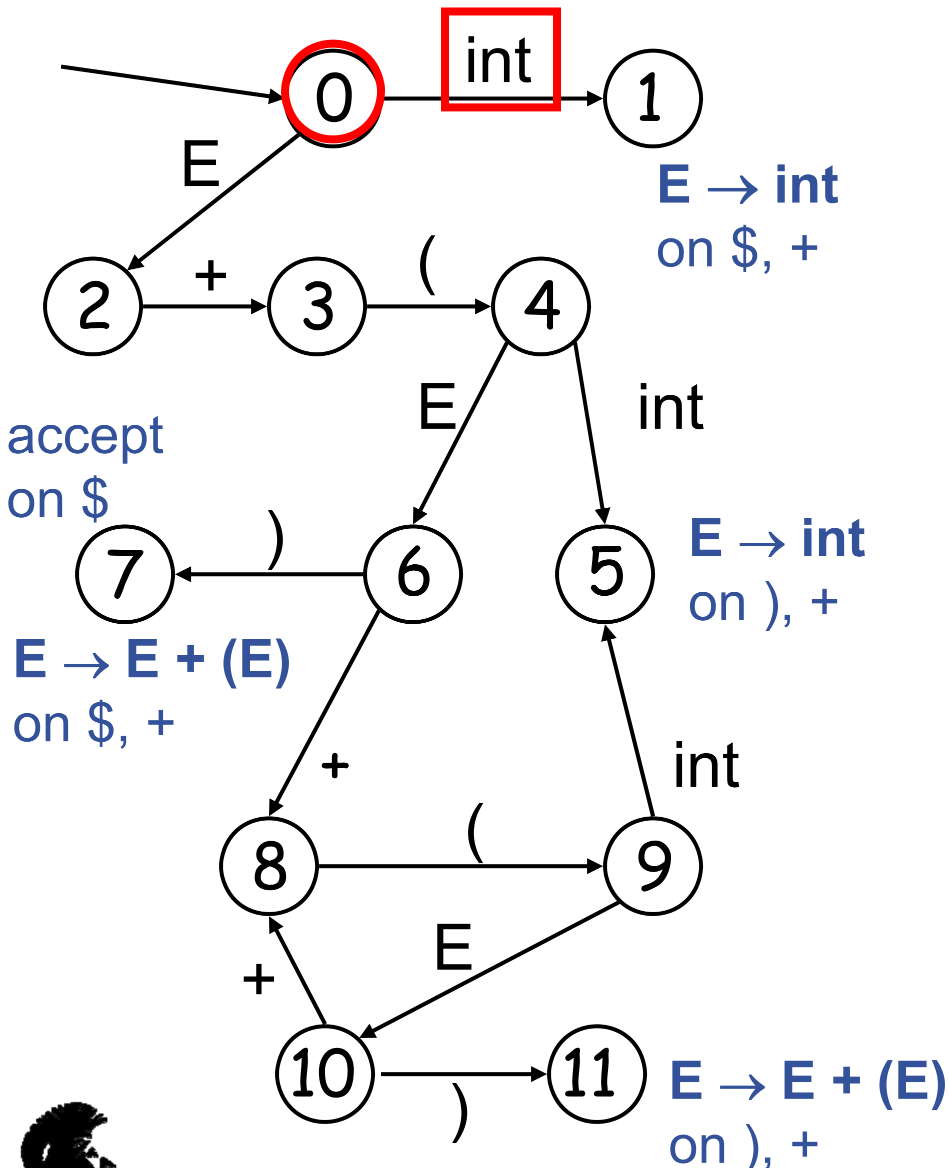
Shift/reduce DFA

- Using the DFA
  - At each parsing step run DFA on stack contents
  - Examine the resulting state X and the token **t** immediately following | in the input stream
    - If X has an outgoing edge labeled **t**, then **shift**
    - if X is labeled "**A** $\rightarrow$ β **on t**", then **reduce**
- Example:

#	*Production rule*
1	`E → E + ( E )`
2	`  | int`

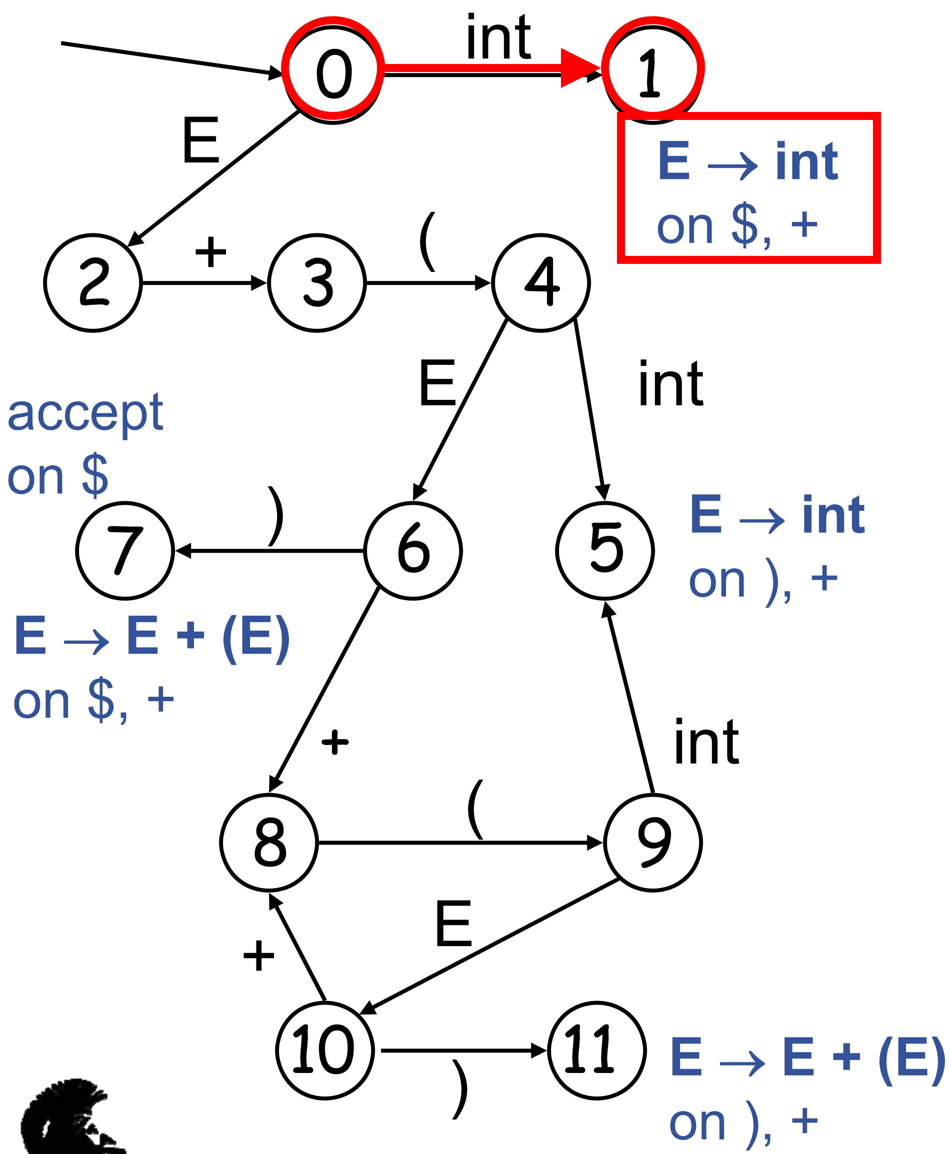
  - First, we'll look at how to use such a DFA…

Example



- int + (int) + (int)$

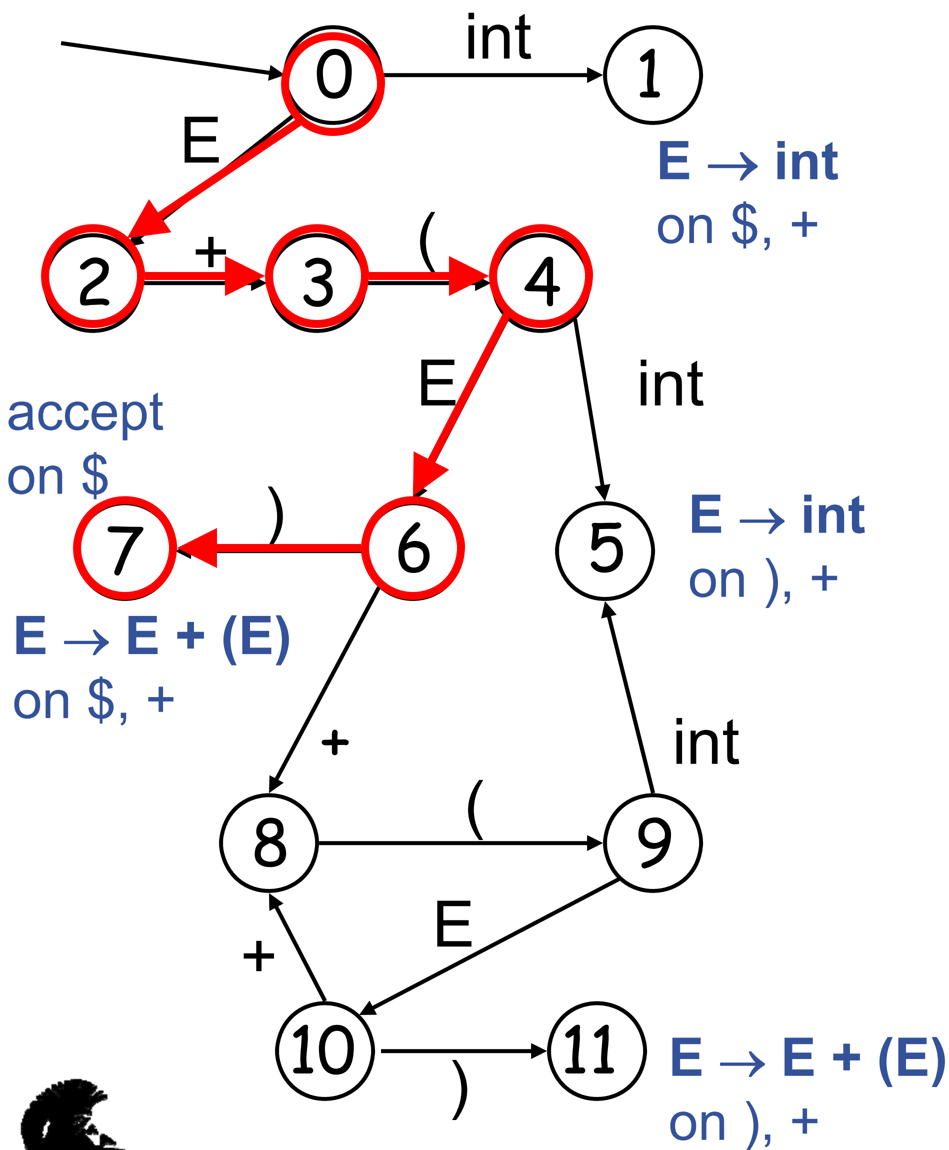
Example



▶ int + (int) + (int)$ shift

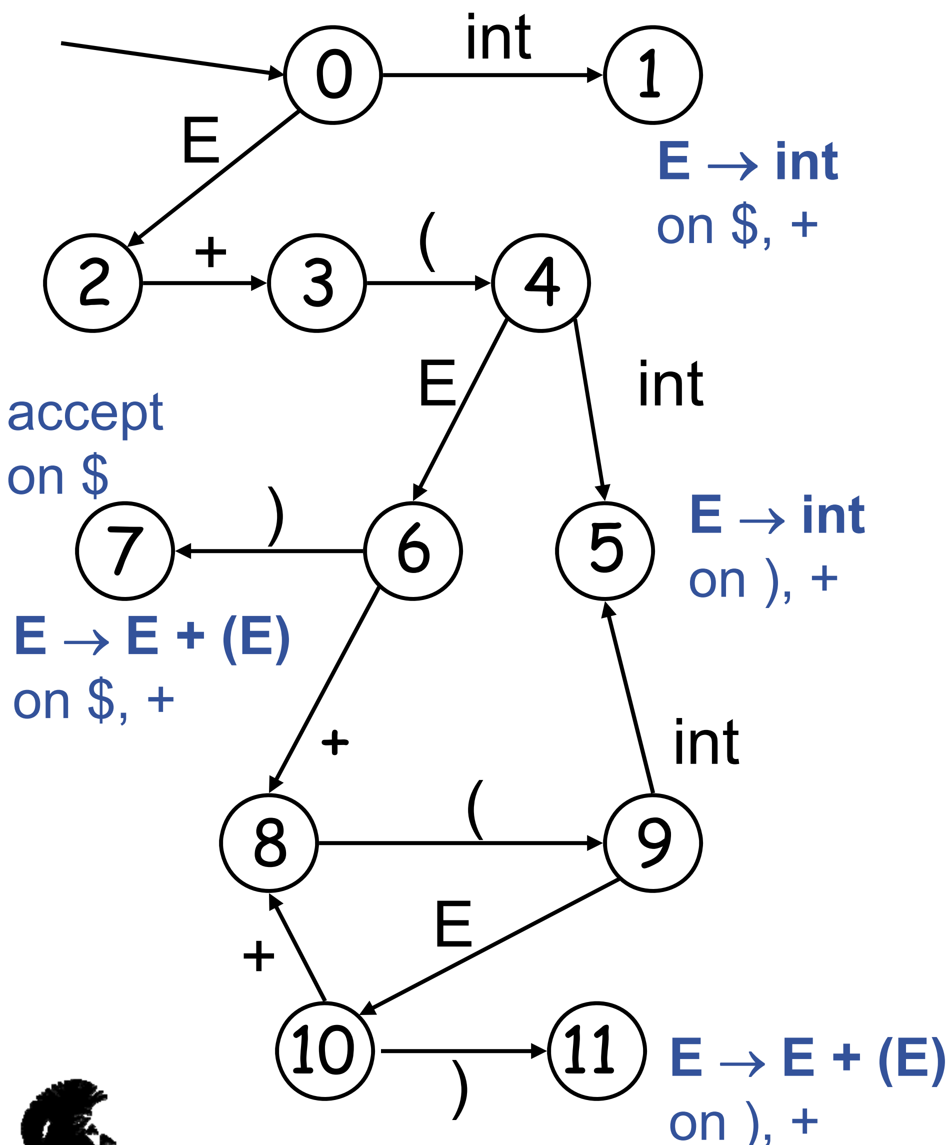
int ▶ + (int) + (int)$

Example



► int + (int) + (int)$	shift
int ► + (int) + (int)$	E --> int
E ► + (int) + (int)$	shift(x3)
E + (int ►) + (int)$	E --> int
E + (E ►) + (int)$	shift
E + (E) ► + (int)$	

Example

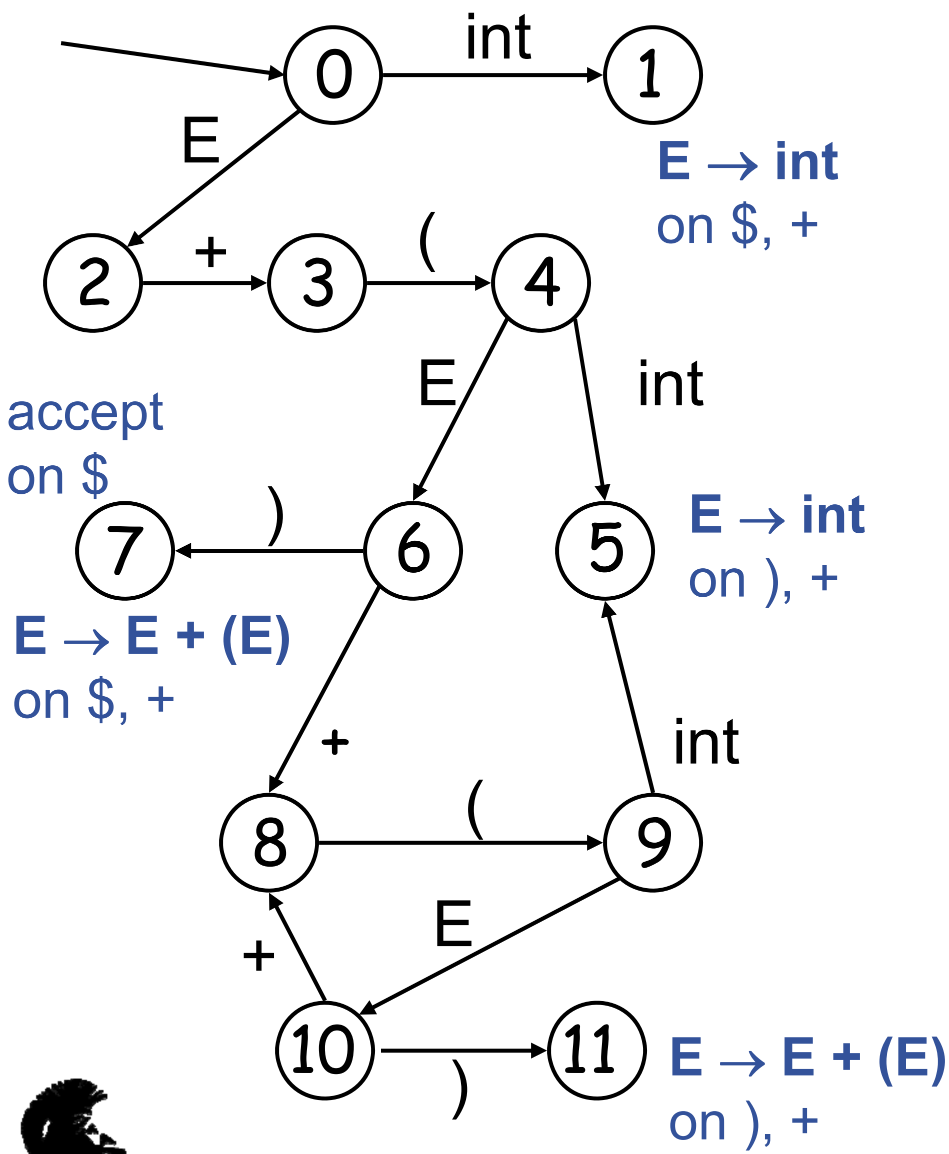


▶ int + (int) + (int)$	shift
int ▶ + (int) + (int)$	E --> int
E ▶ + (int) + (int)$	shift(x3)
E + (int ▶) + (int)$	E --> int
E + (E ▶) + (int)$	shift
E + (E) ▶ + (int)$	E --> E+(E)
E ▶ + (int)$	shift (x3)
E + (int ▶)$	E --> int
E + (E ▶)$	shift
E + (E) ▶ $	E --> E+(E)
E ▶ $	accept

Improvements

- Each DFA state represents stack contents
  - At each step, we rerun the DFA to compute the new state
  - Can we avoid this?
  - Two actions:
    - Shift: Push a new token
    - Reduce: Pop some symbols off, push a new symbol
- **Idea**:
  - For each symbol on the stack, remember the DFA state that represents the contents up to that point
    - Push a new token = go forward in DFA
    - Pop a sequence of symbols = "unwind" DFA to previous state

Example



▶ int + (int) + (int)$ shift

int ▶ + (int) + (int)$ E --> int

E ▶ + (int) + (int)$ shift(x3)

At state 2

go forward in DFA

2-->3-->4-->5

E + (int ▶) + (int)$ E --> int

Back up to state 4

Go forward with E

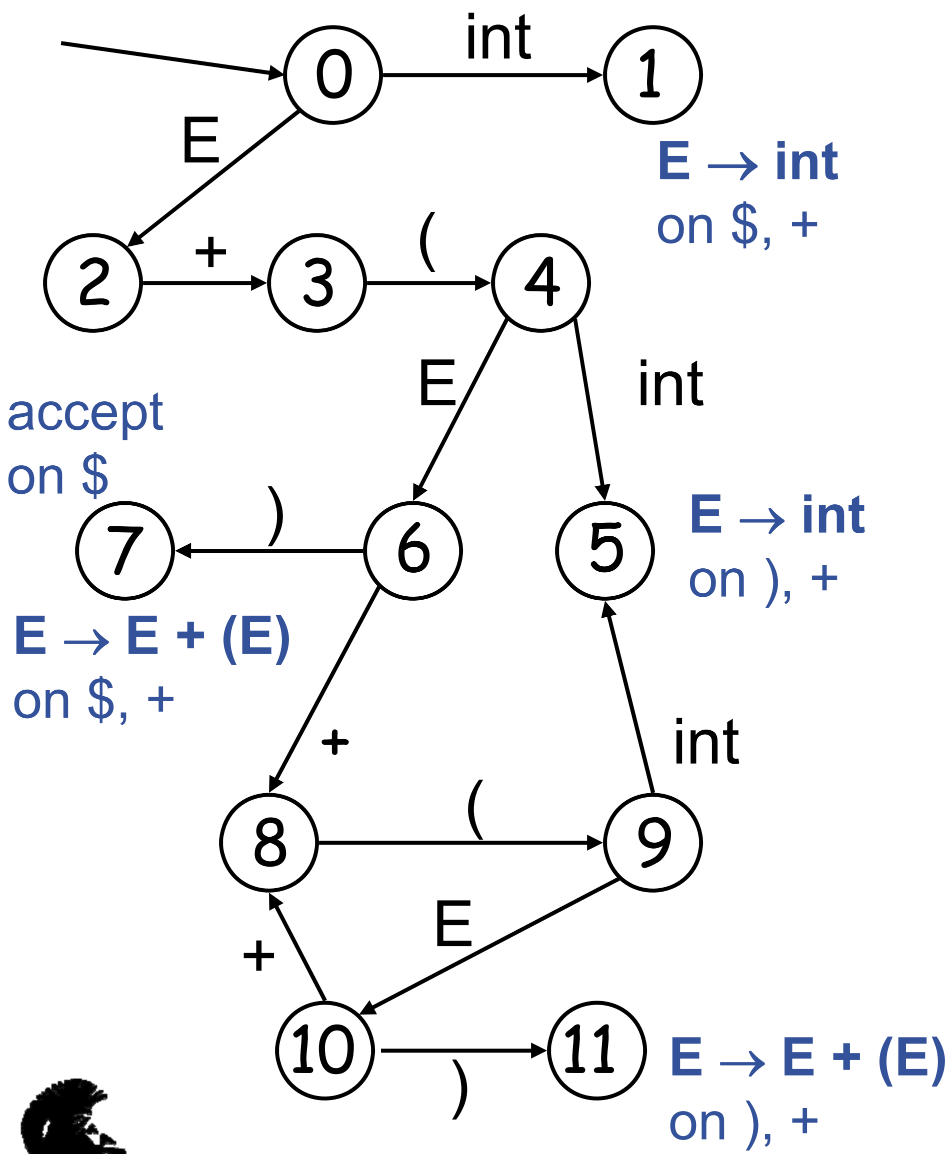
4-->6

Algorithm components

- Stack
  - String of the form : $\langle \mathrm{sym}_1, \mathrm{state}_1 \rangle \ldots \langle \mathrm{sym}_n, \mathrm{state}_n \rangle$
  - ***sym_i*** : grammar symbol (left part of string)
  - ***$state_i$*** : DFA state
    - Intuitively: represents what we've seen so far
    - state_k is the final state of the DFA on $\mathrm{sym}_1 \ldots \mathrm{sym}_k$
    - And, captures what we're looking for next

- Represent as two tables:
  - ***action*** – whether to shift, reduce, accept, error
  - ***goto*** – next state

Example



push <_,0>

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

push <int,1>

int | + (int) + (int)$ E --> int

pop <int, 1>

push <E, goto(0, E)=2>

E | + (int) + (int)$ shift(x3)

push <+,3>, <(,4>, <int,5>

E + (int |) + (int)$ E --> int

pop <int,5>

push <E, goto(4, E) = 6>

etc….

Tables

- **Action**

  Given state and the next token, ***action***$[s_i, \underline{a}]$ =
  - **Shift s'**, where s' is the next state on edge <u>a</u>
  - **Reduce** by a grammar production **A** → β
  - **Accept**
  - **Error**

- **Goto**

  Given a state and a grammar symbol, ***goto***$[s_i, X]$ =
  - After reducing an X production
  - Unwind to state ending with X (to keep going)

Algorithm

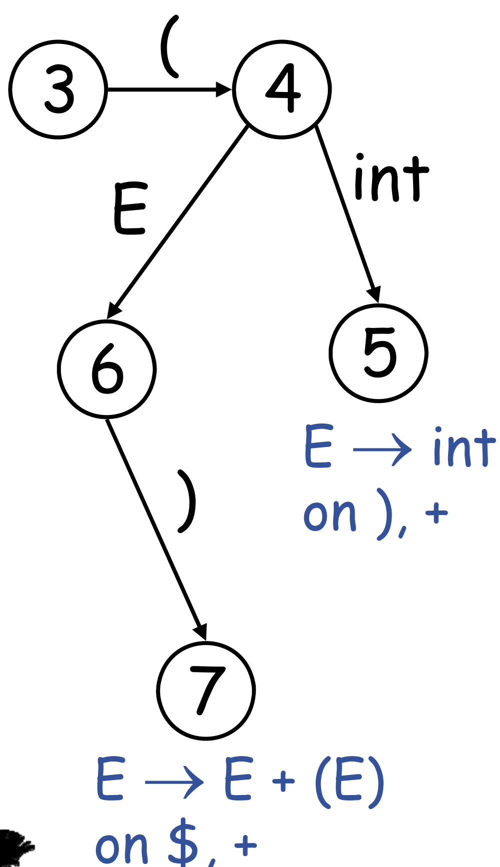
```
push s0 on stack
token = scanner.next_token()
repeat
    s = state at top of stack
    if action[s, token] = reduce A → β then
        pop | β |  pairs (X_i, s_m) off the stack
        s' = top of stack
        push A on stack
        push goto[s', A] on stack
    else if action[s, token] = shift s' then
        push token on stack
        push s' on stack
        token = scanner.next_token()
    else if action(s, token) = accept  then
        return true
    else error()
```

Top of stack is handle $A \rightarrow \beta$

- Work
  - Shift each token
  - Pop each token
- Errors
  - Input exhausted
  - Error entry in table

Representing the DFA

- Combined table:



	action(state, token)					*goto*
	int	+	(	)	$	E
…						
3			s4			
4	s5					g6
5		$r_{E\text{-->int}}$		$r_{E\text{-->int}}$		
6	s8		s7			
7		$r_{E\text{-->E+(E)}}$			$r_{E\text{-->E+(E)}}$	
…						

How is the DFA Constructed?

- What's on the stack?
  - Viable prefix – a piece of a sentential form

    E + (

    E + (int

    E + (E + (

  - **Idea**: we're part-way through some production
  - **Problem**: Productions can share pieces
  - DFA state represents the set of candidate productions
    - Represents all the productions we ***could be*** working on
    - Notation: ***LR(1) item*** shows where we are and what we need to see

LR Items

- An ***LR(1) item*** is a pair:

  $$[\ A \rightarrow \alpha \bullet \beta,\ \underline{a}\ ]$$

  - $A \rightarrow \alpha\beta$ is a production
  - $\underline{a}$ is a terminal (the lookahead terminal)
  - LR(1) means 1 lookahead terminal

- $[A \rightarrow \alpha \bullet \beta,\ \underline{a}]$ describes a context of the parser
  - We are trying to find an **A** followed by an $\underline{a}$, and
  - We have seen an α
  - We need to see a string derived from $\beta\ \underline{a}$

LR Items

- In context containing

$$[\ E \rightarrow E + \bullet\ (\ E\ ),\ +\ ]$$

  - If **“(“** is next then we can a ***shift*** to context containing

$$[\ E \rightarrow E + (\ \bullet\ E\ ),\ +\ ]$$

- In context containing

$$[\ E \rightarrow E + (\ E\ )\ \bullet,\ +\ ]$$

  - We can ***reduce*** with $E \rightarrow E + (\ E\ )$
  - But only if a “**+**” follows

LR Items

- Consider the item

$$E \rightarrow E + (\bullet E) , +$$

  - We expect a string derived from E) +
  - There are two productions for E

$$E \rightarrow \text{int} \quad \textit{and} \quad E \rightarrow E + (E)$$

- We extend the context with two more items:

$$E \rightarrow \bullet \text{ int},)$$
$$E \rightarrow \bullet E + (E) ,)$$

- Each DFA state:

  The set of items that represent all the possible productions we could be working on – called the ***closure*** of the set of items

Closure operation

- Observation:
  - At $A \rightarrow \alpha \bullet B\beta$ we expect to see $B\beta$ next
  - Means if $B \rightarrow \gamma$ is a production, then we could see a γ
- ***Algorithm:***

  closure(*Items*) =

  repeat

  for each $[A \rightarrow \alpha \bullet B\beta, \underline{a}]$ in *Items*

  for each production $B \rightarrow \gamma$

  add $[B \rightarrow \bullet\gamma, ?]$ to *Items*

  until *Items* is unchanged

What is the lookahead?

Closure operation

- **Algorithm:**

closure(*Items*) =

repeat

for each $[A \to \alpha \bullet B\beta, \underline{a}]$ in *Items*

for each production $\mathbf{B} \to \gamma$

for each $\underline{\mathbf{b}} \in$ **FIRST**$(\beta\underline{\mathbf{a}})$

add $[B \to \bullet\gamma, \underline{b}]$ to *Items*

until *Items* is unchanged

Building the DFA – part 1

- Starting context = ***closure***({**S → • E, $**})

$$
\begin{aligned}
&S \rightarrow \bullet E,\ \$ \\
&E \rightarrow \bullet E{+}(E),\ \$ \\
&E \rightarrow \bullet \mathrm{int},\ \$ \\
&E \rightarrow \bullet E{+}(E),\ + \\
&E \rightarrow \bullet \mathrm{int},\ +
\end{aligned}
$$

- Abbreviated:

$$
\begin{aligned}
&S \rightarrow \bullet\ E,\ \$ \\
&E \rightarrow \bullet\ E{+}(E),\ \$/+ \\
&E \rightarrow \bullet\ \mathrm{int},\ \$/+
\end{aligned}
$$

Building the DFA – part 2

- DFA states
  - Each DFA state is a closed set of LR(1) items
  - Start state: **closure**({**S → • E, $**})

- Reductions
  - Label each item **[A → αβ •, <u>x</u>]** with "Reduce with **A** → αβ on lookahead **<u>x</u>**"

- What about transitions?

DFA transitions

- **Idea**:
  - If the parser was in state **[A $\rightarrow$ α•Xβ]** and then recognized an instance of X, then the new state is **[A $\rightarrow$ αX•β]**
  - *Note*: X could be a terminal or non-terminal

- **Algorithm**:

Given a set of items I (DFA stats) and a symbol X

transition(I, X) =

J = {}

for each **[A $\rightarrow$ α•Xβ, $\underline{b}$]** $\in$ I

add **[A $\rightarrow$ αX•β , $\underline{b}$]** to J

return ***closure***(J)

DFA construction

- Data structure:
  - **T** – set of states (each state is a set of items)
  - **E** – edges of the form $I \xrightarrow{X} J$

    where I, J ∈ T and X is a terminal or non-terminal

- **Algorithm**:

  T = {**closure**({**S → • Y, $**}, E = {}

  repeat

  for each state I in T

  for each item **[A → α•Xβ, b]** ∈ I

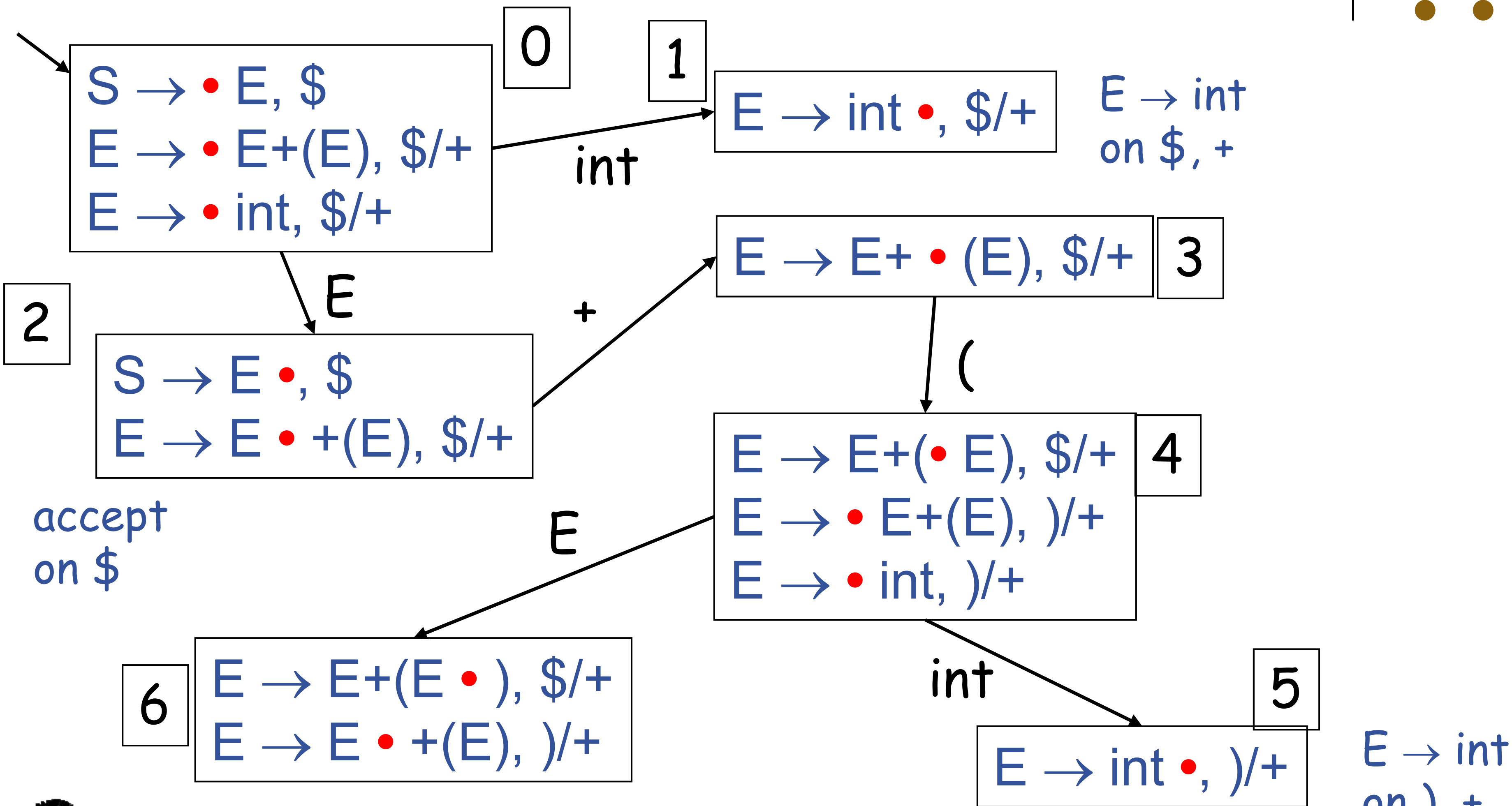
  let J = ***transition***(I, **X**)

  T = T + J

  E = E + { I → J }

  until E and T no longer change

Example DFA



and so on…

To form into tables

- Two tables
  - action(I, token)
  - goto(I, symbol)
- Layout:
  - One row for each state – each I in T
  - One column for each symbol
- Entries:
  - For each edge $\mathbf{I} \xrightarrow{\mathbf{X}} \mathbf{J}$
    - If X is a terminal, add shift J at position (I, X) in action
    - if X is a non-terminal, add goto J at position (I, X) goto
  - For each state **[A** $\rightarrow \alpha\beta$ **•, x]** in I
    - Add reduce n at position (I, x) in action (where n is |rhs|)

Issues with LR parsers

- What happens if a state contains:
  [$\mathbf{X} \to \alpha \bullet \underline{\mathbf{a}}\beta$, $\underline{\mathbf{b}}$] and [$\mathbf{Y} \to \gamma \bullet$, $\underline{\mathbf{a}}$]

- Then on input "<u>a</u>" we could either
  - Shift into state [$\mathbf{X} \to \alpha\underline{\mathbf{a}} \bullet \beta$, $\underline{\mathbf{b}}$], or
  - Reduce with $\mathbf{Y} \to \gamma$

- This is called a ***shift-reduce conflict***
  - Typically due to ambiguity
  - Like what?

Shift/Reduce conflicts

- Classic example: the dangling else

  S → if E then S | if E then S else S | OTHER

- Will have DFA state containing

  [S → if E then S •, else]

  [S → if E then S • else S, x]

- Practical solutions:
  - Painful: modify grammar to reflect the precedence of else
  - Many LR parsers default to “shift”
  - Often have a precedence declaration

Another example

- Consider the ambiguous grammar

  $E \rightarrow E + E \mid E * E \mid \text{int}$

- Part of the DFA:

$$\boxed{\begin{array}{l}[E \rightarrow E * \bullet E,\ +] \\ [E \rightarrow \bullet E + E,\ +] \\ \quad \ldots\end{array}} \xrightarrow{\quad E \quad} \boxed{\begin{array}{l}[E \rightarrow E * E \bullet,\ +] \\ [E \rightarrow E \bullet + E,\ +] \\ \quad \ldots\end{array}}$$

- We have a shift/reduce on input +
- What do we want to happen?
  - Consider: **x * y + z**
  - We *need* to reduce (* binds more tightly than +)
  - Default action is shift

Precedence

- Declare relative precedence
  - Explicitly resolve conflict
  - Tell parser: we prefer the action involving ***** over **+**

$[E \rightarrow E * \bullet E,\ +]$	$\xrightarrow{E}$	$[E \rightarrow E * E \bullet,\ +]$
$[E \rightarrow \bullet E + E,\ +]$		$[E \rightarrow E \bullet + E,\ +]$

- In practice:
  - Parser generators support a precedence declaration for operators
  - What is the alternative?

More…

- Still a problem?

$$\begin{array}{|l|}\hline [E \to E + \bullet E,\ \ +] \\ [E \to \bullet E + E,\ \ +] \\ \hline \end{array} \xrightarrow{\quad E \quad} \begin{array}{|l|}\hline [E \to E + E \bullet,\ \ +] \\ [E \to E \bullet + E,\ \ +] \\ \hline \end{array}$$

- Shift/reduce conflict on **+**
  - Do we care?
  - Maybe: we want left associativity
    parse: "a+b+c" as "((a+b)+c)"
  - Which rule should we choose?
  - Also handled by a declaration "**+** is left-associative"

Other problems

- If a DFA state contains both

  [$X \rightarrow \alpha \bullet$, a] and [$Y \rightarrow \beta \bullet$, a]

  - What's the problem here?
  - Two reductions to choose from when next token is a

- This is called a ***reduce/reduce*** conflict
  - Usually a serious ambiguity in the grammar
  - Must be fixed in order to generate parser
  - Think about relationship between α and β

Reduce/Reduce conflicts

- Example: a sequence of identifiers

  $S \to \varepsilon \ | \ \text{id} \ | \ \text{id } S$

- There are two parse trees for the string id

  $S \to \text{id}$

  $S \to \text{id } S \to \text{id}$

- How does this confuse the parser?

Reduce/Reduce conflicts

- Consider the DFA states:

$[G \to \bullet\, S,\quad \$]$
$[S \to \bullet,\quad \$]$
$[S \to \bullet\, \underline{id},\quad \$]$
$[S \to \bullet\, \underline{id}\; S,\quad \$]$

$\xrightarrow{\underline{id}}$

$[S \to \underline{id}\, \bullet,\quad \$]$
$[S \to \underline{id}\, \bullet\, S,\quad \$]$
$[S \to \bullet,\quad \$]$
$[S \to \bullet\, \underline{id},\quad \$]$
$[S \to \bullet\, \underline{id}\; S,\quad \$]$

- Reduce/reduce conflict on input \$

$$G \to S \to id$$

$$G \to S \to id\ S \to id$$

- Fix: rewrite the grammar: $S \to \varepsilon \mid id\ S$

Practical issues

We use an LR parser generator…

- Question: how many DFA states are there?
  - Does it matter?
  - What does that affect?
    - Parsing time is the same
    - Table size: occupies memory
- Even simple languages have 1000s of states

  Most LR parser generators don't construct the DFA as described

LR(1) Parsing tables

- But many states are similar, e.g.

$\boxed{1}$ $\boxed{E \rightarrow \text{int} \bullet,\ \$/+}$ $E \rightarrow \text{int}$ on $\$$, $+$

and

$\boxed{5}$ $\boxed{E \rightarrow \text{int} \bullet,\ )/+}$ $E \rightarrow \text{int}$ on $)$, $+$

- How can we exploit this?
  - Same reduction, different lookahead tokens
  - **Idea**: merge the states…

$\boxed{1'}$ $\boxed{E \rightarrow \text{int} \bullet,\ \$/+/)}$ $E \rightarrow \text{int}$ on $\$$, $+$, $)$

The core of a set of LR Items

- When can states be merged?

- *Def*: the ***core*** of a set of LR items is:
  - Just the production parts of the items
  - Without the lookahead terminals

- Example: the core of

$$\{\, [\mathbf{X} \rightarrow \alpha \bullet \beta,\ \underline{\mathbf{b}}],\ [\mathbf{Y} \rightarrow \gamma \bullet \delta,\ \underline{\mathbf{d}}] \,\}$$

  is

$$\{\, \mathbf{X} \rightarrow \alpha \bullet \beta,\ \ \mathbf{Y} \rightarrow \gamma \bullet \delta \,\}$$

Merging states

- Consider for example the LR(1) states

$$\{\, [X \to \alpha\bullet,\ \underline{a}],\ [Y \to \beta\bullet,\ \underline{c}] \,\}$$

$$\{\, [X \to \alpha\bullet,\ \underline{b}],\ [Y \to \beta\bullet,\ \underline{d}] \,\}$$

- They have the same core and can be merged
- Resulting state is:

$$\{\, [X \to \alpha\bullet,\ \underline{a}/\underline{b}],\ [Y \to \beta\bullet,\ \underline{c}/\underline{d}] \,\}$$

Does this state do the same thing?

- These are called ***LALR(1) states***
  - Stands for LookAhead LR
  - Typically 10X fewer LALR(1) states than LR(1)

The LALR(1) DFA

- **Algorithm**:

  repeat

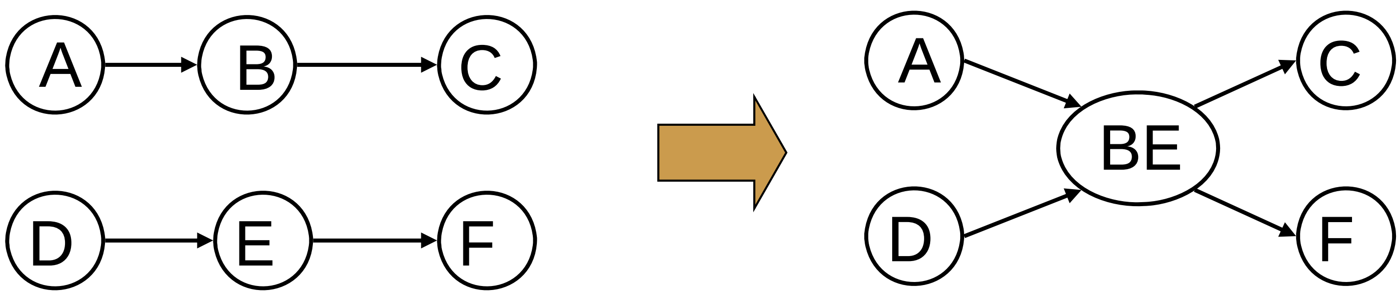
  Choose two states with same core

  Merge the states by combining the items

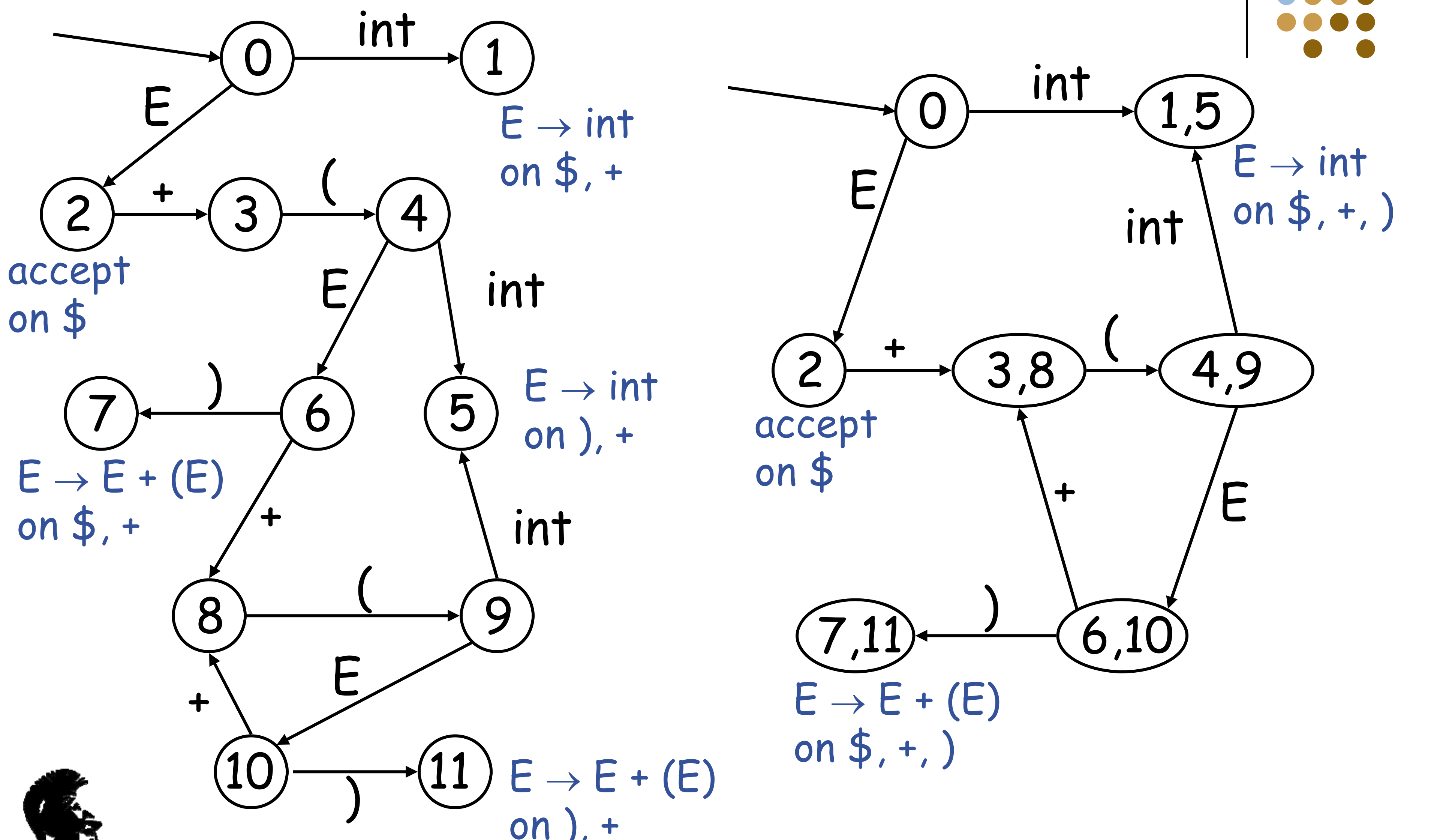
  Point edges from predecessors to new state

  New state points to all the previous successors

  until all states have distinct core



Conversion LR(1) to LALR(1).



LALR states

- Consider the LR(1) states:

  $\{\ [X \rightarrow \alpha\bullet,\ \underline{a}],\ [Y \rightarrow \beta\bullet,\ \underline{b}]\ \}$

  $\{\ [X \rightarrow \alpha\bullet,\ \underline{b}],\ [Y \rightarrow \beta\bullet,\ \underline{a}]\ \}$

- And the merged LALR(1) state

  $\{\ [X \rightarrow \alpha\bullet,\ \underline{a}/\underline{b}],\ [Y \rightarrow \beta\bullet,\ \underline{a}/\underline{b}]\ \}$

- What's wrong with this?
  - Introduces a new reduce-reduce conflict
  - In practice such cases are rare

LALR vs. LR Parsing

- LALR is an efficiency hack on LR languages
- Any "reasonable" programming language has a LALR(1) grammar
  - Languages that are not LALR(1) are weird, unnatural languages
- LALR(1) has become a standard for programming languages and for parser generators

Another variation

- Lookahead symbol
  - How is it computed in LR, LALR parser?
  - In closure operation

    for each $[A \to \alpha \bullet B\beta, \underline{a}]$ in *Items*

    for each production $B \to \gamma$

    for each $\underline{b} \in$ **FIRST**$(\beta\underline{a})$

    add $[B \to \bullet\gamma, \underline{b}]$ to *Items*
  - Based on context of use

- Simplify this process:
  - What symbol (set of symbols) could I use for $[B \to \bullet\gamma, \underline{?}]$
  - FOLLOW(B)
  - Called ***SLR*** (Simple LR) parser

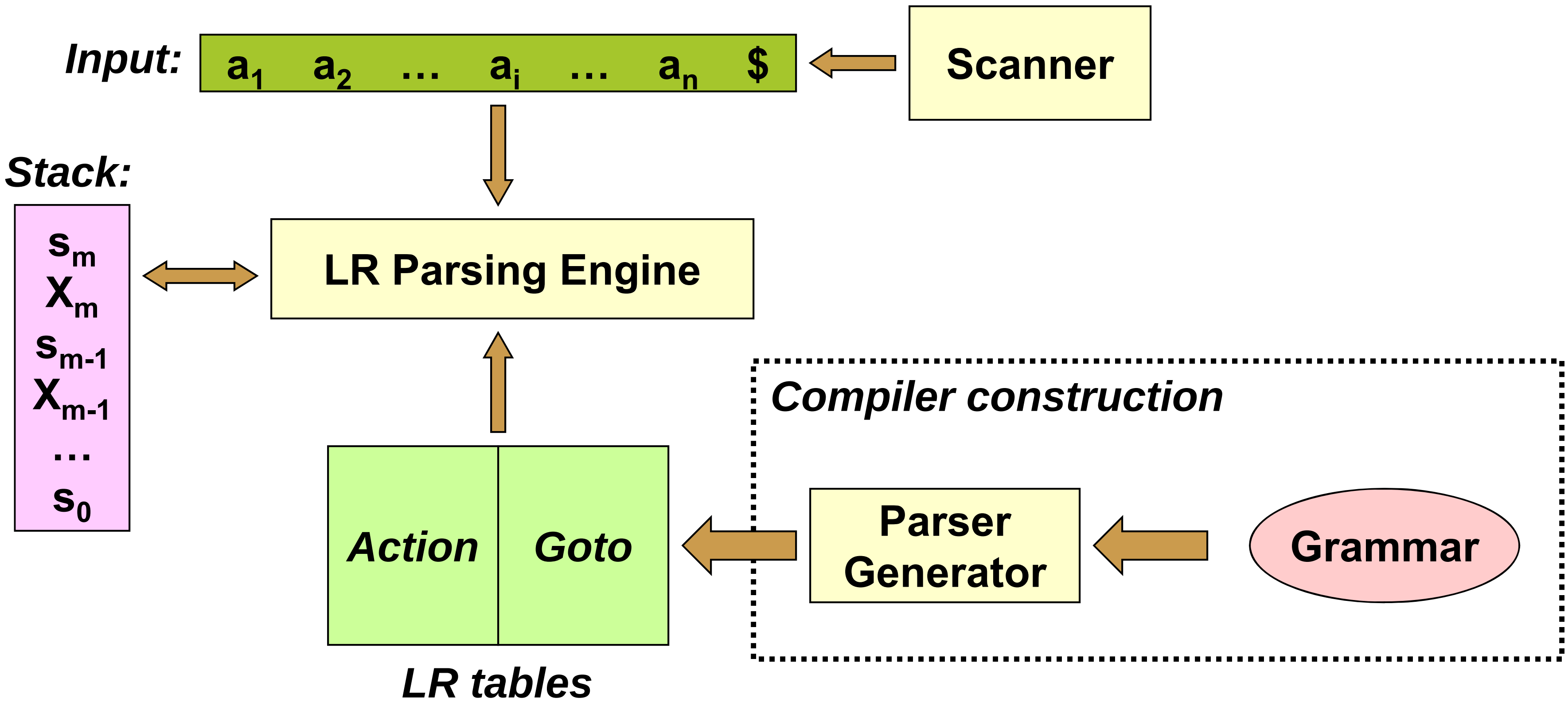
More **power?**

- So far:
  - LALR and SLR: reduce size of tables
  - Also reduce space of languages
  - What if I want to ***expand*** the space of languages?
- What could I do at a reduce/reduce conflict?
  - Try both reductions!
  - GLR parsing
    - At a choice: split the stack, explore both possibilities
    - If one doesn't work out, kill it
  - Run-time proportional to "amount of ambiguity"
  - Must design the stack data structure very carefully

General algorithms

- Parsers for full class of context-free grammars
  - Mostly used in linguistics – constructive proof of decidability
- CYK (1965)
  - Bottom-up ***dynamic programming*** algorithm
  - $O(n^3)$
- Earley's algorithm (1970)
  - Top-down dynamic programming algorithm
  - Developed the "•" notation for partial production
  - Worst-case $O(n^3)$ running time
  - But, $O(n^2)$ even for unambiguous grammars
- GLR
  - Worse-case $O(n^3)$, but $O(n)$ for unambiguous grammars

LR parsing



Input:
a1 a2 … ai … an $
Scanner
Stack:
sm
Xm
sm-1
Xm-1
…
s0
LR Parsing Engine
Compiler construction
Action
Goto
Parser Generator
Grammar
LR tables

Real world parsers

- Real generated code
  - lex, flex, yacc, bison
- Interaction between lexer and parser
  - C typedef problem
  - Merging two languages
- Debugging
  - Diagnosing reduce/reduce conflicts
  - How to step through an LR parser

Parser generators

- Example: JavaCUP
  - LALR(1) parser generator
  - Input: grammar specification
  - Output: Java classes
    - Generic engine
    - Action/goto tables
- Separate scanner specification
- Similar tools:
  - SableCC
  - yacc and bison generate C/C++ parsers
  - JavaCC: similar, but generates LL(1) parser

JavaCUP example

- Simple expression grammar
  - Operations over numbers only

```
// Import generic engine code
import java_cup.runtime.*;

/* Preliminaries to set up and use the scanner. */
init with {: scanner.init(); :};
scan with {: return scanner.next_token(); :};
```

  - Note: interface to scanner

    One issue: how to agree on names of the tokens

Example

- Define terminals and non-terminals
- Indicate operator precedence

```
/* Terminals (tokens returned by the scanner). */
terminal SEMI, PLUS, MINUS, TIMES, DIVIDE, MOD;
terminal UMINUS, LPAREN, RPAREN;
terminal Integer NUMBER;

/* Non terminals */
non terminal         expr_list, expr_part;
non terminal Integer expr, term, factor;

/* Precedences */
precedence left PLUS, MINUS;
precedence left TIMES, DIVIDE, MOD;
```

Example

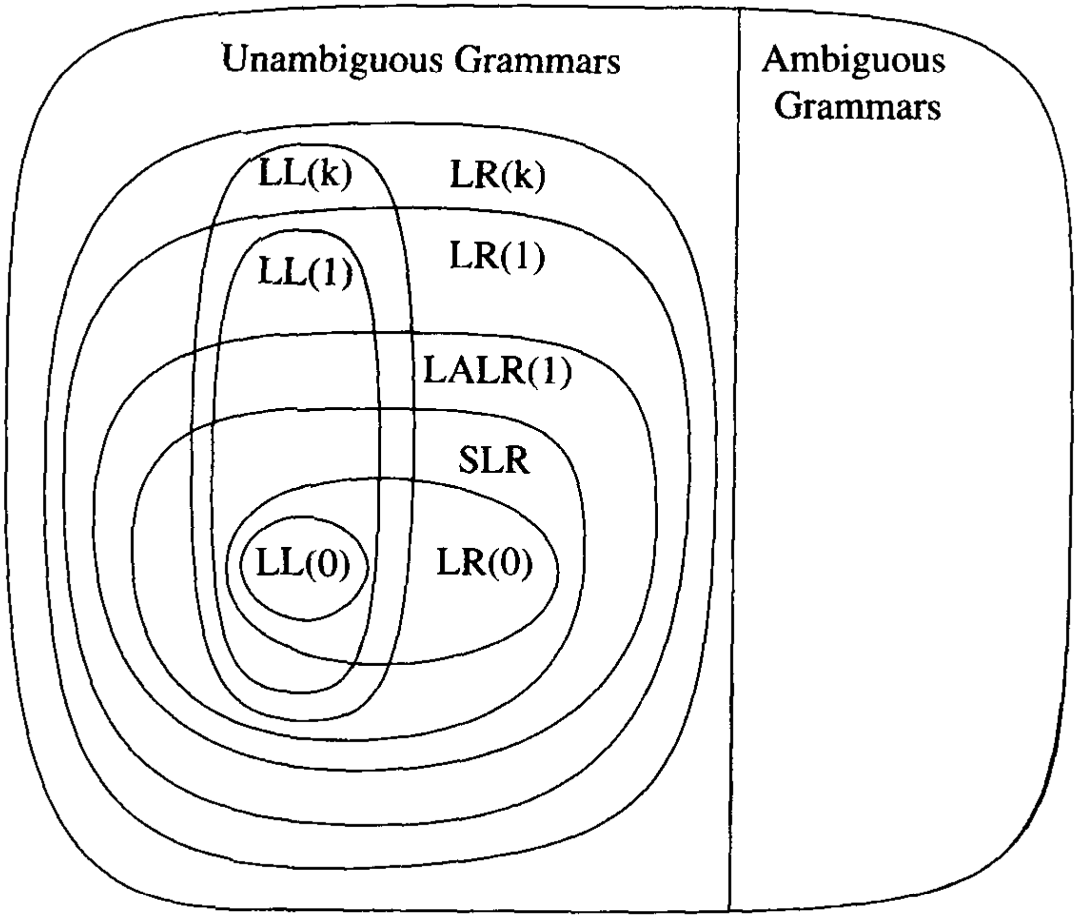
- Grammar rules

```
expr_list ::= expr_list expr_part
            |  expr_part ;

expr_part ::= expr SEMI ;

expr ::= expr PLUS expr
       | expr MINUS expr
       | expr TIMES expr
       | expr DIVIDE expr
       | expr MOD expr
       | LPAREN expr RPAREN
       | NUMBER ;
```

Summary of parsing

- Parsing
  - A solid foundation: context-free grammars
  - A simple parser: LL(1)
  - A more powerful parser: LR(1)
  - An efficiency hack: LALR(1)
  - LALR(1) parser generators

A Hierarchy of Grammar Classes



From Andrew Appel, "Modern Compiler Implementation in Java"