Regular Exp for Lexical Syntax ?

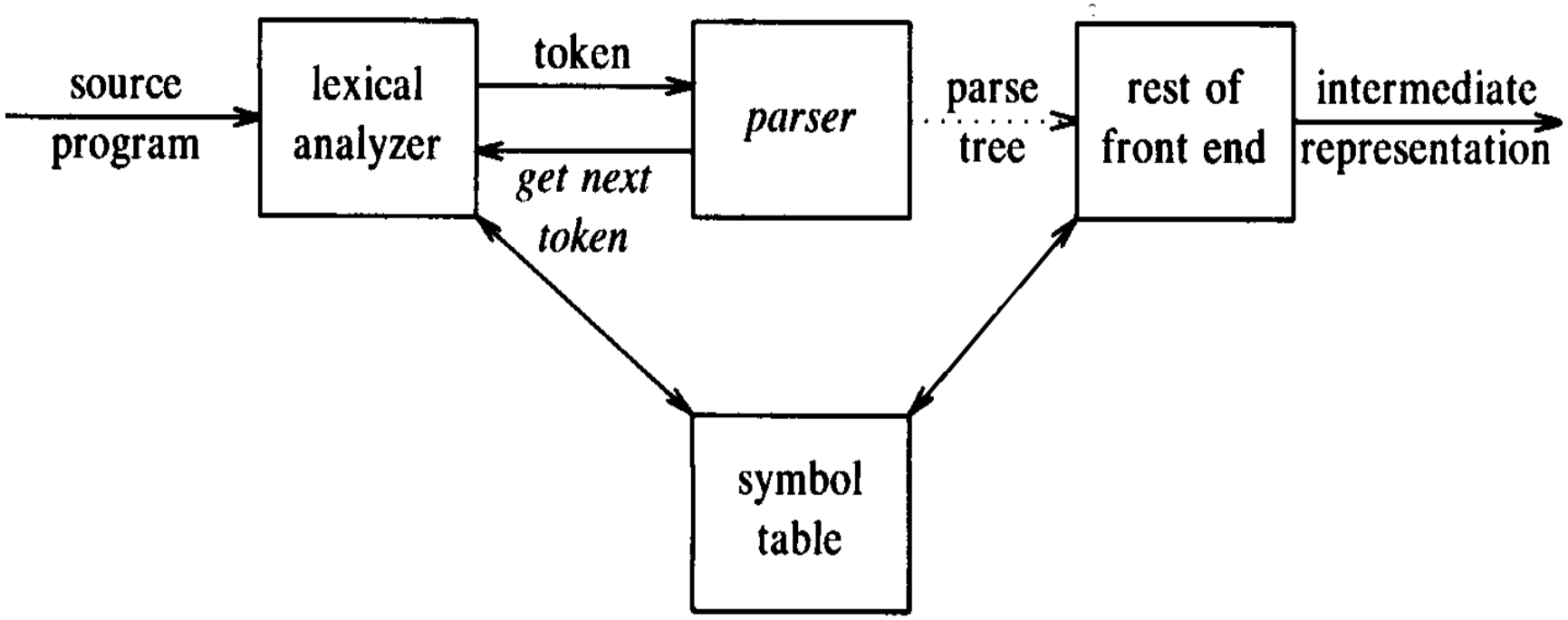
- Frequently quite simple, and to describe them we do not require powerful notations
- RE provide a more concise easy to understand notations for TOKENs
- More efficient lexical analyzers can be constructed automatically from REs
- Separation into lexical and non-lexical provides modular structure of front end of compiler

Why CFG or BNF ?

Syntax of the Programming Language constructs describe by CFG or BNF: WHY ?

1. Specification is Easy to understand
2. From certain classes of grammar we can automatically construct an efficient parser
3. Syntactic Ambiguities can be identified
4. Useful for translation of source program into correct object code
5. New construct can be added very easily to the existing implementation.

The Role of Parser



Position of parser in compiler model.

Parsing

- The scanner recognizes words
- The parser recognizes syntactic units
- Parser operations:
  - Check and verify syntax based on specified syntax rules
  - Report errors
  - Build IR

Parsing

(1) The dog bites the man. (1) = boring,

(2) The man bites the dog. (2) = interesting,

(3) The dog bites man the. (3) = not English

- Check and verify syntax based on specified syntax rules
  - Are regular expressions sufficient for describing syntax?
    - Example 1: Infix expressions
    - Example 2: Nested parentheses
  - We use Context-Free Grammars (CFGs) to specify context-free syntax.
    - A CFG describes how a sentence of a language may be generated.
      - Example:
        - `EvilLaugh` → **mwa** `EvilCackle`
        - `EvilCackle` → **ha** `EvilCackle`
        - `EvilCackle` → **ha!**
    - Use this grammar to generate the sentence **mwa ha ha ha!**

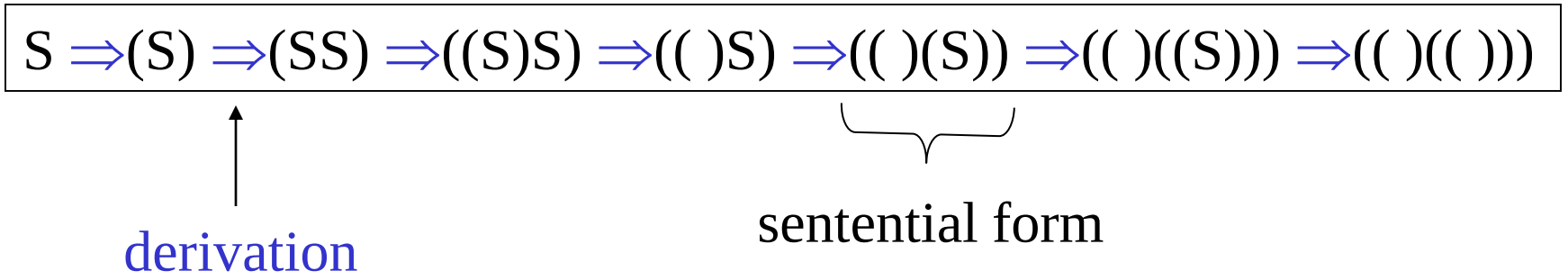
CFGs

- A CFG is a quadruple (N, T, R, S) where
  - N is the set of non-terminal symbols
  - T is the set of terminal symbols
  - $S \in N$ is the starting symbol
  - $R \subseteq N \times (N \cup T)^*$ is a set of rules
- Example: The grammar of nested parentheses

  G = (N, T, R, S) where
  - N = {S}
  - T ={ **(**, **)** }
  - R ={ S→ **(**S**)** , S→SS, S→ε }

Derivations

- The language described by a CFG is the set of strings that can be derived from the start symbol using the rules of the grammar.
- At each step, we choose a non-terminal to replace.



This example demonstrates a **leftmost derivation** : one where we always expand the leftmost non-terminal in the sentential form.

Derivations and parse trees

- We can describe a derivation using a graphical representation called **parse tree**:
  - the root is labeled with the start symbol, S
  - each internal node is labeled with a non-terminal
  - the children of an internal node A are the right-hand side of a production $A \rightarrow \alpha$
  - each leaf is labeled with a terminal
- A parse tree has a unique leftmost and a unique rightmost derivation (however, we cannot tell which one was used by looking at the tree)

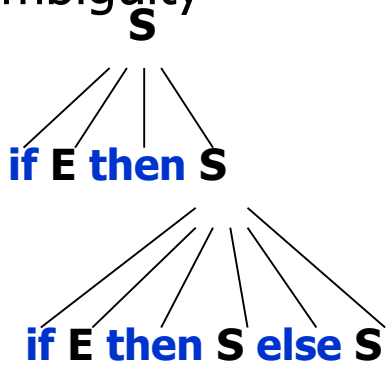
Derivations and parse trees

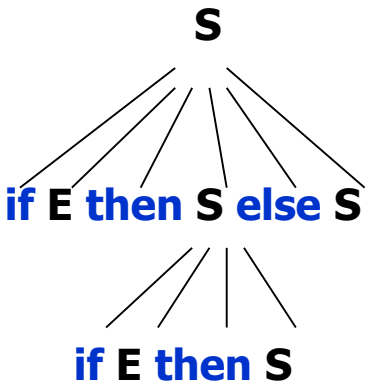
- So, how can we use the grammar described earlier to verify the syntax of "(( ) ((( ))))"?
  - We must try to find a derivation for that string.
  - We can work top-down (starting at the root/start symbol) or bottom-up (starting at the leaves).
- Careful!
  - There may be more than one grammars to describe the same language.
  - Not all grammars are suitable

Problems in parsing

- Consider S → **if** E **then** S **else** S | **if** E **then** S
  - What is the parse tree for

    if E **then if** E **then** S **else** S
  - There are two possible parse trees! This problem is called ambiguity





- A CFG is **ambiguous** if one or more terminal strings have multiple leftmost derivations from the start symbol.

Ambiguity

- There is no general algorithm to tell whether a CFG is ambiguous or not.
- There is no standard procedure for eliminating ambiguity.
- Some languages are *inherently ambiguous*.
  - In those cases, any grammar we come up with will be ambiguous.

Ambiguity

- In general, we try to eliminate ambiguity by rewriting the grammar.
- Example:
  - $E \rightarrow E + E \mid E * E \mid \mathbf{id}$ becomes:

$E \rightarrow E + T \mid T$

$T \rightarrow T * F \mid F$

$F \rightarrow \mathbf{id}$

Ambiguity

- In general, we try to eliminate ambiguity by rewriting the grammar.
- Example:
  - S→**if** E **then** S **else** S | **if** E **then** S | **other** becomes:

S → matched_stmt | unmatched_stmt
matched_stmt → **if** E **then** matched_stmt **else** matched_stmt
| **other**
unmatched_stmt → **if** E **then** S
| **if** E **then** matched_stmt **else** unmatched_stmt

Top-down parsing

- Main idea:
  - Start at the root, grow towards leaves
  - Pick a production and try to match input
  - May need to backtrack

Grammar problems

- In leftmost derivation by scanning the input from left to right, grammars of the form **A** $\rightarrow$ **A x** may cause endless recursion.
- Such grammars are called **left-recursive** and they must be transformed if we want to use a top-down parser.

Left recursion

- A grammar is left recursive if for a non-terminal A,

  there is a derivation $A \stackrel{*}{\Rightarrow} A\alpha$

- There are three types of left recursion:
  - direct ($A \rightarrow A\ x$)
  - indirect ($A \rightarrow B\ C$, $B \rightarrow A$)
  - hidden ($A \rightarrow B\ A$, $B \rightarrow \varepsilon$)

Left recursion

- To eliminate direct left recursion replace

$$\mathbf{A} \rightarrow \mathbf{A}\alpha_1 \mid \mathbf{A}\alpha_2 \mid \ldots \mid \mathbf{A}\alpha_m \mid \beta_1 \mid \beta_2 \mid \ldots \mid \beta_n$$

with

$$\mathbf{A} \rightarrow \beta_1\mathbf{B} \mid \beta_2\mathbf{B} \mid \ldots \mid \beta_n\mathbf{B}$$
$$\mathbf{B} \rightarrow \alpha_1\mathbf{B} \mid \alpha_2\mathbf{B} \mid \ldots \mid \alpha_m\mathbf{B} \mid \varepsilon$$

Left recursion

- How about this:

$S \rightarrow E$
$E \rightarrow E+T$
$E \rightarrow T$
$T \rightarrow E-T$
$T \rightarrow \mathbf{id}$

There is direct recursion: $E \rightarrow E+T$
There is indirect recursion: $T \rightarrow E+T$, $E \rightarrow T$

Algorithm for eliminating indirect recursion

List the nonterminals in some order $A_1, A_2, \ldots, A_n$
for i=1 to n
    for j=1 to i-1
        if there is a production $A_i \rightarrow A_j\alpha$,
            replace A_j with its rhs
    eliminate any direct left recursion on A_i

Left recursion Algo…

- To show that the previous algorithm actually works all we need notice is that iteration i only changes productions with A_i on the left-hand side. And m > i in all productions of the form $A_i \rightarrow A_m a$.
- This can be easily shown by induction.
  - It is clearly true for i=1.
  - If it is true for all i < k, then when the outer loop is executed for i = k, the inner loop will remove all productions $A_i \rightarrow A_m a$ with m < i.
  - Finally, with the elimination of self recursion, m in the $A_i \rightarrow A_m a$ productions is forced to be > i.
- So, at the end of the algorithm, all derivations of the form $A_i \stackrel{*}{\Rightarrow} A_m \alpha$ will have m > i and therefore left recursion would not be possible.

Eliminating indirect left recursion

ordering: S, E, T, F

$S \rightarrow E$
$E \rightarrow E+T$
$E \rightarrow T$
$T \rightarrow E-T$
$T \rightarrow F$
$F \rightarrow E*F$
$F \rightarrow id$

i=S

$S \rightarrow E$
$E \rightarrow E+T$
$E \rightarrow T$
$T \rightarrow E-T$
$T \rightarrow F$
$F \rightarrow E*F$
$F \rightarrow \mathbf{id}$

i=E

$S \rightarrow E$
$E \rightarrow TE'$
$E' \rightarrow +TE' \mid \varepsilon$
$T \rightarrow E-T$
$T \rightarrow F$
$F \rightarrow E*F$
$F \rightarrow \mathbf{id}$

i=T, j=E

$S \rightarrow E$
$E \rightarrow TE'$
$E' \rightarrow +TE' \mid \varepsilon$
$T \rightarrow TE'-T$
$T \rightarrow F$
$F \rightarrow E*F$
$F \rightarrow \mathbf{id}$

$S \rightarrow E$
$E \rightarrow TE'$
$E' \rightarrow +TE' \mid \varepsilon$
$T \rightarrow FT'$
$T' \rightarrow E'-TT' \mid \varepsilon$
$F \rightarrow E*F$
$F \rightarrow \mathbf{id}$

Eliminating indirect left recursion

i=F, j=E

$S \rightarrow E$
$E \rightarrow TE'$
$E' \rightarrow +TE' \mid \varepsilon$
$T \rightarrow FT'$
$T' \rightarrow E'-TT' \mid \varepsilon$
$F \rightarrow TE'*F$
$F \rightarrow \mathbf{id}$

i=F, j=T

$S \rightarrow E$
$E \rightarrow TE'$
$E' \rightarrow +TE' \mid \varepsilon$
$T \rightarrow FT'$
$T' \rightarrow E'-TT' \mid \varepsilon$
$F \rightarrow FT'E'*F$
$F \rightarrow \mathbf{id}$

$S \rightarrow E$
$E \rightarrow TE'$
$E' \rightarrow +TE' \mid \varepsilon$
$T \rightarrow FT'$
$T' \rightarrow E'-TT' \mid \varepsilon$
$F \rightarrow \mathbf{id}F'$
$F' \rightarrow T'E'*FF' \mid \varepsilon$

Grammar problems

- Consider S → **if** E **then** S **else** S | **if** E **then** S
  - Which of the two productions should we use to expand non-terminal S when the next token is **if**?
  - We can solve this problem by factoring out the common part in these rules. This way, we are postponing the decision about which rule to choose until we have more information (namely, whether there is an **else** or not).
  - This is called **left factoring**

Left factoring

$A \rightarrow \alpha\beta_1 \mid \alpha\beta_2 \mid ... \mid \alpha\beta_n \mid \gamma$

becomes

$A \rightarrow \alpha B \mid \gamma$

$B \rightarrow \beta_1 \mid \beta_2 \mid ... \mid \beta_n$

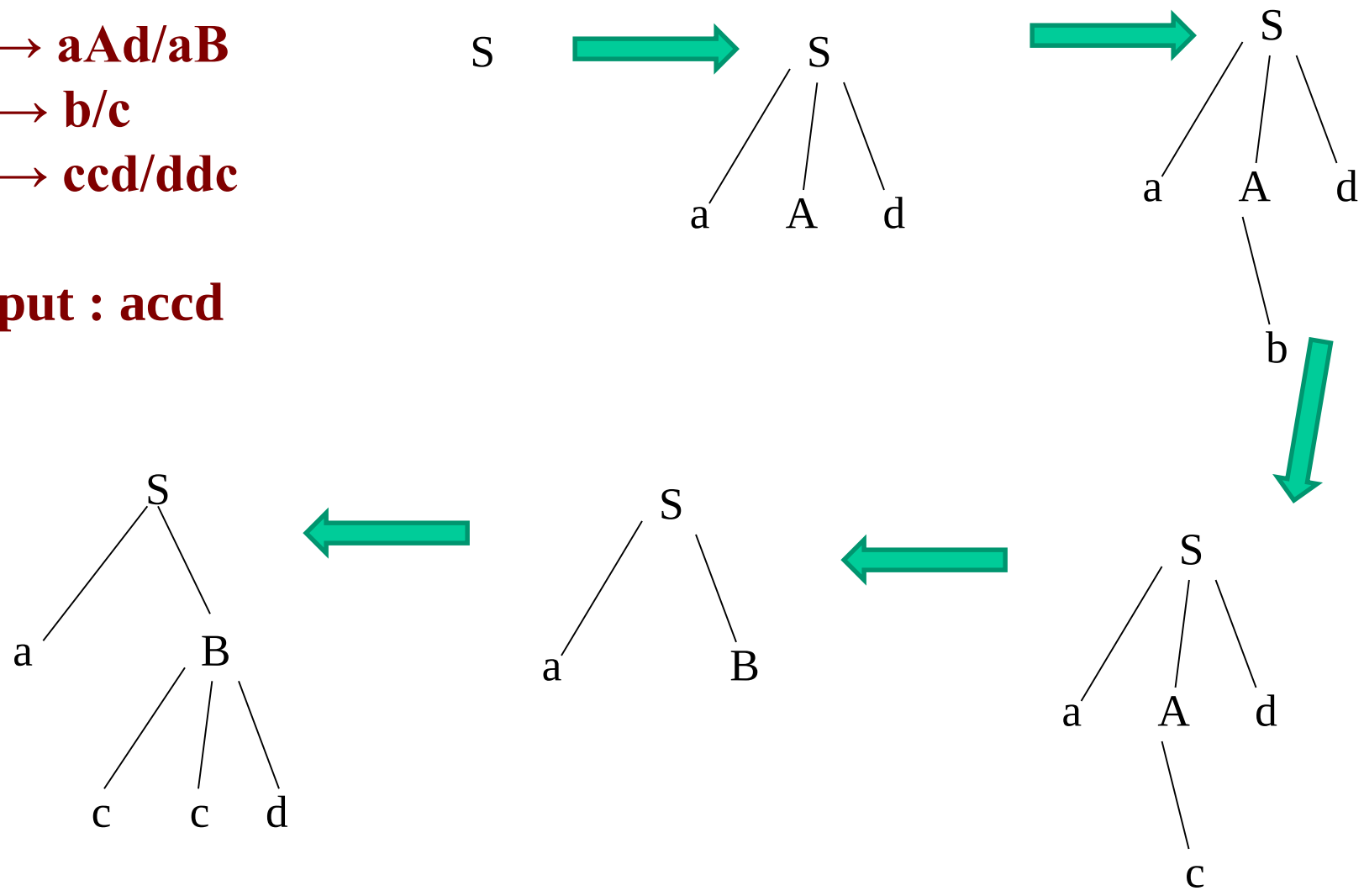
Top Down Parsing

- Brute Force Parsing Approach (full backup)
- Recursive-Descent Parsing (backup is not allowed)
- Predictive Parsing (special form of RDP)
- LL(1) Parsing

Brute Force Parsing

S → aAd/aB
A → b/c
B → ccd/ddc

Input : accd



Recursive Descent Parsing

- Basic idea:
  - Write a routine to recognize each LHS (non-terminal)
  - This produces a parser with mutually recursive routines.
  - Good for hand-coded parsers.
- Example:

<term> ::= <factor> <multiply_operator> <term> or T → F*T

Recursive rule for <term>

```
public boolean term()
{
        if (!factor()) return false;
        if (!multiplyOperator()) return true;
        if (!term()) error("No term after '*' or '/' ");
        return true;
}
```

Example

- **<term> ::= <factor> { <multiply_operator> <factor> }**

- public boolean term()

Iterative rule for <term>

```
{
        if (!factor()) return false;
        while (multiplyOperator())
       {
            if (!factor()) error("No factor after '*' or '/' ");
        }
        return true;
}
```

Advantage and Disadvantages

Advantages:

- They are exceptionally simple
- They can be constructed from recognizers simply by doing some extra work specifically, building a parse tree

Disadvantages:

- not very faster
- It is difficult to provide really good error messages
- They cannot do parses that require arbitrarily long lookaheads

Recursive-Descent Parsing Cont'd…

- **Main challenges:**

1. **back-tracking** is messy, difficult and inefficient (solution: use input "lookahead" to help make the right choice)

2. **more alternatives** --- even if we use one lookahead input char, there are still more than 1 rules to choose --- *A* -> **ab** | **a** (**solution**: rewrite the grammar by **left-factoring**)

3. **left-recursion** might cause infinite loop

   what is the procedure for *E* -> *E* **+** *E* ?

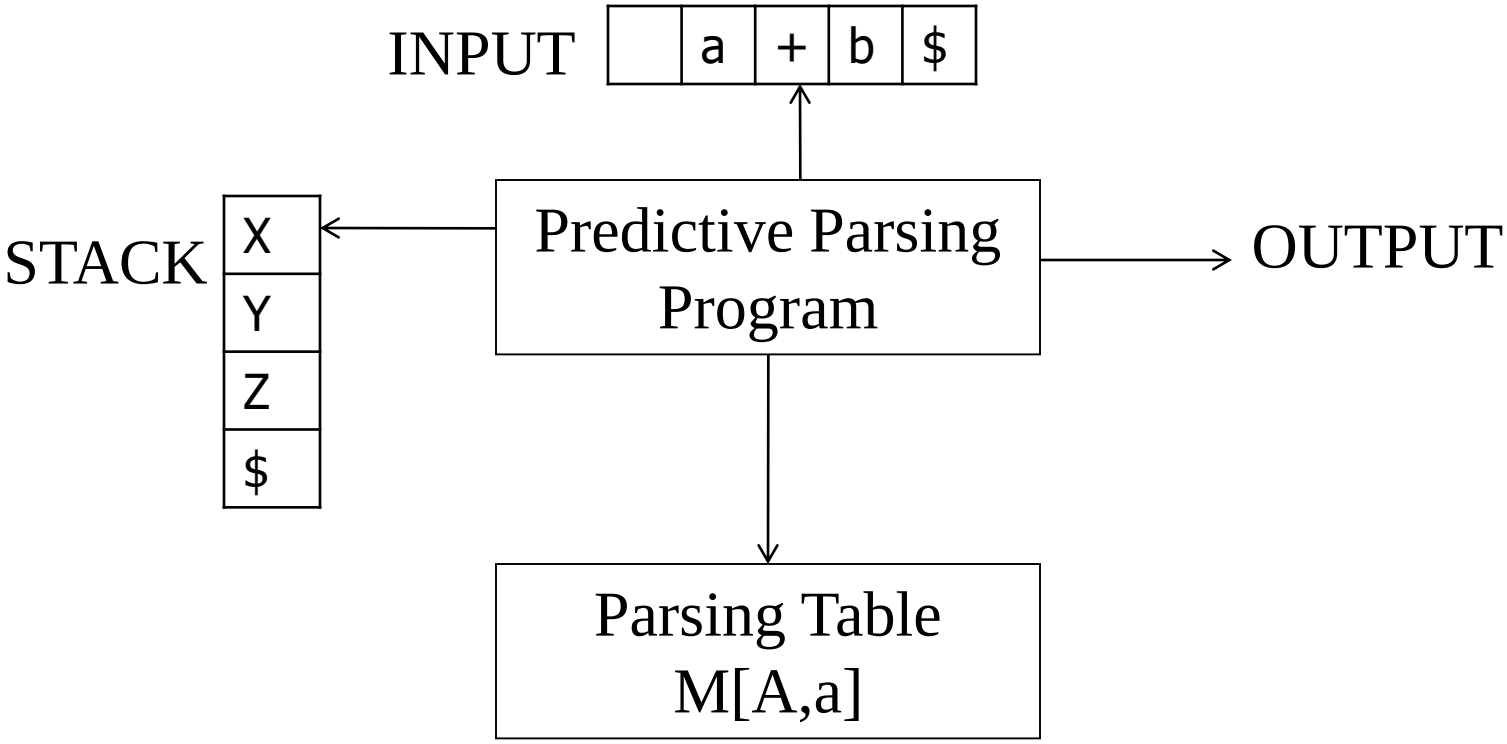
   (**solution**: rewrite the grammar by **eliminating left-recursions**)

4. **error handling --- ** errors detected "far away" from actual source.

Predictive parsing

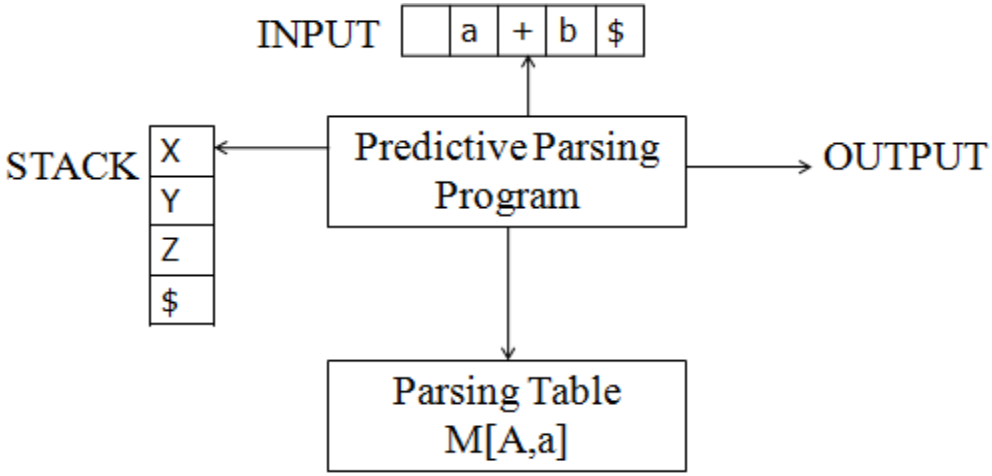
- Special form of RDP
- Can we avoid the backtracking?
  - Given $A \rightarrow \alpha \mid \beta$ the parser should be able to choose between α and β
- How?
  - What if we do some "preprocessing" to answer the question: Given a non-terminal A and **lookahead** t, which (if any) production of A is guaranteed to start with a t?

Non Recursive Predictive Parsing



Non Recursive Predictive Parsing

- Program behavior based on X, top of the stack, and a, the current input symbol
  1) If X = a = $, Success and parser halts
  2) If X = a ≠ $, pop X and advances the input pointer to next input symbol (symbol match)
  3) If X is non-terminal, find the entry M[X,a] in parsing table. This entry will be either X-production or an error entry. e.g. if entry is **X → W*V** then parser replaces X top of the stack by W*V (V , *, W order) else it call error recovery routine.



How to create a Parsing Table

Predictive Parsing

- If we have two productions: $A \rightarrow \alpha \mid \beta$, we want a distinct way of choosing the correct one.
- If α is any string of grammar symbols, **FIRST(α)** is the set of terminals that begin the strings derived from α. If $\alpha \Rightarrow \varepsilon$ then ε is also in FIRST(α).
- Define:
  - for $\alpha \in G$, $\mathbf{x} \in$ FIRST(α) iff $\alpha \Rightarrow^* \mathbf{x}\gamma$
- If FIRST(α) and FIRST(β) contain no common symbols, we will know whether we should choose $A \rightarrow \alpha$ or $A \rightarrow \beta$ by looking at the lookahead symbol.

Compute FIRST(X)

To compute FIRST(X) for all grammar symbols X, apply the following rules until no more terminals or ϵ can be added to any FIRST set.

1. If X is terminal, then FIRST(X) is $\{X\}$.
2. If $X \rightarrow \epsilon$ is a production, then add ϵ to FIRST(X).
3. If X is nonterminal and $X \rightarrow Y_1 Y_2 \cdots Y_k$ is a production, then place a in FIRST(X) if for some i, a is in FIRST(Y_i), and ϵ is in all of FIRST(Y_1), . . . , FIRST(Y_{i-1}); that is, $Y_1 \cdots Y_{i-1} \stackrel{*}{\Rightarrow} \epsilon$. If ϵ is in FIRST(Y_j) for all $j = 1, 2, \ldots, k$, then add ϵ to FIRST(X). For example, everything in FIRST(Y_1) is surely in FIRST(X). If Y_1 does not derive ϵ, then we add nothing more to FIRST(X), but if $Y_1 \stackrel{*}{\Rightarrow} \epsilon$, then we add FIRST($Y_2$) and so on.

$$
\begin{aligned}
E &\rightarrow TE' \\
E' &\rightarrow +TE' \mid \epsilon \\
T &\rightarrow FT' \\
T' &\rightarrow *FT' \mid \epsilon \\
F &\rightarrow (E) \mid \mathbf{id}
\end{aligned}
$$

$\text{FIRST}(E) = \text{FIRST}(T) = \text{FIRST}(F) = \{(, \mathbf{id}\}.$

$\text{FIRST}(E') = \{+, \epsilon\}$

$\text{FIRST}(T') = \{*, \epsilon\}$

Compute FIRST(X) Examples…

Ex.1

S → aAbc | BCf A → C | ε

B → Cd | c C → df | ε

Ex.2

S → qABC A → a | bbD B → a | ε

C → b | ε D → c | ε

Ex.3

A → B | C B → n | i

C → (D) D → DA | A

Predictive parsing

- What if we have a "candidate" production $A \rightarrow \alpha$ where $\alpha = \varepsilon$ or $\alpha \Rightarrow^* \varepsilon$?
- We could expand if we knew that there is some sentential form where the **current input symbol appears after A.**
- Define:
  - for $A \in N$, $\mathbf{x} \in FOLLOW(A)$ iff $\exists\ S \Rightarrow^* \alpha A \mathbf{x} \beta$

Compute FOLLOW(A)

To compute FOLLOW(A) for all nonterminals A, apply the following rules until nothing can be added to any FOLLOW set.

1. Place \$ in FOLLOW(S), where S is the start symbol and \$ is the input right endmarker.
2. If there is a production $A \rightarrow \alpha B \beta$, then everything in FIRST(β) except for ϵ is placed in FOLLOW(B).
3. If there is a production $A \rightarrow \alpha B$, or a production $A \rightarrow \alpha B \beta$ where FIRST(β) contains ϵ (i.e., $\beta \stackrel{*}{\Rightarrow} \epsilon$), then everything in FOLLOW(A) is in FOLLOW(B).

$$
\begin{aligned}
E &\rightarrow TE' \\
E' &\rightarrow +TE' \mid \epsilon \\
T &\rightarrow FT' \\
T' &\rightarrow *FT' \mid \epsilon \\
F &\rightarrow (E) \mid \mathbf{id}
\end{aligned}
$$

FIRST(E) = FIRST(T) = FIRST(F) = {(, **id**}.

FIRST(E') = {+, ϵ}

FIRST(T') = {*, ϵ}

FOLLOW(E) = FOLLOW(E') = {), \$}

FOLLOW(T) = FOLLOW(T') = {+,), \$}

FOLLOW(F) = {+, *,), \$}

Compute FOLLOW(X) Examples…

Ex.1

S → aAbc | BCf A → C | ε

B → Cd | c C → df | ε

Ex.2

S → qABC A → a | bbD B → a | ε

C → b | ε D → c | ε

Ex.3

A → B | C B → n | i

C → (D) D → DA | A

Predictive parsing

- Armed with
  - FIRST
  - FOLLOW
- we can build a parser where no backtracking is required!

Predictive parsing (w/table)

- For each production $A \rightarrow \alpha$ do:
  - For each terminal **a** $\in$ FIRST(α) add $A \rightarrow \alpha$ to entry M[A,**a**]
  - If $\varepsilon \in$FIRST(α), add $A \rightarrow \alpha$ to entry M[A,**b**] for each terminal **b** $\in$ FOLLOW(A).
  - If $\varepsilon \in$FIRST(α) and \$ $\in$FOLLOW(A), add $A \rightarrow \alpha$ to M[A,\$]
- Use table and stack to simulate recursion.

Building a parser

- Original grammar:

$$E \rightarrow E+E$$
$$E \rightarrow E*E$$
$$E \rightarrow (E)$$
$$E \rightarrow id$$

- This grammar is left-recursive, ambiguous and requires left-factoring. It needs to be modified before we build a predictive parser for it:

Remove ambiguity:

$$E \rightarrow E+T$$
$$T \rightarrow T*F$$
$$F \rightarrow (E)$$
$$F \rightarrow id$$

Remove left recursion:

$$E \rightarrow TE'$$
$$E' \rightarrow +TE' \mid \varepsilon$$
$$T \rightarrow FT'$$
$$T' \rightarrow *FT' \mid \varepsilon$$
$$F \rightarrow (E)$$
$$F \rightarrow id$$

Building a parser

- The grammar:

$E \rightarrow TE'$
$E' \rightarrow +TE' \mid \varepsilon$
$T \rightarrow FT'$
$T' \rightarrow *FT' \mid \varepsilon$
$F \rightarrow (E)$
$F \rightarrow id$

FIRST(E) = FIRST(T) = FIRST(F) = {(, **id**}
FIRST(E') = {+, ε}
FIRST(T') = {*, ε}
FOLLOW(E) = FOLLOW(E') = {$,)}
FOLLOW(T) = FOLLOW(T') = {+, $,)}
FOLLOW(F) = {*, +, $,)}

Now, we can either build a table or design a recursive descend parser.

Predictive parsing (w/table)

- For each production $A \rightarrow \alpha$ do:
  - For each terminal **a** $\in$ FIRST(α) add $A \rightarrow \alpha$ to entry M[A,**a**]
  - If $\varepsilon \in$ FIRST(α), add $A \rightarrow \alpha$ to entry M[A,**b**] for each terminal **b** $\in$ FOLLOW(A).
  - If $\varepsilon \in$ FIRST(α) and \$ $\in$ FOLLOW(A), add $A \rightarrow \alpha$ to M[A,\$]
- Use table and stack to simulate recursion.

Parsing table

	+	*	(	)	id	$
E			E→TE'		E→TE'	
E′	E'→+TE'			E'→ε		E'→ε
T			T→FT'		T→FT'	
T′	T'→ε	T'→*FT'		T'→ε		T'→ε
F			F→(E)		F→id	

	+	*	(	)	id	$
E			E→TE'		E→TE'	
E'	E'→+TE'			E'→ε		E'→ε
T			T→FT'		T→FT'	
T'	T'→ε	T'→*FT'		T'→ε		T'→ε
F			F→(E)		F→id	

Step	Stack	Input	Next Action
1	$E	id*id$	E→TE'
2	$E'T	id*id$	T→FT'
3	$E'T'F	id*id$	F→id
4	$E'T'id	id*id$	match id
5	$E'T'	*id$	T'→*FT'
6	$T'F*	*id$	match *
7	$T'F	id$	F→id
8	$T'id	id$	match id
9	$T'	$	T'→ε
10	$	$	accept

Recursive descend parser

```
parse() {
   token = get_next_token();
   if (E() and token == '$')
   then return true
   else return false
}
```

```
E() {
     if (T())
     then return Eprime()
     else return false
}
```

```
Eprime() {
     if (token == '+')
     then token=get_next_token()
             if (T())
             then return Eprime()
             else return false
     else if (token==')' or token=='$')
     then return true
     else return false
}
```

The remaining procedures are similar.

LL(1) parsing

- Our parser scans the input Left-to-right, generates a Leftmost derivation and uses 1 symbol of lookahead.
- It is called an LL(1) parser.
- If you can build a parsing table with no multiply-defined entries, then the grammar is LL(1).
- Ambiguous grammars are never LL(1)
- Non-ambiguous grammars are not necessarily LL(1)

LL(1) Conditions

- No ambiguous or left recursive grammar can be LL(1).
- Grammar G is LL(1) **iff** whenever $A \rightarrow \alpha / \beta$ are two distinct productions of G and:
  - For no terminal *a* do both α and β derive strings beginning with *a*.

    FIRST(α)$\cap$FIRST(β)=$\varnothing$
  - At most one of α and β can derive the empty string.
  - If $\beta^{*} \Rightarrow \varepsilon$, the α does not derive any string beginning with a terminal in FOLLOW(A).

    FIRST(αFOLLOW(A))$\cap$FIRST(βFOLLOW(A))=$\varnothing$

LL(1) parsing

- For example, the following grammar will have two possible ways to expand S' when the lookahead is **else**.

$S \rightarrow$ **if** E **then** S S' | other
$S' \rightarrow$ **else** S | ε
$E \rightarrow$ **id**

  - It may expand S'$\rightarrow$ **else** S or S'$\rightarrow \varepsilon$
  - We can resolve the ambiguity by instructing the parser to always pick S'$\rightarrow$ **else** S. This will match each **else** to the closest previous **then**.

LL(1) parsing

- Here's an example of a grammar that is NOT LL(k) for any k:

$$S \rightarrow Ca \mid Cb$$
$$C \rightarrow cC \mid c$$

  - Why? Suppose the grammar was LL(k) for some k. Consider the input string $\mathbf{c^{k+1}a}$. With only k lookaheads, the parser would not be able to decide whether to expand using S $\rightarrow$ C**a** or S $\rightarrow$ C**b**
  - Note that the grammar is actually regular: it generates strings of the form $c^+(a|b)$

Error detection in LL(1) parsing

- An error is detected whenever an empty table slot is encountered.
- We would like our parser to be able to recover from an error and continue parsing.
- Phase-level recovery
  - We associate each empty slot with an error handling procedure.
- Panic mode recovery
  - Modify the stack and/or the input string to try and reach state from which we can continue.

Error recovery in LL(1) parsing

- Panic mode recovery
  - Idea:
    - Decide on a set of synchronizing tokens.
    - When an error is found and there's a non-terminal at the top of the stack, discard input tokens until a synchronizing token is found.
    - Synchronizing tokens are chosen so that the parser can recover quickly after one is found
      - e.g. a semicolon when parsing statements.
    - If there is a terminal at the top of the stack, we could try popping it to see whether we can continue.
      - Assume that the input string is actually missing that terminal.

Error recovery in LL(1) parsing

- Panic mode recovery
  - Possible synchronizing tokens for a nonterminal A
    - the tokens in FOLLOW(A)
      - When one is found, pop A of the stack and try to continue
    - the tokens in FIRST(A)
      - When one is found, match it and try to continue
    - tokens such as semicolons that terminate statements

Bottom-Up Parsing

- Also known as Shift-reduce parsing
- Attempts to construct a parse tree for an input string beginning at the leaves (bottom) and working up towards the root (top).
- "Reducing" a string **w** to the start symbol of a grammar.
- At each step, decide on some substring that matches the RHS of some production
  - Replace this string by the LHS (called **reduction**).
- If the substring is chosen correctly at each step, it is the trace of a rightmost derivation in reverse.

Example

- Grammar:

  S → aABe

  A → Abc | b

  B → d

$$S \underset{rm}{\Longrightarrow} aABe \underset{rm}{\Longrightarrow} aAde \underset{rm}{\Longrightarrow} aAbcde \underset{rm}{\Longrightarrow} abbcde$$

- Reduction:

  abbcde ←- - - - A → b

  aAbcde ←- - - - A → Abc

  aAde ←- - - - B → d

  aABe ←- - - - S → aABe

  S

Bottom-Up Stack Parsing

E → E + T / T
T → T * F / F
F → (E) / n

Stack	Input	Action
$	n + n * n $	S
$ n	+ n * n $	R T → F
$ F	+ n * n $	R E → T
$ T	+ n * n $	R
$ E	+ n * n $	S
$ E +	n * n $	S
$ E + n	* n $	R F → n
$ E + F	* n $	R T → F
$ E + T	* n $	S
$ E + T *	n $	S
$ E + T * n	$	R F → n
$ E + T * F	$	R T → T * F
$ E + T	$	R E → E + T
$ E	$	

Derivation

- Bottom-up parsing produces a rightmost derivation…
- …in reverse
- Together, the parsing stack and the input form the right sentential forms of the rightmost derivation

Derivation

$ n	+ n * n $	n + n * n ⇐
$ F	+ n * n $	F + n * n ⇐
$ T	+ n * n $	T + n * n ⇐
$ E	+ n * n $	E + n * n ⇐
$ E +	n * n $	E + n * n ⇐
$ E + n	* n $	E + n * n ⇐
$ E + F	* n $	E + F * n ⇐
$ E + T	* n $	E + T * n ⇐
$ E + T *	n $	E + T * n ⇐
$ E + T * n	$	E + T * n ⇐
$ E + T * F	$	E + T * F ⇐
$ E + T	$	E + T ⇐
$ E	$	E

E → E + T / T
T → T * F / F
F → (E) / n

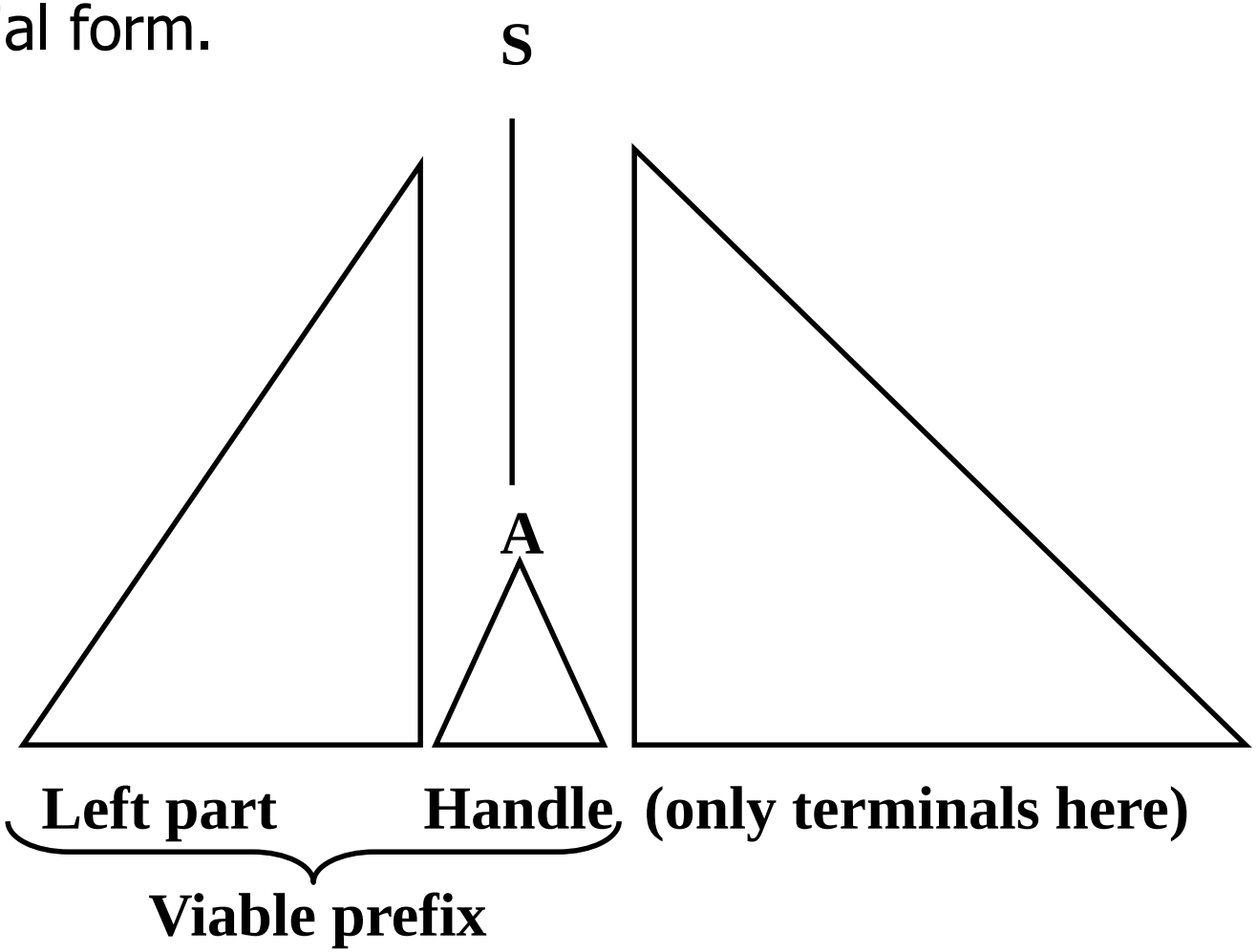
Handles

- A handle is the substring of a right sentential form that gets reduced in the (reverse) derivation
  - E + T * F
- A shift-reduce parser shifts input until a handle appears on the parsing stack, then reduces the handle
  - E + T * F $\Leftarrow$ E + T

So we scan tokens from left to right, find the handle, and replace it by corresponding LHS (called handle pruning)

Handle Pruning, II

- Consider the cut of a parse-tree of a certain right sentential form.



Handles

$	n + n * n $	S
$ n	+ n * n $	R F → n
$ F	+ n * n $	R T → F
$ T	+ n * n $	R E → T
$ E	+ n * n $	S
$ E +	n * n $	S
$ E + n	* n $	R F → n
$ E + F	* n $	R T → F
$ E + T	* n $	S
$ E + T *	n $	S
$ E + T * n	$	R F → n
$ E + T * F	$	R T → T * F
$ E + T	$	R E → E + T
$ E	$	

E → E + T / T
T → T * F / F
F → (E) / n

Viable Prefixes

The string on the parsing stack is a prefix of a right sentential form

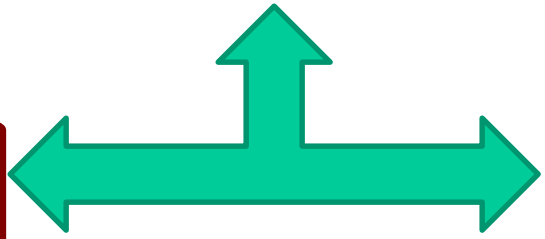
E + T * (n or F or …)

That does not extend past the end of the handle

E + n * (cannot occur on the stack)

Such prefixes are called viable prefixes

LR Parser Operations



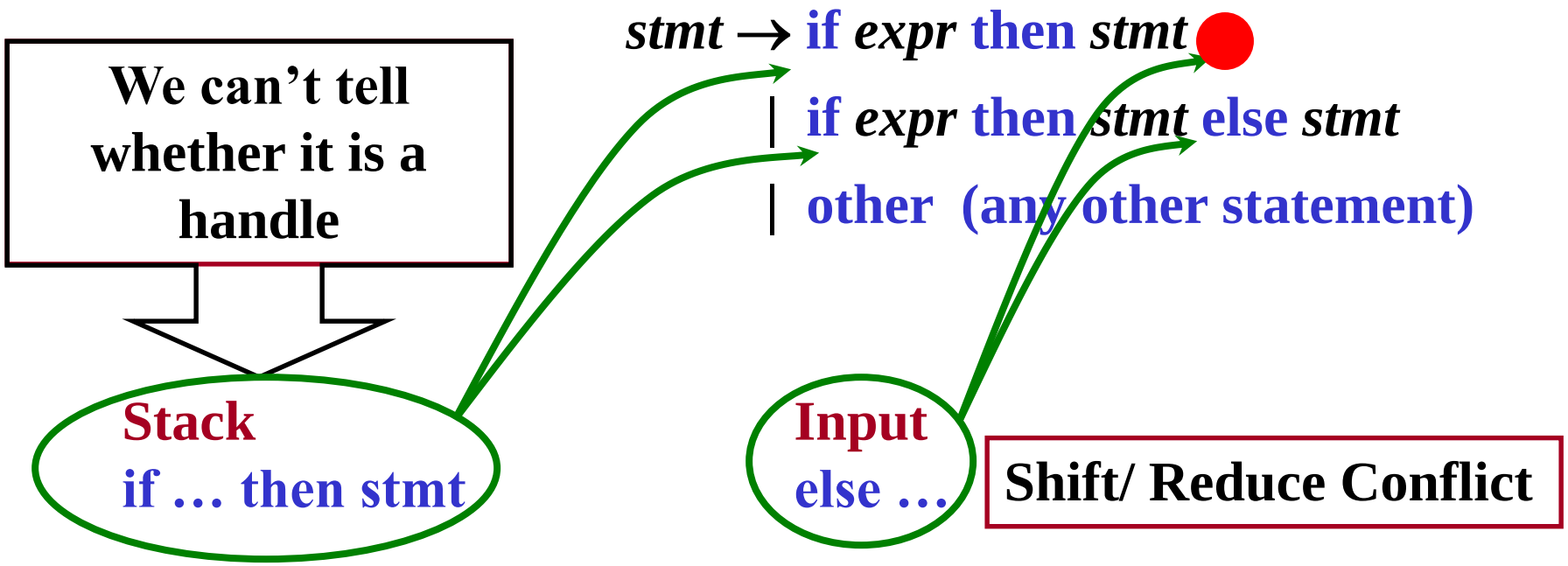
Shift Reduce Parsing

1. SHIFT
2. REDUCE
3. ACCEPT
4. ERROR

Conflicts in Shift-Reduce Parsing

"shift/reduce" or "reduce/reduce"

Example:



More Conflicts

stmt → id (*parameter-list*)

stmt → *expr* := *expr*

parameter-list → *parameter-list* , *parameter* | *parameter*

parameter → id

expr-list → *expr-list* , *expr* | *expr*

expr → id | id (*expr-list*)

Consider the string A(I,J)

Corresponding token stream is id(id, id)

After three shifts:

Stack = … id(id Input = , id)…

Reduce/Reduce Conflict … what to do?

(it really depends on what is A, -- an array? or a procedure?

How the symbol third from the top of stack determines the reduction to be made, even though it is not involved in the reduction. Shift-reduce parsing can utilize information far down in the stack to guide the parse

Operator Precedence Parsing

- **Operator grammar**
  - small, but an important class of grammars
  - we may have an efficient operator precedence parser (a shift-reduce parser) for an operator grammar.
- In an ***operator grammar***, no production rule can have:
  - ε at the right side
  - two adjacent non-terminals at the right side.
- Ex:

E→AB A→a B→b not operator grammar	E→EOE E→id O→+|*|/ not operator grammar	E→E+E | E→ E*E | E→ E/E | id **Operator grammar**

Precedence Relations

- Here we define three disjoint precedence relations between certain pairs of terminals.

$a \lessdot b$	b has higher precedence than a
$a \doteq b$	b has same precedence as a
$a \gtrdot b$	b has lower precedence than a

- The determination of correct precedence relations between terminals are based on the traditional notions of associativity and precedence of operators. (Unary minus causes a problem).

Using Operator-Precedence Relations

- The intention of the precedence relations is to find the **handle** of a right-sentential form,
  - $<\cdot$ with marking the left end,
  - $=\cdot$ appearing in the interior of the handle, and
  - $\cdot>$ marking the right hand.

- In our input string **$\$a_1a_2\ldots a_n\$$**, we insert the precedence relation between the pairs of terminals (the precedence relation holds between the terminals in that pair).

Using Operator -Precedence Relations

$E \rightarrow E + E \mid E - E \mid E * E \mid E / E \mid E \wedge E \mid (E) \mid - E \mid id$

The partial operator-precedence table for this grammar

	id	+	*	$
id		·>	·>	·>
+	<·	·>	<·	·>
*	<·	·>	·>	·>
$	<·	<·	<·	

- Then the input string **id + id * id** with the precedence relations inserted will be:

$ <· id ·> + <· id ·> * <· id ·> $

To Find The Handles

1. Scan the string from left end until the first ·> is encountered.
2. Then scan backwards (to the left) over any =· until a <· is encountered.
3. The handle contains everything to left of the first ·> and to the right of the <· is encountered.

$ <. id .> + <. id .> * <. id .> $	E → id	$ id + id * id $
$ <. E + <. id .> * <. id .> $	E → id	$ E + id * id $
$ <. E + <. E * <. id .> $	E → id	$ E + E * id $
$ <. E + <. E * E .> $	E → E * E	$ E + E * E $
$ <. E + E .> $	E → E + E	$ E + E $
$ $		$ E $

Operator-Precedence Parsing Algorithm

The input string is w$, the initial stack is $ and a table holds precedence relations between certain terminals

Algorithm:

```
set p to point to the first symbol of w$ ;
repeat forever
    if ( $ is on top of the stack and p points to $ ) then return
    else {
        let a be the topmost terminal symbol on the stack and let b be the symbol pointed
        to by p;
        if ( a <· b or a =· b ) then
            { push b onto the stack;              /* SHIFT */
              advance p to the next input symbol;
            }
        else if ( a ·> b ) then
            repeat pop stack                      /* REDUCE */
            until ( the top of stack terminal is related by <· to the terminal most recently
                    popped );
        else error();
    }
```

Operator-Precedence Parsing Algorithm -- Example

stack	***input***	***action***		
$	id+id*id$	$ <· id	shift	
$id	+id*id$	id ·> +	reduce	E → id
$	+id*id$	shift		
$+	id*id$	shift		
$+id	*id$	id ·> *	reduce	E → id
$+	*id$	shift		
$+*	id$	shift		
$+*id	$	id ·> $	reduce	E → id
$+*	$	* ·> $	reduce	E → E*E
$+	$	+ ·> $	reduce	E → E+E
$	$	accept		

	id	**+**	*****	**$**
id		·>	·>	·>
+	<·	·>	<·	·>
*****	<·	·>	·>	·>
$	<·	<·	<·	

How to Create Operator-Precedence Relations

- We use associativity and precedence relations among operators.
  1. If operator θ_1 has higher precedence than operator θ_2,

     ➔ $\theta_1 \cdot> \theta_2$ and $\theta_2 <\cdot \theta_1$
  2. If operator θ_1 and operator θ_2 have equal precedence,

     they are left-associative ➔ $\theta_1 \cdot> \theta_2$ and $\theta_2 \cdot> \theta_1$

     they are right-associative ➔ $\theta_1 <\cdot \theta_2$ and $\theta_2 <\cdot \theta_1$
  3. For all operators θ

     $\theta <\cdot$ id, id $\cdot>$ θ, $\theta <\cdot$ (, ($<\cdot \theta$, $\theta \cdot>$),) $\cdot>$ θ, $\theta \cdot>$ \$, and

     \$ $<\cdot \theta$
  4. Also, let

     (=·) \$ <· (\$ <. id

     (<· (id ·> \$) .> \$

     (<· id id .>)) .>)

Operator-Precedence Relations

	+	-	*	/	^	id	(	)	$
+									
-									
*									
/									
^									
id						-	-		
(									-
)						-	-		
$								-	-

Operator-Precedence Relations

	+	-	*	/	^	id	(	)	$
+	·>	·>	<·	<·	<·	<·	<·	·>	·>
-	·>	·>	<·	<·	<·	<·	<·	·>	·>
*	·>	·>	·>	·>	<·	<·	<·	·>	·>
/	·>	·>	·>	·>	<·	<·	<·	·>	·>
^	·>	·>	·>	·>	<·	<·	<·	·>	·>
id	·>	·>	·>	·>	·>			·>	·>
(	<·	<·	<·	<·	<·	<·	<·	=·	
)	·>	·>	·>	·>	·>			·>	·>
$	<·	<·	<·	<·	<·	<·	<·		

Handling Unary Minus

- Operator-Precedence parsing cannot handle the unary minus when we also have the binary minus in our grammar.
- The best approach to solve this problem, let the lexical analyzer handle this problem.
  - The lexical analyzer will return two different tokens for the unary minus and the binary minus.
  - The lexical analyzer will need a lookhead to distinguish the binary minus from the unary minus.
  - Problem with this scheme is that lexical analyzer has to remember last token.
- Then, we make

  $\theta \lessdot$ unary-minus for any operator

  unary-minus $\gtrdot \theta$ if unary-minus has higher precedence than θ

  unary-minus $\lessdot \theta$ if unary-minus has lower (or equal) precedence than θ

Precedence Functions

- Compilers using operator precedence parsers do not need to store the table of precedence relations.
- The table can be encoded by two precedence functions **f** and **g** that map terminal symbols to integers.
- For symbols a and b.

  f(a) < g(b) whenever a $\lessdot$ b

  f(a) = g(b) whenever a $\doteq$ b

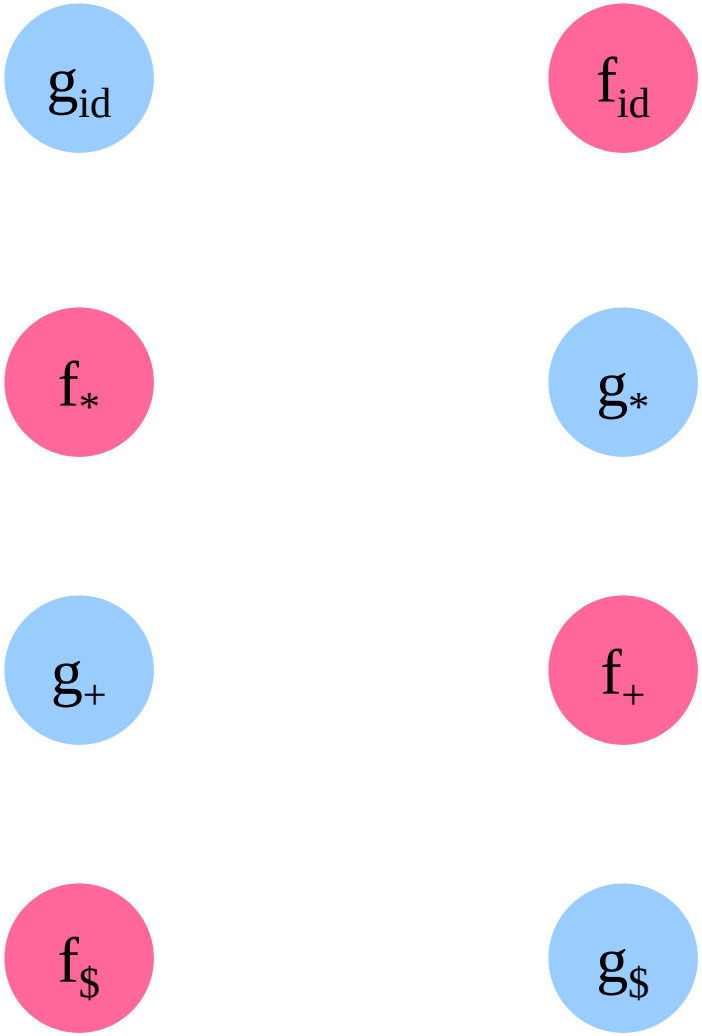
  f(a) > g(b) whenever a $\gtrdot$ b

Constructing precedence functions

Method:

1. Create symbols f_a and g_a for each **a** that is a terminal or $.
2. Partition the created symbols into as many groups as possible, in such a way that if a =. b, then f_a and g_b are in the same group.
3. Create a directed graph whose nodes are the groups found in (2). For any a and b, if a <.b , place an edge from the group of g_b to the group of f_a. Of a .> b, place an edge from the group of f_a to that of g_b.
4. If the graph constructed in (3) has a cycle, then no precedence functions exist. If there are no cycle, let f(a) be the length of the longest path beginning at the group of f_a; let g(a) be the length of the longest path beginning at the group of g_a.

Example

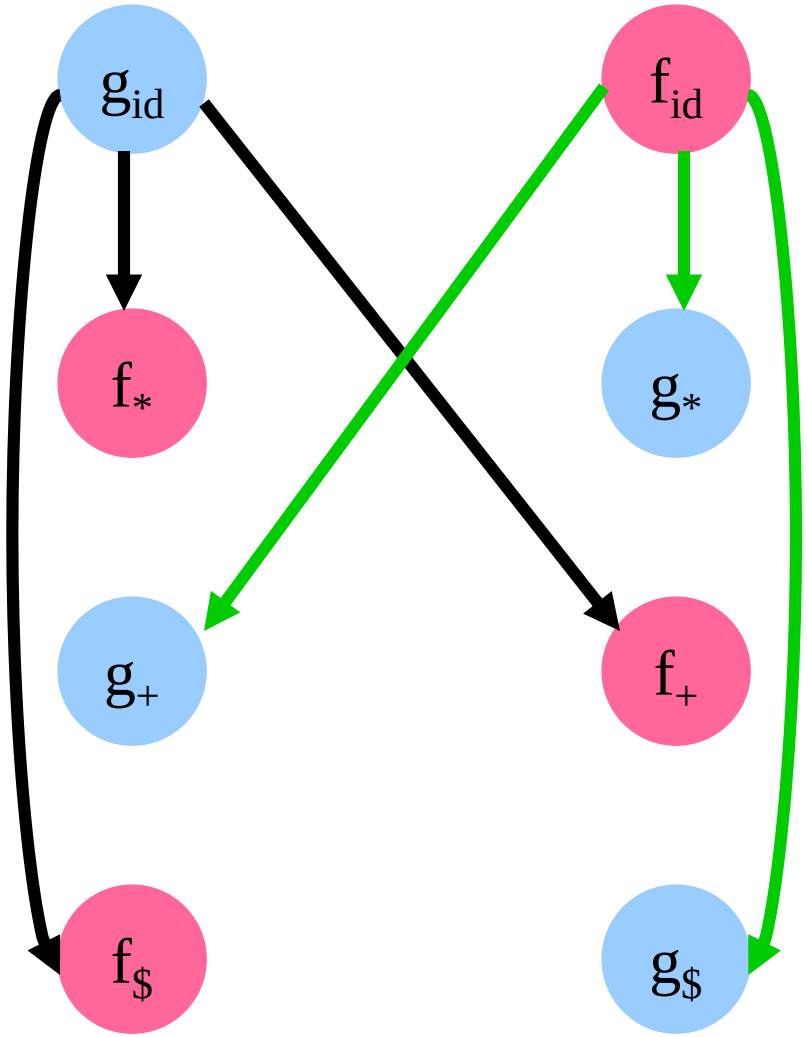


	id	+	*	$
id		·>	·>	·>
+	<·	·>	<·	·>
*	<·	·>	·>	·>
$	<·	<·	<·	

Create symbols f_a and g_a for each *a* that is a terminal or $.

No equal relationship

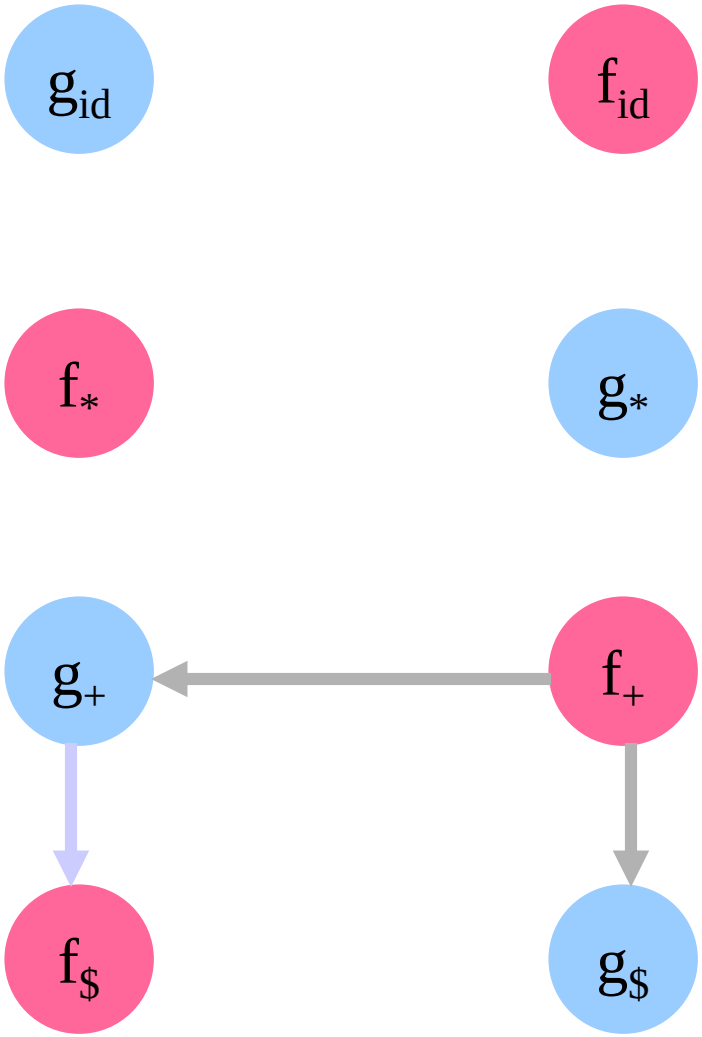
Example



	id	+	*	$
id		·>	·>	·>
+	<·	·>	<·	·>
*	<·	·>	·>	·>
$	<·	<·	<·	

For any *a* and *b*, if *a* <·*b* , place an edge from the group of g_b to the group of f_a. Of *a* ·> *b*, place an edge from the group of f_a to that of g_b.

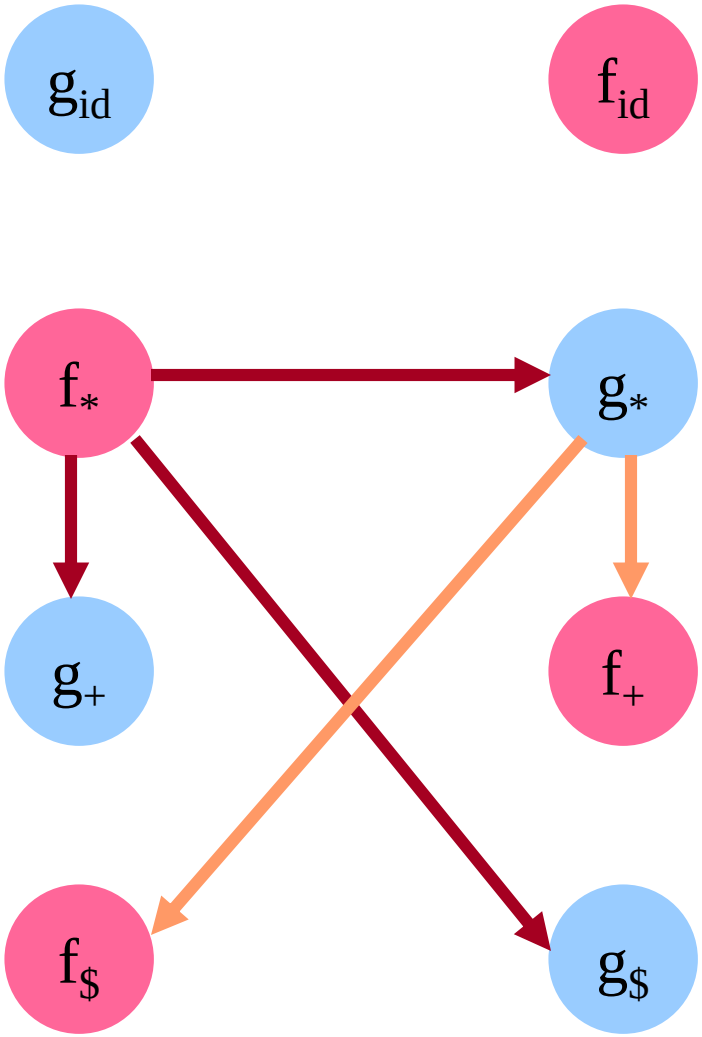
Example



	id	+	*	$
id		·>	·>	·>
+	<·	·>	<·	·>
*****	<·	·>	·>	·>
$	<·	<·	<·	

For any *a* and *b*, if *a* <·*b* , place an edge from the group of g_b to the group of f_a. Of *a* ·> *b*, place an edge from the group of f_a to that of g_b.

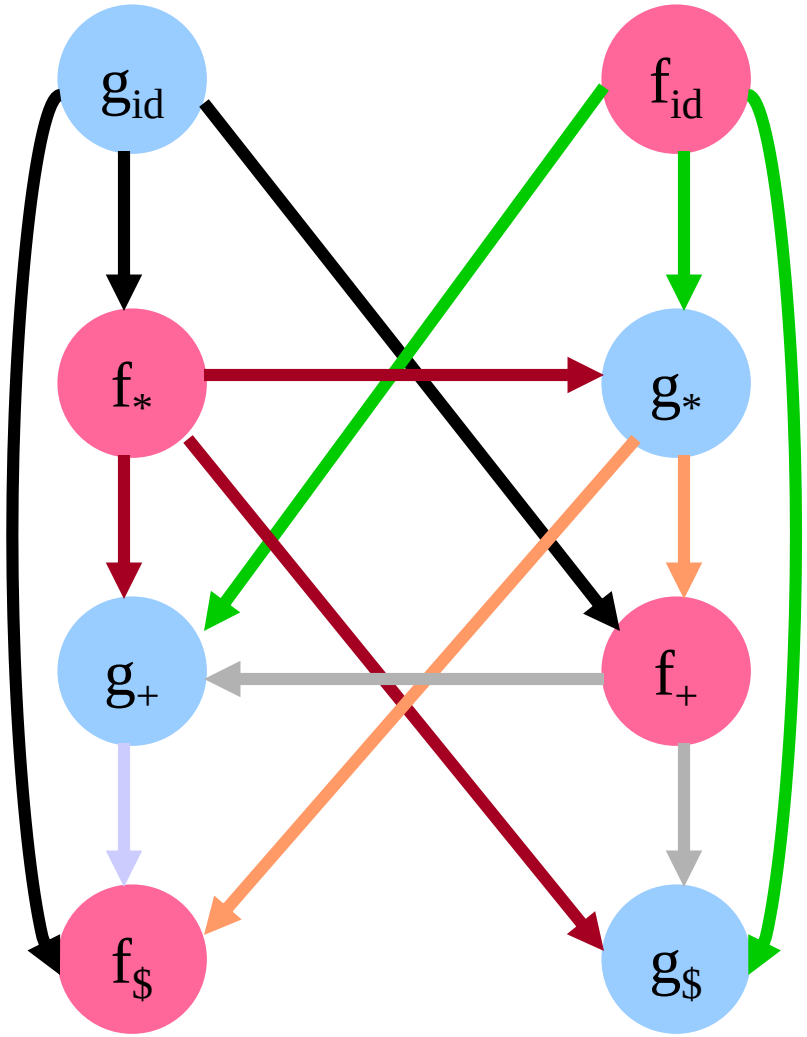
Example



	id	+	*	$
id		·>	·>	·>
+	<·	·>	<·	·>
*	<·	·>	·>	·>
$	<·	<·	<·	

For any *a* and *b*, if *a* <·*b* , place an edge from the group of g_b to the group of f_a. Of *a* ·> *b*, place an edge from the group of f_a to that of g_b.

Example

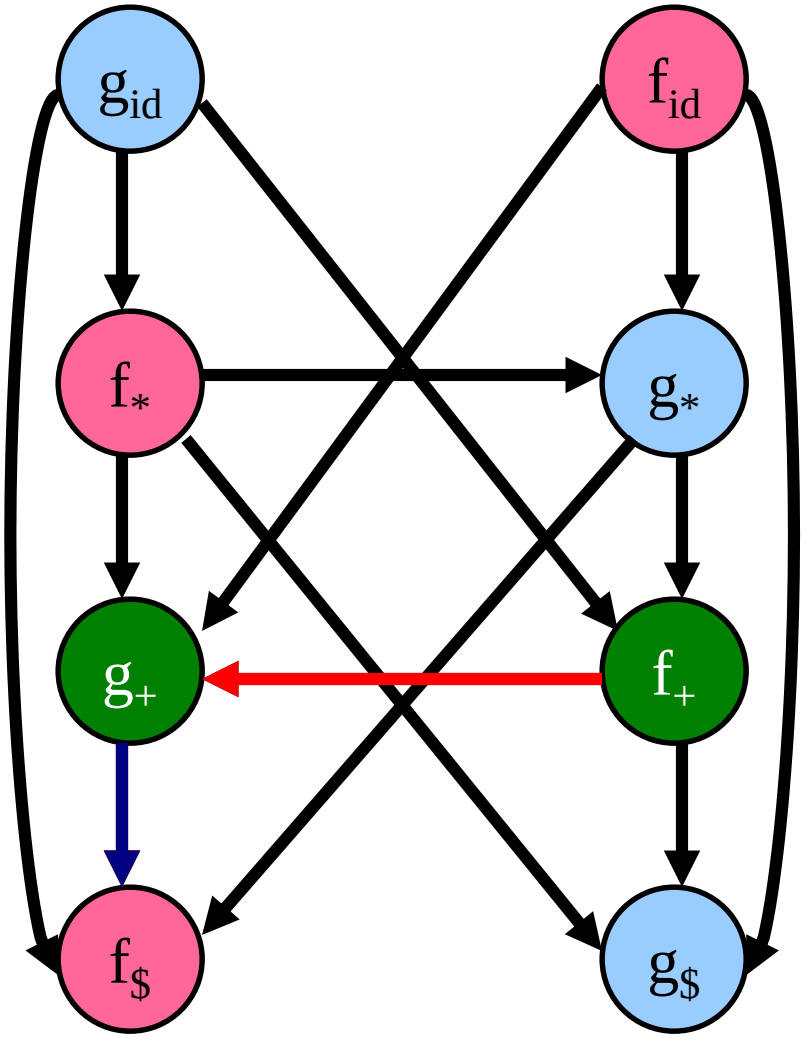


	id	+	*	$
id		·>	·>	·>
+	<·	·>	<·	·>
*	<·	·>	·>	·>
$	<·	<·	<·	

	+	*	Id	$
f				
g				

LONGEST PATH

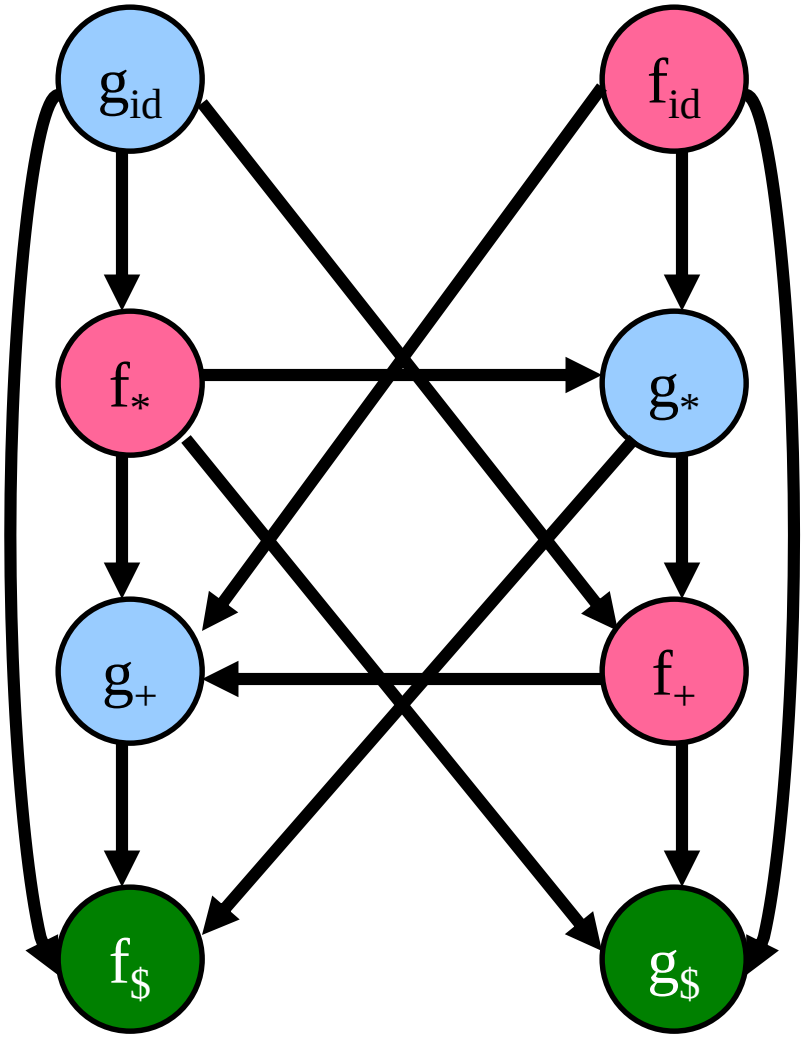
Example



	id	+	*	$
id		·>	·>	·>
+	<·	·>	<·	·>
*	<·	·>	·>	·>
$	<·	<·	<·	

	+	*	Id	$
f	2			
g	1			

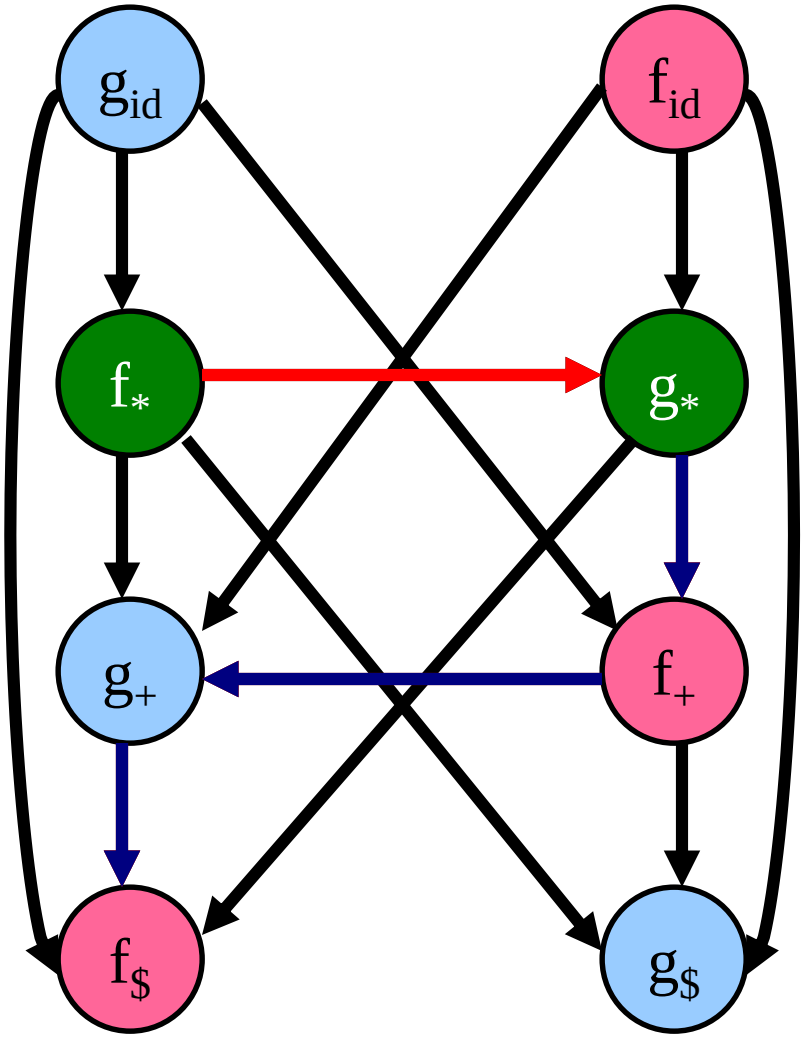
Example



	id	+	*	$
id		·>	·>	·>
+	<·	·>	<·	·>
*	<·	·>	·>	·>
$	<·	<·	<·	

	+	*	Id	$
f	2			0
g	1			0

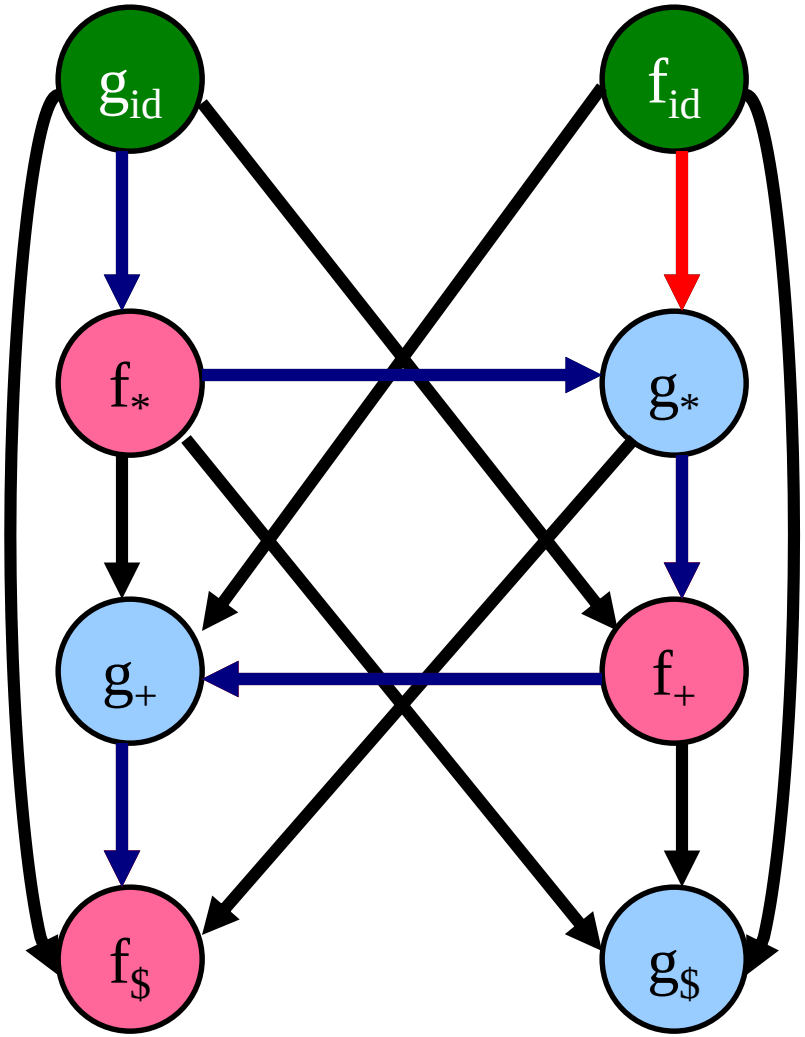
Example



	id	+	*	$
id		·>	·>	·>
+	<·	·>	<·	·>
*	<·	·>	·>	·>
$	<·	<·	<·	

	+	*	Id	$
f	2	4		0
g	1	3		0

Example



	id	+	*	$
id		·>	·>	·>
+	<·	·>	<·	·>
*	<·	·>	·>	·>
$	<·	<·	<·	

	+	*	Id	$
f	2	4	4	0
g	1	3	5	0

Disadvantages of Operator Precedence Parsing

- **Disadvantages**:
  - It cannot handle the unary minus (the lexical analyzer should handle the unary minus).
  - Small class of grammars.
  - Difficult to decide which language is recognized by the grammar.

- **Advantages**:
  - simple
  - powerful enough for expressions in programming languages

Error Recovery in Operator-Precedence Parsing

Error Cases:

1. No relation holds between the terminal on the top of stack and the next input symbol.
2. A handle is found (reduction step), but there is no production with this handle as a right side

Error Recovery:

1. Each empty entry is filled with a pointer to an error routine.
2. Decides the popped handle "looks like" which right hand side. And tries to recover from that situation.

Handling Shift/Reduce Errors

When consulting the precedence matrix to decide whether to shift or reduce, we may find that no relation holds between the top stack and the first input symbol.

To recover, we must modify (insert/change)

1. Stack or
2. Input or
3. Both.

We must be careful that we don't get into an infinite loop.

Example

e1: Called when : whole expression is missing

insert **id** onto the input

issue diagnostic: 'missing operand'

e2: Called when : expression begins with a right parenthesis delete) from the input

issue diagnostic: 'unbalanced right parenthesis'

e3: Called when : id or) is followed by id or (

insert **+** onto the input

issue diagnostic: 'missing operator'

e4: Called when : expression ends with a left parenthesis

pop (from the stack

issue diagnostic: 'missing right parenthesis'

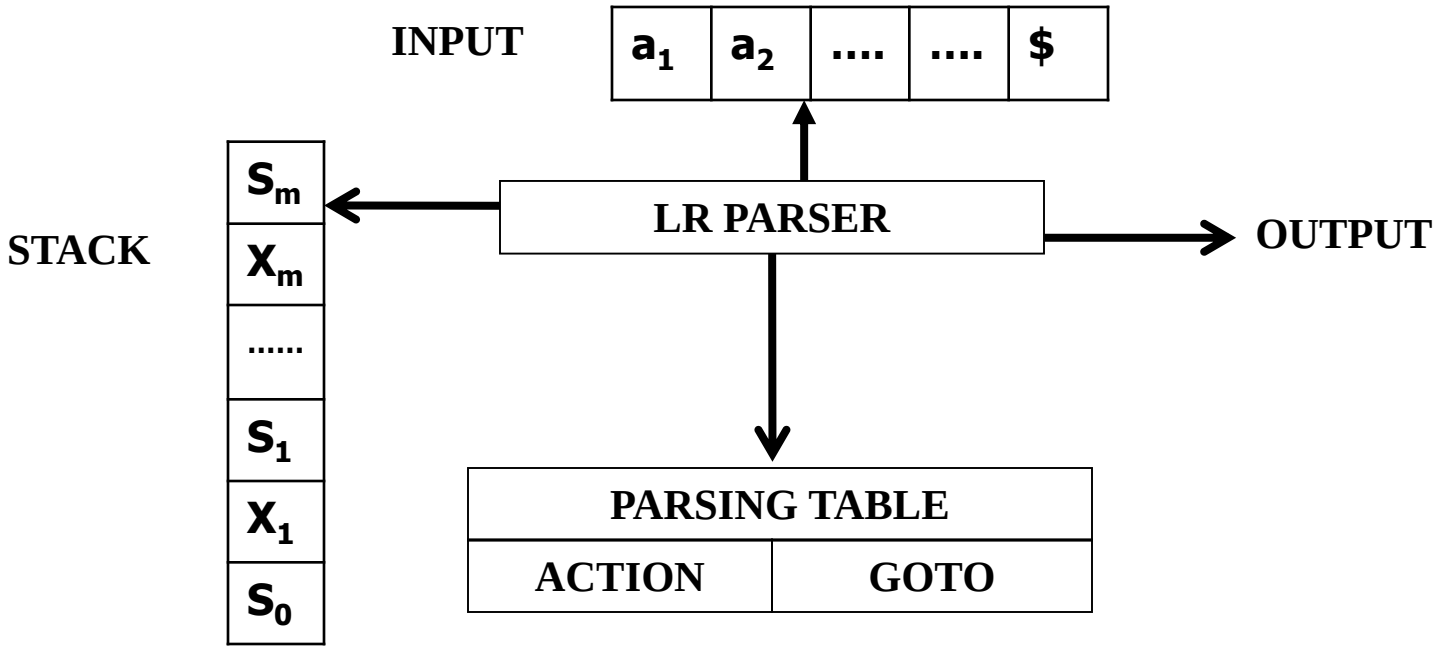
	id	(	)	$
id	e3	e3	·>	·>
(	<·	<·	=.	e4
)	e3	e3	·>	·>
$	<·	<·	e2	e1

LR(k) – Bottom Up Parsing

L: left-to-right scanning of the input

R: constructing a rightmost derivation in reverse

k: the number of input symbols of lookahead that are used in making parsing decisions. (default value is 1)



LR(k) Parsing

- **Parsing tables**:
  - ✓ Indexed by the state at the stack top and the current input symbol; composed of action[ ] & goto[ ] tables
  - ✓ **action[s_m, a_i]** (**s_m**:state, **a_i** :terminal)

    - ➢shift *s*, where *s* is a state
    - ➢reduce by A → β
    - ➢accept
    - ➢error

  - ✓ **goto[s. X]** (**s:** state, **X:** non-terminal) produces a state (transition function of a deterministic finite automaton)

Types of LR(K) parsers: SLR, Canonical LR and LALR

LR(k) Parsing

action[s_m, a_i] (**s_m**:state, **a_i** :terminal)

Configuration of LR PARSER is a

$$(s_0X_1\ s_1X_2\ s_2\ldots\ldots\ s_{m-1}X_m\mathbf{s_m}\ ,\ \mathbf{a_i}a_{i+1}a_{i+2}\ldots\ldots\ldots a_n\$)$$

This configuration represent the right sentential form

$$X_1X_2\ldots\ldots\ X_m\ a_ia_{i+1}a_{i+2}\ldots\ldots\ldots a_n$$

When action = shift s:

$$(s_0X_1s_1X_2s_2\ldots\ldots\ s_{m-1}X_ms_m\mathbf{a_is}\ ,\ a_{i+1}a_{i+2}\ldots\ldots\ldots a_n\$)$$

When action = reduce $A \rightarrow \beta$

$$(s_0X_1s_1X_2s_2\ldots\ldots X_{m-r}s_{m-r}\mathbf{A\ s}\ ,\ a_ia_{i+1}\ldots\ldots\ldots a_n\$)$$

Where s = goto[s_{m-r}, A] and r is the length of β

If accept = action, parsing is completed.

If action = error, the parser has discovered an error and calls an error recovery routine.

LR Parsing Program

```
stack.push(INVALID);  stack.push(s0);
not_found = true;
token = scanner.next_token();
do while (not_found) {
        s = stack.top();
        if ( ACTION[s,token] = "reduce A → β" ) then {
                stack.popnum(2 * | β |); // pop 2 * | β | symbols
        s = stack.top();
        stack.push( A );
        stack.push(GOTO[ s, A]); }
        else if ( ACTION[ s, token] == "shift si" ) then {
                stack.push( token ); stack.push( si);
                token ← scanner.next_token(); }
        else if ( ACTION[ s , token] == "accept" & token = EOF )
        then not_found = false;
        else report a syntax error and recover;
}
report success;
```

Items (Parser States)

- A shift-reduce parser is always looking for the next handle
- A parser state, or **LR(0) item (SLR), is a production and a position (indicated by "·")**
- **Production $A \rightarrow XYZ$ yields the four items**
  - **$A \rightarrow .XYZ$**
  - **$A \rightarrow X.YZ$**
  - **$A \rightarrow XY.Z$**
  - **$A \rightarrow XYZ.$**

$A \rightarrow \varepsilon$ then only one item , **$A \rightarrow .$**

- **$A \rightarrow a.g$**, which means that **a** is on the stack and the parser is looking for **g**
- If the parser finds **g**, then it has the whole handle

Valid Items

- An item $A \rightarrow \alpha \cdot \gamma$ is valid for a viable prefix $\sigma\alpha$ if $S \Rightarrow_r^* \sigma A z \Rightarrow_r^* \sigma\alpha\gamma z$
- First construct DFA from Grammar to recognize Viable Prefix. In each state we group some items.

Canonical LR(0) Item Set

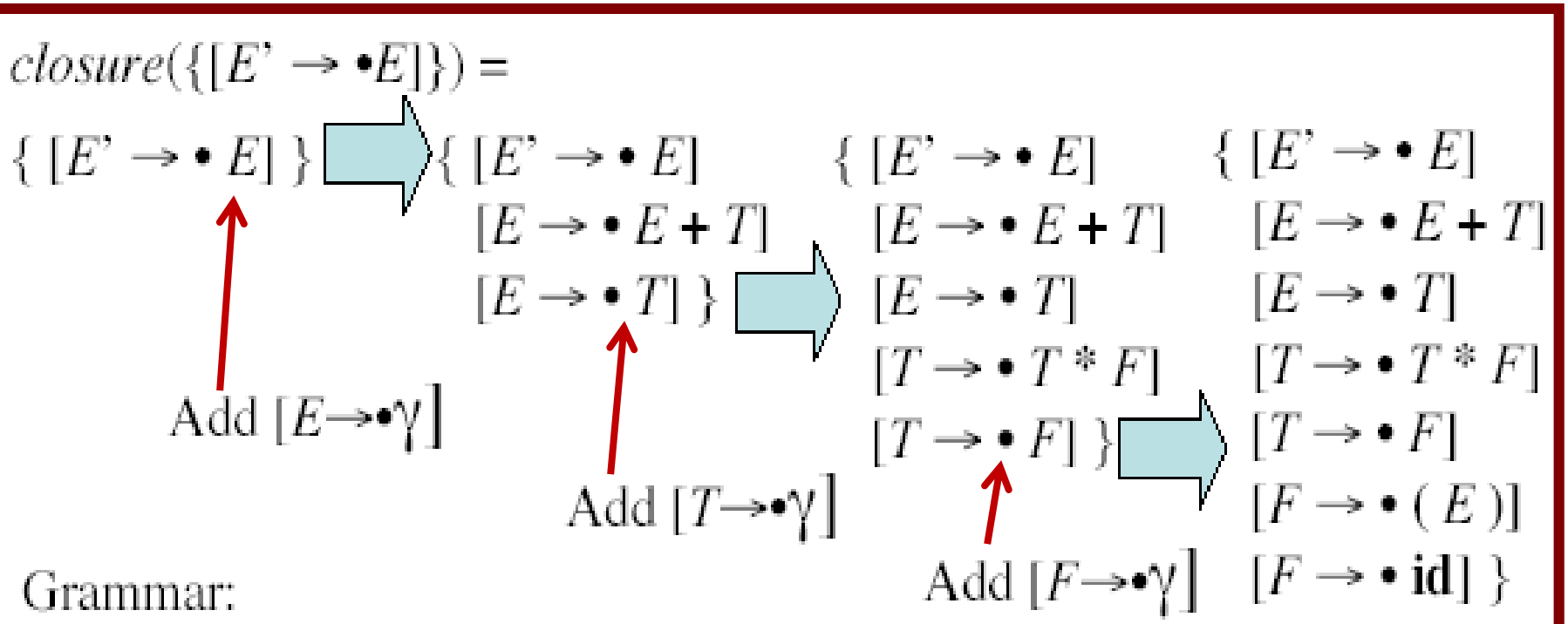
- We require Augmented Grammar and two Function
  - Closure Operation
  - GOTO
- Augmented Grammar for a grammar G(starting symbol S) is G', start with a new starting symbol S', with S' $\rightarrow$ S

Closure Operation

- If I is a set of items for a grammar G, then *closure(I)* is the set of items constructed form I by two rules:

1. Start with $closure(I) = I$
2. If $[A \rightarrow \alpha \bullet B\beta] \in closure(I)$ then for each production $B \rightarrow \gamma$ in the grammar, add the item $[B \rightarrow \bullet\gamma]$ to I if not already in I
3. Repeat 2 until no new items can be added

Closure Operation for LR(0) Items…Example



Grammar:

$E \rightarrow E + T \mid T$

$T \rightarrow T * F \mid F$

$F \rightarrow (E)$

$F \rightarrow \mathbf{id}$

Closure Operation Algo…

function ***closure(I)*** ;

begin

J:= I;

repeat

for each item $A \rightarrow \alpha \cdot B\beta$ in ***J*** and each production $B \rightarrow \gamma$ of ***G*** such that $B \rightarrow .\gamma$ is not in ***J*** do

add $B \rightarrow .\gamma$ to J

until no more items can be added to ***J***;

return ***J***

end

GOTO Operation

- *GOTO(I,X)* : *I* is a set of items and *X* is a grammar symbol

1. For each item $[A \rightarrow \alpha \bullet X \beta] \in I$, add the set of items *closure*($\{[A \rightarrow \alpha X \bullet \beta]\}$) to *goto*(*I*,*X*) if not already there
2. Repeat step 1 until no more items can be added to *goto*(*I*,*X*)
3. Intuitively, *goto*(*I*,*X*) is the set of items that are valid for the viable prefix γX when *I* is the set of items that are valid for γ

GOTO Operation for LR(0) Items…Example

Suppose $I = \{\ [E' \rightarrow \bullet\ E]$
$[E \rightarrow \bullet\ E + T]$
$[E \rightarrow \bullet\ T]$
$[T \rightarrow \bullet\ T * F]$
$[T \rightarrow \bullet\ F]$
$[F \rightarrow \bullet\ (\ E\ )]$
$[F \rightarrow \bullet\ \mathbf{id}]\ \}$

Then $goto(I,E)$
$= closure(\{[E' \rightarrow E \bullet, E \rightarrow E \bullet + T]\})$
$= \{\ [E' \rightarrow E \bullet]$
$[E \rightarrow E \bullet + T]\ \}$

Grammar:

$E \rightarrow E + T \mid T$
$T \rightarrow T * F \mid F$
$F \rightarrow (\ E\ )$
$F \rightarrow \mathbf{id}$

GOTO Operation for LR(0) Items…Example

Suppose $I = \{ [E' \rightarrow E \bullet], [E \rightarrow E \bullet + T] \}$

Then $goto(I,+) = closure(\{[E \rightarrow E + \bullet T]\}) = \{ [E \rightarrow E + \bullet T]$
$[T \rightarrow \bullet T * F]$
$[T \rightarrow \bullet F]$
$[F \rightarrow \bullet (E)]$
$[F \rightarrow \bullet \mathbf{id}] \}$

Grammar:

$E \rightarrow E + T \mid T$

$T \rightarrow T * F \mid F$

$F \rightarrow (E)$

$F \rightarrow \mathbf{id}$

Set of Item Construction Algo…

procedure items(G');

begin

$I_0 = CLOSURE(\{S' \rightarrow .S\})$ and $C = \{I_0\}$

repeat

for each $I \in C$ and grammar symbol X

such that $GOTO(I, X)$ is not empty and not in C do

add $GOTO(I, X)$ to C

until no more sets of items can be added to C

end

Facts About Valid Items

- If $A \rightarrow \alpha \cdot B\gamma$ is valid for $\sigma\alpha$,

  then $B \rightarrow \cdot\beta$ is valid for $\sigma\alpha$

  - $S \Rightarrow_r^* \sigma Az \Rightarrow_r^* \sigma\alpha B\gamma z \Rightarrow_r^* \sigma\alpha Byz \Rightarrow_r^* \sigma\alpha\beta yz$

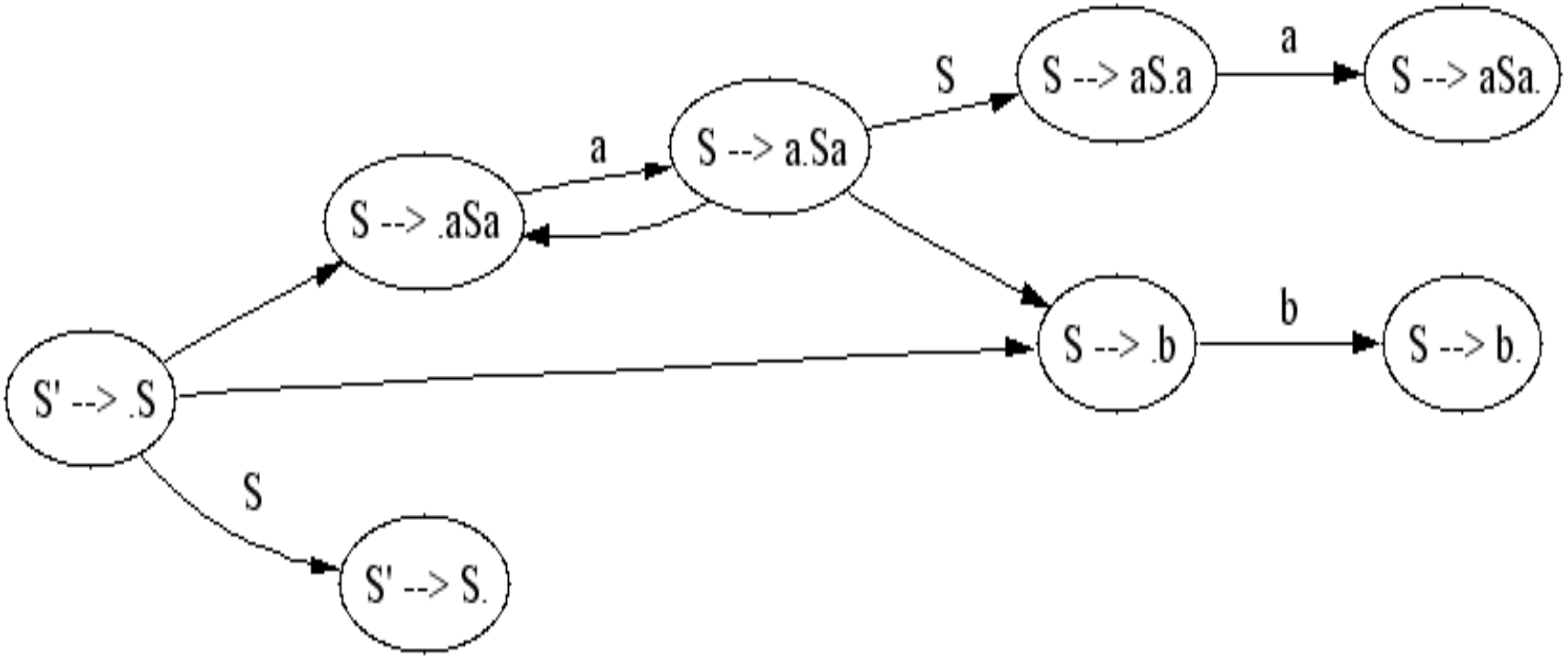
- If $A \rightarrow \alpha \cdot X\gamma$ is valid for $\sigma\alpha$,

  then $A \rightarrow \alpha X \cdot \gamma$ is valid for $\sigma\alpha X$

  - $S \Rightarrow_r^* \sigma Az \Rightarrow_r^* \sigma\alpha X\gamma z$

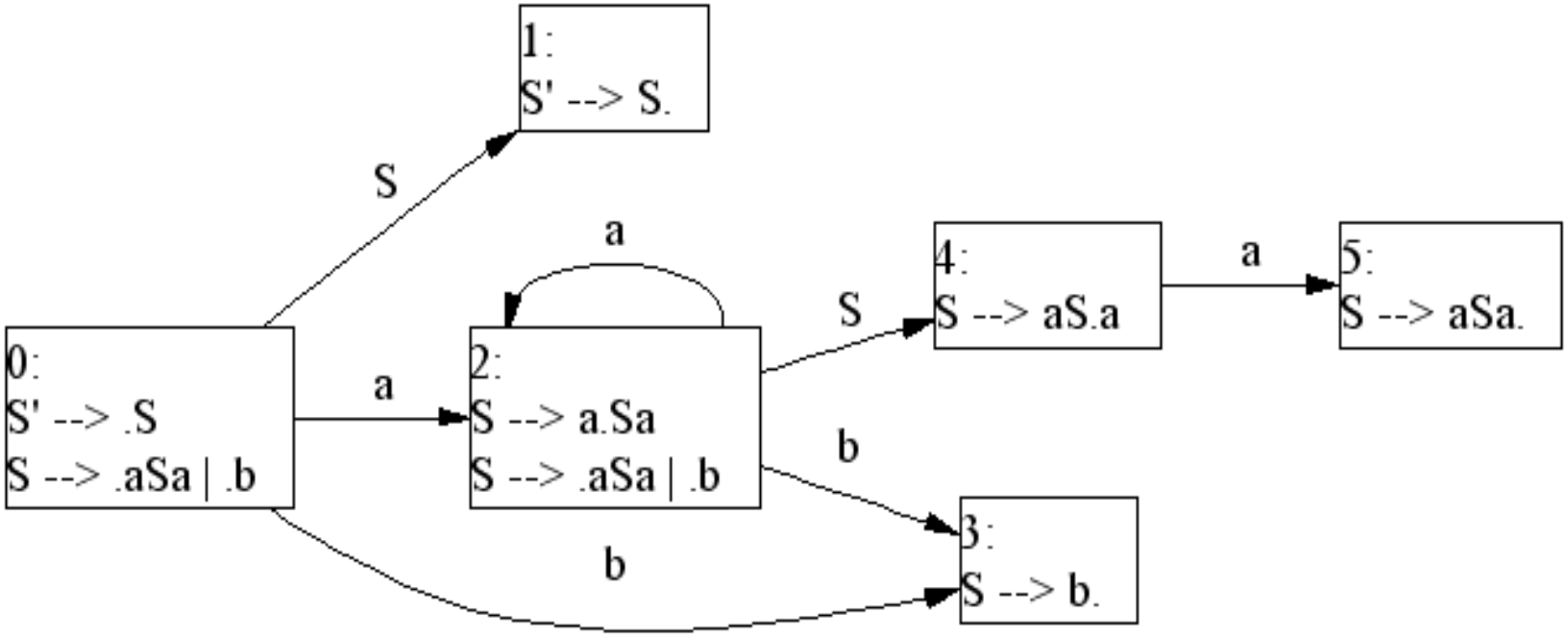
LR(0) Item Automaton

- The items (parser states) act like the states of an NFA!
  - S → aSa | b

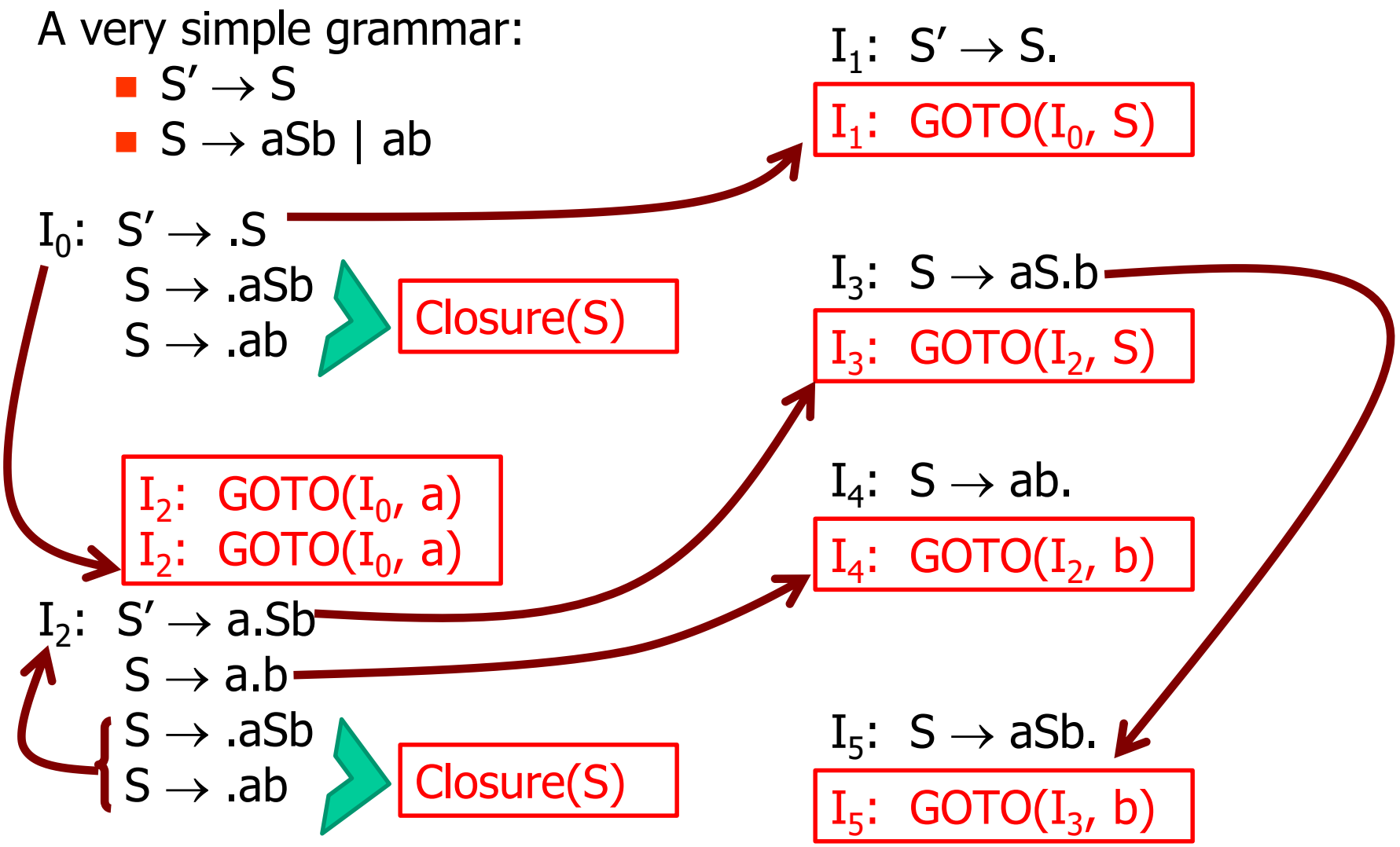


LR(0) Item Automaton

- The items (parser states) act like the states of an NFA!
  - S → aSa | b
- Equivalent DFA



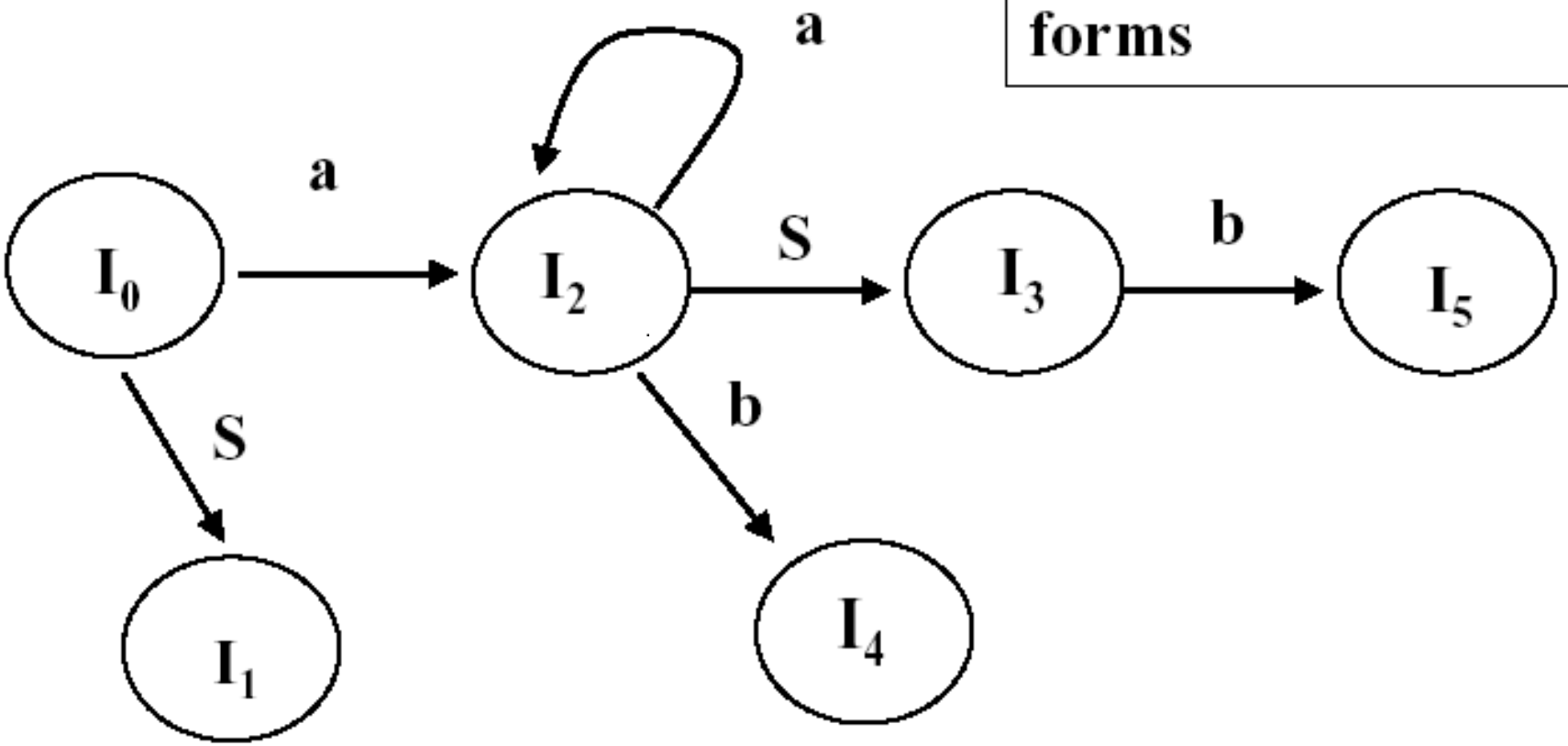
LR(0) Items…Example



LR(0) Items…Example

Example

Recognizes prefixes of right sentential forms



Ex: $E' \rightarrow E$
$E \rightarrow E + T \mid T$
$T \rightarrow T * F \mid F$
$F \rightarrow (E) \mid id$

LR(0) Items…Example

I_0: $E' \rightarrow . E$
$E \rightarrow . E + T$
$E \rightarrow . T$
$T \rightarrow . T * F$
$T \rightarrow . F$
$F \rightarrow . (E)$
$F \rightarrow . id$

I_1: $E' \rightarrow E .$
$E \rightarrow E . + T$

I_2: $E \rightarrow T.$
$T \rightarrow T. * F$

I_3: $T \rightarrow F.$

I_4: $F \rightarrow (. E)$
$E \rightarrow . E + T$
$E \rightarrow . T$
$T \rightarrow . T * F$
$T \rightarrow . F$
$F \rightarrow . (E)$
$F \rightarrow . id$

I_5: $F \rightarrow id .$

I_6: $E \rightarrow E + . T$
$T \rightarrow . T * F$
$T \rightarrow . F$
$F \rightarrow . (E)$
$F \rightarrow . id$

I_7: $T \rightarrow T * . F$
$F \rightarrow . (E)$
$F \rightarrow . id$

I_8: $F \rightarrow (E .)$
$E \rightarrow E . + T$

I_9: $E \rightarrow E + T.$
$T \rightarrow T . * F$

I_{10}: $T \rightarrow T * F .$

I_{11}: $F \rightarrow (E).$

LR(0) Item Automaton

- Recognizes prefixes of right sentential forms
  - Move proceeds through rule RHS
  - Closure expands rightmost nonterminal
- Recognizes viable prefixes
  - Paths end at end of rule RHS (handle)
- Produces valid items for viable prefixes

Valid Items and Parsing

- Valid items are informative
  - If $A \rightarrow \beta \cdot \gamma$ is valid for parsing stack $\alpha\beta$, then shift (and keep looking for γ)
  - If $A \rightarrow \beta \cdot$ is valid for parsing stack $\alpha\beta$, then the handle is there, so reduce β to A

Construction of an SLR Paring Table

1. Augment the grammar with $S' \rightarrow S$
2. Construct the set $\boldsymbol{C = \{I_0, I_1, \ldots, I_n\}}$ **of** ***LR(0) items***
3. Compute the action function entries:

   -- If $[A \rightarrow \alpha \bullet a\, \beta] \in I_i$ and $goto(I_i, a) = I_j$ then set $action[i,a] = shift\ j$ *(a is terminal)*

   -- If $[A \rightarrow \alpha \bullet] \in I_i$ then set $action[I, a] = reduce\ A \rightarrow \alpha$ for all $a \in FOLLOW(A)$ (apply only if $A \neq S'$)

   -- If $[S' \rightarrow S \bullet]$ is in I_i then set $action[i,\$] = accept$

   -- all undefined entries are **error** entries (i.e. blank)
4. Compute the GOTO (next) state entries:

   -- If $goto(I_i, A) = I_j$ then set $goto[i,A] = j$
5. all undefined entries are error entries (i.e. blank)
6. Repeat 3 to 5 until no more entries added
7. Initial state of the parser is the one constructed from the set of items containing [S′ → .S].

ACTION Function

Compute the action function entries:

1. If $[A \rightarrow \alpha \bullet a\ \beta] \in I_i$ and $goto(I_i, a) = I_j$ then set $action[i,a] = shift\ j$ *(a is terminal)*
2. If $[A \rightarrow \alpha \bullet] \in I_i$ then set $action[I, a] = reduce\ A \rightarrow \alpha$ for all $a \in FOLLOW(A)$ (apply only if $A \neq S'$)
3. If $[S' \rightarrow S \bullet]$ is in I_i then set $action[i,\$] = accept$
4. all undefined entries are **error** entries (i.e. blank)

GOTO Function

If $goto(I_i, A) = I_j$ then set $goto[i, A] = j$

SLR(1) Parsing Table

- SLR(1) ("simple LR")

Uses **input lookahead** in addition to the **LR(0) table** to determine parsing actions

0: E′ → E

1: E → E + T

2: E → T

3: T → T * F

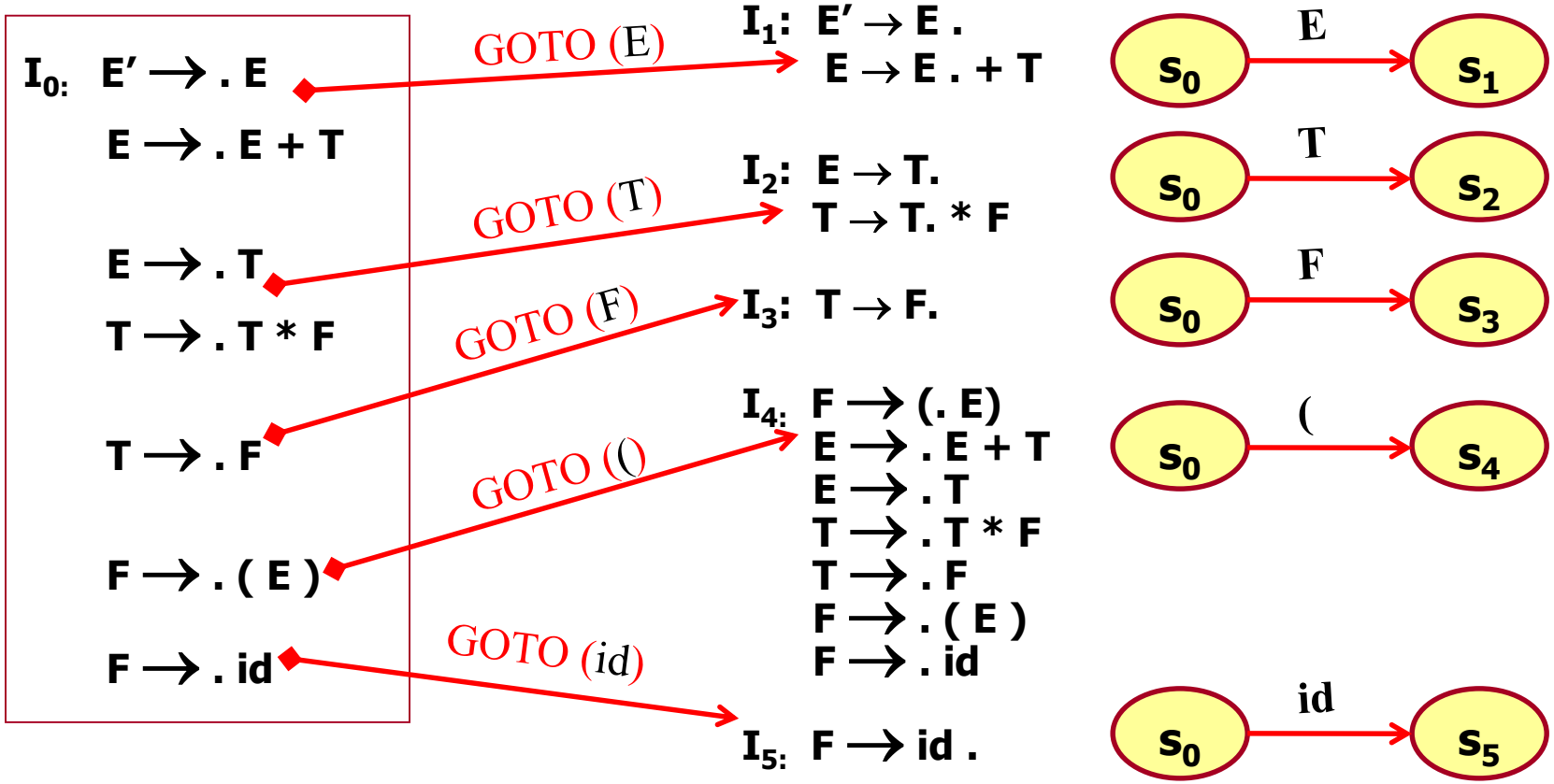
4: T → F

5: F → (E)

6: F → id

Ex: $E' \rightarrow E$
$E \rightarrow E + T \mid T$
$T \rightarrow T * F \mid F$
$F \rightarrow (E) \mid id$

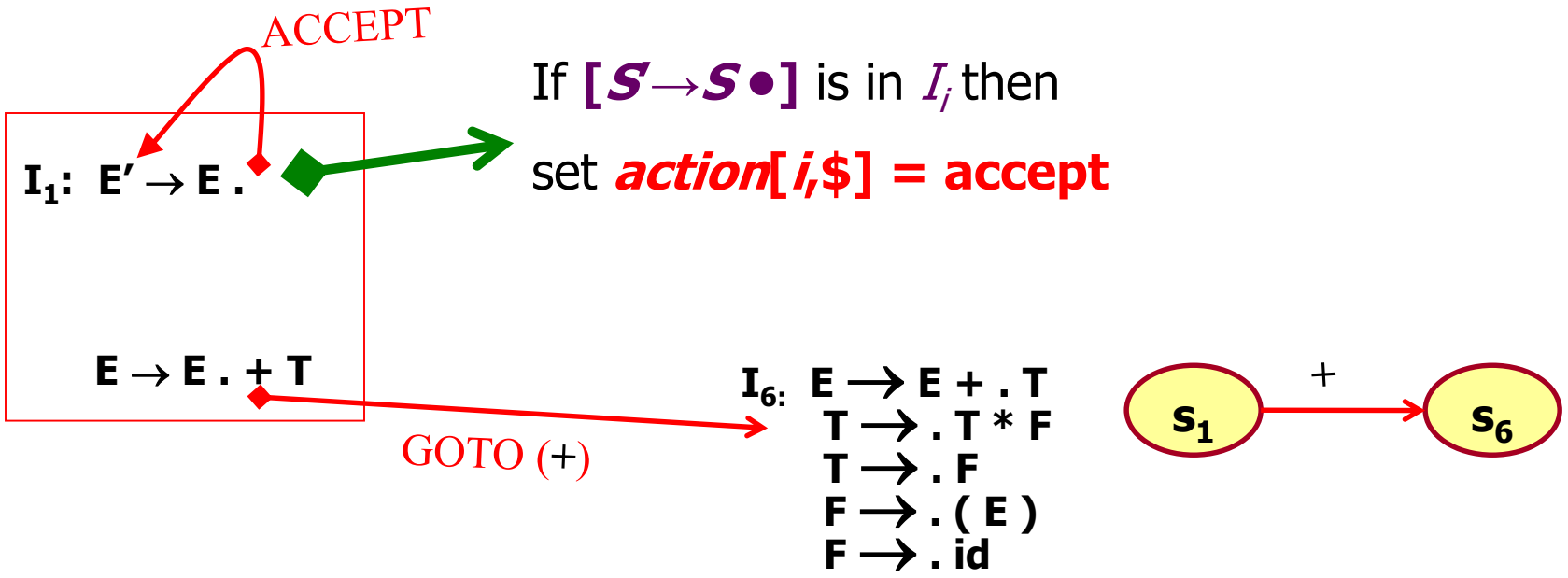
SLR(1) Parsing Table



STATE	ACTION						GOTO		
	id	+	*	(	)	$	E	T	F
0	S5			S4			1	2	3

Ex: $E' \rightarrow E$
$E \rightarrow E + T \mid T$
$T \rightarrow T * F \mid F$
$F \rightarrow (E) \mid id$

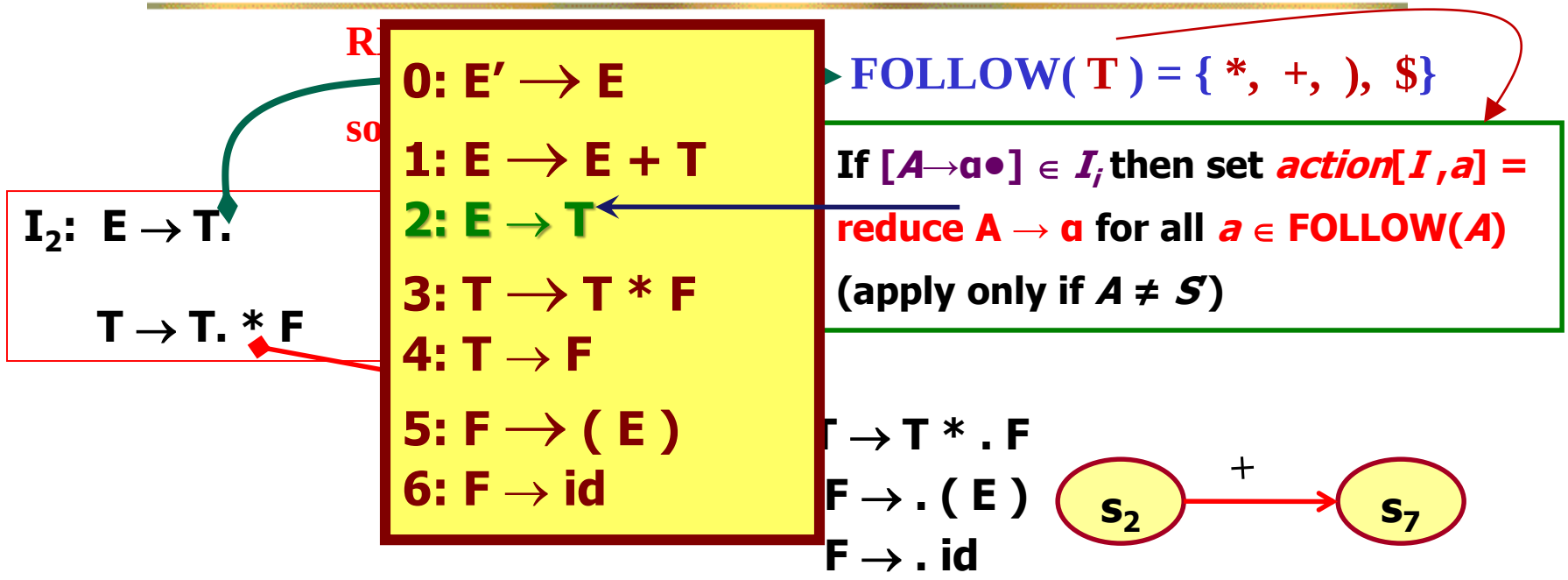
SLR(1) Parsing Table



STATE	ACTION						GOTO		
	id	+	*	(	)	$	E	T	F
0	S5			S4			1	2	3
1		S6				ACCEPT			

Ex: $E' \rightarrow E$
$E \rightarrow E + T \mid T$
$T \rightarrow T * F \mid F$
$F \rightarrow (E) \mid id$

SLR(1) Parsing Table



STATE	ACTION						GOTO		
	id	+	*	(	)	$	E	T	F
0	S5			S4			1	2	3
1		S6				ACCEPT			
2		R2	S7		R2	R2			

Ex: $E' \rightarrow E$
$E \rightarrow E + T \mid T$
$T \rightarrow T * F \mid F$
$F \rightarrow (E) \mid id$

SLR(1) Parsing Table

STATE	ACTION						GOTO		
	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			

SLR(1) Parsing Example

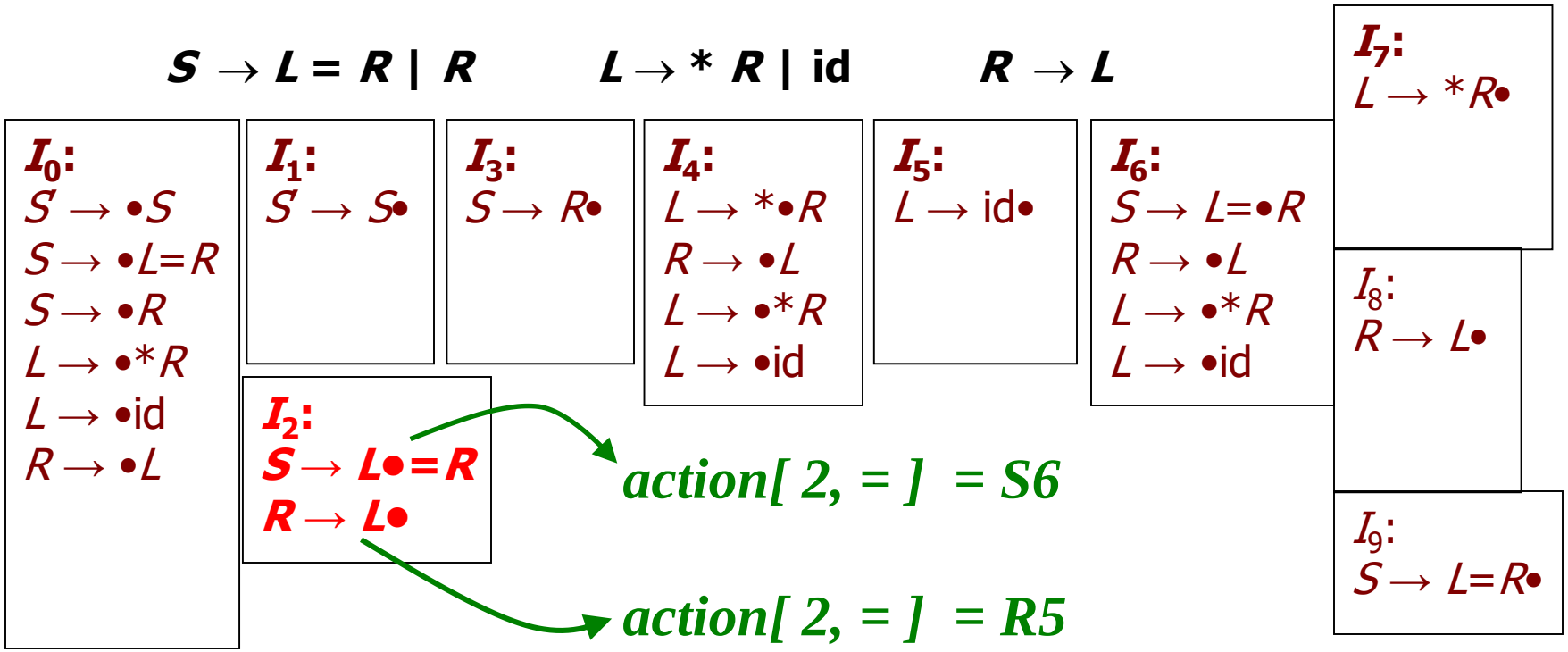
Stack	Input	Action
$ 0	n * n + n $	Shift 5
$ 0 n 5	* n + n $	Reduce 6
$ 0 F 3	* n + n $	Reduce 4
$ 0 T 2	* n + n $	Shift 7
$ 0 T 2 * 7	n + n $	Shift 5
$ 0 T 2 * 7 n 5	+ n $	Reduce 6
$ 0 T 2 * 7 F 10	+ n $	Reduce 3
$ 0 T 2	+ n $	Reduce 2
$ 0 E 1	+ n $	Shift 6
$ 0 E 1 + 6	n $	Shift 5
$ 0 E 1 + 6 n 5	$	Reduce 6
$ 0 E 1 + 6 F 3	$	Reduce 4
$ 0 E 1 + 6 T 9	$	Reduce 1
$ 0 E 1	$	ACCEPT

SLR(1) Parsing Conflicts

- If a state contains inconsistent items, then the grammar is not LR(0)
  - $E \rightarrow E + T\cdot$, $T \rightarrow T\cdot * F$ both are in same state
    - SHIFT/REDUCE conflict
  - $E \rightarrow E + E\cdot \mid E * E\cdot$ both are in same state
    - REDUCE/REDUCE conflict
- Most "real" grammars are not LR(0)

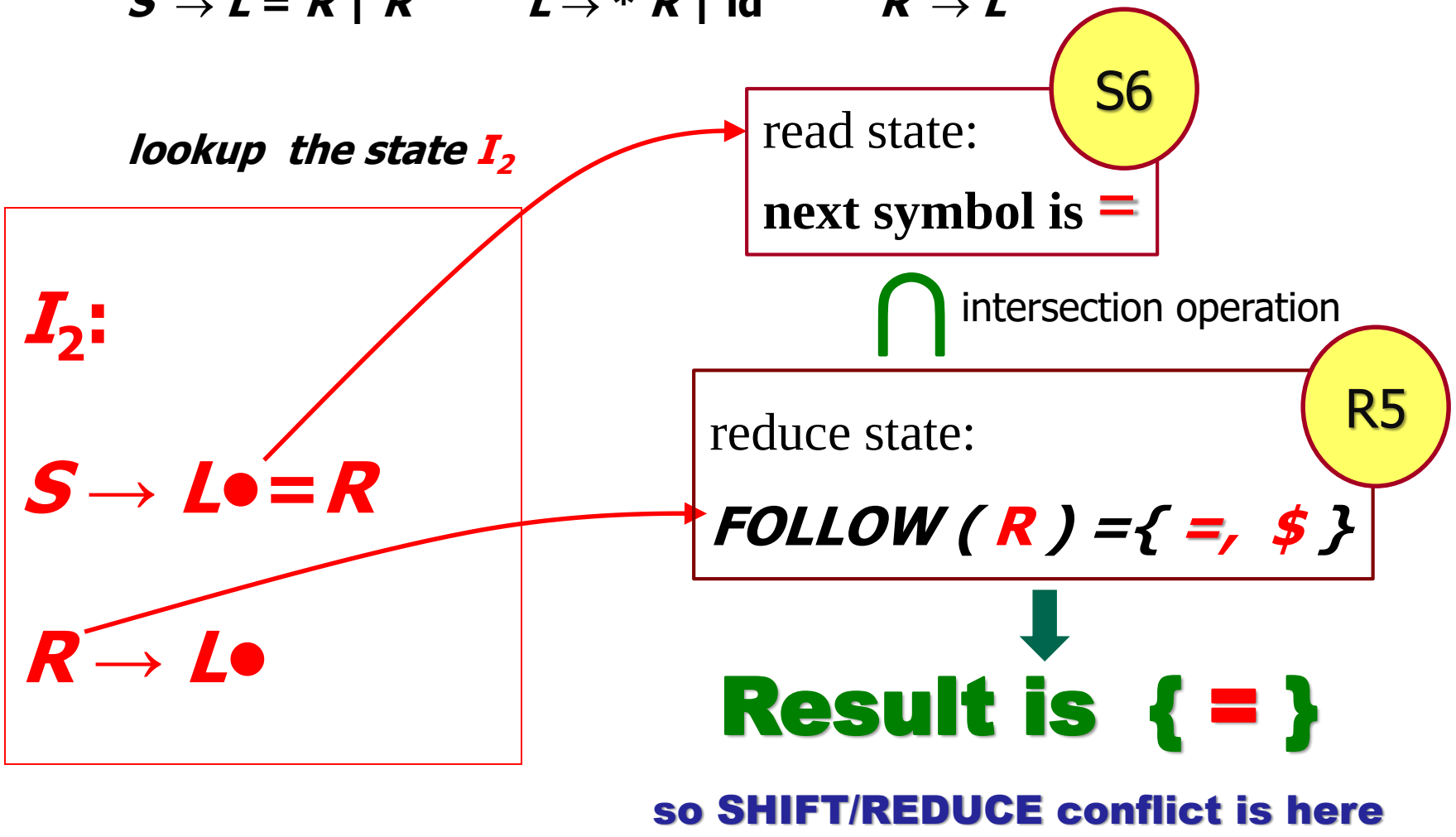
SLR(1) and Ambiguity

- **Every SLR grammar is unambiguous, but not every unambiguous grammar is SLR**
- Consider for example the unambiguous grammar



SLR and Ambiguity

$S \rightarrow L = R \mid R$ $L \rightarrow * R \mid id$ $R \rightarrow L$



Exercise

Ex.1 $S \rightarrow CC$ $C \rightarrow cC$ $C \rightarrow d$

Ex.2 $S \rightarrow AaAb$ $S \rightarrow BbBa$ $A \rightarrow \varepsilon$ $B \rightarrow \varepsilon$

Ex.3 $S \rightarrow xAy \mid xBy \mid xAz$ $A \rightarrow aS \mid q$ $B \rightarrow q$

Ex.4 $S \rightarrow aSbS \mid bSaS \mid \varepsilon$

Exercise

Ex.5 $S \rightarrow Aa \mid aAc \mid Bc \mid bBa \quad A \rightarrow d \quad B \rightarrow d$

Ex.6 $A \rightarrow AB \mid \varepsilon \quad B \rightarrow aB \mid b$

Ex.7 stmt → **if** expr **then** stmt | matched_stmt

matched_stmt → **if** expr **then** matched_stmt **else** stmt

| **other**

LL(1) and SLR(1)

- LL(1) but not LR(0)

  $E \rightarrow TE'$

  $E' \rightarrow + TE' \mid e$

  $T \rightarrow n \mid (E)$

- LR(0) but not LL(1)

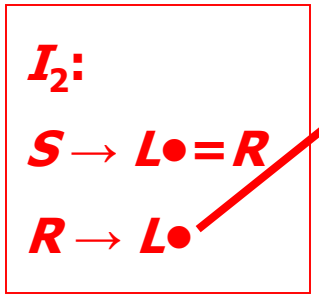
  $A \rightarrow aa \mid ab$

Other Bottom-Up Algorithms

- LR(1) ("canonical LR")
  - Incorporates input lookahead into LR items (**LR(1) items**)
- LALR(1) ("lookahead LR")
  - Uses more **compact LR(1) items**

Canonical LR(1) Parsing

$S \rightarrow L = R \mid R$ $\quad$ $L \rightarrow * R \mid \text{id}$ $\quad$ $R \rightarrow L$



- Reduce state so L can be reduce into R
- But there is no **Right sentential form** that begins **R = ...** in the grammar
- So it is not a valid reduce operation for the parsing
- How this type of information is added with parsing method
- Concept of **LR(1) item set** (i.e. item set with one lookahead symbol)

LR(1) Items

- **LR(1) item = LR(0) item + lookahead**

LR(0) item: **LR(1) item:**

$[A \rightarrow \alpha \bullet \beta]$ **$[A \rightarrow \alpha \bullet \beta, a]$**

An *LR(1) item* **$[A \rightarrow \alpha \bullet \beta, a]$** contains a *lookahead* terminal ***a***, meaning α already on top of the stack, expect to see **βa**

- For items of the form

$[A \rightarrow \alpha \bullet, a]$

the lookahead *a* is used to reduce ***$A \rightarrow \alpha$*** only if the next input is ***a***

- For items of the form

$[A \rightarrow \alpha \bullet \beta, a]$

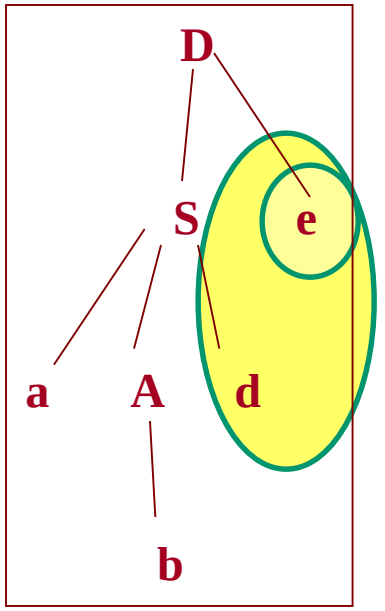
with **$\beta \neq \varepsilon$** the lookahead has no effect

The Closure Operation for LR(1) Items

1. Start with closure(I) = I

2. If **[A → α•Bβ, a]** ∈ **closure(I)** then for each production **B → γ** in the grammar and each terminal **b ∈ FIRST(βa)**, add the item **[B → •γ, b]** to I if it is not already in I

3. Repeat 2 until no new items can be added



LR(1) Closure Operation

- Initial state: start with (S' → . S , $), then apply closure operation
- Example: sum grammar

S' → S $
S → E + S | E
E → num

State: S_0

S' → . S , $

State: S_1

S' → . S , $
S → . E + S , $
S → . E , $
E → . num , +,$

The GOTO Operation for LR(1) Items

1. For each item **[A → α•Xβ, a] ∈ I**, add the set of items c**losure({[A → αX•β, a]})** to **goto(I,X)** if not already there

2. Repeat step 1 until no more items can be added to **goto(I,X)**

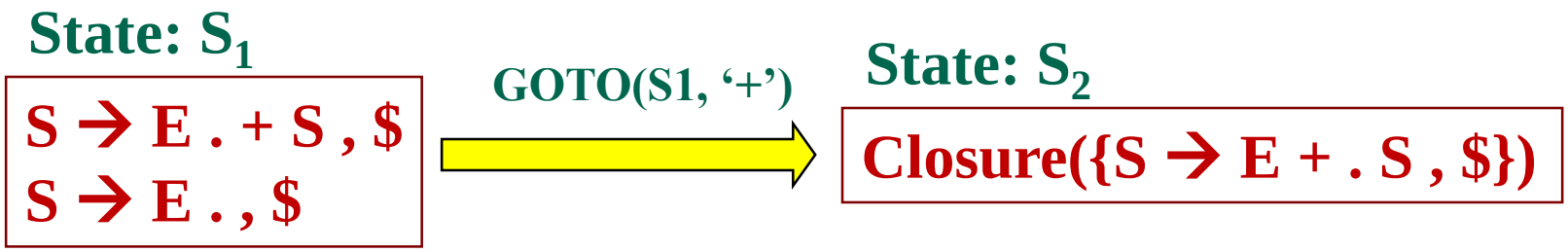
LR(1) GOTO Operation

Grammar:

S' → S$

S → E + S | E

E → num



Constructing the set of LR(1) Items of a Grammar

1. Augment the grammar with a new start symbol S′ and production **S′ → S**

2. Initially, set **C = closure({[S′ → •S, $]})** (this is the start state of the DFA)

3. For each set of items I ∈ C and each grammar symbol X ∈ (NT *U* T) such that **goto(I,X) ∉ C** and **goto(I,X) ≠ ϕ**, add the set of items goto(I,X) to C

4. Repeat 3 until no more sets can be added to C

Ex:
$S' \rightarrow S$
$S \rightarrow C\ C$
$C \rightarrow c\ C \mid d$

LR(1) Items…Example

I_0 : $S' \rightarrow .\ S,\ \$$ — $GOTO(I_0, S) = I_1$
$S \rightarrow .\ C\ C,\ \$$ — $GOTO(I_0, C) = I_2$
$C \rightarrow .\ c\ C,\ c/d$ — $GOTO(I_0, c) = I_3$
$C \rightarrow .\ d,\ c/d$ — $GOTO(I_0, d) = I_4$

I_1: $S' \rightarrow S\ .\ ,\ \$$

I_2: $S \rightarrow C\ .\ C,\ \$$ — $GOTO(I_2, C) = I_5$
$C \rightarrow .\ c\ C,\ \$$ — $GOTO(I_2, c) = I_6$
$C \rightarrow .\ d,\ \$$ — $GOTO(I_2, d) = I_7$

I_3: $C \rightarrow c\ .\ C,\ c/d$ — $GOTO(I_3, C) = I_8$
$C \rightarrow\ .\ c\ C,\ c/d$ — $GOTO(I_3, c) = I_3$
$C \rightarrow .d,\ c/d$ — $GOTO(I_3, d) = I_4$

I_4: $C \rightarrow d\ .\ ,\ c/d$

I_5: $S \rightarrow\ C\ C\ .\ ,\ \$$

I_6: $C \rightarrow c\ .\ C,\ \$$ — $GOTO(I_6, C) = I_9$
$C \rightarrow .\ c\ C,\ \$$ — $GOTO(I_6, c) = I_6$
$C \rightarrow .d,\ \$$ — $GOTO(I_6, d) = I_7$

I_7: $C \rightarrow d\ .\ ,\ \$$

I_8: $C \rightarrow c\ C\ .\ ,\ c/d$

I_9: $C \rightarrow c\ C\ .\ ,\ \$$

Canonical LR(1) Example 1

$S \rightarrow L = R \mid R$
$L \rightarrow * R \mid \text{id}$
$R \rightarrow L$

I_0: $[S' \rightarrow \bullet S,\ \$]$ goto$(I_0,S)=I_1$
$[S \rightarrow \bullet L=R,\ \$]$ goto$(I_0,L)=I_2$
$[S \rightarrow \bullet R,\ \$]$ goto$(I_0,R)=I_3$
$[L \rightarrow \bullet *R,\ =/\$]$ goto$(I_0,*)=I_4$
$[L \rightarrow \bullet \text{id},\ =/\$]$ goto$(I_0,\text{id})=I_5$
$[R \rightarrow \bullet L,\ \$]$ goto$(I_0,L)=I_2$

I_1: $[S' \rightarrow S\bullet,\ \$]$

I_2: $[S \rightarrow L\bullet=R,\ \$]$ goto$(I_0,=)=I_6$
$[R \rightarrow L\bullet,\ \$]$

I_3: $[S \rightarrow R\bullet,\ \$]$

I_4: $[L \rightarrow *\bullet R,\ =/\$]$ goto$(I_4,R)=I_7$
$[R \rightarrow \bullet L,\ =/\$]$ goto$(I_4,L)=I_8$
$[L \rightarrow \bullet *R,\ =/\$]$ goto$(I_4,*)=I_4$
$[L \rightarrow \bullet \text{id},\ =/\$]$ goto$(I_4,\text{id})=I_5$

I_5: $[L \rightarrow \text{id}\bullet,\ =/\$]$

I_6: $[S \rightarrow L=\bullet R,\ \$]$ goto$(I_6,R)=I_4$
$[R \rightarrow \bullet L,\ \$]$ goto$(I_6,L)=I_{10}$
$[L \rightarrow \bullet *R,\ \$]$ goto$(I_6,*)=I_{11}$
$[L \rightarrow \bullet \text{id},\ \$]$ goto$(I_6,\text{id})=I_{12}$

I_7: $[L \rightarrow *R\bullet,\ =/\$]$

I_8: $[R \rightarrow L\bullet,\ =/\$]$

I_9: $[S \rightarrow L=R\bullet,\ \$]$

I_{10}: $[R \rightarrow L\bullet,\ \$]$

I_{11}: $[L \rightarrow *\bullet R,\ \$]$ goto$(I_{11},R)=I_{13}$
$[R \rightarrow \bullet L,\ \$]$ goto$(I_{11},L)=I_{10}$
$[L \rightarrow \bullet *R,\ \$]$ goto$(I_{11},*)=I_{11}$
$[L \rightarrow \bullet \text{id},\ \$]$ goto$(I_{11},\text{id})=I_{12}$

I_{12}: $[L \rightarrow \text{id}\bullet,\ \$]$

I_{13}: $[L \rightarrow *R\bullet,\ \$]$

Constructing Canonical LR(1) Parsing Tables

1. Augment the grammar with $S' \rightarrow S$
2. Construct the set $C = \{I_0, I_1, \ldots, I_n\}$ of LR(1) items
3. Repeat through step 5 for each state i in the state set
4. Compute the Action entries
   - -- If $[A \rightarrow \alpha \bullet a\beta, b] \in I_i$ and $goto(I_i, a) = I_j$ then set **$action[i,a] = shift\ j$**
   - -- If $[A \rightarrow \alpha \bullet, a] \in I_i$ then set **$action[i,a] = reduce\ A \rightarrow \alpha$** (apply only if $A \neq S'$)
   - -- If $[S' \rightarrow S \bullet, \$]$ is in I_i then set **$action[i,\$] = accept$**
   - -- all undefined entries are error entries (i.e. blank)
5. Compute the Goto entries
   - -- If $goto(I_i, A) = I_j$ then set **$goto[i,A] = j$**
6. Repeat 3 & 5 until no more entries added
7. all undefined entries are error entries (i.e. blank)
8. the start state of the parser is state 0

ACTION Function

Compute the F(action) entries

1. If $[A \rightarrow \alpha \bullet a\beta, b] \in I_i$ and $goto(I_i, a) = I_j$ then set **$action[i,a] = shift\ j$**
2. If $[A \rightarrow \alpha \bullet, a] \in I_i$ then set **$action[i,a] = reduce\ A \rightarrow \alpha$** (apply only if $A \neq S'$)
3. If $[S' \rightarrow S \bullet, \$]$ is in I_i then set **$action[i,\$] = accept$**
4. all undefined entries are error entries (i.e. blank)

GOTO Function

If $goto(I_i, A) = I_j$ then set $goto[i, A] = j$

Ex:
S′ → S
S → C C
C → c C / d

CANONICAL Parsing Table

STATE	ACTION			GOTO	
	c	d	$	S	C
0	S3			1	2
1			ACCEPT		
2	S6	S7			5
3	S3	S4			8
4	R3	R3			
5			R1		
6	S6	S7			9
7			R3		
8	R2	R2			
9			R2		

LALR(1) Parsing

- LR(1) parsing tables have many states
- LALR(1) parsing (Look-Ahead LR) combines LR(1) states to reduce table size
- Space savings over LR(k)
- Less powerful than LR(1)
  - Will not introduce shift-reduce conflicts, because shifts do not use lookahead
  - May introduce reduce-reduce conflicts, but seldom do so for grammars of programming languages
- Combine 2 states if they have the same core items and compatible goto functions
- If cannot combine 2 states, then grammar is not LALR(1) but still LR(1).

LALR(1) Items…Example

Ex:
$S' \rightarrow S$
$S \rightarrow C\ C$
$C \rightarrow c\ C \mid d$

I_0 : $S' \rightarrow .\ S,\ \$$ — $GOTO(I_0, S) = I_1$
$S \rightarrow .\ C\ C,\ \$$ — $GOTO(I_0, C) = I_2$
$C \rightarrow .\ c\ C,\ c/d$ — $GOTO(I_0, c) = I_3$
$C \rightarrow .\ d,\ c/d$ — $GOTO(I_0, d) = I_4$

I_1: $S' \rightarrow S\ .\ ,\ \$$

I_2: $S \rightarrow C\ .\ C,\ \$$ — $GOTO(I_2, C) = I_5$
$C \rightarrow .\ c\ C,\ \$$ — $GOTO(I_2, c) = I_6$
$C \rightarrow .\ d,\ \$$ — $GOTO(I_2, d) = I_7$

I_3: $C \rightarrow c\ .\ C,\ c/d$ — $GOTO(I_3, C) = I_8$
$C \rightarrow\ .\ c\ C,\ c/d$ — $GOTO(I_3, c) = I_3$
$C \rightarrow .d,\ c/d$ — $GOTO(I_3, d) = I_4$

I_4: $C \rightarrow d\ .\ ,\ c/d$

I_5: $S \rightarrow\ C\ C\ .\ ,\ \$$

I_6: $C \rightarrow c\ .\ C,\ \$$ — $GOTO(I_6, C) = I_9$
$C \rightarrow .\ c\ C,\ \$$ — $GOTO(I_6, c) = I_6$
$C \rightarrow .d,\ \$$ — $GOTO(I_6, d) = I_7$

I_7: $C \rightarrow d\ .\ ,\ \$$

I_8: $C \rightarrow c\ C\ .\ ,\ c/d$

I_9: $C \rightarrow c\ C\ .\ ,\ \$$

LALR(1) Items…Example

Ex:
$S' \rightarrow S$
$S \rightarrow C\ C$
$C \rightarrow c\ C \mid d$

I_3: $C \rightarrow c\ .\ C,\ c/d$
$C \rightarrow\ .\ c\ C,\ c/d$
$C \rightarrow .d,\ c/d$

I_6: $C \rightarrow c\ .\ C,\ \$$
$C \rightarrow .\ c\ C,\ \$$
$C \rightarrow .d,\ \$$

I_{36}: $C \rightarrow c\ .\ C,\ c/d/\$$
$C \rightarrow .\ c\ C,\ c/d/\$$
$C \rightarrow .d,\ c/d/\$$

I_4: $C \rightarrow d\ .\ ,\ c/d$

I_7: $C \rightarrow d\ .\ ,\ \$$

I_{47}: $C \rightarrow d\ .\ ,\ c/d/\$$

I_8: $C \rightarrow c\ C\ .\ ,\ c/d$

I_9: $C \rightarrow c\ C\ .\ ,\ \$$

I_{89}: $C \rightarrow c\ C\ .\ ,\ c/d/\$$

LALR(1) Items…Example

Ex:
$S' \rightarrow S$
$S \rightarrow C\ C$
$C \rightarrow c\ C \mid d$

I_0 : $S' \rightarrow .\ S,\ \$$
$S \rightarrow .\ C\ C,\ \$$
$C \rightarrow .\ c\ C,\ c/d$
$C \rightarrow .\ d,\ c/d$

I_1: $S' \rightarrow S\ .\ ,\ \$$

I_2: $S \rightarrow C\ .\ C,\ \$$
$C \rightarrow .\ c\ C,\ \$$
$C \rightarrow .\ d,\ \$$

I_{36}: $C \rightarrow c\ .\ C,\ c/d/\$$
$C \rightarrow .\ c\ C,\ c/d/\$$
$C \rightarrow .d,\ c/d/\$$

$I_{5:}$ $S \rightarrow C\ C\ .\ ,\ \$$

$I_{47:}$ $C \rightarrow d\ .\ ,\ c/d/\$$

$I_{89:}$ $C \rightarrow c\ C\ .\ ,\ c/d/\$$

Ex:
S′ → S
S → C C
C → c C / d

LALR Parsing Table

STATE	ACTION			GOTO	
	c	d	$	S	C
0	S36			1	2
1			ACCEPT		
2	S36	S47			5
36	S36	S47			89
47	R3	R3			
5			R1		
89	R2	R2			

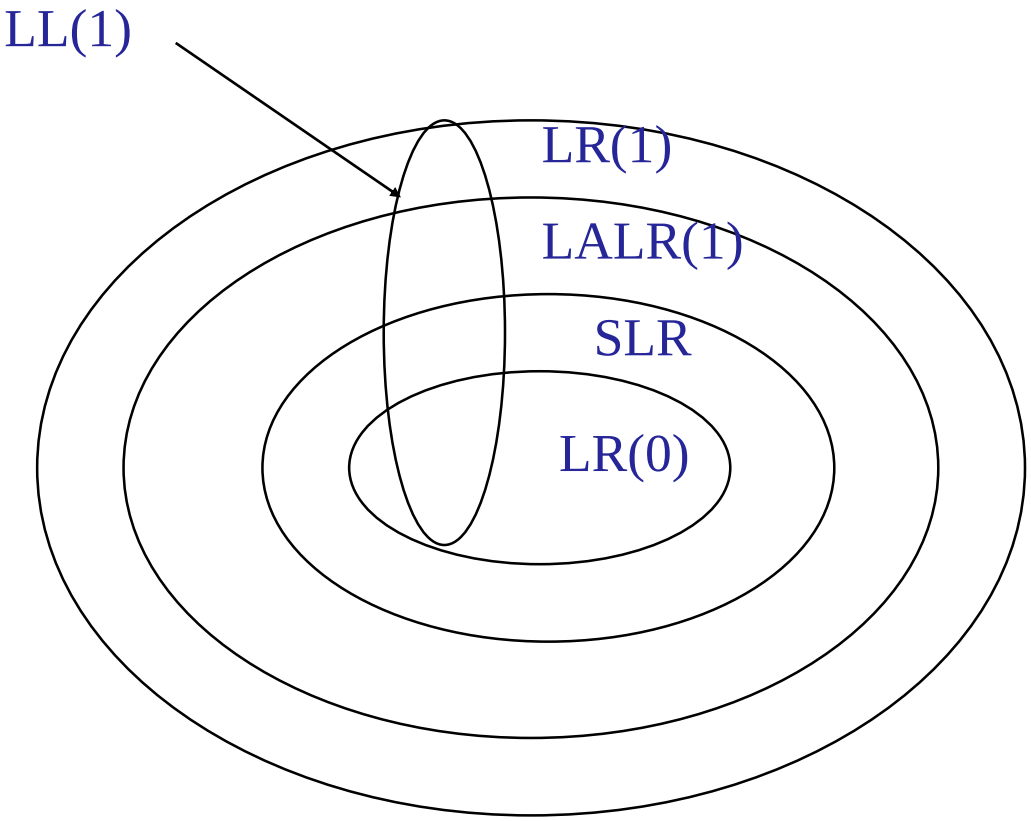
Error Detection in LR Parsing

- Canonical LR parser uses full LR(1) parse tables and will never make a single reduction before recognizing the error when a syntax error occurs on the input
- SLR and LALR may still reduce when a syntax error occurs on the input, but will never shift the erroneous input symbol

Error Recovery in LR Parsing

- **Panic mode**
  - Pop until state with a goto on a nonterminal A is found, (where A represents a major programming construct), push A
  - Discard input symbols until one is found in the FOLLOW set of A
- **Phrase-level recovery**
  - Implement error routines for every error entry in table
- **Error productions**
  - Pop until state has error production, then shift on stack
  - Discard input until symbol is encountered that allows parsing to continue

Classification of Grammars



LR(k) ⊆ LR(k+1)
LL(k) ⊆ LL(k+0)

LL(k) ⊆ LR(k)
LR(0) ⊆ SLR
LALR(1) ⊆ LR(1)

LR(k) versus LL(k)

Finding Reductions

LR(k) : Each reduction in the parse is detectable with

1. the complete left context,
2. the reducible phrase, itself, and
3. the k terminal symbols to its right

LL(k) : Parser must select the reduction based on

1. The complete left context
2. The next k terminals

Thus, LR(k) examines more context

	Advantages	*Disadvantages*
Top-down recursive descent	Fast Good locality Simplicity Good error detection	Hand-coded High maintenance Right associativity
LR(1)	Fast Deterministic langs. Automatable Left associativity	Large working sets Poor error messages Large table sizes

Using Ambiguous Grammar

Use Precedence and Associativity

$E \rightarrow E + E \mid E * E \mid (E) \mid id$

$E \rightarrow E + T \mid T$

$T \rightarrow T * F \mid F$

$F \rightarrow (E) \mid id$

Two extra reductions : $E \rightarrow T$ and $T \rightarrow F$

These two only for give higher precedence to * over +

Ex: E′ → E
E → E + E | E * E
E → (E) | id

Ambiguous Grammar

I_0: $E' \to \cdot E$
$E \to \cdot E + E$
$E \to \cdot E * E$
$E \to \cdot (E)$
$E \to \cdot \mathbf{id}$

I_1: $E' \to E \cdot$
$E \to E \cdot + E$
$E \to E \cdot * E$

I_2: $E \to (\cdot E)$
$E \to \cdot E + E$
$E \to \cdot E * E$
$E \to \cdot (E)$
$E \to \cdot \mathbf{id}$

I_3: $E \to \mathbf{id} \cdot$

I_4: $E \to E + \cdot E$
$E \to \cdot E + E$
$E \to \cdot E * E$
$E \to \cdot (E)$
$E \to \cdot \mathbf{id}$

I_5: $E \to E * \cdot E$
$E \to \cdot E + E$
$E \to \cdot E * E$
$F \to \cdot (E)$
$E \to \cdot \mathbf{id}$

I_6: $E \to (E \cdot)$
$E \to E \cdot + E$
$E \to E \cdot * E$

I_7: $E \to E + E \cdot$
$E \to E \cdot + E$
$E \to E \cdot * E$

I_8: $E \to E * E \cdot$
$E \to E \cdot + E$
$E \to E \cdot * E$

I_9: $E \to (E) \cdot$

Ex: $E' \rightarrow E$
$E \rightarrow E + E \mid E * E$
$E \rightarrow (E) \mid id$

Ambiguous Grammar

STATE	action						goto
	id	+	*	(	)	$	E
0	s3			s2			1
1		s4	s5			acc	
2	s3			s2			6
3		r4	r4		r4	r4	
4	s3			s2			8
5	s3			s2			8
6		s4	s5		s9		
7		r1	s5		r1	r1	
8		r2	r2		r2	r2	
9		r3	r3		r3	r3	

Ex: $S' \rightarrow S$
$S \rightarrow iSeS \mid Is \mid a$

Ambiguous Grammar

I_0: $S' \rightarrow \cdot S$
$S \rightarrow \cdot iSeS$
$S \rightarrow \cdot iS$
$S \rightarrow \cdot a$

I_1: $S' \rightarrow S \cdot$

I_2: $S \rightarrow i \cdot SeS$
$S \rightarrow i \cdot S$
$S \rightarrow \cdot iSeS$
$S \rightarrow \cdot iS$
$S \rightarrow \cdot a$

I_3: $S \rightarrow a \cdot$

I_4: $S \rightarrow iS \cdot eS$
$S \rightarrow iS \cdot$

I_5: $S \rightarrow iSe \cdot S$
$S \rightarrow \cdot iSeS$
$S \rightarrow \cdot iS$
$S \rightarrow \cdot a$

I_6: $S \rightarrow iSeS \cdot$

Ex: $S' \rightarrow S$
$S \rightarrow iSeS \mid Is \mid a$

Ambiguous Grammar

STATE	action				goto
	i	*e*	*a*	$	*S*
0	s2		s3		1
1				acc	
2	s2		s3		4
3		r3		r3	
4		s5		r2	
5	s2		s3		6
6		r1		r1	

THANKS