Chapter 4
Syntax Analysis

Outline

- Introduction to the parser
- Context-free grammars
- Writing a grammar
- Top-down parsing
- Bottom-up parsing
- Introduction to LR parsing: simple LR
- More powerful LR parsers
- Using ambiguous grammars
- Parser generator *Yacc*

Introduction to the Parser

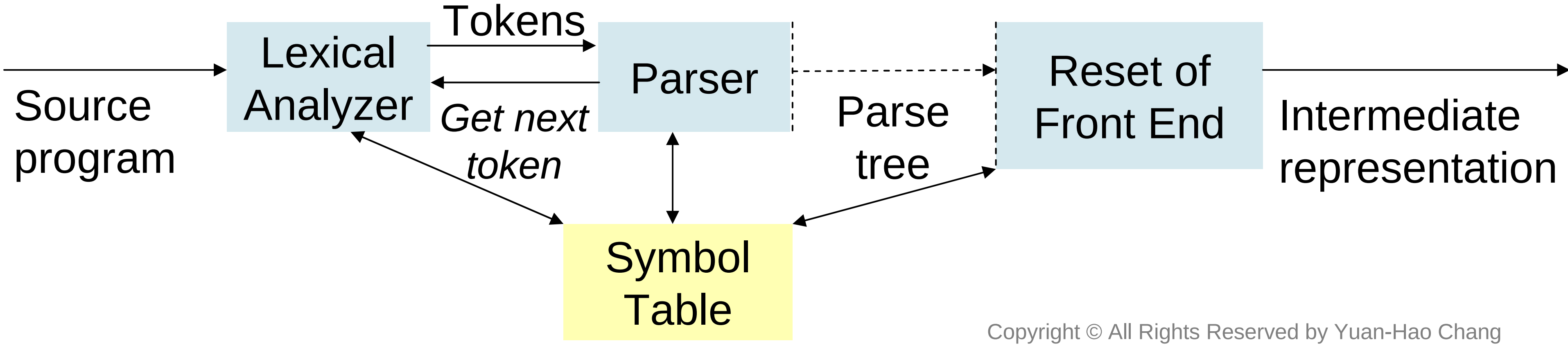
Benefits of Grammars for Programming Languages

- Give a precise syntactic specification of a programming language.
- Construct automatically a parser that determines the syntactic structure of a source program.
  - Parser-construction process could reveal syntactic ambiguities and trouble spots.
- Make the source program translation and error detection easier.
- Make the adding of new constructs for a language easier.

The Role of the Parser

- The parser obtains a string of tokens from the lexical analyzer, and verifies them with the grammar for the language.
  - Collect information about various tokens into the symbol table.
  - Perform type checking and other semantic analysis.
  - Generate intermediate code.
- In practice, parsers are expected to
  - Report syntax errors and
  - Recover from commonly occurring errors.

No strategy is proven universally acceptable, and the simplest approach for the parser is to quit with an error message when it detects the first error.



Types of Parsers

- There are three general types of parsers:
  - Universal parsing
    - These general methods are too inefficient to use in production compilers.
    - E.g., Cocke-Younger-Kasami algorithm and Earley’s algorithm
  - Top-down parsing
    - Build parse trees from the root to the leaves.
  - Bottom-up parsing
    - Build pares trees from the leaves to the root.

Representative Grammars

- Belong to LR grammars, suitable for bottom-up parsing
- Easy to add additional operators and precedence levels
- Not suitable for top-down parsing due to its left-recursion.

$$\begin{aligned} E &\rightarrow E + T \mid T \\ T &\rightarrow T * F \mid F \\ F &\rightarrow (E) \mid id \end{aligned} \qquad (4.1)$$

- Left-recursion elimination to be a non-left recursion.
- Suitable for top-down parsing

$$\begin{aligned} E &\rightarrow TE' \\ E' &\rightarrow + TE' \mid \varepsilon \\ T &\rightarrow FT' \\ T' &\rightarrow * FT' \mid \varepsilon \\ F &\rightarrow (E) \mid id \end{aligned} \qquad (4.2)$$

We will concentrate on expressions because of the associativity and precedence of operators.

To demonstrate the handling of ambiguities

$$E \rightarrow E + E \mid E * E \mid (E) \mid id \qquad (4.3)$$

E: expressions consisting of terms separated by + signs.
T: terms consisting of factors separated by * signs.
F: factors that can be either parenthesized expressions or identifiers.

Common Programming Errors

- Lexical errors
  - Misspellings of identifiers, keywords, or operators.
    - E.g., use *elipseSize* instead of *ellipseSize* (楕圓形).
  - Missing quotes around text intended as a string.
- Syntactic errors
  - Misplaced semicolons
  - Extra or missing braces “{“ and “}”
- Semantic errors
  - E.g., type mismatches
- Logical errors
  - Incorrect reasoning on the part of the programmer
    - E.g., in a C program, the comparison operator is == instead of the assignment operator =.

Error Handling in a Parser

- Parsers should have the *viable-prefix property*.
  - Viable-prefix property is to detect errors as soon as the stream of tokens from the lexical analyzer cannot be parsed further.
- Accurate detection of semantic and logical errors at compile time is a difficult but important task for parsers.
- The goal of the error handler in a parser:
  - Report the presence of errors clearly and accurately.
  - Recover from each error quickly enough to detect subsequent errors.
  - Add minimal overhead to the processing of correct programs.
- Most programming language specifications do not describe how a compiler should respond to errors.
  - A common place to report errors is where an error is detected in the source program.

Error Recovery Strategies

- Panic-mode recovery
  - On discovering an error, the parser discards input symbols until one of *synchronizing tokens* is found.
  - Synchronizing tokens are usually delimiters such as semicolon or }.
- Phrase-level recovery
  - On discovering an error, the parser replace a prefix of the remaining input by some string that allows the parser to continue.
    - E.g., replace a comma by a semicolon, delete an extraneous semicolon, or insert a missing semicolon.
  - Major drawback is the difficulty on coping with the actual error occurring before the point of detection.
  - Need careful replacement selection to prevent from infinite loops.
- Error productions
  - Augment the grammar with error productions to detect the anticipated errors when an error production is used.
- Global correction
  - Given an incorrect input *x*, find a parse tree for *y* such that the number of insertions, deletions, and changes of tokens needed to transform x to y is minimized.
  - This method is lack of efficiency, but used for finding optimal replacement strings for phrase-level recovery.

Context-Free Grammars

Context-Free Grammar

- Context-free grammar is also called *grammar* for short. It consists of
  - Terminals
    - The basic symbols from which strings are formed.
    - The token name is a synonym for “terminal”.
  - Nonterminals
    - Nonterminals are syntactic variables denote sets of strings that help define the language generated by the grammar.
    - Nonterminals impose a hierarchical structure that is key to syntax analysis and translation.
  - Start symbol
    - Start symbol is the first nonterminal of the grammar and can generate the language.
  - Productions
    - Productions of a grammar specify the manner in which the terminals and nonterminals are combined to form strings. Each production consists of:
      - A nonterminal called the *head* or *left side*.
      - The symbol →
      - A *body* or *right side* that consists of zero or more terminals and nonterminals.

A Grammar to Define Arithmetic Expressions

- A grammar to define arithmetic expressions
  - 7 terminals or terminal symbols: id + - * / ( )
  - 3 nonterminals: *expression*, *term*, *factor*

```
expression → expression + term
expression → expression – term
expression → term
term       → term * factor
term       → term / factor
term       → factor
factor     → ( expression )
factor     → id
```

Notational Conventions

- These symbols are terminals:
  - Lowercase letters early in the alphabet, e.g., a, b, c
  - Operator symbols, e.g., +, -, *, /, and so on
  - Punctuation symbols, e.g., parentheses, comma, and so on
  - The digits 0, 1, …, 9.
  - Boldface strings each of which represents a single termminal symbol, e.g., **id**.
- These symbols are nonterminals:
  - Uppercase letters early in the alphabet, e.g., A, B, C
  - The letter S represent the start symbol
  - Lowercase, *italic names*, e.g., *expr* or *stmt*.
  - When discussing programming constructs, uppercases letters may be used to represent nonterminals for the constructs, e.g., E, T, and F. (E: expression, T: term, F: factor)

Notational Conventions (Cont.)

- Grammar symbols (either nonterminals or terminals)
  - Uppercase letters late in the alphabet, e.g., X, Y, Z
- Strings of terminals
  - Lowercase letters late in the alphabet, e.g., u, v, …, z
- Strings of grammar symbols
  - Lowercase Greek letters, e.g., α, β, γ
  - E.g., $A \rightarrow \alpha$
- A-productions
  - A set of productions $A\rightarrow \alpha_1$, $A\rightarrow \alpha_2$, …, $A\rightarrow \alpha_K$ with a common head A.
  - It can be written $A\rightarrow \alpha_1 \mid \alpha_2 \mid \ldots \mid \alpha_K$,
  - α_1, α_2 , …, α_K are the alternatives for A.
- The head of the first production is the start symbol.

Grammar with Notational Convention

expression → *expression* + *term*

expression → *expression* – *term*

expression → *term*

term → *term* * *factor*

term → *term* / *factor*

term → *factor*

factor → (*expression*)

factor → **id**

E → E + T | E – T | T

T → T * F | T / F | F

F → (E) | **id**

Derivations

- The construction of a parse tree can be made precise by taking a derivational view.
  - Productions are treated as rewriting rules.
  - Beginning with the start symbol, each rewriting step replaces a nonterminal by the body of one of its productions.
    - The leftmost derivation corresponds to the top-down parsing.
    - The rightmost derivation corresponds to the bottom-up parsing.
  - E.g., $E \rightarrow E + E \mid E * E \mid -E \mid (E) \mid id$ (4.7)

read as “E derives –E”

The replacement of a single E by –E is described by: $E \Rightarrow -E$

E.g., $E \Rightarrow -E \Rightarrow -(E) \Rightarrow -(id)$

A derivation of –(id) from E

Derivations (Cont.)

- Consider a nonterminal A in the middle of a sequence of grammar symbol, as in $\alpha A\beta$.
  - α and β are arbitrary strings of grammar symbols (either nontermials or terminals).
  - If $A \rightarrow \gamma$, then $\alpha A\beta \Rightarrow \alpha\gamma\beta$ ($\alpha A\beta$ derives $\alpha\gamma\beta$ in one step)
- $\alpha_1 \Rightarrow \alpha_2 \Rightarrow \ldots \Rightarrow \alpha_n$ (α_1 derives α_n)

$\Rightarrow$ Derive in one step

$\overset{*}{\Rightarrow}$ Derive in zero or more steps

$\overset{+}{\Rightarrow}$ Derive in one or more steps

1. $\alpha \overset{*}{\Rightarrow} \alpha$, for any string

2. If $\alpha \overset{*}{\Rightarrow} \beta$ and $\beta \overset{*}{\Rightarrow} \gamma$, then $\alpha \overset{*}{\Rightarrow} \gamma$

Derivations (Cont..)

- A language that can be generated by a (context-free) grammar is a context-free language.
- If two grammars generate the same language, the grammars are equivalent.
- The language generated by a grammar is the set of sentences of the grammar.
  - A sentential form may contain both terminals and nonterminals, and may be empty.
  - A sentence of grammar G is a sentential form with no nonterminals.
    - A string of terminals *w* is in *L(G)* iff *w* is a sentence of G

$$\text{If } S \overset{*}{\Rightarrow} \alpha, \text{ where } S \text{ is the start symbol of grammar G}$$

α is a sentential form of G

Leftmost Derivation and Rightmost Derivation

- The string –(**id** + **id**) is a sentence of grammar (4.7)
- Leftmost derivation:

E → E + E | E * E | -E | (E) | id (4.7)

  - The leftmost nonterminal in each sentential is always chosen. We write $\alpha \underset{lm}{\Rightarrow} \beta$
  - E.g.,

$$E \underset{lm}{\Rightarrow} -E \underset{lm}{\Rightarrow} -(E) \underset{lm}{\Rightarrow} -(E + E) \underset{lm}{\Rightarrow} -(\mathbf{id} + E) \underset{lm}{\Rightarrow} -(\mathbf{id} + \mathbf{id}) \qquad (4.8)$$

- Rightmost (or canonical(標準的)) derivation:
  - The rightmost nonterminal in each sentential is always chosen. We write $\alpha \underset{rm}{\Rightarrow} \beta$
  - E.g.,

$$E \underset{rm}{\Rightarrow} -E \underset{rm}{\Rightarrow} -(E) \underset{rm}{\Rightarrow} -(E + E) \underset{rm}{\Rightarrow} -(E + \mathbf{id}) \underset{rm}{\Rightarrow} -(\mathbf{id} + \mathbf{id}) \qquad (4.9)$$

Left-Sentential and Right-Sentential Form

- Every leftmost step can be written as $wA\gamma \Rightarrow w\delta\gamma$, where
  - – w consists of terminals only.
  - – $A \rightarrow \delta$ is the production applied.
  - – γ is a string of grammar symbols.
- If $S \overset{*}{\underset{lm}{\Rightarrow}} \alpha$, then α is a left-sentential form of the grammar.
- If $S \overset{*}{\underset{rm}{\Rightarrow}} \alpha$, then α is a right-sentential form of the grammar.

Parse Trees and Derivations

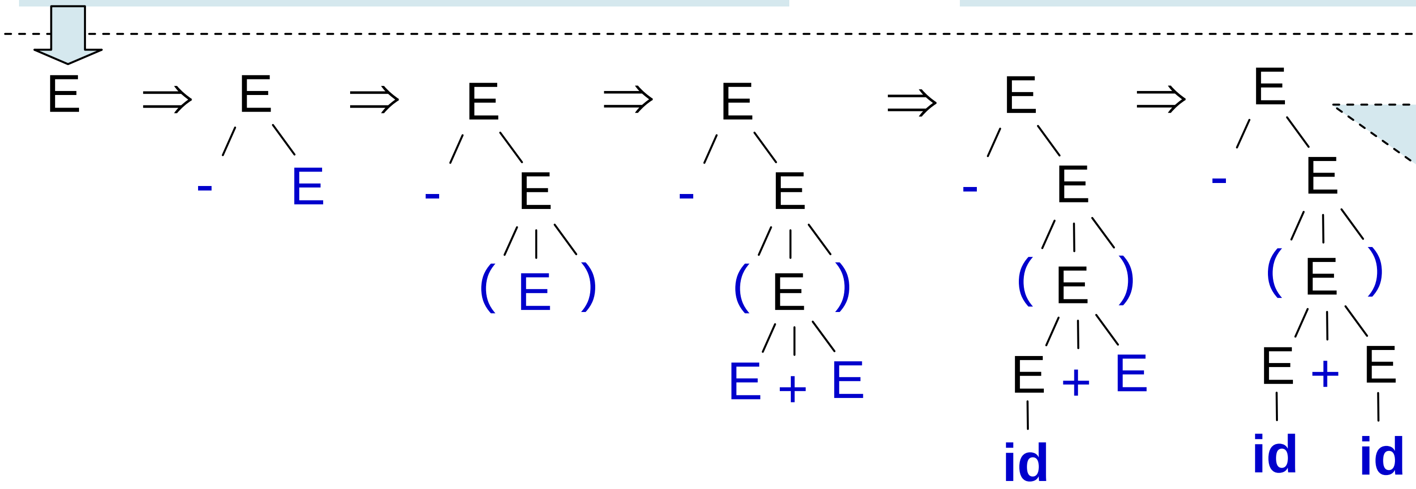
- A parse tree is a graphical representation of a derivation, and filters out the order in which productions are applied to replace nonterminals.
  - Each interior node labeled with the nonterminal in the head of the production to represent the application of the production.
  - The children of an interior node are labeled (from left to right) by the symbols in the body of the corresponding production.
  - During derivation, the head of the production is replaced by the body of the corresponding production.
- *Yield* or *frontier* of the tree:
  - Read the leaves of a parse tree from left to right to constitute a sentential form.

Parse Trees and Derivations (Cont.)

Parse string -(**id**+**id**) with the grammar: E → E + E | E * E | -E | (E) | id (4.7)

Derivation with leftmost derivation:
E ⇒ -E
⇒ -(E)
⇒ -(E + E) (4.8)
⇒ -(**id** + E)
⇒ -(**id** + **id**)

Derivation with rightmost derivation:
E ⇒ -E
⇒ -(E)
⇒ -(E + E) (4.9)
⇒ -(E + **id**)
⇒ -(**id** + **id**)



Yield: read leaves from left to right

Sequence of parse trees for leftmost derivation (4.8)

Relationship Induction between Derivations and Parse Trees

- Consider any derivation $\alpha_1 \Rightarrow \alpha_2 \Rightarrow \ldots \Rightarrow \alpha_n$, where α_1 is a single nonterminal A.
  - For each sentential form α_i, we can construct a parse tree whose yied is α_i.
- Induction process on *i*
  - **BASIS**: the tree for $\alpha_1 = A$ is a single node labeled A.
  - **INDUCTION**:
    - Suppose we have constructed a parse tree with yield $\alpha_{i-1} = X_1 X_2 \ldots X_k$ (where X_i is either a nonterminal or a terminal)
    - Suppose α_i is derived from α_{i-1} by replacing X_j with β where $X_j \rightarrow \beta$, and $\beta = Y_1 Y_2 \ldots Y_m$ → $\alpha_i = X_1 X_2 \ldots X_{j-1}\, \beta\, X_{j+1} \ldots X_k$
    - To model this step:
      - Find the j^{th} leaf from the left in the current parse tree.
      - Let this leaf X_j
      - Give this leaf m children labeled Y_1, Y_2, … Y_m

Ambiguity

- A grammar that produces more than one parse tree for some sentence is said to be *ambiguous*.
  - There should be a one-to-one relationship between parse trees and it rightmost (or rightmost) derivation.
  - In other words, every parse tree has associated with a unique leftmost and a unique rightmost derivation.
- E.g., Produce the sentence **id+id*id** with the grammar:

E → E + E | E * E | (E) | id (4.3)

Derivation with leftmost derivation:

E ⇒ E + E
⇒ **id** + E
⇒ **id** + E * E (4.8)
⇒ **id** + **id** * E
⇒ **id + id * id**

Produce the same sentence

Derivation with another leftmost derivation:

E ⇒ E * E
⇒ E + E * E
⇒ **id** + E * E (4.8)
⇒ **id** + E * **id**
⇒ **id + id * id**

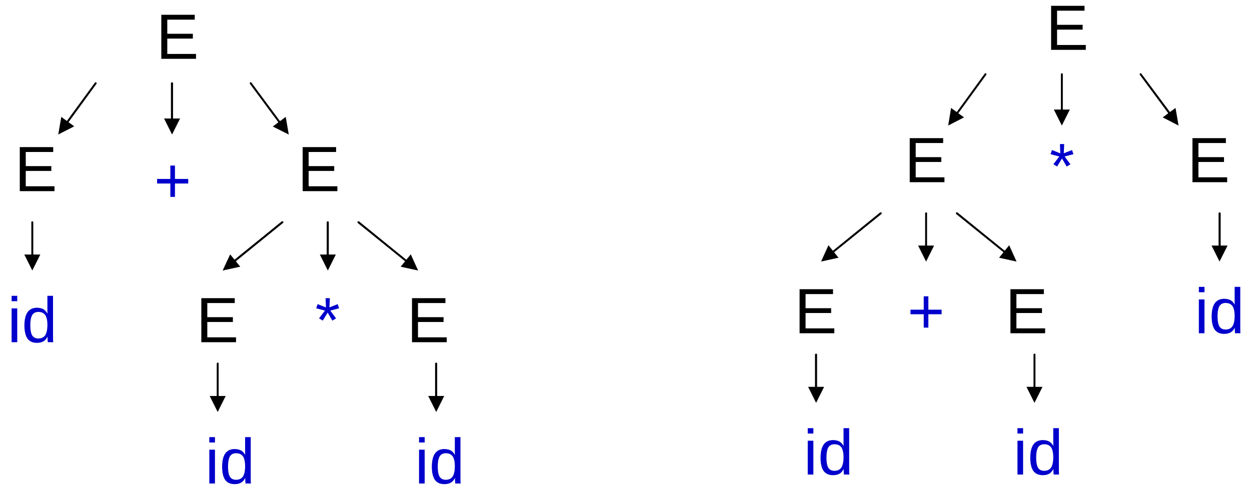
Ambiguity (Cont.)

Derivation with leftmost derivation:

E $\Rightarrow$ E + E
$\Rightarrow$ **id** + E
$\Rightarrow$ **id** + E * E (4.8)
$\Rightarrow$ **id** + **id** * E
$\Rightarrow$ **id + id * id**

Derivation with another leftmost derivation:

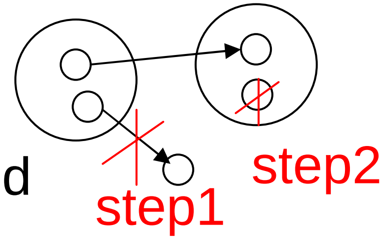
E $\Rightarrow$ E * E
$\Rightarrow$ E + E * E
$\Rightarrow$ **id** + E * E (4.8)
$\Rightarrow$ **id** + **id** * E
$\Rightarrow$ **id + id * id**



Two parse trees for **id+id*id**

Verifying the Language Generated by a Grammar

- A proof that a grammar *G* generates a Language *L* has two parts:
  - Step 1. Show that every string generated by *G* is in *L*.
  - Step 2. Show that every string in *L* can be generated by *G*.



- Consider S → (S) S | ε that generates all strings of balanced parentheses and only such strings.
  - **PROOF STEP 1**: Show every sentence derivable from *S* is balanced
  - **INDUCTIVE PROOF** (歸納法) on the number of steps *n*:
    - **BASIS**: When n = 1, the only string of terminals is the empty string.
    - **INDUCTION**:
      - Assume that all derivations of fewer than n steps produce balanced sentences.
      - Consider that a leftmost derivation of exactly *n* steps is of the form:

$$S \underset{lm}{\Rightarrow} (S)\,S \overset{*}{\underset{lm}{\Rightarrow}} (x)S \overset{*}{\underset{lm}{\Rightarrow}} (x)y$$

The derivation of *x* and *y* from S take fewer than *n* steps. By the inductive hypothesis, *x* and *y* are balanced, so the string *(x)y* must be balanced.

Verifying the Language Generated by a Grammar (Cont.)

- Consider S → (S) S | ε that generates all strings of balanced parentheses and only such strings.
  - PROOF STEP 2: Show every balanced string is derivable from *S*
  - **INDUCTIVE PROOF** (歸納法) on the length of a string
    - **BASIS**: If the string is of length 0, it must be ε, which is balanced.
    - **INDUCTION**:
      - Observation: every balanced string has even length.
      - Assume that every balanced string of length less than *2n* is derivable from S.
      - Consider a balanced string *w* of length *2n*, n ≥ 1.
      - Let *(x)* be the shortest nonempty prefix of *w* and have an equal number of left and right parentheses.
      - Then *w* can be written as *w* = *(x)y*, where x and y are balanced.
      - Thus we can find a derivation of the form:

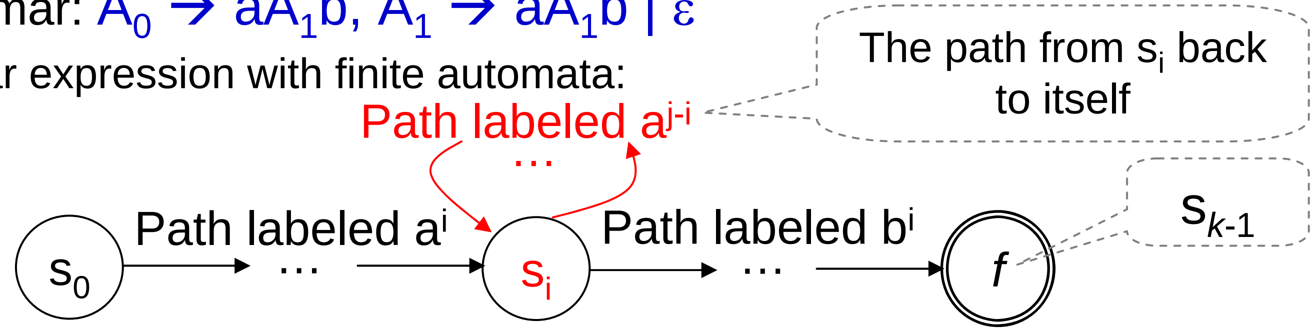
$$S \Rightarrow (S)\ S \overset{*}{\Rightarrow} (x)S \overset{*}{\Rightarrow} (x)y$$

so that w=(x)y is derivable from S.

Since *x* and *y* are of length less than *2n*, they are derivable from S by the inductive hypothesis.

Context-Free Grammars vs. Regular Expression

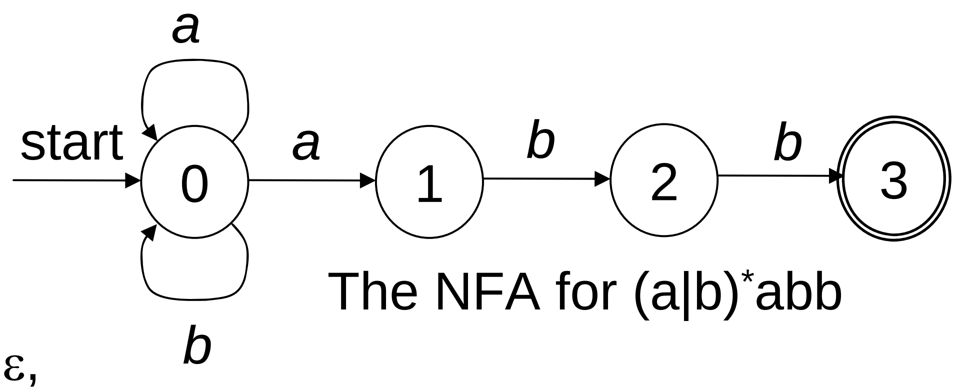
- Grammars are more powerful than regular expressions.
  - Every construct that can be described by a regular expression can be described a grammar, but not vice-versa.
  - Every regular language is a context-free language, but not vice-versa.
- E.g., the language $L = \{a^n b^n \mid n \geq 1\}$ with an equal number a's and b's.
  - Grammar: $A_0 \rightarrow aA_1b$, $A_1 \rightarrow aA_1b \mid \varepsilon$
  - Regular expression with finite automata:



- Construct a DFA D with a finite number of states *k* to accept the language *L*.
  - For an input beginning with more than *k* a's, *D* must enter some state twice (i.e., s_i)
  - $a^i b^i$ is in the language, but there is also a path labeled $a^j b^i$. Not in the language

Construct a Grammar from NFA

- Construct a grammar to recognize the same language as an NFA as follows:
  - 1. For each state *i*, create a nonterminal A_i.
  - 2. If state *i* has a transition to state *j* on input *a*, add the production $A_i \rightarrow aA_j$.
    If state i goes to state j on input ε, add the production $A_i \rightarrow A_j$
  - 3. If *i* is an accepting state, add $A_i \rightarrow \varepsilon$
  - 4. If *i* is the start state, make A_i be the start symbol.



The NFA for $(a|b)^*abb$

$A_0 \rightarrow aA_0 \mid bA_0 \mid aA_1$
$A_1 \rightarrow bA_2$
$A_2 \rightarrow bA_3$
$A_3 \rightarrow \varepsilon$

The grammar for $(a|b)^*abb$

Writing a Grammar

Lexical vs. Syntactic Analysis

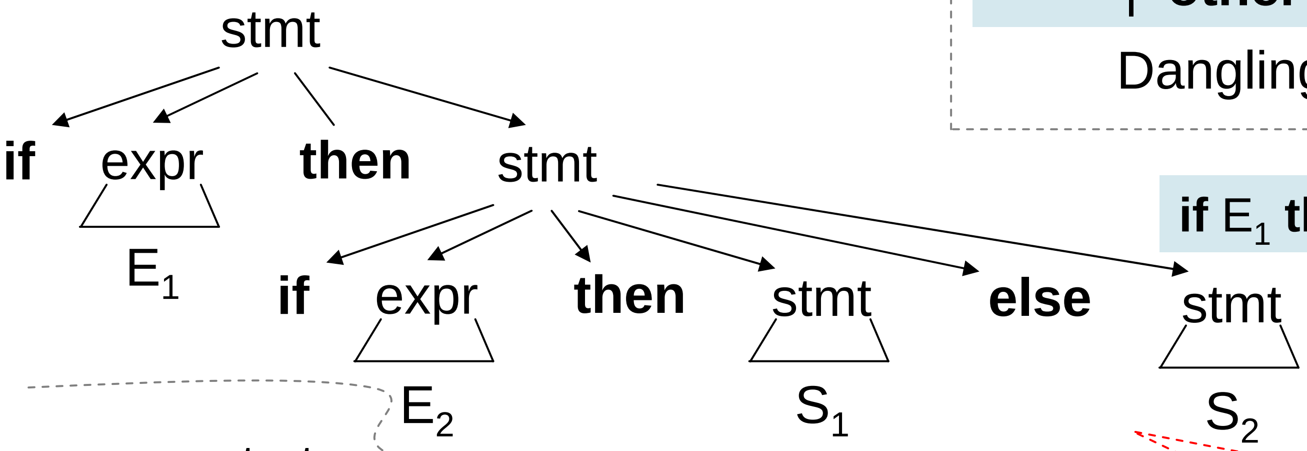
- Everything described by a regular expression can be described by a grammar. Why use regular expressions in the lexical analysis?
  - Separate the syntactic structure of a language into lexical and non-lexical parts for modularization.
  - Lexical rules of a language are frequently quite simple.
  - Regular expressions generally provide a more concise and easier-to-understand notation for tokens.
  - More efficient lexical analyzers can be constructed automatically from regular expressions.

Eliminating Ambiguity

E.g., **if** E_1 **then if** E_2 **then** S_1 **else** S_2

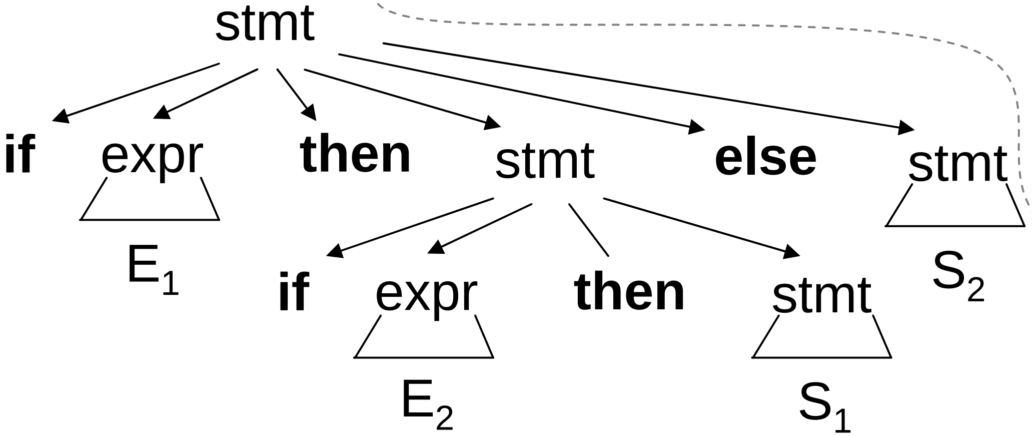
stmt → **if** expr **then** stmt
    | **if** expr **then** stmt **else** stmt
    | **other**

Dangling-else grammar



if E_1 **then** (**if** E_2 **then** S_1 **else** S_2)

$E_1(/S_1)$ and $E_2\,(/S_2)$ are different occurrences of the same nonterminal.



Preferred: match each **else** with the closest **unmatched then**.

if E_1 **then** (**if** E_2 **then** S_1) **else** S_2

Unambiguous Grammar for if-then-else Statements

Unambiguous if-then-else grammar

(Associate each **else** with the closest previous **unmatched then**)

stmt → matched_stmt

| open_stmt

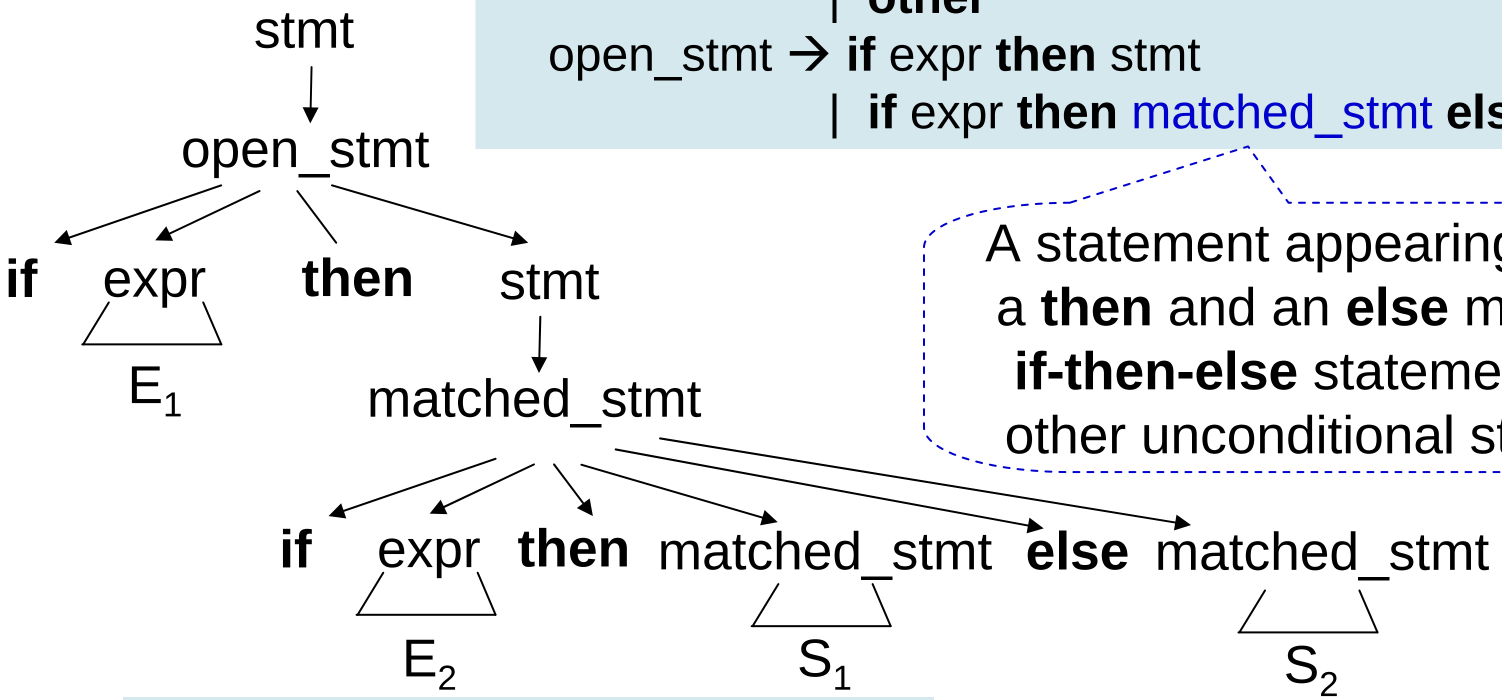
matched_stmt → **if** expr **then** matched_stmt **else** matched_stmt

| **other**

open_stmt → **if** expr **then** stmt

| **if** expr **then** matched_stmt **else** open_stmt

A statement appearing between a **then** and an **else** must be an **if-then-else** statement or any other unconditional statement.



E.g., **if** E_1 **then if** E_2 **then** S_1 **else** S_2

Elimination of Left Recursion

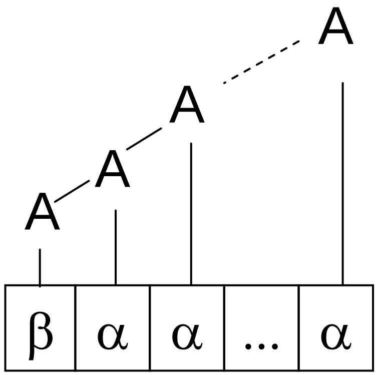
- Top-down parsing methods can not handle left-recursive grammars. E.g., $A \overset{+}{\Rightarrow} A\alpha$
- Left-recursion elimination:

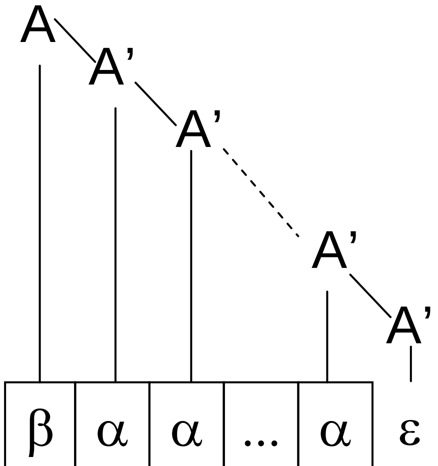
$A \rightarrow A\alpha \mid \beta$ ⇨ $A \rightarrow \beta A'$
$A' \rightarrow \alpha A' \mid \varepsilon$

$E \rightarrow E + T \mid T$
$T \rightarrow T * F \mid F$ (4.1)
$F \rightarrow (E) \mid id$

⇩

$E \rightarrow TE'$
$E' \rightarrow + TE' \mid \varepsilon$
$T \rightarrow FT'$ (4.2)
$T' \rightarrow * FT' \mid \varepsilon$
$F \rightarrow (E) \mid id$





Immediate Left Recursion Elimination

Begin with A

Not begin with A

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

⇓ Left recursion elimination

$$A \rightarrow \beta_1 A' \mid \beta_2 A' \mid \ldots \mid \beta_n A'$$

$$A' \rightarrow \alpha_1 A' \mid \alpha_2 A' \mid \ldots \mid \alpha_m A' \mid \varepsilon$$

This can not eliminate left recursion involving derivations of two or more steps.

Left Recursion Elimination

- **Algorithm**: Eliminating left recursion
- **INPUT**: Grammar G with no cycles or ε-productions.
- **OUTPUT**: An equivalent grammar with no left recursion.
- **METHOD**:

```
arrange the nonterminals in some order $A_1, A_2, \ldots, A_n$.
for ( each i from 1 to n) {
    for ( each j from 1 to i-1) {
        replace each production of the form $A_i \rightarrow A_j\gamma$ by
        the productions $A_i \rightarrow \delta_1\gamma \mid \delta_2\gamma \mid \ldots \mid \delta_k\gamma$,
        where $A_j \rightarrow \delta_1 \mid \delta_2 \mid \ldots \mid \delta_k$ are all current $A_j$-productions
    }
    Eliminate the immediate left recursion among the $A_i$-productions
}
```

Left Recursion Elimination (Cont.)

- E.g.,

$S \rightarrow Aa \mid b$
$A \rightarrow Ac \mid Sd \mid \varepsilon$

S is left recursive because
$S \Rightarrow Aa \Rightarrow Sda$.
Therefore, $A \Rightarrow Sd \Rightarrow Aad$
(left recursive)

⇩

1. Sort nonterminals S, A
2. Use S-productions to replace S in A-productions
   $A \rightarrow Ac \mid Aad \mid bd \mid \varepsilon$
3. Eliminate the immediate left recursion among A-productions:

⇩

$S \rightarrow Aa \mid b$
$A \rightarrow bdA' \mid A'$
$A' \rightarrow cA' \mid adA' \mid \varepsilon$

Left Factoring

- Left factoring is a grammar transformation for producing a grammar suitable for predictive (top-down) parsing.
  - When the choice between two alternative A-productions is not clear, the production is rewritten to defer the decision until enough of the input has been seen.
- Left factoring: find the longest prefix α common to two or more of its alternatives.

$$A \rightarrow \alpha\beta_1 \mid \alpha\beta_2 \quad \Longrightarrow \quad \begin{aligned} A &\rightarrow \alpha A' \\ A' &\rightarrow \beta_1 \mid \beta_2 \end{aligned}$$

$$A \rightarrow \alpha\beta_1 \mid \alpha\beta_2 \mid \dots \mid \alpha\beta_n \mid \gamma \quad \Longrightarrow \quad \begin{aligned} A &\rightarrow \alpha A' \mid \gamma \\ A' &\rightarrow \beta_1 \mid \beta_2 \mid \dots \mid \beta_n \end{aligned}$$

Left Factoring (Cont.)

- E.g., on seeing the input **if**, we cannot tell which production to choose to expand *stmt.*

stmt → **if** expr **then** stmt
     | **if** expr **then** stmt **else** stmt
     | **other**

- Abstracted dangling-else:

$$S \rightarrow i\ E\ t\ S\underline{\ } \mid i\ E\ t\ S\ \underline{e\ S} \mid a$$
$$E \rightarrow b$$

⇓

$$S \rightarrow i\ E\ t\ S\ S' \mid a$$
$$S' \rightarrow eS \mid \varepsilon$$
$$E \rightarrow b$$

i: **if**
t: **then**
e: **else**
E: conditional expression
S: statement

Non-Context-Free Language Constructs

- Non-context-free language constructs are syntactic constructs that cannot be specified by using context-free grammars alone.
  - Note: context-free grammars are context independent and only nonterminals (excluding terminal/context) appearing at the head of productions.
- E.g., In C and Java, identifiers need to be declared before they are used in a program. They are presented in the form *wcw*:
  - The first *w* represents the declaration of an identifier *w*.
  - The *c* represents an intervenigng program fragment.
  - The second *w* represents the use of the identifier *w*.
  - E.g., $L_1 = \{\, wcw \mid w \text{ is in } (a|b)^* \,\}$

    Abstract language

    - L_1 consists of all words composed of a repeated of *a's* and *b's* separated by *c* such as aabcaab.

    - L_1 cannot be represented by context-free grammar, so that the correctness needs to be checked in the semantic-analysis phase.

Non-Context-Free Language Constructs (Cont.)

- E.g., Checking that the number of formal parameters in the declaration of a function agrees with the number of actual parameters in a use of function.
  - E.g., strings of the form $a^nb^mc^nd^m$
    - a^n and b^m represent the formal-parameter lists of two functions declared to have n and m arguments, respectively.
    - c^n and d^m represent the actual-parameter lists of two functions declared to have n and m arguments, respectively.
  - E.g., the abstract language $L_2 = \{ a^nb^mc^nd^m \mid n \geq 1 \text{ and } m \geq 1 \}$
    - L_2 consists of strings in the language generated by the regular expression a*b*c*d* such that the numbers of a's and c's are equal and the numbers of b's and d's are equal, so that L_2 is not context-free.
  - A function call in C:

stmt → **id** (*expr_list*)
expr_list → *expr_list*, *expr* | *expr*

Checking whether the number of parameters in a call is correct is usually done during the semantic-analysis phase.

Non-Context-Free Language Constructs (Cont.)

- E.g., non-context-free language $L_3 = \{ a^n b^n c^n \mid n \geq 1 \}$

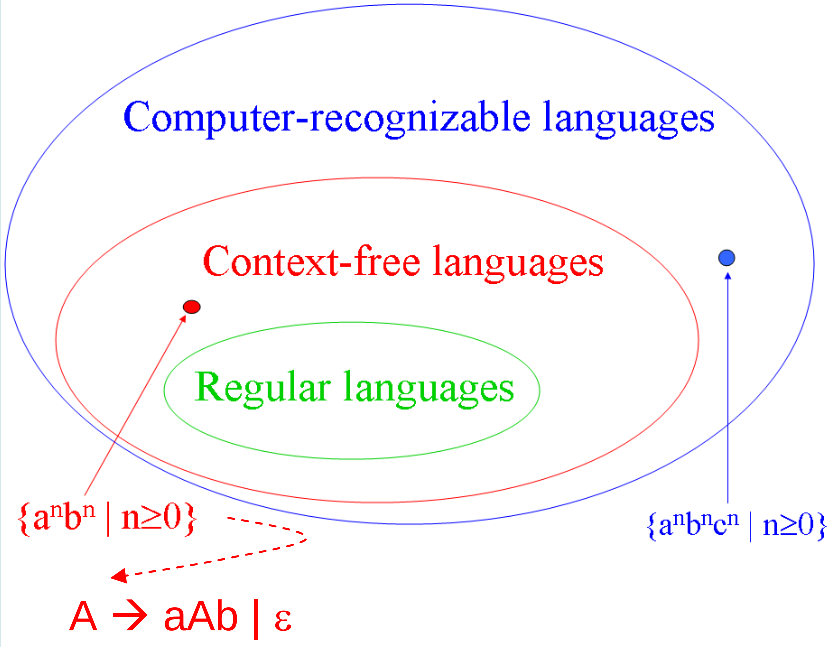
S → aSBC
S → aBC
CB → HB
HB → HC
HC → BC
aB → ab
bB → bb
bC → bc
cC → cc

Grammar

Non-context free

S ⇒ aSBC ⇒ aaSBCBC
⇒ aaaBCBCBC
⇒ aaaBHBCBC
⇒ aaaBHCCBC
⇒ aaaBBCCBC
⇒ aaaBBCHBC
⇒ aaaBBCHCC
⇒ aaaBBCBCC
⇒ aaaBBHBCC
⇒ aaaBBHCCC
⇒ aaaBBBCCC
⇒ aaabBBCCC
⇒ aaabbBCCC
⇒ aaabbbCCC
⇒ aaabbbcCC
⇒ aaabbbccC
⇒ aaabbbccc

To generate aaabbbccc


Computer-recognizable languages
Context-free languages
Regular languages
$\{a^n b^n \mid n \geq 0\}$
$\{a^n b^n c^n \mid n \geq 0\}$
A → aAb | ε

Top-Down Parsing

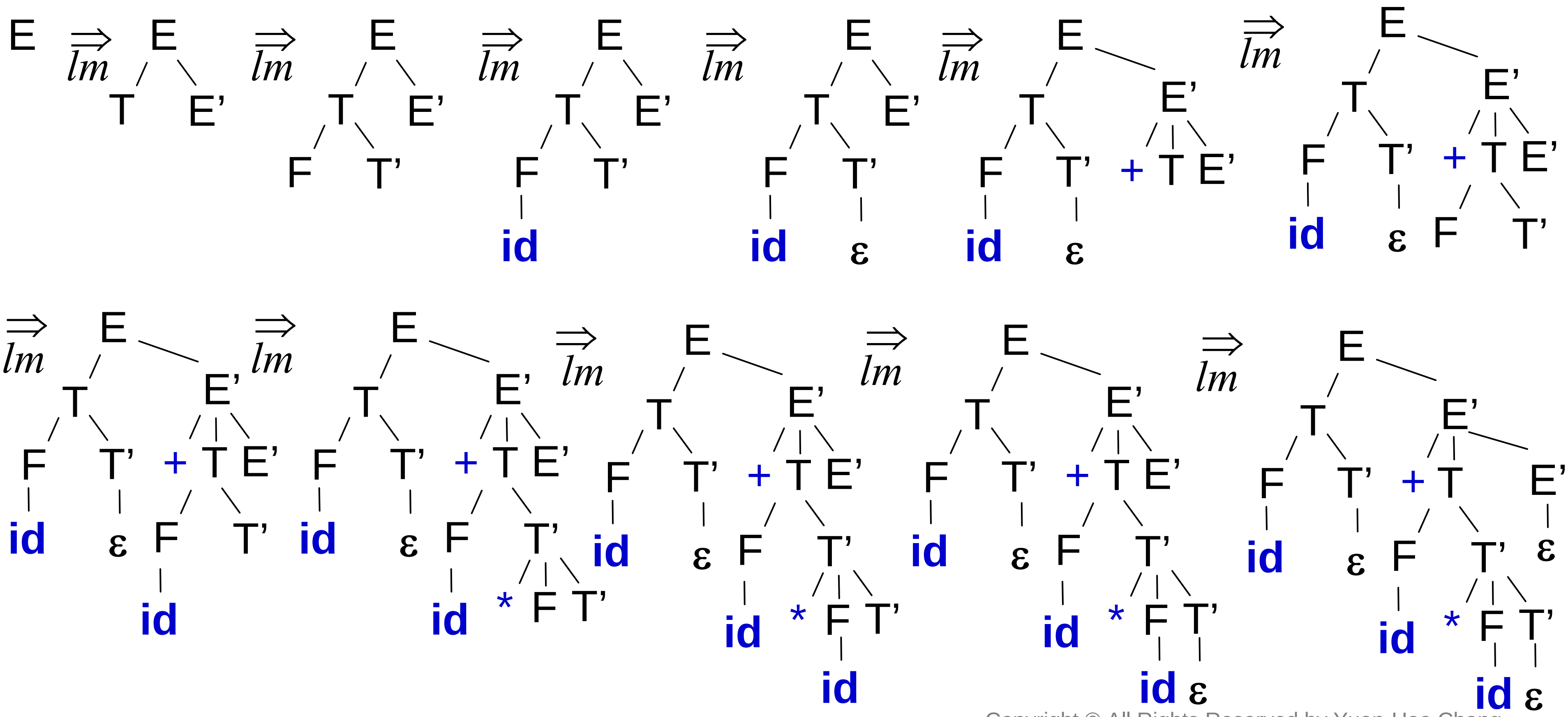
Top-Down Parsing

- Top-down parsing can be viewed as
  - Constructing a parse tree for the input string from the root
  - Creating the nodes of the parse tree in preorder
  - Finding a leftmost derivation for an input string.
- The key problem is to determine the production to be applied for a nonterminal.
  - Once a production is chosen, the rest of the parsing process is to match the terminal symbols in the production body with the input string.

Top-Down Parsing (Cont.)

- Top-down parse for **id+id*id**

E → TE'
E' → + TE' | ε
T → FT' (4.2)
T' → * FT' | ε
F → (E) | id



E
$\underset{lm}{\Rightarrow}$
E T E'
$\underset{lm}{\Rightarrow}$
E T E' F T'
$\underset{lm}{\Rightarrow}$
E T E' F T' id
$\underset{lm}{\Rightarrow}$
E T E' F T' id ε
$\underset{lm}{\Rightarrow}$
E T E' F T' + T E' id ε
$\underset{lm}{\Rightarrow}$
E T E' F T' + T E' id ε F T'
$\underset{lm}{\Rightarrow}$
E T E' F T' + T E' id ε F T' id
$\underset{lm}{\Rightarrow}$
E T E' F T' + T E' id ε F T' id * F T'
$\underset{lm}{\Rightarrow}$
E T E' F T' + T E' id ε F T' id * F T' id
$\underset{lm}{\Rightarrow}$
E T E' F T' + T E' id ε F T' id * F T' id ε
$\underset{lm}{\Rightarrow}$
E T E' F T' + T E' id ε F T' ε id * F T' id ε

Recursive-Descent Parsing

- A recursive-descent parsing consists of a set of procedures, each of which is for one nonterminal.
- Backtracking might be needed to repeat scans over the input.
  - NOTE: backtracking is not very efficient, and tabular methods such as the dynamic programming algorithm is preferred.
- Left-recursive grammar can cause a recursive-decent parser to go into an infinite loop. (i.e., A production might be expanded repeatedly without consuming any input.)

```
    void A() {
1)     Choose an A-production, A→$X_1X_2$ … $X_k$
2)      for ( i = 1 to k) {
3)          if ( $X_i$ is a nonterminal )
4)              call procedure $X_i$();
5)          else if ( $X_i$ equals the current input symbol $a$)
6)              advance the input to the next symbol;
7)          else /* an error has occurred */
        }
   }
```

To allow backtracking, this should try each production in some order

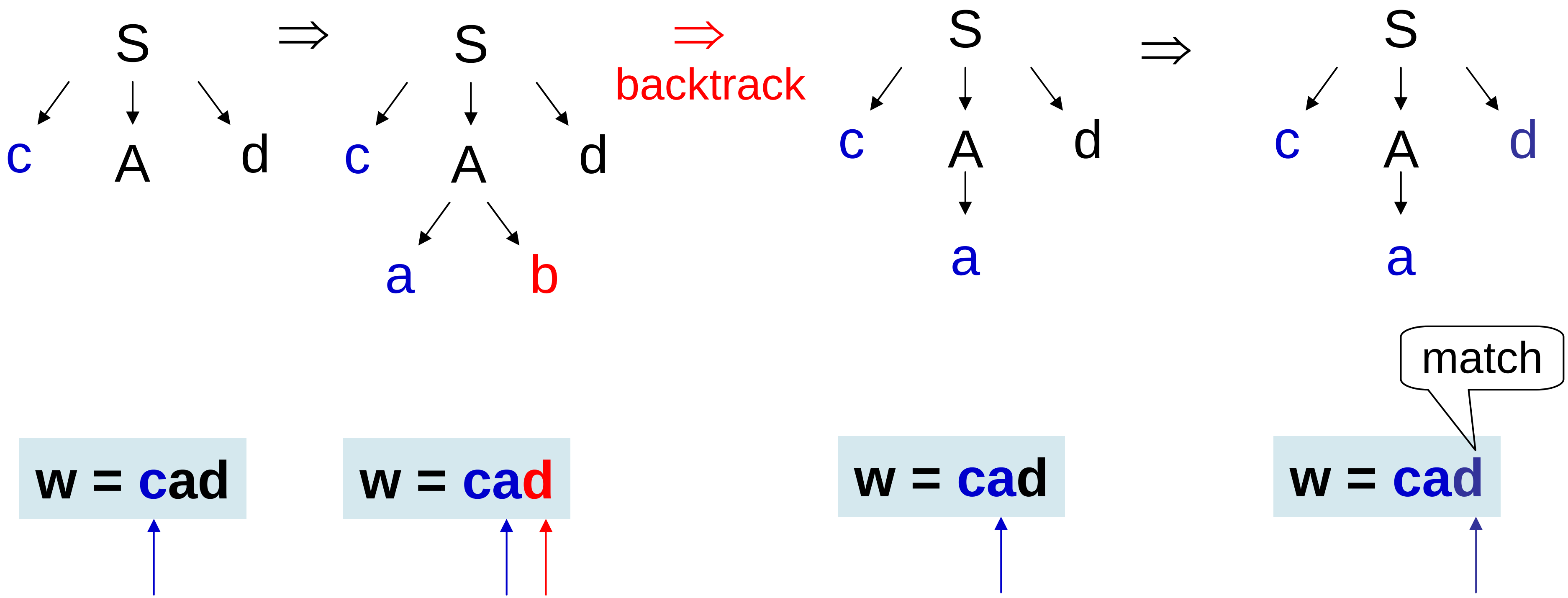
To allow backtracking, this should return to line (1) and try another A-production until no more A-productions to try.

A typical procedure for a nonterminal in a top-down parser

Recursive-Descent Parsing (Cont.)

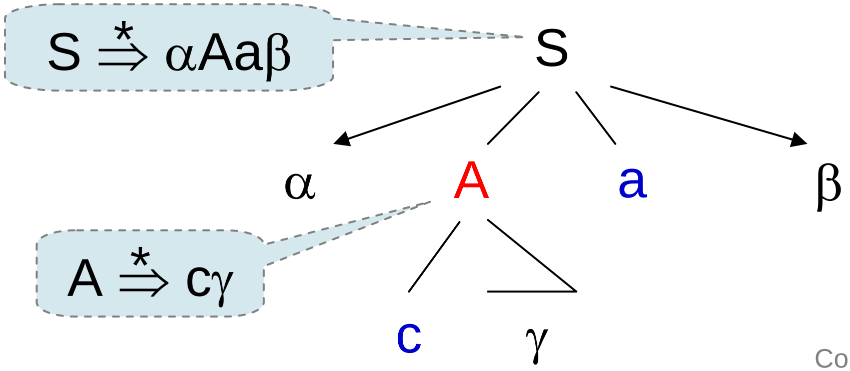
- Input string **w = cad**.

S → cAd
A → ab | a
Grammar



FIRST and FOLLOW

- FIRST and FOLLOW allow us to choose which production to apply, based on the next input symbol.
  - FIRST(α) is the set of terminals that begin strings derived from α, where α is any string of grammar symbols. If $\alpha \overset{*}{\Rightarrow} \varepsilon$, then ε is also in FIRST(α).
  - FOLLOW(α) is (for nonterminal A) the set of terminals *a* that can appear immediately to the right of A in some sentential form.
    - If A can be the rightmost symbol in some sentential form, then $ is in FOLLOW(A), where $ is a special “endmarker” symbol.



$S \overset{*}{\Rightarrow} \alpha A a \beta$

$A \overset{*}{\Rightarrow} c\gamma$

c is in FIRST(A)
a is in FOLLOW(A)

FIRST

- Compute FIRST(X) for all grammar symbols X:
  - If X is a terminal, then FIRST(X) = {X}.
  - If X is a nonterminal and $X \rightarrow Y_1 Y_2 \ldots Y_k$ is a production for some $k \geq 1$,
    - Everything in FIRST(Y_1) is surely in FIRST(X).
    - If Y_1 does not derive ε, then nothing more is added to FIRST(X).
    - If $Y_1 \overset{*}{\Rightarrow} \varepsilon$, then FIRST($Y_2$) is added to FIRST(X), and so on.
  - If $X \rightarrow \varepsilon$ is a production, then add ε to FIRST(X).

E → TE'
E' → + TE' | ε
T → FT' (4.2)
T' → * FT' | ε
F → (E) | **id**

- FIRST(F) = { (, id }
- FIRST(T') = {*, ε}
  - The two productions for T' begins with * and ε.
- FIRST(T) = FIRST(F) = { (, id }
  - T has one production beginning with F.
- FIRST(E') = {+, ε}
  - The two productions for E' begins with + and ε.
- FIRST(E) = FIRST(T) = { (, id }
  - E has one production beginning with T.

FOLLOW

- Compute FOLLOW(A) for all nonterminals A
  - Place $ in FOLLOW(S), where S is the start symbol and $ is the input right endmarker.
  - If there is a production $A \rightarrow \alpha B \beta$, then everything in FIRST(β) except ε is in FOLLOW(B).
  - If there is a production $A \rightarrow \alpha B$ (or $A \rightarrow \alpha B \beta$ with FIRST(β) contains ε), then everything in FOLLOW(A) is in FOLLOW(B).

- FIRST(E) = { (, id }
- FIRST(E’) = { +, ε }
- FIRST(T) = { (, id }
- FIRST(T’) = { *, ε }
- FIRST(F) = { (, id }

$$
\begin{array}{lll}
E & \rightarrow & TE' \\
E' & \rightarrow & +\,TE' \mid \varepsilon \\
T & \rightarrow & FT' \qquad (4.2) \\
T' & \rightarrow & *\,FT' \mid \varepsilon \\
F & \rightarrow & (E) \mid \mathbf{id}
\end{array}
$$

- FOLLOW(E) = {), $ }
  - E is the start symbol with the production body (E)
- FOLLOW(E’) = FOLLOW(E) = {), $ }
  - E’ appears at the ends of the body of E-productions.
- FOLLOW(T) = { +,), $ }
  - T only appears in the body followed by E’. Everything in FIRST(E’) except ε is in FOLLOW(T). → +
  - In E→TE’, $E' \overset{*}{\Rightarrow} \varepsilon$ so that everything in FOLLOW(E) is in FOLLOW(T).
- FOLLOW(T’) = FOLLOW(T) = { +,), $ }
  - In T→FT’, everything in FOLLOW(T) is in FOLLOW(T’).
- FOLLOW(F) = { +, *,), $ }
  - In T→FT’, everything in FIRST(T’) except ε is in FOLLOW(F)
  - In T→FT’, $T' \overset{*}{\Rightarrow} \varepsilon$ so that everthing in FOLLOW(T) is in FOLLOW(F) → +,), $

LL(1) Grammars

- LL(1) grammar:
  - First L: scan the input from left to right.
  - Second L: produce a leftmost derivation.
  - The “1”: use one input symbol of lookahead at each step to make parsing action decisions.
- No left-recursive or ambiguous grammar can be LL(1).
- A grammar is LL(1) iff whenever A→ α | β are two distinct productions of G, the following conditions should hold to prevent multiply defined entries in the parsing table:
  - 1. For no terminal *a* do both α and β derive strings beginning with *a*.
  - 2. At most one of α and β can derive the empty string.
  - 3. If $\beta \overset{*}{\Rightarrow} \varepsilon$, then α does not derive any string beginning with a terminal in FOLLOW(A), and likewise if ε is in FIRST(α).

FIRST(α) and FIRST(β) are disjoint.

If ε is in FIRST(β), then FIRST(α) and FOLLOW(A) are disjoint.

Predictive Parsers for LL(1) Grammars

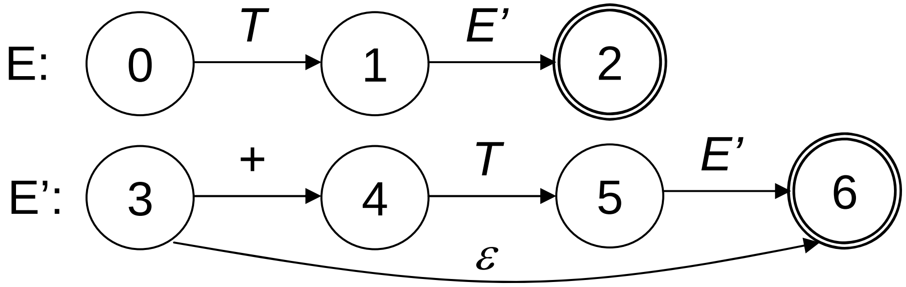
- Predictive parsers
  - Are recursive-descent parsers that need no backtracking.
  - Look only at the current input symbol on applying the proper production for a nonterminal.
  - Can be constructed for a class of grammars called LL(1).
- E.g., we have the following productions:

stmt → **if** (*expr*) *stmt* **else** *stmt*
   | **while** (*expr*) *stmt*
   | { *stmt_list* }

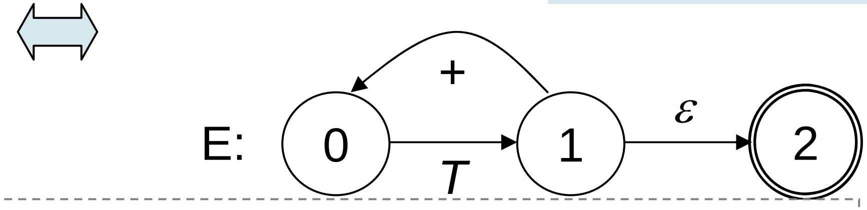
The keywords **if**, **while** and the symbol **{** tell us which alternative is the only one that could possibly succeed if we are to find a statement.

Transition Diagrams for Predictive Parsers

- To construct the transition diagram from a grammar:
  - First eliminate left recursion, and left factor the grammar.
  - Then, for each nonterminal A,
    - 1. Create an initial and final (return) state.
    - 2. For each production $A \rightarrow X_1X_2 \ldots X_k$, create a path from the initial to the final state, with edges labeled $X_1, X_2, \ldots, X_k$.
- Parsers have one diagram for each nonterminal.
  - The labels of edges can be tokens (terminals) or nonterminals.
    - A transition on a token means that the token is the next input symbol.
    - A transition on a nonterminal *A* is a call of the procedure for *A*.



E → TE'
E' → + TE' | ε



Use the diagram E' to substitute E' in the diagram E with tail-recursion removal.

Predictive Parsing Table

- A predictive parsing table *M[A, a]* is a two-dimensional array, where *A* is a nonterminal, and *a* is a terminal or the symbol $ (the input endmarker).
  - The production $A \rightarrow \alpha$ is chosen if the next input symbol *a* is in FIRST(α).
  - When $\alpha = \varepsilon$ or $\alpha \overset{*}{\Rightarrow} \varepsilon$, we should choose $A \rightarrow \alpha$ if
    - The current input symbol is in FOLLOW(A) or
    - The $ on the input has been reached and $ is in FOLLOW(A).

Predictive Parsing Table (Cont.)

- **Algorithm**: Construction of a predictive parsing table
- **INPUT**: Grammar *G*.
- **OUTPUT**: Parsing table *M*.
- **METHOD**: For each production $A \rightarrow \alpha$ of the grammar, do the following:
  - For each terminal *a* in FIRST(A), add $A \rightarrow \alpha$ to *M[A, a]*.
  - If ϵ is in FIRST(α), then for each terminal *b* in FOLLOW(A), add $A \rightarrow \alpha$ to *M[A, b]*.
  - If ϵ is in FIRST(α) and $ is in FOLLOW(A), add $A \rightarrow \alpha$ to *M[A, $]*.
  - If (after performing the above) there is no production in *M[A, a]*, then set *M[A, a]* to **error** or an empty entry.

Predictive Parsing Table (Cont.)

E → TE’
E’ → + TE’ | ε
T → FT’ (4.2)
T’ → * FT’ | ε
F → (E) | **id**

- E→TE’: FIRST(TE’) = FIRST(T) = { (, **id** }
- E’→+TE’: FIRST(+TE’) = {+}
- E’→ε: FOLLOW(E’)={), $ }
- T→FT’: FIRST(FT’)=FIRST(F)={ (, **id** }
- T’→*FT’: FIRST(*FT’)={ * }
- T’→ε: FOLLOW(T’)={ +,), $}
- F→(E): FIRST((E))={ (}
- F→**id**: FIRST(id) = { **id** }

- FIRST(E) = { (, id }
- FIRST(E’) = { +, ε }
- FIRST(T) = { (, id }
- FIRST(T’) = { *, ε }
- FIRST(F) = { (, id }

- FOLLOW(E) = {), $ }
- FOLLOW(E’) = {), $ }
- FOLLOW(T) = { +,), $ }
- FOLLOW(T’) = { +,), $ }
- FOLLOW(F) = { +, *,), $ }

NON-TERMINAL	INPUT SYMBOL					
	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→**id***			*F→(E)*		

Predictive Parsing Table (Cont.)

- For every LL(1) grammar, each parsing-table entry uniquely identifies a production or signals an error.
  - If G is left-recursive or ambiguous, then *M* will have at least one multiply defined entry.
  - Although left-recursion elimination and left factoring are easy to do, some grammars have no corresponding LL(1) grammar.
- E.g.,
  - S→iEtSS': FIRST(iEtSS') = { i }
  - S→a: FIRST(a) = { a }
  - S'→eS: FIRST(eS) = { e }
  - S'→ε: FOLLOW(S')= {e, $}
  - E→b: FIRST(b) = {b}

- FOLLOW(S') = FOLLOW(S)
- FOLLOW(S) = {$}: start symbol
- FOLLOW(S)=FIRST(S')={e}

S → *i* E *t* S S' | a
S' → eS | ε
E → b

Grammar

ambiguity

NON-TERMINAL	INPUT SYMBOL					
	a	**b**	**e**	**i**	**t**	**$**
S	*S→a*			*S→iEtSS'*		
S'			*S'→ε* *S'→eS*			*S'→ε*
E		*E→b*				

Nonrecursive Predictive Parsing

- A nonrecursive predictive parser is a table-driven parser that maintains a stack explicitly instead of recursive calls.
- If *w* is the matched input so far, then the stack holds a sequence of grammar symbols α such that $S \overset{*}{\underset{lm}{\Rightarrow}} w\alpha$

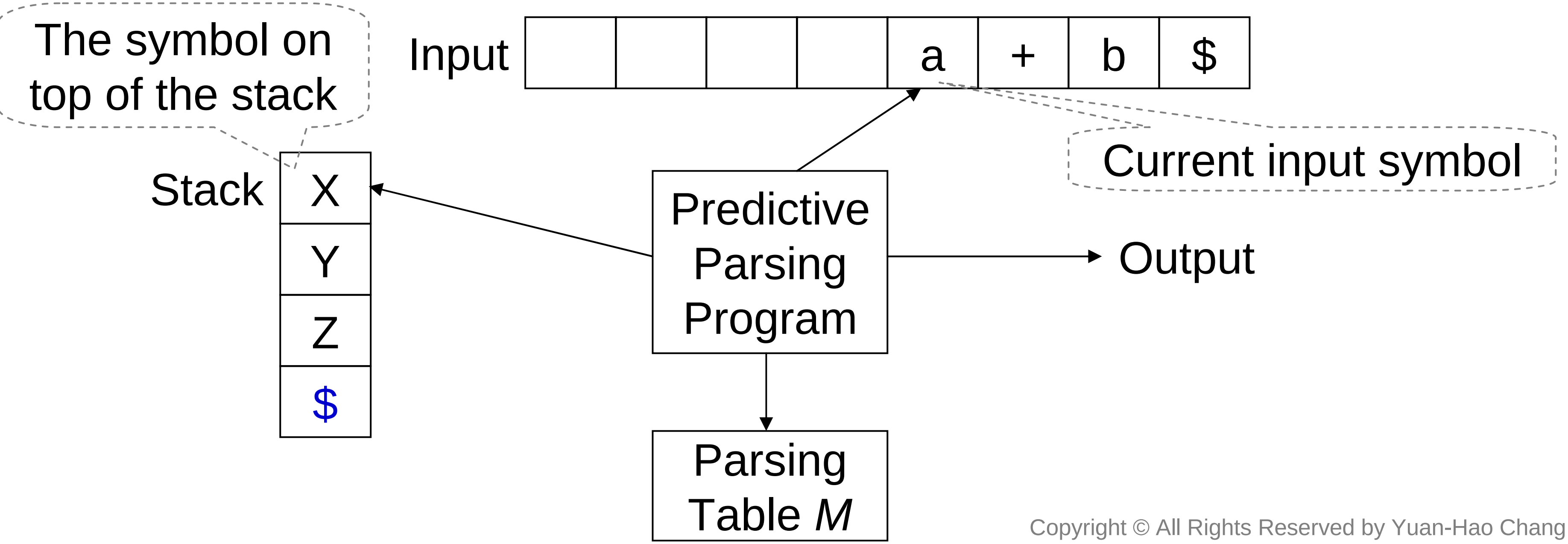


Table-Driven Predictive Parsing

- **Algorithm**: Table-driven predictive parsing
- **INPUT**: A string *w* and a parsing table *M* for grammar *G*.
- **OUTPUT**: If *w* is in *L(G)*, a leftmost derivation of *w*; otherwise, an error indication.
- **METHOD**: Initially, the parser is in a configuration with *w$* in the input buffer, and the start symbol *S* of *G* on top of the stack, above *$*.

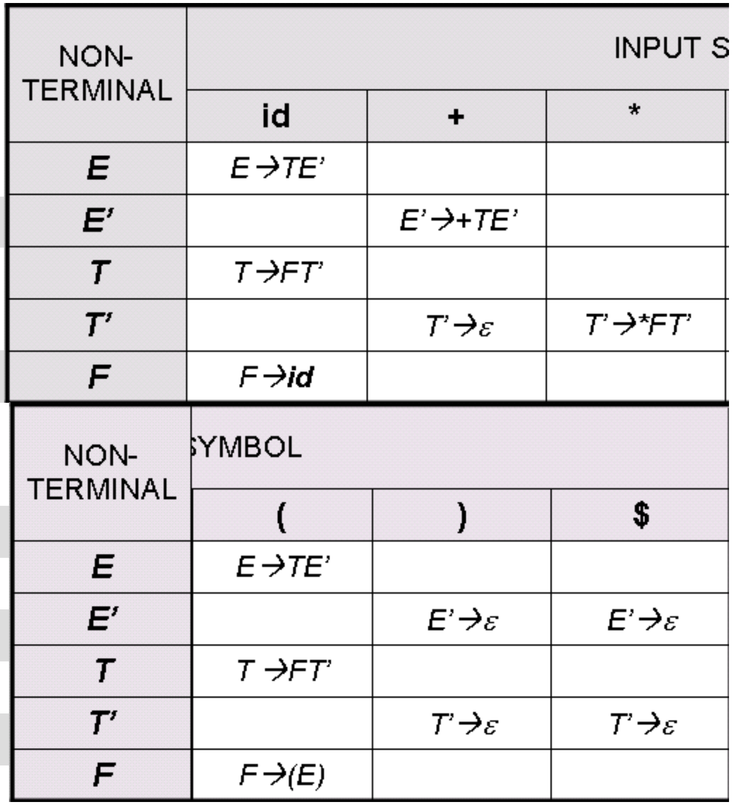
```
set ip to the first symbol of w;
set X to the top stack symbol; /* a is the current input symbol */
while (X≠$) { /* stack is not empty */
    if( X is a ) pop the stack and advance ip; /* pop X */
    else if (X is a terminal) error();
    else if (M[X, a] is an error entry) error();
    else if (M[X,a]=X→$Y_1Y_2…Y_k$) {
        output the production X→ $Y_1Y_2…Y_k$
        pop the stack; /* pop X  */
        push $Y_kY_{k-1}…Y_1$ onto the stack with $Y_1$ on top;
    }
    set X to the top stack symbol;
}
```

E → TE'
E' → + TE' | ε
T → FT' (4.2)
T' → * FT' | ε
F → (E) | **id**

Table-Driven Predictive Parsing (Cont.)

- Input: **id+id*id**

MATCHED	STACK	INPUT	ACTION
	E$	**id**+**id***id**$	
	TE'$	**id**+**id***id**$	output E→TE'
	FT'E'$	**id**+**id***id**$	output T→FT'
	id*T'E'$*	**id**+**id***id**$	output F→**id**
id	*T'E'$*	+**id***id**$	match **id**
id	*E'$*	+**id***id**$	output T'→ε
id	*+TE'$*	+**id***id**$	output E'→+TE'
id+	*TE'$*	**id***id**$	match +
id+	*FT'E'$*	**id***id**$	output T→FT'
id+	**id***T'E'$*	**id***id**$	output F→**id**
id+**id**	*T'E'$*	***id**$	match **id**
id+**id**	**FT'E'$*	***id**$	output T'→*FT'
id+**id***	*FT'E'$*	**id**$	match *
id+**id***	**id***T'E'$*	**id**$	output F→**id**
id+**id*id**	*T'E'$*	$	match **id**
id+**id*id**	*E'$*	$	output T'→ε
id+**id*id**	*$*	$	output E'→ε
			match $



NON-TERMINAL	INPUT SYMBOL					
	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→**id***			*F→(E)*		

Error Recovery in Predictive Parsing

- An error is detected during predictive parsing
  - When the terminal on top of the stack does not match the next input symbol. Or
  - When nonterminal *A* is on top of the stack, *a* is the next input symbol, and *M[A, a]* is **error**.
- Error recovery methods:
  - Panic mode
    - Skip symbols on the input until a token in a selected set of synchronizing tokens appears.
    - The effectiveness depends on the choice of synchronizing set.
  - Phrase-level recovery
    - Fill in the blank entries in the predictive parsing table with pointers to error routines.
    - Error routines may change, insert, or delete symbols on the input and issue appropriate error messages.
    - An infinite loop must be prevented: checking that any recovery action eventually consumes input symbols.

Panic-Mode Error Recovery

- Some heuristics to select synchronizing set:
  - All symbols in FOLLOW(A) as the synchronizing set for nontermial A
    - Skip tokens until an element of FOLLOW(A) is seen and pop A.
  - The symbols that begin higher-level constructs as the synchronizing set of a lower-level construct
    - E.g., add keywords that begin statements to the synchronizing sets for the nonterminals generating expressions.
  - The symbols in FIRST(A) as the synchronizing set for nonterminal A.
  - If a nonterminal can generate the empty string, the production deriving ε can be used as a default.
    - To postpone some error detection, but cannot miss an error.
  - If a terminal on top of the stack cannot be matched, pop the terminal, issue a message saying that the terminal was inserted, and continue parsing.
    - This approach takes the synchronizing set of a token to consist of all of other tokens.

Panic-Mode Error Recovery (Cont.)

E → TE'
E' → + TE' | ε
T → FT' (4.2)
T' → * FT' | ε
F → (E) | **id**

- Obtain synchronizing tokens from the FOLLOW set of the nonterminal.
  - If checked M[A, a] is blank, skip the input symbol *a*.
  - If the entry is *synch*, pop the nonterminal on top of the stack.
  - If a token on top of the stack does not match the input symbol, pop the token from the stack.

- FOLLOW(E) = {), $ }
- FOLLOW(E') = {), $ }
- FOLLOW(T) = { +,), $ }
- FOLLOW(T') = { +,), $ }
- FOLLOW(F) = { +, *,), $ }

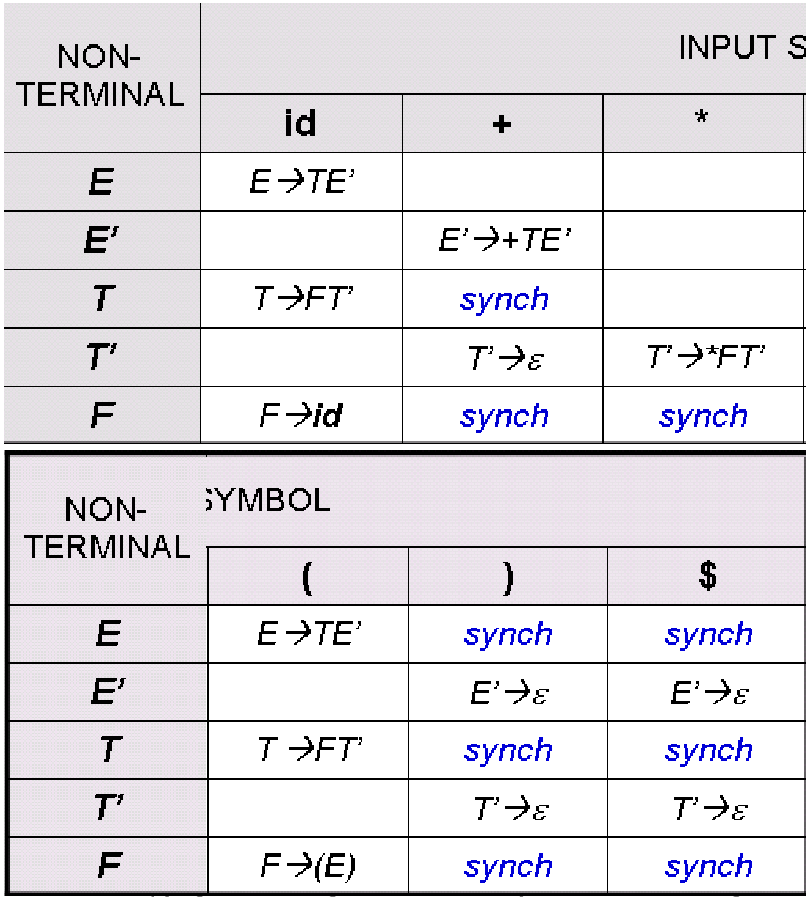
NON-TERMINAL	INPUT SYMBOL					
	id	**+**	*****	**(**	**)**	**$**
E	*E→TE'*			*E→TE'*	*synch*	*synch*
E'		*E'→+TE'*			*E'→ε*	*E'→ε*
T	*T→FT'*	*synch*		*T→FT'*	*synch*	*synch*
T'		*T'→ε*	*T'→*FT'*		*T'→ε*	*T'→ε*
F	*F→**id***	*synch*	*synch*	*F→(E)*	*synch*	*synch*

Panic-Mode Error Recovery (Cont.)

- Input: **+id*+id**

E → TE'
E' → + TE' | ε
T → FT' (4.2)
T' → * FT' | ε
F → (E) | **id**

MATCHED	STACK	INPUT	Remark
	E$	+id*+id$	error, skip +
	E$	id*+id$	**id** is in FIRST(E)
	TE'$	id*+id$	
	FT'E'$	id*+id$	
	idT'E'$	id*+id$	
id	T'E'$	*+id$	match **id**
id	*FT'E'$	*+id$	
id*	FT'E'$	+id$	match *
id*	FT'E'$	+id$	Error, M[F, +]=synch
id*	T'E'$	+id$	F has been popped
id*	E'$	+id$	
id*	+TE'$	+id$	
id*+	TE'$	id$	match +
id*+	FT'E'$	id$	
id*+	**id**T'E'$	id$	
id*+id	T'E'$	$	match **id**
id*+id	E'$	$	
id*+id	$	$	

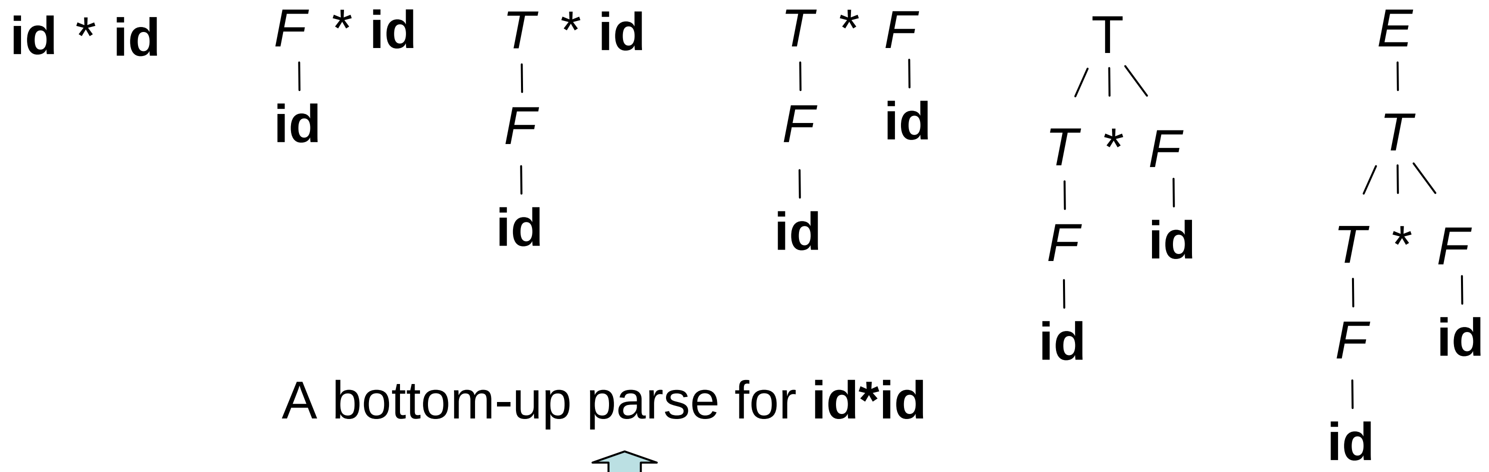


NON-TERMINAL	INPUT SYMBOL					
	id	+	*	(	)	$
E	E→TE'			E→TE'	synch	synch
E'		E'→+TE'			E'→ε	E'→ε
T	T→FT'	synch		T→FT'	synch	synch
T'		T'→ε	T'→*FT'		T'→ε	T'→ε
F	F→id	synch	synch	F→(E)	synch	synch

Bottom-Up Parsing

Bottom-Up Parse

- A bottom-up parse constructs a parse tree for an input string beginning at the leaves towards the root.
  - It describes parsing as the process of building parse trees.



A bottom-up parse for **id*id**

The derivation corresponds to the parse:

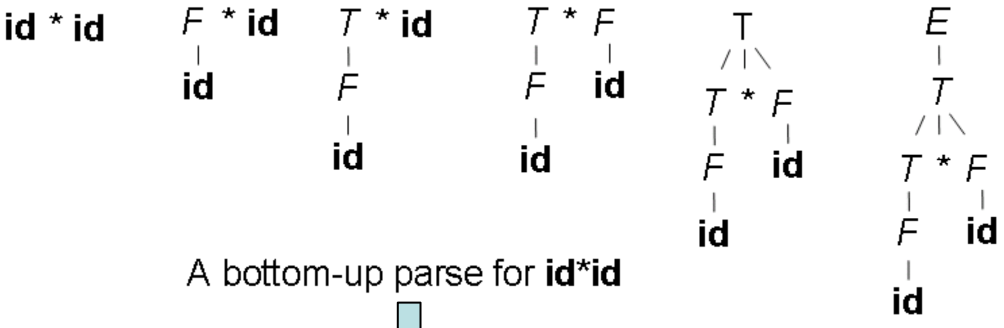
$E \Rightarrow T \Rightarrow T*F \Rightarrow T*\textbf{id} \Rightarrow F*\textbf{id} \Rightarrow \textbf{id}*\textbf{id}$

A rightmost derivation

$E \rightarrow E + T \mid T$
$T \rightarrow T * F \mid F$ (4.1)
$F \rightarrow (E) \mid \textbf{id}$

Reductions

- Bottom-up parsing is the process of “reducing” a string *w* to the start symbol of the grammar.
  - The goal is to construct a derivation in reverse.
  - At each reduction step, a specific substring matching the body of a production is replaced by the nonterminal at the head of the production.
  - Key decisions: When to reduce and what production to apply



A bottom-up parse for **id*id**

$E \rightarrow E + T \mid T$
$T \rightarrow T * F \mid F$ (4.1)
$F \rightarrow (E) \mid \textbf{id}$

Reduction sequence:
id*id**, F*id**, T****id**, T*F, T, E*

A reduction is the reverse of a step in a derivation.

Handle Pruning

- Bottom-up parsing during a left-to-right scan of the input constructs a right-most derivation in reverse.
  - Handle: a handle is a substring that matches the body of a production and
  - Reduction: the reduction of a handle represents one step along the reverse of a rightmost derivation.

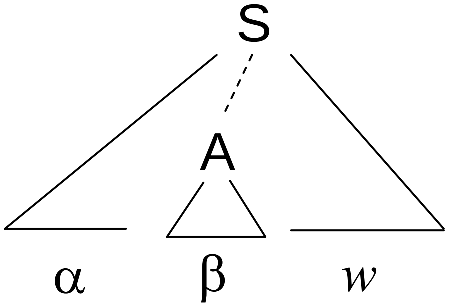
$$\begin{aligned} E &\rightarrow E + T \mid T \\ T &\rightarrow T * F \mid F \qquad (4.1) \\ F &\rightarrow (E) \mid \mathbf{id} \end{aligned}$$

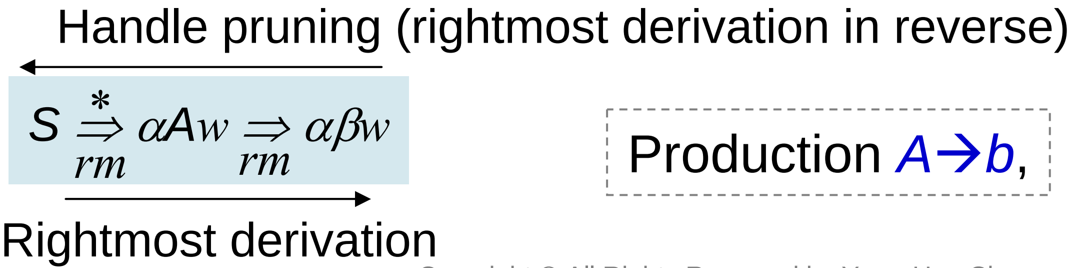
Right sentential form	Handle	Reducing production
$\mathbf{id}_1 * \mathbf{id}_2$	$\mathbf{id}_1$	$F \rightarrow \mathbf{id}$
$F * \mathbf{id}_2$	F	$T \rightarrow F$
$T * \mathbf{id}_2$	$\mathbf{id}_2$	$F \rightarrow \mathbf{id}$
$T * F$	$T * F$	$E \rightarrow T * F$

a, b, c: a terminal
w, x, y, z: strings of terminals
A,B, C: a nonterminal
W, X, Y, Z: a grammar symbol (termina or nonterminal)
α, β, γ: strings of grammar symbols

Handle Pruning (Cont.)

- If S $\overset{*}{\Rightarrow} \alpha A w \Rightarrow \alpha\beta w$, given a production $A \rightarrow \beta$,
  - The β (or $A \rightarrow \beta$) is a handle of $\alpha\beta w$.
- Given a right sentential form γ, a handle β of γ, a production $A \rightarrow \beta$, and a *position* of γ where β may be found, replace β at that *position* by *A* to produce the previous right-sentential form in a rightmost derivation of γ.
- Every right-sentential form of the grammar has exactly one handle, except ambiguous grammars.
  - A rightmost derivation in reverse can be obtaine by "handle pruning".




Handle pruning (rightmost derivation in reverse)

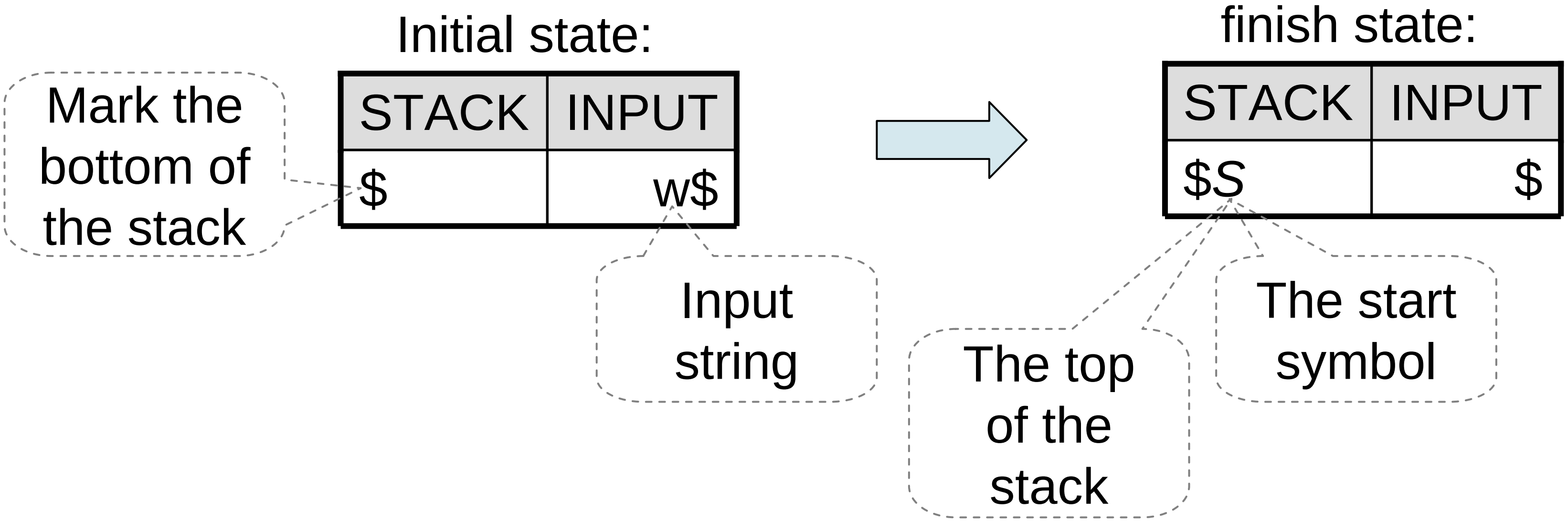
$$S \underset{rm}{\overset{*}{\Rightarrow}} \alpha A w \underset{rm}{\Rightarrow} \alpha\beta w$$

Rightmost derivation

Production *A→b*,

Shift-Reduce Parsing

- Shift-reduce parsing is a form of bottom-up parsing in which
  - a stack holds grammar symbols and
  - an input buffer holds the rest of the string to be parsed.
- The handle always appears at the top of the stack just before it is identified as the handle.



Initial state:

STACK	INPUT
$	w$

finish state:

STACK	INPUT
$*S*	$

Mark the bottom of the stack

Input string

The top of the stack

The start symbol

Shift-Reduce Parsing (Cont.)

- Operations of shift-reduce parsing:
  - Shift: Shift the next input symbol onto the top of the stack.
  - Reduce: Locate the left end of the string within the stack and decide with what nonterminal to replace the string.
  - Accept: Announce successful completion of parsing.
  - Error: Discover a syntax error and call an error recovery routine.

$E \rightarrow E + T \mid T$

$T \rightarrow T * F \mid F$ (4.1)

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

STACK	INPUT	ACTION
$\$$	$\mathbf{id}_1 * \mathbf{id}_2 \$$	shift
$\$\mathbf{id}_1$	$* \mathbf{id}_2 \$$	reduce by $F \rightarrow \mathbf{id}$
$\$F$	$* \mathbf{id}_2 \$$	reduce by $T \rightarrow F$
$\$T$	$* \mathbf{id}_2 \$$	shift
$\$T*$	$\mathbf{id}_2 \$$	shift
$\$T*\mathbf{id}_2$	$\$$	reduce by $F \rightarrow \mathbf{id}$
$\$T*F$	$\$$	reduce by $T \rightarrow T*F$
$\$T$	$\$$	reduce by $E \rightarrow T$
$\$E$	$\$$	accept

E.g., parse $\mathbf{id}_1 * \mathbf{id}_2$

Shift-Reduce Parsing (Cont.)

The *handle* will always eventually appear on top of the stack.

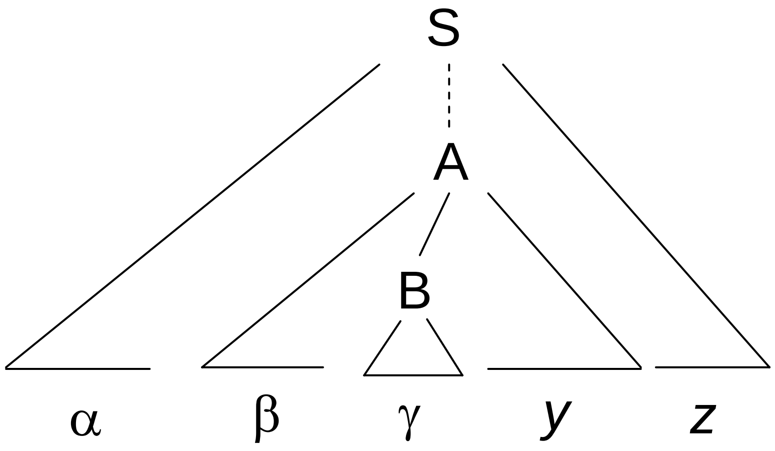
Case 1

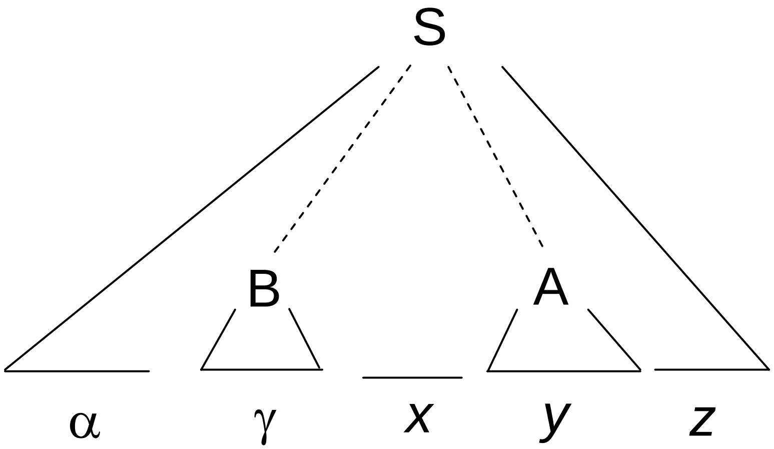
$S \overset{*}{\underset{rm}{\Rightarrow}} \alpha A z \underset{rm}{\Rightarrow} \alpha \beta B y z \underset{rm}{\Rightarrow} \alpha \beta \gamma y z$

Leftmost derivation

Case 2

$S \overset{*}{\underset{rm}{\Rightarrow}} \alpha B x A z \underset{rm}{\Rightarrow} \alpha B x y z \underset{rm}{\Rightarrow} \alpha \gamma x y z$





STACK	INPUT	ACTION
$αβγ	yz $	reduce B→γ
$αβB	yz $	shift
$αβBy	z $	reduce A→βBy
$ αA	z $	

STACK	INPUT	ACTION
$αγ	xyz $	reduce B→γ
$αB	xyz $	shift xy
$αBxy	z $	reduce A→y
$ αBxA	z $	

Conflicts During Shift-Reduce Parsing

- Some context-free grammars could let the shift-reduce parsing encounter conflicts on deciding the next action.
  - Shift/reduce conflict
    - Cannot decide whether to shift or to reduce
    - E.g., shift-reduce conflict

stmt → **if** expr **then** stmt
| **if** expr **then** stmt **else** stmt
| **other**

Dangling-else grammer

STACK	INPUT
… **if** *expr* **then** *stmt*	**else …** $

Cannot determine whether to shift or to reduce

  - Reduce/reduce conflict
    - Cannot decide which production should be adopted to reduce

Conflicts During Shift-Reduce Parsing (Cont.)

- E.g., a grammar for function call and array for the input **p(i,j)**
  - A function called with parameters surrounded by parentheses.
  - Indices of arrays are surrounded by parentheses.

(1)	*stmt* →	**id** (*parameter_list*)
(2)	*stmt* →	*expr* := *expr*
(3)	*parameter_list* →	*parameter_list*, *parameter*
(4)	*parameter_list* →	*parameter*
(5)	*parameter* →	**id**
(6)	*expr* →	**id** (*expr_list*)
(7)	*expr* →	**Id**
(8)	*expr_list* →	*expr_list*, *expr*
(9)	*expr_list* →	*expr*

One solution to resolve this problem is to change production into
stmt → **procid** (*parameter_list*)
For the token name of procedures.

STACK	INPUT
… **procid** (**id**	, **id**) … $

A procedure call is encountered

STACK	INPUT
… **id** (**id**	, **id**) … $

An array is encountered

Input: p(i,j) is converted to the token string **id**(**id**, **id**)
The correct choice is production (5) if p is a procedure call.
The correct choice is production (7) if p is an array.

Introduction to LR Parsing: Simple LR (SLR)

LR Parsers

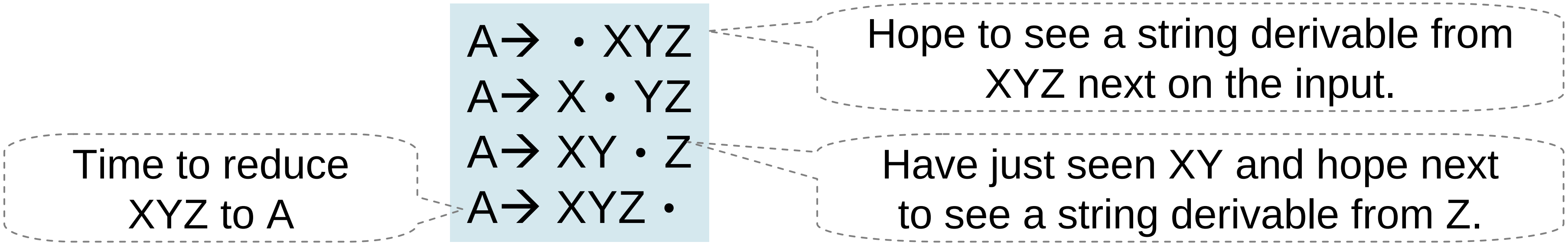
- The most prevalent type of bottom-up parser is the *LR(k)* parsing:
  - *L* stands for left-to-right scanning of the input.
  - *R* stands for constructing a rightmost derivation in reverse.
  - *k* is number of input symbols of lookahead.
    - The cases *k=0* or *k=1* are of practical interest.
    - When *(k)* is omitted, k is assumed to be 1.
- Simple LR (SLR)
  - The easiest method for constructing shift-reduce parsers.
- LR parsers are table driven.
  - An *LR grammar* is a grammar whose parsing table could be constructed by LR parsers.

Why LR Parsers?

- Advantages:
  - LR parsers can recognize almost every programming-language constructs written by context-free grammars.
    - Non-LR context-free grammars exists, but they usually can be avoided.
  - The LR-parsing method is the most general nonbacktracking shift-reduce parsing method.
  - An LR parser can detect a syntactic error as soon as it is possible to do so on a left-to-right scan of the input.
  - LR methods are a proper superset of the LL or predictive methods.
    - With *k* input symbols of lookahead, an *LR(k)* parser can recognize the occurrence of a production, but an *LL* parser can not guarantee this.
- Drawbacks:
  - It is too much work to construct an LR parser by hand for a typical programming-language grammar.

Items and the LR(0) Automation

- An LR parser makes shift-reduce decisions by maintaining states to keep track of where we are in a parse.
  - A state represents a set of "items".
  - An LR(0) item (*item* for short) of a grammar G is a production of G with a dot at some position of the body.
  - An item indicates how much of a production we have seen at a given time.
    - E.g., production A→XYZ yields the four items:



A→ • XYZ

A→ X • YZ

A→ XY • Z

A→ XYZ •

Hope to see a string derivable from XYZ next on the input.

Have just seen XY and hope next to see a string derivable from Z.

Time to reduce XYZ to A

- The production A→ε generates only one item A→ •

Items and the LR(0) Automation (Cont.)

- Canonical LR(0) collection is one collection of sets of LR(0) items.
  - Provide the basis for constructing a deterministic finite automation (called an LR(0) automation) that is used to make parsing decisions.
- To construct the canonical LR(0) collection, an augmented grammar and two functions (CLOSURE and GOTO) are needed:
  - An augmented grammar G’ of G has a new start symbol S’ and production S’→S, if S is the start symbol of G. The new production is to indicate when it should stop parsing and announce acceptance of the input.

The Function CLOSURE

- If *I* is a set of items for a grammar *G*, then CLOSURE(*I*) is the set of items constructed from *I* by the two rules:
  - 1. Initially, add every item in *I* to CLOSURE(*I*).
  - 2. If $A \rightarrow \alpha \cdot B\beta$ is in CLOSURE(I) and $B \rightarrow \gamma$ is a production, add the item $B \rightarrow \cdot\, \gamma$ to CLOSURE(*I*) if it is not there.
    Apply this rule until no more new items can be added to CLOSURE(*I*).
- Intuitively, $A \rightarrow \alpha \cdot B\beta$ in CLOSURE(*I*) indicates that we might next see a substring derivable from $B\beta$ as input.
  - Therefore, if $B \rightarrow \gamma$ is a production, we also add $B \rightarrow \cdot\, \gamma$ in CLOSURE(*I*).

Computation of CLOSURE

- If *I* is the set of one item {[E’→ • E]}, then CLOSURE(*I*) contains the set of items

E’ → *E*
E → *E* + *T* | *T*
T → *T* * *F* | *F*
F → (*E*) | **id**

Augmented grammar

E’→ • E

E → • E+T
E → • T
T → • T*F
T → • F
F → • (E)
F → • **id**

CLOSURE(*I*)

Computation of CLOSURE (Cont.)

```
SetOfItems CLOSURE(I) {
    J = I;
    repeat
        for (each item $A \rightarrow \alpha \cdot B\beta$ in J)
            for (each production $B \rightarrow \gamma$ of G)
                add $B \rightarrow \cdot \gamma$ to J;
    until no more items are added to J on one round;
    return J;
}
```

Computation of CLOSURE

- A list of the nonterminals B whose productions were added to *I* by CLOURSE is suffice.
  - If one B-production is added to the closure, then all B-productions will be similarly added to the closure.

Kernel Items and Nonkernel Items

- Sets of items can be divided into two classes:
  - Kernel items:
    - The initial item, $S' \rightarrow \cdot S$, and all items whose dots are not at the left end.
  - Nonkernel items:
    - All items with their dots at the left end, except for $S' \rightarrow \cdot S$.
- Each set of items is formed by taking the closure of a set of kernel items.
- Items added in the closure can never be kernel items.

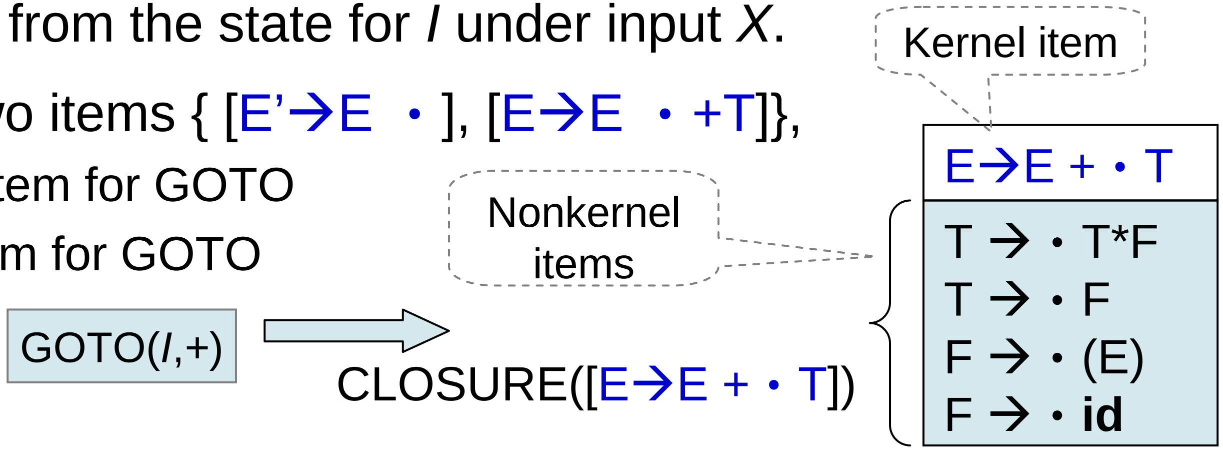
The Function GOTO

a, b, c: a terminal
w, x, y, z: strings of terminals
A,B, C: a nonterminal
W, X, Y, Z: a grammar symbol (terminal or nonterminal)
α, β, γ: strings of grammar symbols

- GOTO(*I*, *X*) is defined to be the closure of the set of all items [A→αX · β] such that [A→α · Xβ] is in *I*, where
  - *I*: a set of items
  - *X*: a grammar symbol

E' → *E*
E → *E* + *T* | *T*
T → *T* * *F* | *F*
F → (*E*) | **id**
Augmented grammar

- The GOTO function is used to define the transitions in the LR(0) automation for a grammar.
- The states of the automation correspond to sets of items, and GOTO(*I*,*X*) specifies the transition from the state for *I* under input *X*.
- E.g., If *I* is the set of two items { [E'→E ·], [E→E · +T]},
  - [E'→E ·] is not the item for GOTO
  - [E→E · +T] is the item for GOTO
    →[E→E + · T]

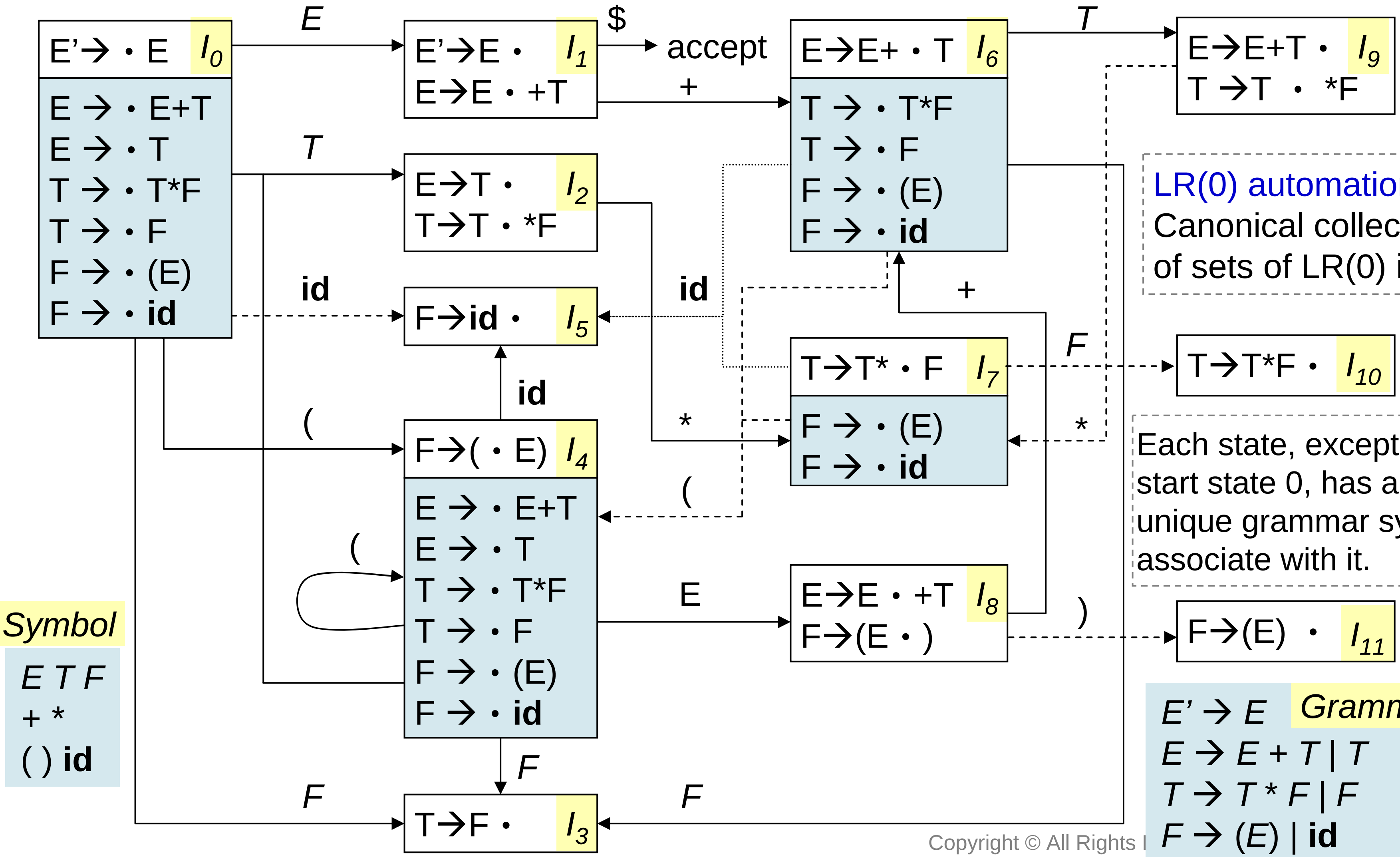


Canonical Collection of Sets of LR(0) Items

- The canonical collection ***C*** of sets of LR(0) items can be computed as follows:

```
void items(G’) {
    C = CLOSURE( { [S’→ · S] } );
    repeat
        for (each set of items I in C)
            for (each grammar symbol X)
                if (GOTO(I, X) is not empty and not in C)
                    add GOTO(I, X) to C;
    until no new sets of items are added to C on a round;
}
```

Canonical Collection of Sets of LR(0) Items (Cont.)



I_0: E'→ · E; E → · E+T; E → · T; T → · T*F; T → · F; F → · (E); F → · id

I_1: E'→E ·; E→E · +T

I_2: E→T ·; T→T · *F

I_3: T→F ·

I_4: F→(· E); E → · E+T; E → · T; T → · T*F; T → · F; F → · (E); F → · **id**

I_5: F→**id** ·

I_6: E→E+ · T; T → · T*F; T → · F; F → · (E); F → · **id**

I_7: T→T* · F; F → · (E); F → · **id**

I_8: E→E · +T; F→(E ·)

I_9: E→E+T ·; T →T · *F

I_{10}: T→T*F ·

I_{11}: F→(E) ·

LR(0) automation = Canonical collection of sets of LR(0) items

Each state, except the start state 0, has a unique grammar symbol associate with it.

Symbol

E T F
+ *
( ) **id**

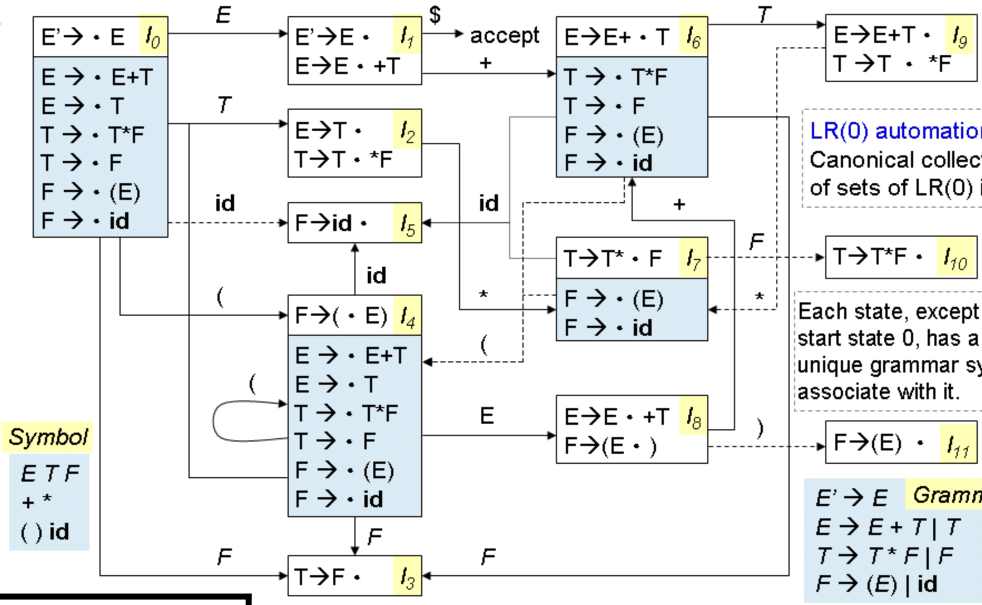
Grammar G'

E' → E
E → E + T | T
*T → T * F | F*
F → (E) | **id**

Use of the LR(0) Automation

- The central idea of SLR parsing is the construction from the grammar of the LR(0) automation.
  - The states of the LR(0) automation are the sets of items from the canonical LR(0) collections.
  - The transitions are given by the GOTO function.
  - The start state of the LR(0) automation is CLOSURE({[S'→ · S]}), where S' is the start symbol of the augmented grammar.
  - All states are accepting states.
  - "State j" refers to the state corresponding the set of items I_j.
- The LR(0) automation helps with shift-reduce decisions on when to shift and when to reduce.
  - Shift on the next symbol *a* if state *j* has a transition *a*.
  - Otherwise, reduce with the production indicated by the items in state *j*.

Parse id*id



LR(0) automation = Canonical collection of sets of LR(0) items

Each state, except the start state 0, has a unique grammar symbol associate with it.

Grammar G'

$E' \rightarrow E$

$E \rightarrow E + T \mid T$

$T \rightarrow T * F \mid F$

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

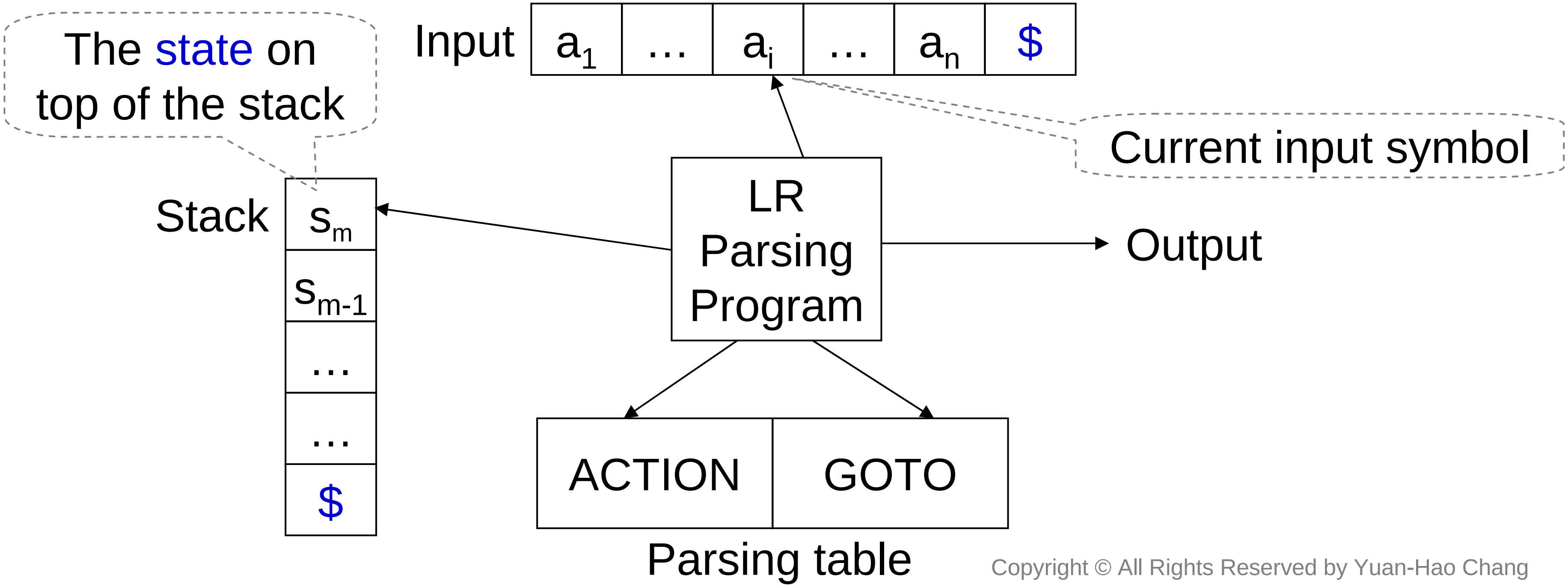
At line (1), the next input symbol is id so state 0 has a transition to state 5 on id.

LINE	STACK	SYMBOL	INPUT	ACTION
(1)	0	$	$\mathbf{id}_1 * \mathbf{id}_2 \$$	shift to 5
(2)	0 5	$\$\mathbf{id}_1$	$* \mathbf{id}_2 \$$	reduce by $F \rightarrow \mathbf{id}$
(3)	0 3	$\$F$	$* \mathbf{id}_2 \$$	reduce by $T \rightarrow F$
(4)	0 2	$\$T$	$* \mathbf{id}_2 \$$	shift to 7
(5)	0 2 7	$\$T*$	$\mathbf{id}_2 \$$	shift to 5
(6)	0 2 7 5	$\$T*\mathbf{id}_2$	$	reduce by $F \rightarrow \mathbf{id}$
(7)	0 2 7 10	$\$T*F$	$	reduce by $T \rightarrow T*F$
(8)	0 2	$\$T$	$	reduce by $E \rightarrow T$
(9)	0 1	$\$E$	$	accept

At line (2), state 5 is pushed onto the stack, and no transition from state 5 on input *, so reduce id with production F→id to pop state 5 from the stack, and put state 3 to the stack (due to the transition from state 0 to state 3 on F.

Model of an LR Parser

- The parsing table changes from one parser to another.
- The parsing program reads characters from an input buffer one at a time.
- An LR parser shifts a state. Each state summarizes the information contained in the stack below it. (A shift-reduce parser shifts a symbol.)

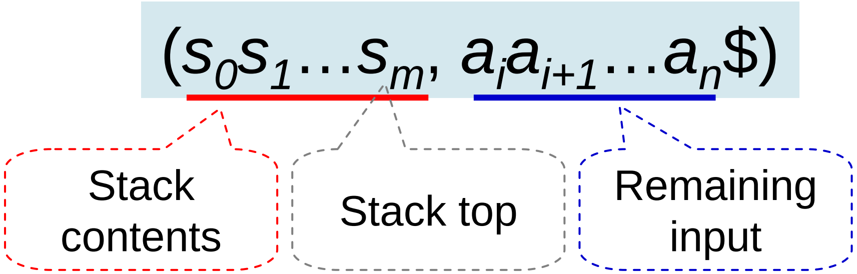


Structure of the LR Parsing Table

- The parsing table consists of two parts:
  - A parsing-action function ACTION
    - ACTION takes a state *i* and a terminal *a* (or $). The value of ACTION [i, a] can have one of four forms:
      - Shift *j*, where *j* is a state: Shift state *j* representing input *a* to the stack.
      - Reduce A→β: Reduce β on the top of the stack to head *A*.
      - Accept: The parser accepts the input and finishes parsing.
      - Error: The parser discovers an error in its input.
  - A goto function GOTO
    - If GOTO[I_i, A] = I_j, then GOTO maps a state *i* and a nonterminal A to state *j*.

LR-Parser Configuration

- The configuration of LR-parsers is to represent the complete state of the parser.
- A configuration of an LR parser is a pair:



$(s_0s_1\ldots s_m,\ a_ia_{i+1}\ldots a_n\$)$

  - This configuration represents the right-sentential form:

    $(X_1X_2\ldots X_m,\ a_ia_{i+1}\ldots a_n)$

    where state s_i represents grammar symbol X_i.
  - Note: the start state s_0 does not represent any grammar symbol. It serves as a bottom-of-stack marker.

Behavior of the LR Parser

- The next move of the parser from the configuration is determined by the entry ACTION[s_m, a_i].
  - s_m: the state on top of the stack
  - a_i: the current input symbol

$(s_0s_1\dots s_m, a_ia_{i+1}\dots a_n\$)$

Current configuration

- The move of ACTION:
  - 1. If ACTION[s_m, a_i]=shift s, it shifts the next state s onto the stack. The symbol a_i need not be held on the stack, since it can be recovered from s.

$(s_0s_1\dots s_ms, a_{i+1}a_{i+2}\dots a_n\$)$

  - 2. If ACTION[s_m, a_i]=reduce $A \rightarrow \beta$, it executes a reduce move, where r is the length of β, $\beta=X_{m-r+1}\dots X_m$, and s=GOTO[s_{m-r} ,A].

$(s_0s_1\dots s_{m-r}s, a_ia_{i+1}\dots a_n\$)$

  - 3. If ACTION[s_m, a_i]=accept, it executes parsing completed.
  - 4. If ACTION[s_m, a_i]=error, it has discovered an error.

LR-Parsing Algorithm

- **Algorithm**: LR-parsing algorithm
- **INPUT**: An input string *w* and an LR-parsing table with functions ACTION and GOTO for a grammar G.
- **OUTPUT**: If *w* is in L(G), the reduction steps of a bottom-up parse for *w*; otherwise, an error indication.
- **METHOD**: Initially, the parser has the initial state s_0 on its stack, where *w*$ in the input buffer.

```
Let a be the fist symbol of w$;
while(1) { /* repeat forever */
    let s be the state on top of the stack;
    if( ACTION[s, a] = shift t ) {
        push t onto the stack;
        Move a to the next input symbol;
    } else if ( ACTION[s, a] = reduce $A \to \beta$ ) {
        pop $|\beta|$ symbols off the stack;
        let state t now be on top of the stack;
        push GOTO[t, A] onto the stack;
        output the production $A \to \beta$;
    } else if ( ACTION[s, a] = accept ) break; /* parsing is done */
    else call error-recovery routine;
}
```

Case shift

Case reduce

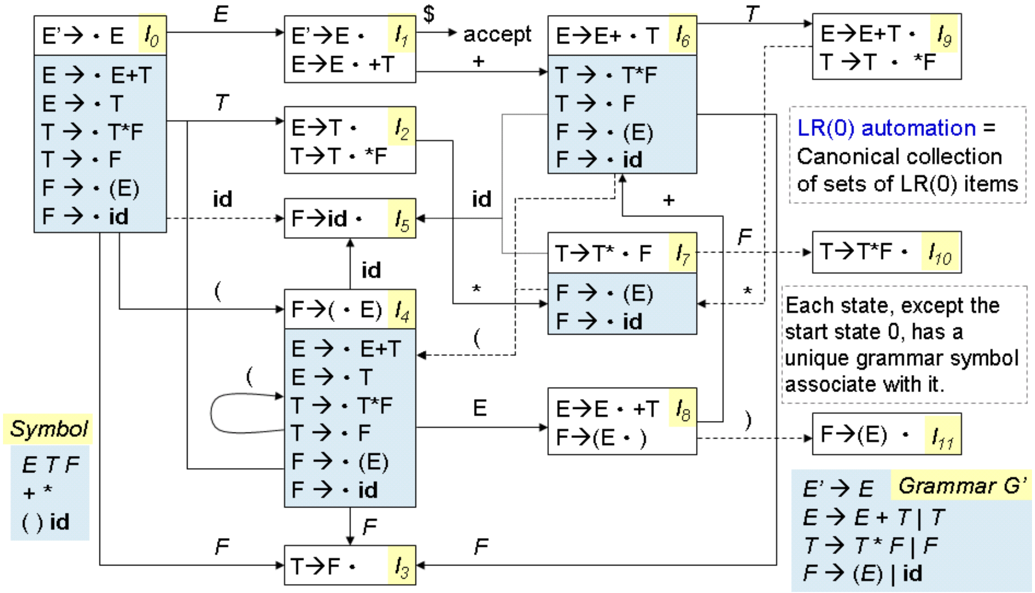
Case accept

Case error

SLR-Parsing Table – SLR(1) Table

- **Algorithm**: Constructing an SLR-parsing table, i.e., SRL(1) Table.
- **INPUT**: An augmented grammar G'.
- **OUTPUT**: The SLR-parsing table functions ACTION and GOTO for G'.
- **METHOD**:
  - 1. Construct C = $\{I_0, I_1, \ldots, I_n\}$, the collection of sets of LR(0) items for G'.
  - 2. State *i* is constructed from I_i. The parsing actions for state *i* are determined as follows:
    - (a) If [A→α · aβ] is in I_i and GOTO(I_i, *a*)=I_j, then set ACTION[*i*, *a*] to "shift *j*", where *a* is a terminal.
    - (b) If [A→α ·] is in I_i, then set ACTION[*i*, *a*] to "reduce A→ α" for all *a* in FOLLOW(*A*), where *A* may not be *S'*.
    - (c) If [S'→S] is in I_i, then set ACTION[*i*, *$*] to "accept".
    - If any conflicting actions result from the above rules, the grammar is not SLR(1) and the algorithm fails to produce a parser for it.
  - 3. If GOTO(I_i, *A*)=I_j, then GOTO[i,A]=*j*.
  - 4. All entries not defined by rules (2) and (3) are made "error."
  - 5. The initial state of the parser is the one constructed from the set of items containing [S'→ · S]

SLR-Parsing Table



LR(0) automation = Canonical collection of sets of LR(0) items

Each state, except the start state 0, has a unique grammar symbol associate with it.

Grammar G'

E' → E
E → E + T | T
*T → T * F | F*
F → (E) | id

- The codes for the actions:
  - si: shift and stack state i.
  - rj: reduce by the production number j
  - acc: accept
  - blank: error

(1) E → E + T
(2) E → T
(3) T → T * F
(4) T → F (4.1)
(5) F → (E)
(6) F → **id**

- FOLLOW(E) = {+,), $ }
- FOLLOW(T) = { *, +,), $ }
- FOLLOW(F) = {*, +,), $ }

STATE	ACTION						GOTO		
	id	+	*	(	)	$	E	T	F
0	s5			s4			1	2	3
1		s6				acc			
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-Parsing Table (Cont.)

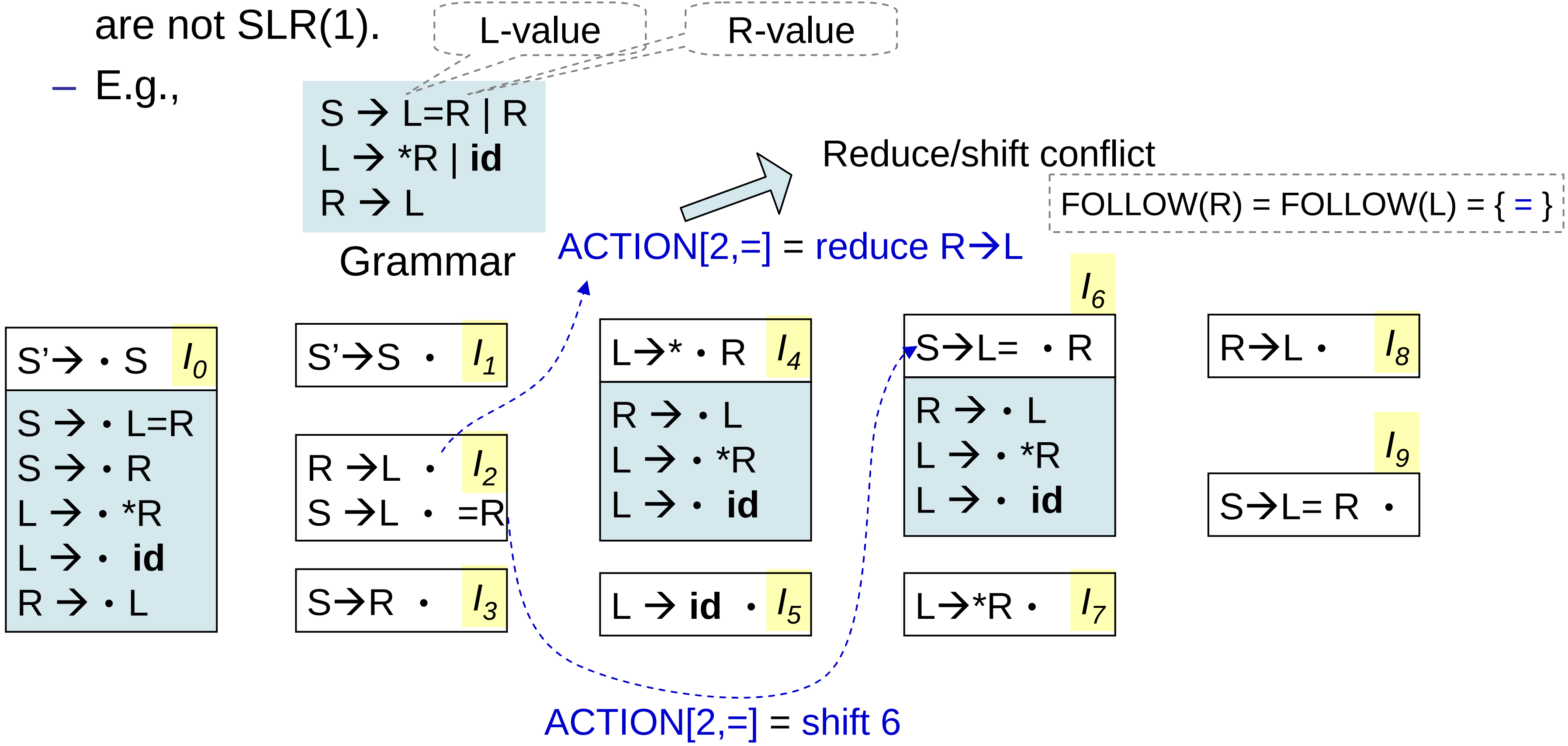
(1) E → E + T
(2) E → T
(3) T → T * F
(4) T → F (4.1)
(5) F → (E)
(6) F → **id**

	STACK	SYMBOLS	INPUT	ACTION
(1)	0		**id**$_1$ * **id**$_2$+**id** $	shift to 5
(2)	0 5	**id**$_1$	* **id**$_2$+**id** $	reduce by *F*→ **id**
(3)	0 3	*F*	* **id**$_2$+**id** $	reduce by *T*→ *F*
(4)	0 2	*T*	* **id**$_2$ +**id**$	shift to 7
(5)	0 2 7	*T**	**id**$_2$+**id** $	shift to 5
(6)	0 2 7 5	*T****id**$_2$	+**id**$	reduce by *F*→ **id**
(7)	0 2 7 10	*T*F*	+**id**$	reduce by *T*→ *T*F*
(8)	0 2	*T*	+**id**$	reduce by *E*→ *T*
(9)	0 1	*E*	+**id**$	shift 6
(10)	0 1 6	*E+*	**id**$	shift 5
(11)	0 1 6 5	*E+***id**	$	reduce by *F*→ **id**
(12)	0 1 6 3	*E+F*	$	reduce by *T*→ *F*
(13)	0 1 6 9	*E+T*	$	reduce by *F*→ *E+T*
(14)	0 1	*E*	$	accept

STA TE	ACTION						GOTO		
	id	+	*	(	)	$	E	T	F
0	s5			s4			1	2	3
1		s6				acc			
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) Grammar

- A grammar using the SLR(1) table is said to be *SLR(1) grammar*.
  - Every SLR(1) grammar is unambiguous, but many unambiguous grammars are not SLR(1).
  - E.g.,



Viable Prefixes

- The LR(0) automation for a grammar characterizes the strings of grammar symbols that can appear on the stack of a shift-reduce parser.
  - The stack contents must be a prefix of a right-sentential form.
  - If the stack holds α and the rest of the input is x, then a sequence of reductions will take αx to S. That is, the derivation $S \overset{*}{\underset{rm}{\Rightarrow}} \alpha x$.
  - The set of valid items for a viable prefix γ is exactly the set of items reached from the initial state along the path labeled γ in the LR(0) automation grammar.
- The prefixes of right sentential forms that can appear on the stack of a shift-reduce parser are called *viable prefixes*.
  - A viable prefix is a right-sentential form that does not continue past the right end of the rightmost handle of that sentential form.
- SLR parsing is based on the fact that LR(0) automata recognize viable prefixes.
  - A viable prefix might have two valid actions to incur conflicts. Such conflicts might be solved by looking at the next input symbol.
  - E.g., $A \rightarrow \beta_1 \cdot \beta_2$ is valid for the prefix $\alpha\beta_1$.
    - If $\beta_2 \neq \varepsilon$, the “shift” actions should be performed.
    - If $\beta_2 \Rightarrow \varepsilon$, it looks whether $A \rightarrow \beta_1$ is a handle, and reduces by $A \rightarrow \beta_1$

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

Viable Prefixes (Cont.)

- The string E+T* is a viable prefix of the grammar, and will be in state 7.

T→T* • F

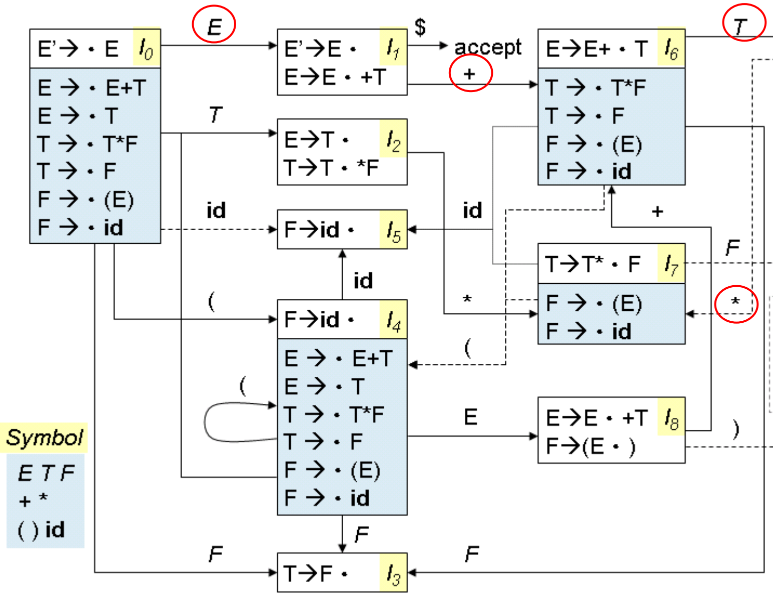
F → • (E)
F → • **id**

Precisely the items *valid* for E+T*

$E' \underset{rm}{\Rightarrow} E \underset{rm}{\Rightarrow} E+T \underset{rm}{\Rightarrow} E+T*F$

$E' \underset{rm}{\Rightarrow} E \underset{rm}{\Rightarrow} E+T \underset{rm}{\Rightarrow} E+T*F \underset{rm}{\Rightarrow} E+T*(E)$

$E' \underset{rm}{\Rightarrow} E \underset{rm}{\Rightarrow} E+T \underset{rm}{\Rightarrow} E+T*F \underset{rm}{\Rightarrow} E+T*\mathbf{id}$



LR(0) automation = Canonical collection of sets of LR(0) items

Each state, except the start state 0, has a unique grammar symbol associate with it.

Grammar G'
$E' \rightarrow E$
$E \rightarrow E + T \mid T$
$T \rightarrow T * F \mid F$
$F \rightarrow (E) \mid$ **id**

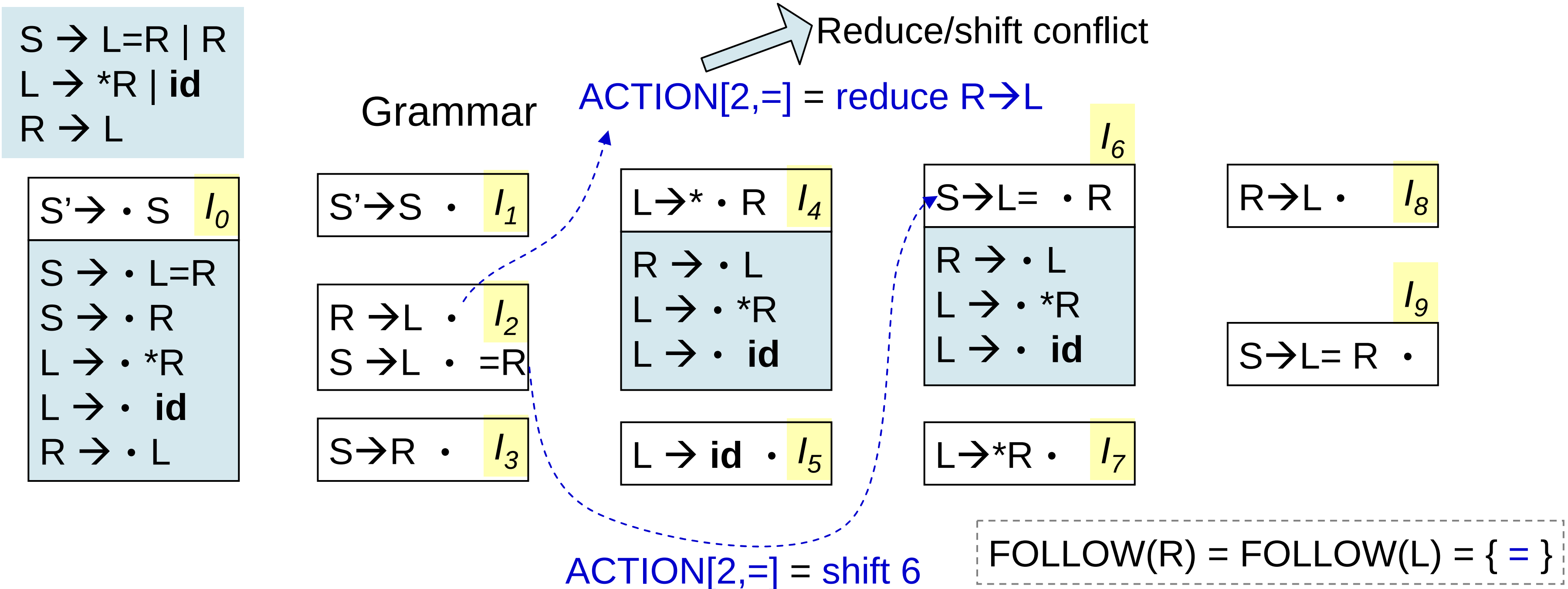
More Powerful LR Parsers

More Powerful LR Parsers

- More powerful LR parsers:
  - 1. Canonical-LR parser (LR parser):
    - Make full use of the lookahead symbol(s) with a large set of LR(1) items.
  - 2. Lookahead-LR parser (LALR parser):
    - By carefully introducing lookaheads into the LR(0) items, the LALR parser can handle more grammars than SLR parsers with the parsing tables that are no bigger than the SLR tables.

Limitations of LR(0) Items or SLR(1) Parsers

- The following grammar has no right-sentential form that begins R=… Thus state 2 (that is the state corresponding to viable prefix L) should not call for reduction with R→L.
- With LR(1) items, the reduce/shift conflict can be avoided.



S → L=R | R
L → *R | **id**
R → L

Grammar

Reduce/shift conflict

ACTION[2,=] = reduce R→L

I_0: S'→ • S; S → • L=R; S → • R; L → • *R; L → • **id**; R → • L

I_1: S'→S •

I_2: R →L •; S →L • =R

I_3: S→R •

I_4: L→* • R; R → • L; L → • *R; L → • **id**

I_5: L → **id** •

I_6: S→L= • R; R → • L; L → • *R; L → • **id**

I_7: L→*R •

I_8: R→L •

I_9: S→L= R •

ACTION[2,=] = shift 6

FOLLOW(R) = FOLLOW(L) = { = }

Canonical LR(1) Items

- Purpose of LR(1) items:
  - Given a production $A \rightarrow \alpha$, exactly indicate which input symbol could follow a handle α when there is a possible reduction to *A*.
- The general form of an LR(1) item:
  - $[A \rightarrow \alpha \cdot \beta, a]$, where $A \rightarrow \alpha\beta$ is a production and *a* is a terminal or \$.
  - 1 refers to the length of the second component, i.e., the lookahead.
    - If β is not ε, the lookahead has no effect in the item $[A \rightarrow \alpha \cdot \beta, a]$.
    - If β is ε, the item $[A \rightarrow \alpha \cdot \beta, a]$ can call for a reduction by $A \rightarrow \alpha$ if the next input symbol is *a*.

a, b, c: a terminal
w, x, y, z: strings of terminals
A,B, C: a nonterminal
W, X, Y, Z: a grammar symbol (a terminal or a nonterminal)
α, β, γ: strings of grammar symbols

Canonical LR(1) Items (Cont.)

- Formally, LR(1) item [A→α · β, a] is *valid* for a viable prefix γ if there is a derivation $S \overset{*}{\underset{rm}{\Rightarrow}} \delta A w \underset{rm}{\Rightarrow} \delta\alpha\beta w$, where
  - 1. $\gamma=\delta\alpha$, and
  - 2. Either *a* is the first symbol of *w*, or *w* is ε and *a* is $.
- E.g.,

  S → BB
  B → aB | b

  - There is a rightmost derivation $S \overset{*}{\underset{rm}{\Rightarrow}} aaBab \underset{rm}{\Rightarrow} aaaBab$
    - Item [B→a · B, a] is valid for a viable prefix γ=aaa by letting δ=aa, A=B, α=a, β=B, and w=ab.
  - There is another rightmost derivation $S \overset{*}{\underset{rm}{\Rightarrow}} BaB \underset{rm}{\Rightarrow} BaaB$
    - Item [B→a · B, $] is valid for prefix Baa by letting δ=Ba, A=B, α=a, β=B, and w=ε.

Constructing LR(1) Sets of Items

- **Algorithm**: Construction of the sets of LR(1) items.
- **INPUT**: An augmented grammar G'.
- **OUTPUT**: The sets of LR(1) items that are the sets of items valid for one or more viable prefixes of G'.
- **METHOD**: The procedures CLOSURE and GOTO and the main routin *items*

```
SetOfItems CLOSURE($I$) {
    repeat
        for (each iteam $[A \to \alpha \cdot B\beta, a]$ in $I$)
            for (each production $B \to \gamma$ in G')
                for ( each terminal $b$ in FIRST($\beta$a) )
                    add $[B \to \cdot \gamma, b]$ to set $I$;
    until no more items are added to $I$;
    return $I$;
}
```

```
SetOfItems GOTO($I$, $X$) {
    initialize $J$ to be the empty set;
    for (each iteam $[A \to \alpha \cdot X\beta, a]$ in $I$)
        add item $[A \to \alpha X \cdot \beta, \alpha]$ to set $J$;
    return CLOSURE($J$);
}
```

```
void items(G') {
    initialize $C$ = CLOSURE( { $[S' \to \cdot S, \$]$ } );
    repeat
        for (each set of items $I$ in $C$)
            for (each grammar symbol $X$)
                if (GOTO($I$, $X$) is not empty and not in $C$)
                    add GOTO($I$, $X$) to $C$;
    until no new sets of items are added to $C$;
}
```

Constructing LR(1) Sets of Items (Cont.)

- Why *b* must be in FIRST(β*a*)
  - Consider an item of the for [A→ $\alpha\cdot B\beta$, *a*] in the set of items *valid* for some viable prefix γ.
  - Then there is a rightmost derivation $S \overset{*}{\underset{rm}{\Rightarrow}} \delta A a x \underset{rm}{\Rightarrow} \delta\alpha B\beta a x$ where $\gamma=\delta\alpha$.
    - Suppose βax derives terminal string *by*.
    - For each production B→ η for some η, we can have derivation $S \overset{*}{\underset{rm}{\Rightarrow}} \gamma B b y \underset{rm}{\Rightarrow} \gamma \eta b y$
    - Thus, [B→ $\cdot\ \eta$, *b*] is valid for γ.
  - For *b*, there are two conditions:
    - 1. *b* is the first terminal derived from β.
    - 2. *b* is *a* if β derives ε. That is, $\beta ax \overset{*}{\underset{rm}{\Rightarrow}} by$
    - So that *b* must be any terminal in FIRST(βax) = FIRST(βa)

a, b, c: a terminal
w, x, y, z: strings of terminals
A,B, C: a nonterminal
W, X, Y, Z: a grammar symbol (terminal or nonterminal)
α, β, γ: strings of grammar symbols

An Example of LR(1) Sets of Items

S' → S
S → CC
C → cC | d

- Consider the grammar, we begin from CLOSURE{[S'→ · S, $]}
  - Step 1:
    - Watch item [S'→ · S, $] with the item [A→$\alpha$ • Bβ, *a*].
      A=S', α=ε, B=S, β=ε, and a = $.
    - Function CLOSURE tells us to add [B→ · γ, *b*] for each production B→γ and terminal *b* in FIRST(β*a*).
      Thus, B→γ must be S→CC. Since β=ε and a = $, so that *b* = FIRST($\beta$*a*) = $.
    - Therefore, we add [S→ · CC, $] to the closure.
  - Step 2:
    - Match [S→ · CC, $] against [A→$\alpha$ • Bβ, *a*].
      A=S, α=ε, B=C, β=C, and a = $.
    - Compute the closure by adding all items [C→ · γ, *b*] for b in FIRST(C$).
      Since b= FIRST(C$) = {*c*, *d*}, we add [C→ · cC, c], [C→ · cC, d], [C→ · d, c], and [C→ · d, d].
  - Step 3:
    - None of the new items has a nonterminal immediately to the right of the dot, so we have completed the first set of LR(i) items.

S'→ · S, $* *$I_0$

S → · CC, $
C → · cC, c/d
C → · d, c/d

An Example of LR(1) Sets of Items (Cont.)

S' → S
S → CC
C → cC | d

- Next, compute GOTO(I_0, *X*) for the various values of *X*.
  - For *X*=*S*, close the item [S'→S · , $] and no additional closure is possible because the dot is at the right end. Thus we have

S'→*S* · , $	I_1

  - For *X*=*C*, close [S→C · C, $] due to [S→ · CC, $] to add C-productions with second component $ to yield:

S→*C* · *C*, $	I_2
C → · *cC*, $ *C* → · *d*, $	

S'→ · *S*, $	I_0
S → · *CC*, $ *C* → · *cC*, *c*/*d* *C* → · *d*, *c*/*d*	

  - For *X*=*c*, close [C→c · C, c/d] to add C-productions with second component c/d to yield:

C→c · C, c/d	I_3
C → · *cC*, c/d *C* → · *d*, c/d	

  - For *X*=*d*, close [C→d · , c/d] to wind up:

C→d · , c/d	I_4

An Example of LR(1) Sets of Items (Cont.)

$S' \rightarrow S$
$S \rightarrow CC$
$C \rightarrow cC \mid d$

- GOTO(I_1, X) goes to no new sets.

  $S' \rightarrow S\cdot, \$$ I_1

- Compute GOTO(I_2, X) for the various values of X
  - For X=C, add $[S \rightarrow CC\cdot, \$]$ and no additional closure is possible.

    $S \rightarrow CC\cdot, \$$ I_5

  - For X=c, we take the closure of $[C \rightarrow c\cdot C, \$]$ to obtain:

    $C \rightarrow c\cdot C, \$$ I_6
    $C \rightarrow \cdot cC, \$$
    $C \rightarrow \cdot d, \$$

    I_6 differs from I_3 only in second components. In LR(0), these sets of LR(1) items will coincide to the same set of LR(0) items.

  - For X=d, we take GOTO(I_2, d)

    $C \rightarrow d\cdot, \$$ I_7

- Compute GOTO(I_3, X).
  - GOTO(I_3, c) and GOTO(I_3, d) are I_3 and I_4, respectively.
  - GOTO(I_3, C) is $C \rightarrow cC\cdot, c/d$ I_8
- GOTO(I_6, C) is $C \rightarrow cC\cdot, \$$ I_9

$S \rightarrow C\cdot C, \$$ I_2
$C \rightarrow \cdot cC, \$$
$C \rightarrow \cdot d, \$$

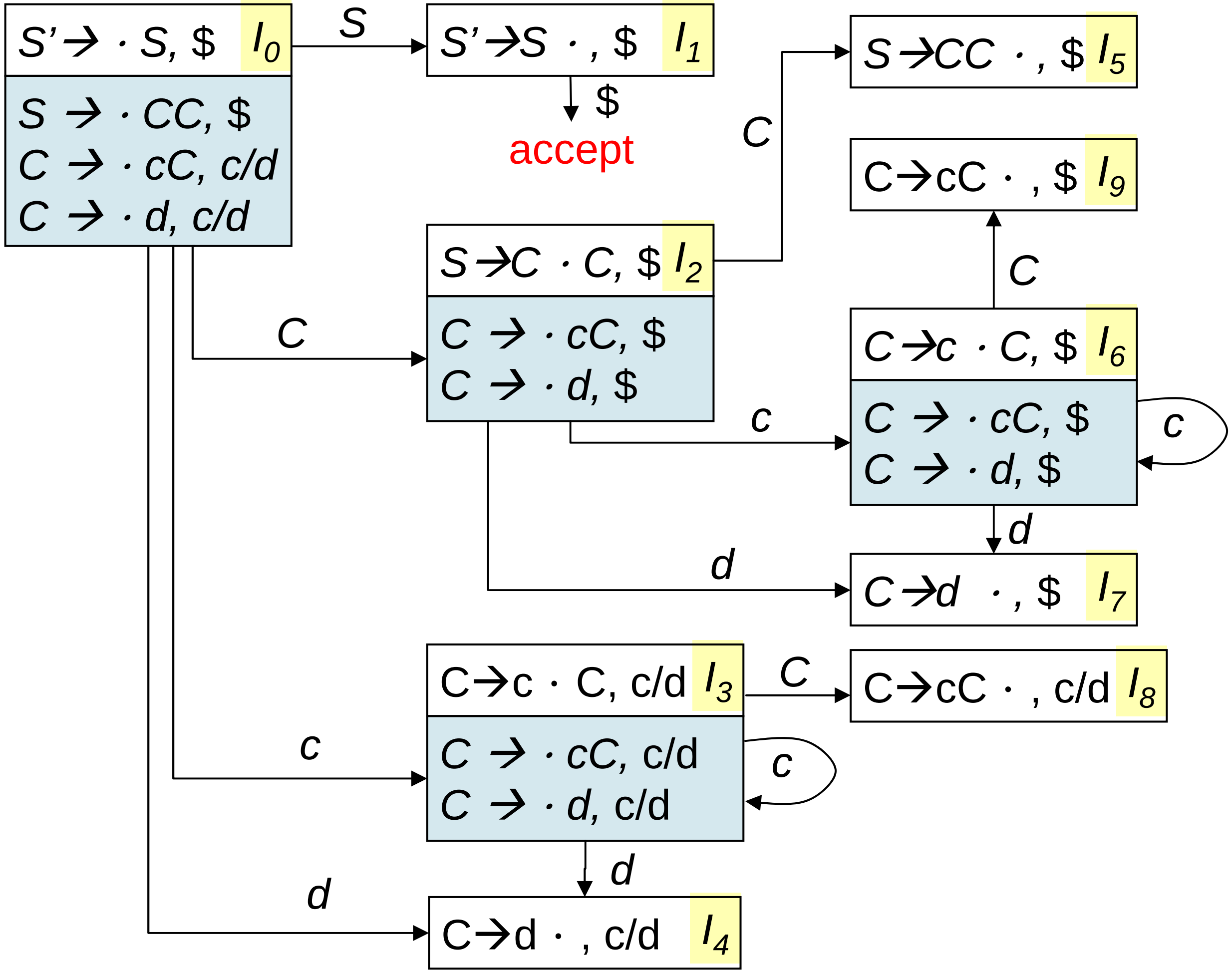
I_4, I_5, I_7, I_8, and I_9 have no GOTOs.

$C \rightarrow c\cdot C, c/d$ I_3
$C \rightarrow \cdot cC, c/d$
$C \rightarrow \cdot d, c/d$

$C \rightarrow d\cdot, c/d$ I_4

GOTO(I_6, c) and GOTO(I_6, d) are I_6 and I_7, respectively.

An Example of LR(1) Sets of Items (Cont.)



I_0: S'→ · S, $ | S → · CC, $ | C → · cC, c/d | C → · d, c/d

I_1: S'→S · , $

I_2: S→C · C, $ | C → · cC, $ | C → · d, $

I_3: C→c · C, c/d | C → · cC, c/d | C → · d, c/d

I_4: C→d · , c/d

I_5: S→CC · , $

I_6: C→c · C, $ | C → · cC, $ | C → · d, $

I_7: C→d · , $

I_8: C→cC · , c/d

I_9: C→cC · , $

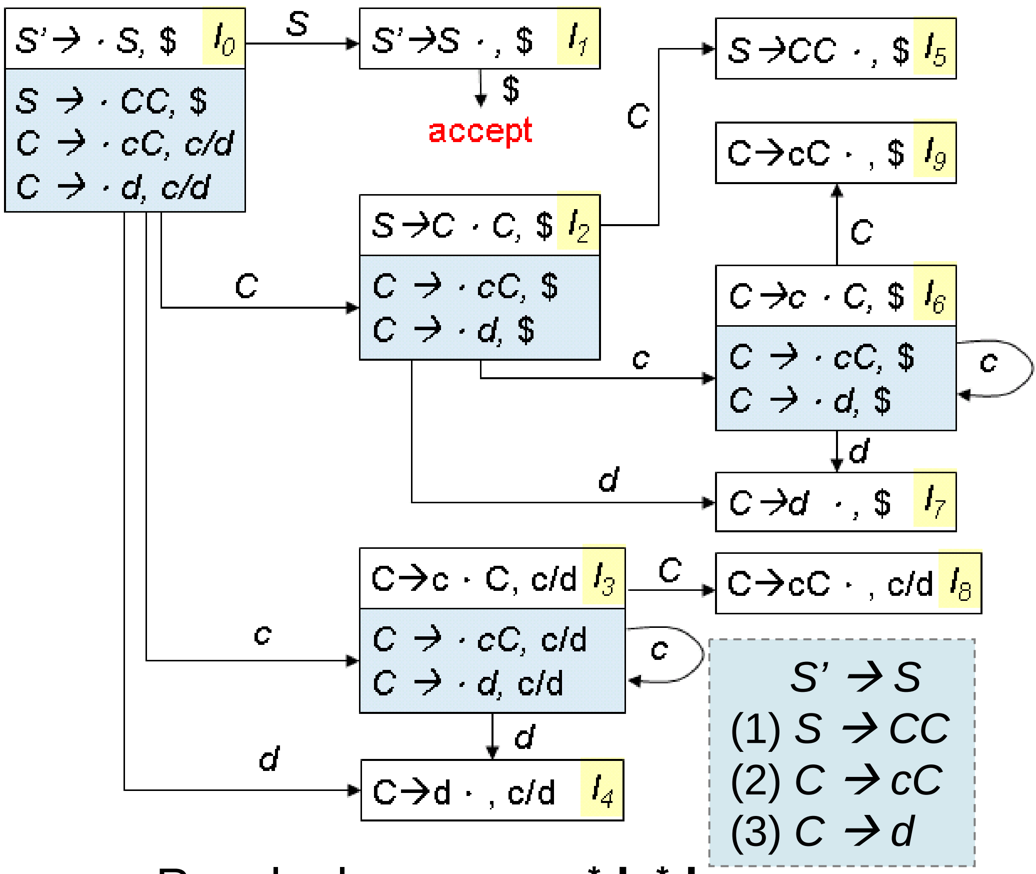
S' → S
S → CC
C → cC | d

Canonical LR(1) Parsing Table

- **Algorithm**: Construction of canonical-LR parsing tables. (Algorithm 4.56)
- **INPUT**: An augmented grammar G'.
- **OUTPUT**: The canonical-LR parsing table functions ACTION and GOTO for G'.
- **METHOD**:
  - 1. Construct C' = $\{I_0, I_1, \ldots, I_n\}$, the collection of sets of LR(1) items for G'.
  - 2. State *i* of the parser is constructed from I_i. The parsing actions for state *i* are determined as follows:
    - (a) If [A→$\alpha \cdot a\beta$, *b*] is in I_i and GOTO(I_i, *a*)=I_j, then set ACTION[*i*, *a*] to "shift *j*", where *a* must be a terminal.
    - (b) If [A→$\alpha \cdot$, *a*] is in I_i, and A≠S', then set ACTION[*i*, *a*] to "reduce A→ α"
    - (c) If [S'→S · , $] is in I_i, then set ACTION[*i*, $] to "accept".
    - If any conflicting actions result from the above rules, the grammar is not LR(1) and the algorithm fails to produce a parser for it.
  - 3. If GOTO(I_i, *A*)=I_j, then GOTO[i,A]=*j*.
  - 4. All entries not defined by rules (2) and (3) are made "error."
  - 5. The initial state of the parser is the one constructed from the set of items containing [S'→ · S, $]

Canonical LR(1) Parsing Table (Cont.)

STATE	ACTION			GOTO	
	c	d	$	S	C
0	s3	s4		1	2
1			acc		
2	s6	s7			5
3	s3	s4			8
4	r3	r3			
5			r1		
6	s6	s7			9
7			r3		
8	r2	r2			
9			r2		



I_0: S'→ · S, $ / S → · CC, $ / C → · cC, c/d / C → · d, c/d

I_1: S'→S · , $

I_2: S→C · C, $ / C → · cC, $ / C → · d, $

I_3: C→c · C, c/d / C → · cC, c/d / C → · d, c/d

I_4: C→d · , c/d

I_5: S→CC · , $

I_6: C→c · C, $ / C → · cC, $ / C → · d, $

I_7: C→d · , $

I_8: C→cC · , c/d

I_9: C→cC · , $

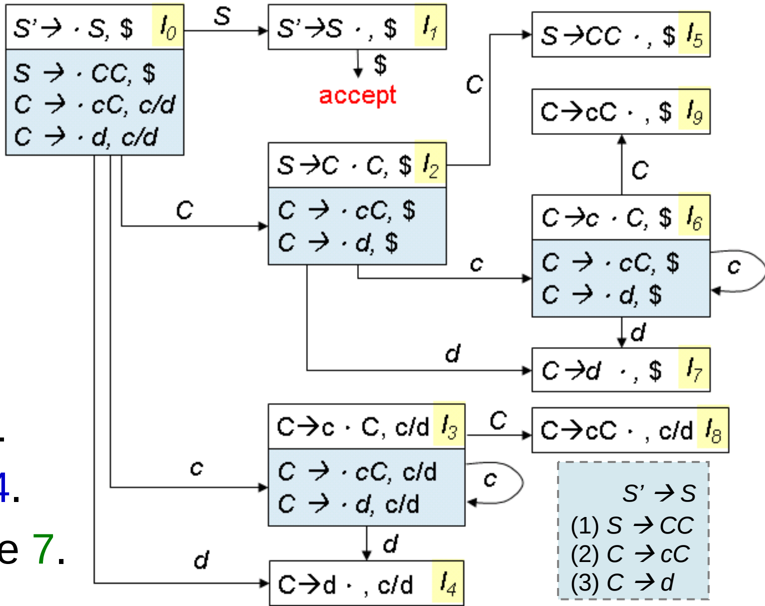
S' → S
(1) S → CC
(2) C → cC
(3) C → d

Regular language: **c*dc*d**

Lookahead-LR (LALR) Parser

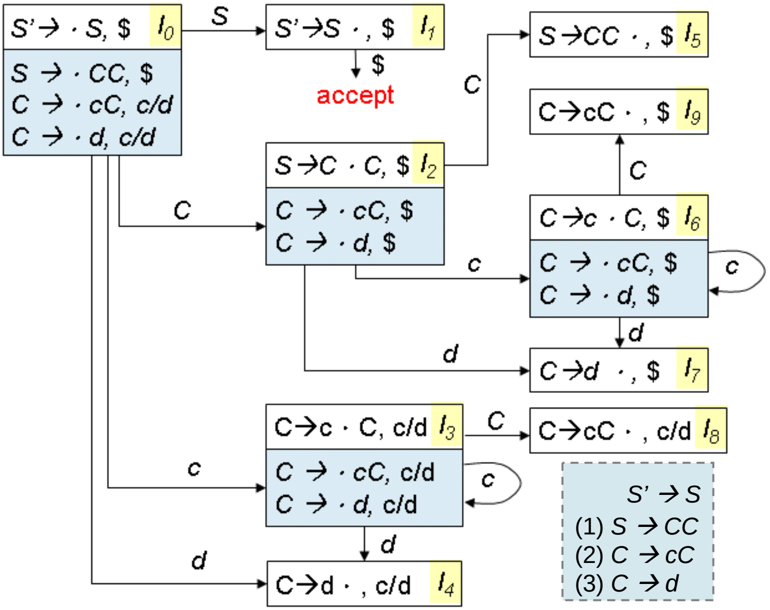
- LALR and SLR tables for a grammar always have the same number of states.
  - E.g., typically several hundred states for a language like C.
- The canonical LR(1) (or simply LR) table would typically have several thousand states for the same size language.
  - Different states in LR parser might consists of the same items (called cores) with different lookheads.
    - E.g., I_4 and I_7, I_3 and I_6 , I_8 and I_9
  - For the regular language **c*dc*d**,
    - When reading *cc…cdcc…cd*
      - The parser shifts the first group of *c*'s and their following *d* to enter state 4, and then reduce *C →d*.
      - The parser enters state 7 after reading the second *d*.
    - If input is *ccd*, declare error after entering state 4.
    - If input is *cdcdc*, declare error after entering state 7.

Regular language: **c*dc*d**



LALR Parser (Cont.)

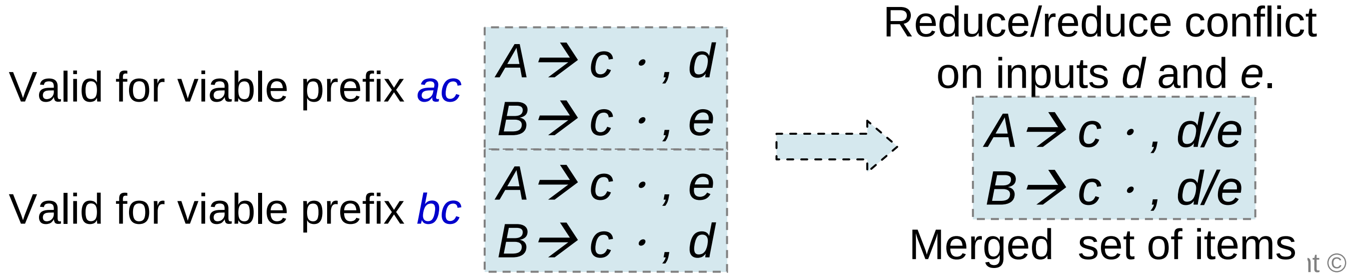
- Revise the parser for the regular language **c*dc*d**
  - I_4 and I_7 → replaced by I_{47} $[C \rightarrow d\ \cdot\ ,\ c/d/\$]$
  - I_3 and I_6 → replaced by I_{36}
    $[C \rightarrow c\ \cdot\ C,\ c/d/\$]$
    $[C \rightarrow\ \cdot\ cC,\ c/d/\$]$
    $[C \rightarrow\ \cdot\ d,\ c/d/\$]\ \}$
  - I_8 and I_9 → replaced by I_{89} $[C \rightarrow cC\ \cdot\ ,\ c/d/\$]$
- The revised parser might reduce C→c where the original parser would declare error. But the error will eventually be caught before any more input symbols are shifted.



Regular language: **c*dc*d**

Reduce/Reduce Conflict

- A merger to merge states of LR parsers
  - Do not produce shift/reduce conflicts.
    - E.g., Suppose in the union, there is a conflict due to the item $[A \rightarrow \alpha\cdot, a]$ for reduce and $[B \rightarrow \beta \cdot a\gamma, b]$ for shift.
      - Some set of items from which the union was formed has item $[A \rightarrow \alpha\cdot, a]$
      - The cores of all states for the same union are the same, it must have an item $[B \rightarrow \beta \cdot a\gamma, c]$ for some *c*.
        → the shift/reduce conflict exists before the union/merging.
  - Might produce a reduce/reduce conflict.
    - E.g., This grammar generates *acd*, *ace*, *bcd*, and *bce*.



Valid for viable prefix *ac*

$A \rightarrow c\cdot, d$
$B \rightarrow c\cdot, e$

Valid for viable prefix *bc*

$A \rightarrow c\cdot, e$
$B \rightarrow c\cdot, d$

Reduce/reduce conflict on inputs *d* and *e*.

$A \rightarrow c\cdot, d/e$
$B \rightarrow c\cdot, d/e$

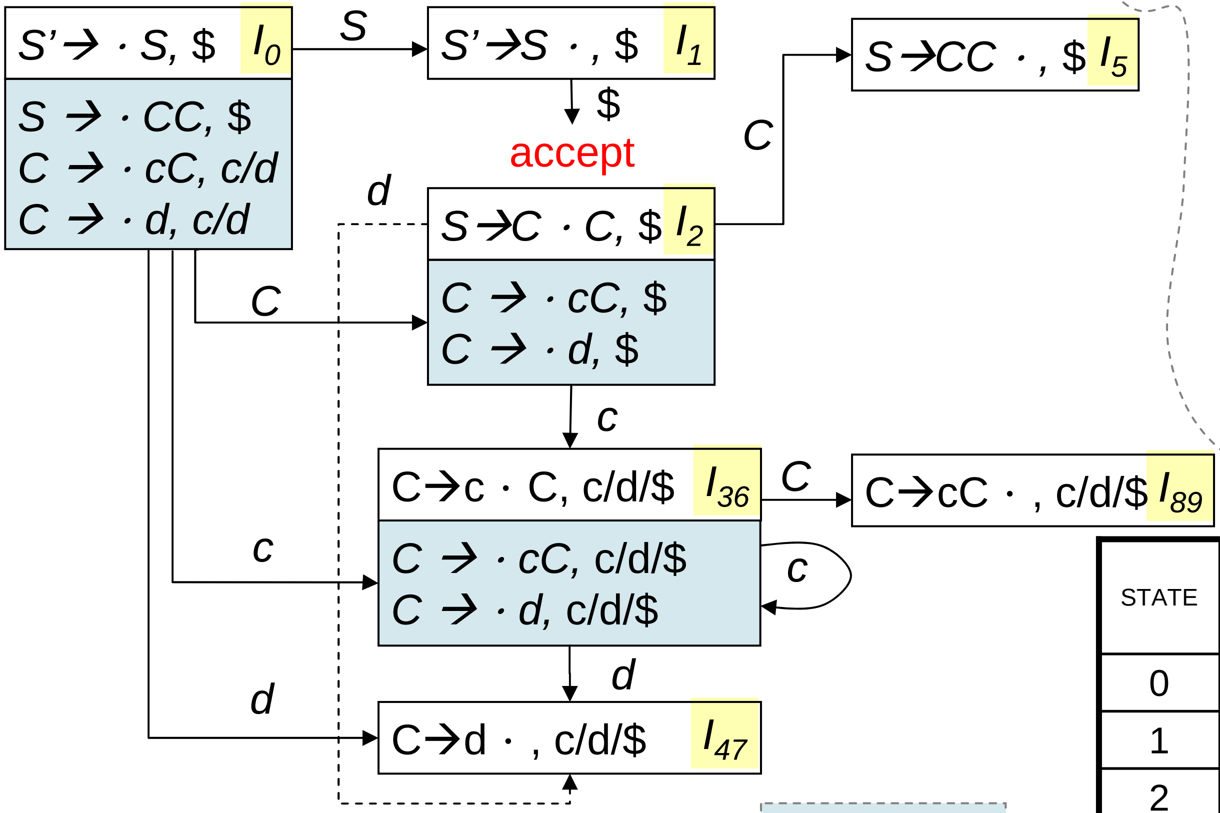
Merged set of items

$S' \rightarrow S$
$S \rightarrow aAd \mid bBd\ aBe \mid bAe$
$A \rightarrow c$
$B \rightarrow c$

LALR(1) Table Construction

- **Algorithm**: An easy, but space-consuming LALR table construction.
- **INPUT**: An augmented grammar G’.
- **OUTPUT**: The LALR parsing-table functions ACTION and GOTO for G’
- **METHOD**:
  - 1. Construct C= {I_0, I_1, …, I_n}, the collection of sets of LR(1) items.
  - 2. For each core present along the set of LR(1) items, find all sets having that core, and replace these sets by their union.
  - 3. Let C’= {J_0, J_1, …, J_m} be the resulting sets of LR(1) items. The parsing actions for state *i* are constructed from J_i in the same manner as in Algorithm 4.56. If there is a parsing action conflict, the algorithm fails to produce an LALR(1) parser for the grammar.
  - 4. If J is the union of one or more sets of LR(1) items, $J= I_1 \cap I_2 \cap \ldots \cap I_k$, then the cores of $GOTO(I_1, X), GOTO(I_2, X), \ldots, GOTO(I_k, X)$ are the same. Let K be the union of all sets of items. Then $GOTO(J,X) = K$.

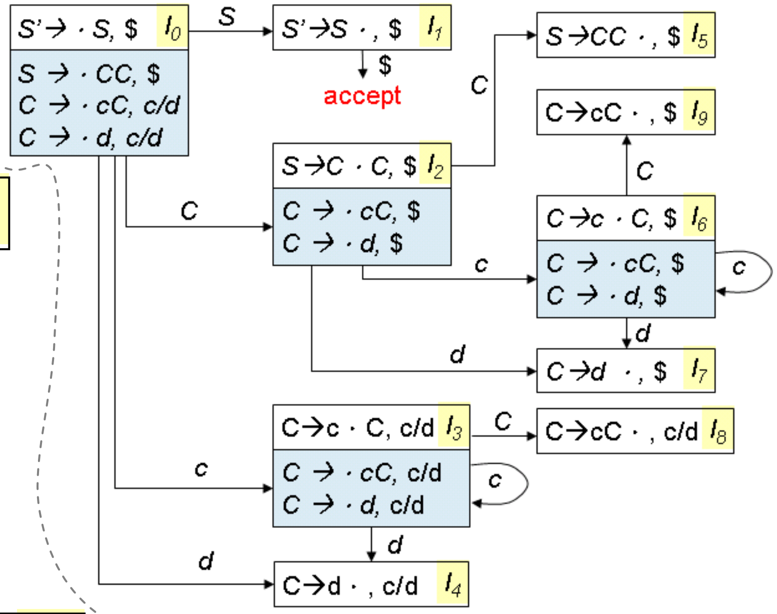
LALR(1) Table Construction (Cont.)



LALR(1) automation

Regular language: **c*dc*d**

S' → S
(1) S → CC
(2) C → cC
(3) C → d

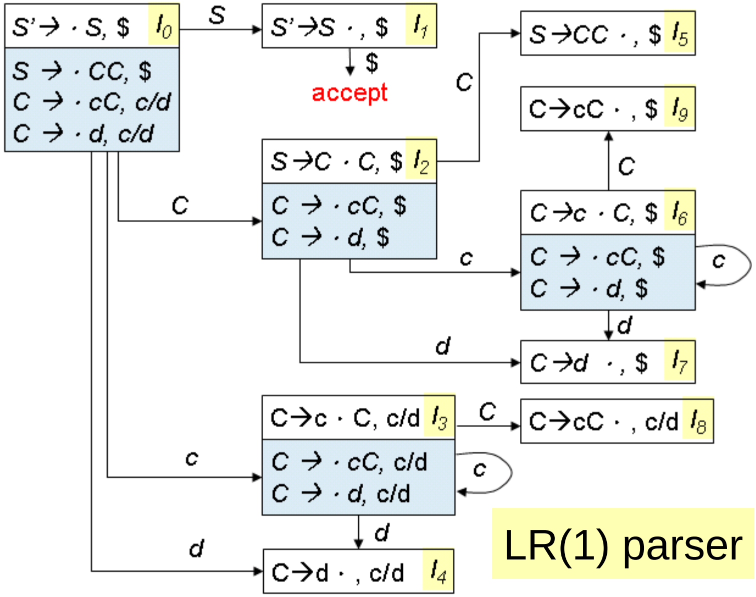


LR(1) automation

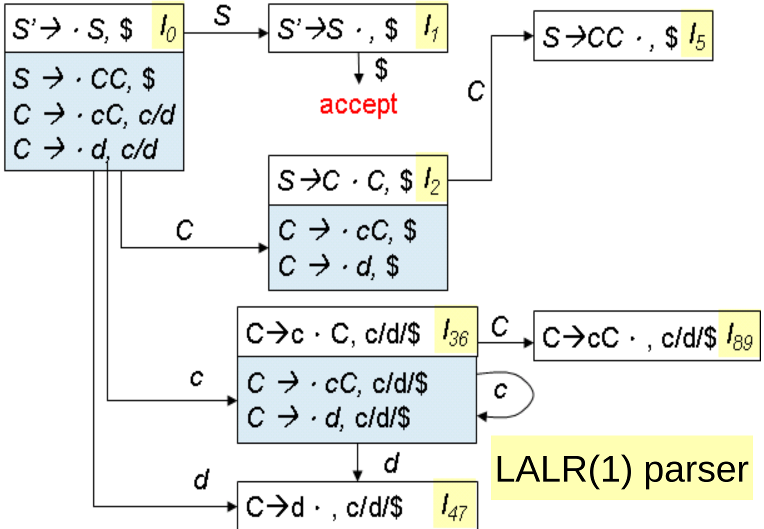
STATE	ACTION			GOTO	
	c	d	$	S	C
0	s36	s47		1	2
1			acc		
2	s36	s47			5
36	s36	s47			89
47	r3	r3	r3		
5			r1		
89	r2	r2	r2		

Erroneous Input

- The LALR parser may proceed to do some reductions after the LR parser has declared an error, but it never shifts another symbol after the LR parser declares an error.
- E.g., on input *ccd* followed by *$*,
  - The LR parser puts 0 3 3 4 to the stack, and discover an error on $.
  - The LALR parser make the corresponding moves:
    - Put 0 36 36 47 on the stack. (prefix: ccd)
    - State 47 on input $ has action reduce C→d. The stack is changed to 0 36 36 89. (prefix: ccC)
    - State 89 on input $ has reduce C→cC. The stack becomes 0 36 89. (prefix: cC)
    - State 89 on input $ has reduce C→cC. The stack becomes 0 2. (prefix: C)
    - Finally, state 2 has action error on input $.



LR(1) parser



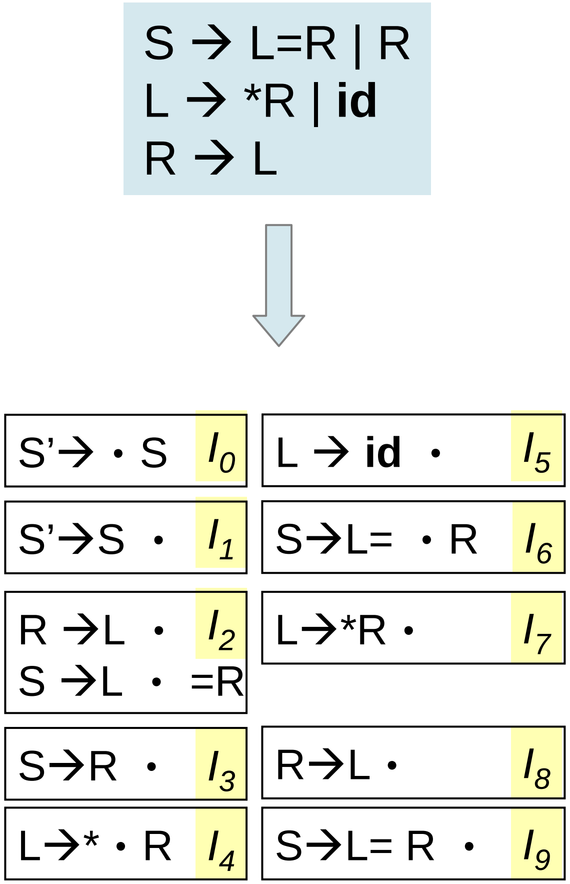
LALR(1) parser

Efficient Construction of LALR Parsing Tables

- Ways to avoid constructing the full collection of sets of LR(1) items during LALR(1) table construction:
  - 1. Represent any set of LR(0) or LR(1) items by its kernel items.
  - 2. Construct the LALR(1) kernel items (or “kernels” for short) from the LR(0) kernel items by a process of ***propagation*** and ***spontaneous*** generation of lookaheads.
  - 3. If we have the LALR(1) kernel items, we can generate the LALR(1) parsing table by closing each kernel item.

LALR(1) Kernels from LR(0) Kernels

- Attach proper lookaheds to the LR(0) kernels to create the kernels of the sets of LALR(1) items.
  - There are two ways a lookahead *b* can get attached to an LR(0) item $B\rightarrow\gamma\cdot\delta$ in some set of LALR(1) items *J*.
    - With set of items *I* with a kernel item $[A\rightarrow\alpha\cdot\beta, a]$, If J = GOTO(*I*, *X*) = GOTO(CLOSURE($[A\rightarrow\alpha\cdot\beta, a]$), X) contains $[B\rightarrow\gamma\cdot\delta, b]$ regardless of *a*
      - Lookahead *b* is generated spontaneously for $B\rightarrow\gamma\cdot\delta$
    - With set of items *I* with a kernel item $[A\rightarrow\alpha\cdot\beta, b]$, If J = GOTO(*I*, *X*) = GOTO(CLOSURE($[A\rightarrow\alpha\cdot\beta, b]$), X) contains $[B\rightarrow\gamma\cdot\delta, b]$.
      - Lookahead *b* is propagated from $A\rightarrow\alpha\cdot\beta$ in the kernel of *I* to $B\rightarrow\gamma\cdot\delta$ in the kernel of *J*.
      - **Either all lookaheads propagate from one item to another, or none do.**
    - Lookhead $ is generated spontaneously for the item $S'\rightarrow\cdot S$ in the initial set of items.
  - **The only kernel items in *J* must have *X* immediately to the left of the dot. That is, they must be of the form $B\rightarrow\gamma X\cdot\delta$**



S → L=R | R
L → *R | **id**
R → L

S'→ · S I_0	L → **id** · I_5
S'→S · I_1	S→L= · R I_6
R →L · I_2 S →L · =R	L→*R · I_7
S→R · I_3	R→L · I_8
L→* · R I_4	S→L= R · I_9

Kernels of the sets of LR(0) items

Lookahead Determination

- **Algorithm**: Determining lookaheads. (Algorithm 4.62)
- **INPUT**: The kernel *K* of a set of LR(0) items *I* and a grammar symbol *X*.
- **OUTPUT**:
  - The lookaheads spontaneously generated by items in *I* for kernel items in *GOTO(I, X)*.
  - The lookaheads propagated to kernel items in *GOTO(I,X)* from the items in *I*.
- **METHOD**:

# represnets any symbol

```
for ( each item $A \rightarrow \alpha \cdot \beta$ in $K$) {
    $J$ := CLOSURE( {[$A \rightarrow \alpha \cdot \beta$, #]} );
    if ( [$B \rightarrow \gamma \cdot X\delta$, $a$] is in $J$, and $a$ is not # )
        conclude that lookahead $a$ is generated spontaneously from item $B \rightarrow \gamma X \cdot \delta$ in $GOTO(I, X)$;
    if ( [$B \rightarrow \gamma \cdot X\delta$, #] is in $J$)
        conclude that lookahead propagate from from $A \rightarrow \alpha \cdot \beta$ in $I$ to $B \rightarrow \gamma X \cdot \delta$ in $GOTO(I, X)$;
}
```

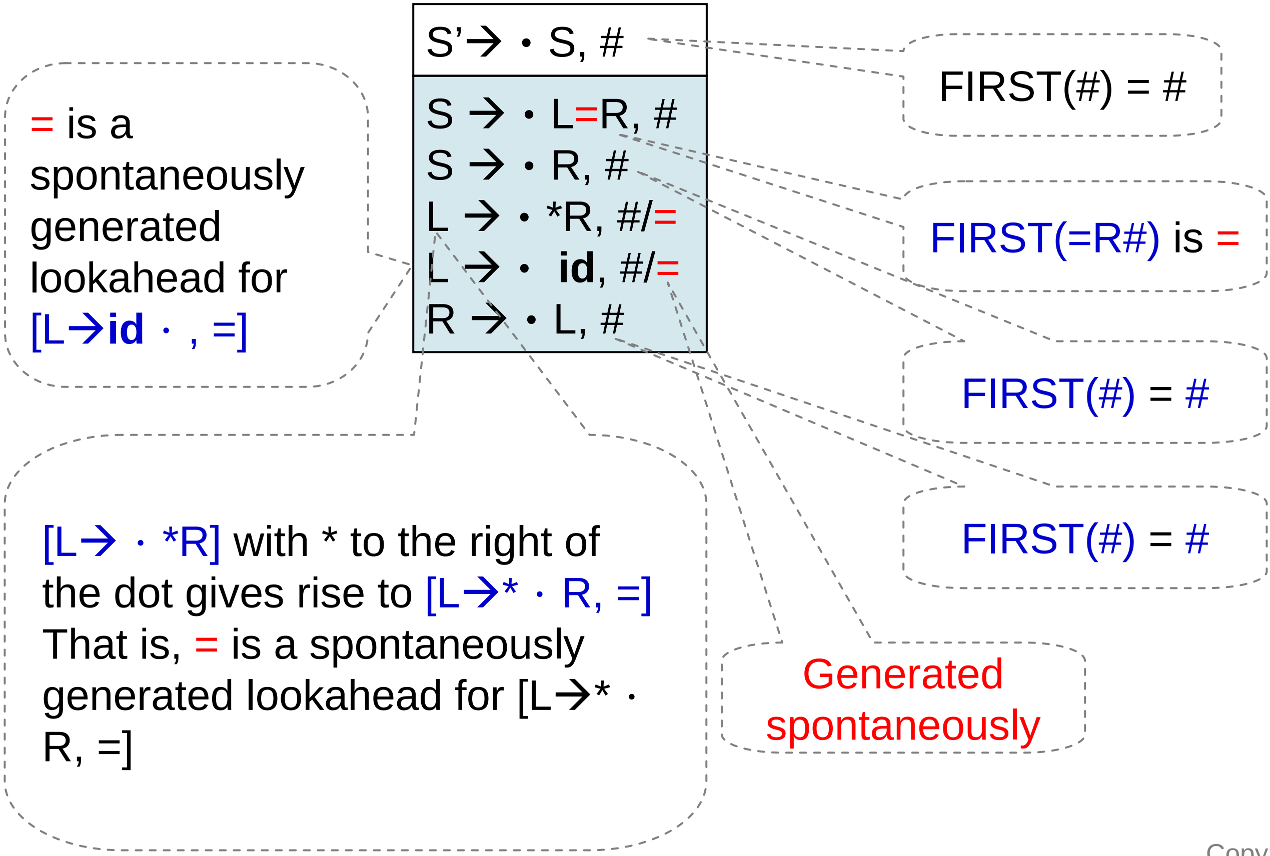
LALR(1) Collection of Sets of Items.

- **Algorithm**: Efficient computation of the kernels of the LALR(1) collection of sets of items. (Algorithm 4.63)
- **INPUT**: An augmented grammar G’.
- **OUTPUT**: The kernels of the LALR(1) collections of sets of items for G’.
- **METHOD**:
  - 1. Construct the kernels of the sets of LR(0) items for G.
  - 2. Apply Algorithm 4.62 to the kernel of each set of LR(0) items and grammar symbol X to determine
    - Which lookaheads are spontaneously generated for kernel items in *GOTO(I, X).*
    - From which items in *I*, lookaheads are propagated to kernel items in *GOTO(I, X).*
  - 3. Initialize a table that gives the associated lookaheads. Initially, each item has associated with those lookaheads that we determined in step (2) and generated spontaneously.
  - 4. Make repeated passes over the kernel items in all sets.
    - When we visit an item *i*, we look up the kernel items for which *i* propagates its lookheads by using information tabulated in step (2).
    - The current set of lookaheads for i is added.
    - We continue making passes over the kernel items until no more new lookaheads are propagrated.

Kernels of the LALR(1) Items

- E.g., Initially, compute CLOSURE({[S'→ · S, #]})

S → L=R | R
L → *R | **id**
R → L



S'→ · S, #

S → · L=R, #
S → · R, #
L → · *R, #/=
L → · **id**, #/=
R → · L, #

FIRST(#) = #

FIRST(=R#) is =

FIRST(#) = #

FIRST(#) = #

= is a spontaneously generated lookahead for [L→**id** · , =]

[L→ · *R] with * to the right of the dot gives rise to [L→* · R, =] That is, = is a spontaneously generated lookahead for [L→* · R, =]

Generated spontaneously

As # is a lookahead for all six items in the closure, we determine that the item S'→ · S in I_0 propagates lookaheads to the following six items:

I_1: S'→S · , #

I_2: S →L · =R, #
I_2: S →R · , #
I_3: L →* · R, #
I_4: L → **id** · , #
I_5: R →L · , #

Kernels of the LALR(1) Items (Cont.)

FROM	TO
I_0: S'→ · S (, #)	**I_1: S'→S · (, #)** I_2: S→L · =R (, #) I_2: R→L · (, #) I_3: S→R · (, #) I_4: L→* · R (, #/=) I_5: L→id · (, #/=)
I_2: S→L · =R (, #)	**I_6: S→L= · R (, #)**
I_4: L→* · R (, #)	I_4: L→* · R (, #) I_5: L→id · (, #) **I_7: L→*R · (, #)** I_8: R→L · (, #)
I_6: S→L= · R (, #)	I_4: L→* · R (, #) I_5: L→**id** · (, #) I_8: R→L · (, #) **I_9: S→L=R · (, #)**

Propagation of lookaheads

Dot reaches the end of the production: no further moves or propagations.

S → L=R | R
L → *R | **id**
R → L

S'→ · S, #

S → · L=R, #
S → · R, #
L → · *R, #/=
L → · **id**, #/=
R → · L, #

CLOSURE({[S'→ · S, #]})

S'→ · S	I_0	L → **id** ·	I_5
S'→S ·	I_1	S→L= · R	I_6
R →L · S →L · =R	I_2	L→*R ·	I_7
S→R ·	I_3	R→L ·	I_8
L→* · R	I_4	S→L= R ·	I_9

Kernels of the sets of LR(0) items

Kernels of the LALR(1) Items (Cont.)

FROM	TO
I_0: S'→ · S (, \$)	**I_1: S'→S · (, \$)** I_2: S→L · =R (, \$) I_2: R→L · (, \$) I_3: S→R · (, \$) I_4: L→* · R (, \$/=) I_5: L→id · (, \$/=)
I_2: S→L · =R (, \$)	**I_6: S→L= · R (, \$)**
I_4: L→* · R (, \$/=)	I_4: L→* · R (, \$/=) I_5: L→id · (, \$/=) **I_7: L→*R · (, \$/=)** I_8: R→L · (, \$/=)
I_6: S→L= · R (, \$)	I_4: L→* · R (, \$) I_5: L→**id** · (, \$) I_8: R→L · (, \$) **I_9: S→L=R · (, \$)**

Propagation of lookaheads

SET	ITEM	LOOKAHEADS			
		INIT	PASS 1	PASS 2	PASS 3
I_0:	S'→ · S	\$	\$	\$	\$
I_1:	S'→S ·		\$	\$	\$
I_2:	S→L · =R R→L		\$	\$	\$
I_3:	S→R ·		\$	\$	\$
I_4:	L→* · R	=	=/\$	=/\$	=/\$
I_5:	L→id ·	=	=/\$	=/\$	=/\$
I_6:	S→L= · R			\$	\$
I_7:	L→*R ·		=	=/\$	=/\$
I_8:	R→L ·		=	=/\$	=/\$
I_9:	S→L=R ·				\$

Computation of lookaheads

Compaction of LR Parsing Tables

- A typical programming language grammar with 50 to 100 terminals and 100 productions may have an LALR parsing table with
  - Several hundred states
  - 20,000 entries in action functions
- Compaction
  - Compaction to the ACTION field:
    - Eliminate identical action entries in different states.
      - E.g., Create a pointer for each state into a one-dimensional array. Pointers for states with the same actions point to the same location.
    - Further space efficiency can be achieved by creating a list of actions with (terminal-symbol, action) pairs.
  - Compaction to the GOTO field
    - Few states have transitions on nonterminals.
    - For each nonterminal A, each pair on the list for A is of the form: GOTO[*currentState*, A] = *nextState*
    - For more space reduction, replace each error entry by the most common non-error entry in its column because the error entries in the goto table are never consulted.

ACTION Table Compaction

- Frequent actions for a state is places at the end of the list.
- “Any” means that if the current symbol has not been found so far on the list, we should do that action no matter what input is.
- The error will be detected later before a shift.

SYMBOL	ACTION
id	s5
(	s4
any	error

States 0, 4, 6, 7

SYMBOL	ACTION
+	s6
$	acc
any	error

State 1

SYMBOL	ACTION
*	s7
any	r2

State 2

SYMBOL	ACTION
any	r4

State 3

SYMBOL	ACTION
any	r6

State 5

SYMBOL	ACTION
any	r3

State 10

SYMBOL	ACTION
any	r5

State 11

SYMBOL	ACTION
+	s6
)	s11
any	error

State 8

SYMBOL	ACTION
*	s7
any	r1

State 9

STATE	ACTION						GOTO		
	id	+	*	(	)	$	E	T	F
0	s5			s4			1	2	3
1		s6				acc			
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			

GOTO Table Compaction

- The error entries in the goto table are never consulted.

currentState	**nextState**
7	10
any	3

Colum F

currentState	**nextState**
6	9
any	2

Colum T

currentState	**nextState**
4	8
any	1

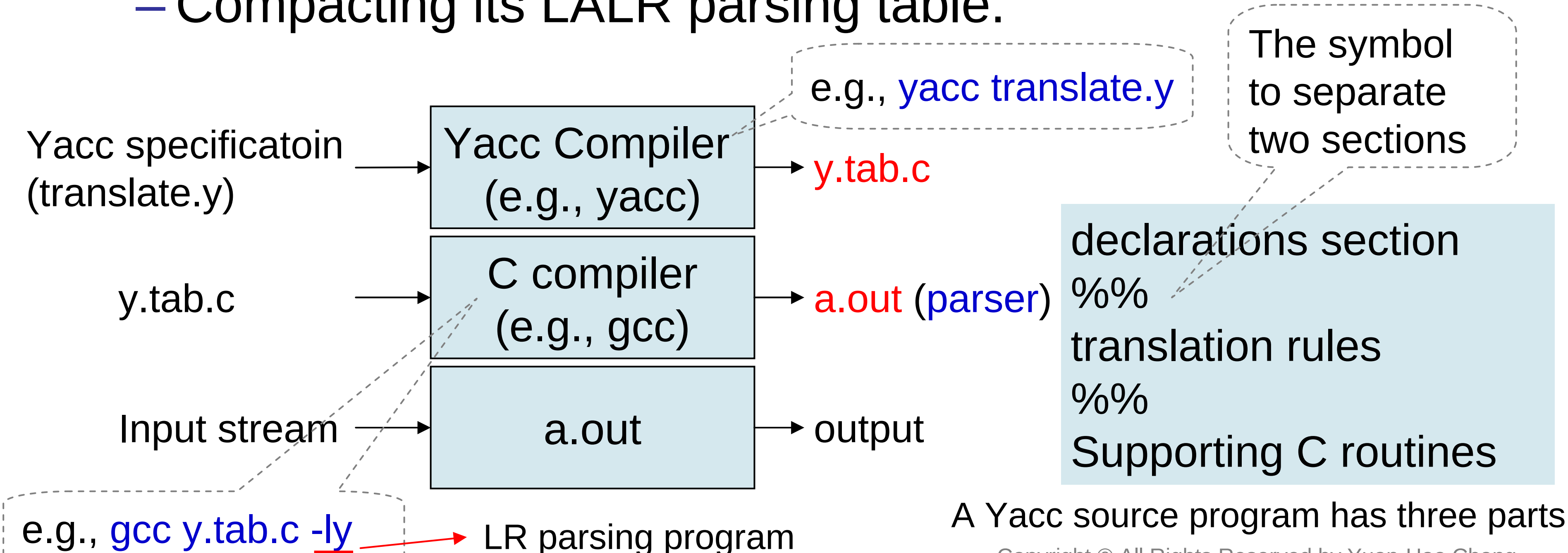
Colum E

STATE	ACTION						GOTO		
	id	+	*	(	)	$	E	T	F
0	s5			s4			1	2	3
1		s6				acc			
2		r2	s7		r2	r2			
3		r4	r4		r4	r4			
4	s5			s4			8	2	3
5		r6	r6		r6	r6			
6	s5			s4				9	3
7	s5			s4					10
8		s6			s11				
9		r1	s7		r1	r1			
10		r3	r3		r3	r3			
11		r5	r5		r5	r5			

Parser Generators

Yacc

- Yacc stands for “yet another compiler-compiler.”
  - Created by S.C. Johnson in the early 1970s.
  - Using the LALR method outlined in Algorithm 4.63.
  - Compacting its LALR parsing table.



Yacc specificatoin (translate.y) → Yacc Compiler (e.g., yacc) → y.tab.c

e.g., yacc translate.y

y.tab.c → C compiler (e.g., gcc) → a.out (parser)

e.g., gcc y.tab.c -ly → LR parsing program

Input stream → a.out → output

The symbol to separate two sections

declarations section
%%
translation rules
%%
Supporting C routines

A Yacc source program has three parts.

Simple Desk Calculator

- Construct a simple desk calculator that reads an arithmetic expression, evaluates it, and then prints its numeric value with the following grammar:

$E \rightarrow E + T \mid T$
$T \rightarrow T * F \mid F$
$F \rightarrow (E) \mid \textbf{digit}$

A single digit between 0 and 9

- If Lex is used to create the lexical analyzer that passes token to the Yacc parser, then these token declarations are also made available to the lexical analyzer generated by Lex.

Get a symbol or a token name with its attribute value.

Declarations section

Ordinary C declarations

Grammar tokens

Start symbol

Translation rules

Supporting C functions

```
%{
#include <stdio.h>
#include <ctype.h>
%}
%token DIGIT
%%
line        : expr '\n'    { printf("%d\n", $1); }
            ;
expr        : expr '+' term { $$ = $1 + $3; }
            | term
            ;
term        : term '*' factor { $$ = $1 * $3; }
            | factor
            ;
factor      : '(' expr ')'   { $$ = $2; }
            | DIGIT
            ;
%%
yylex(){
    int c;
    c = getchar();
    if (isdigit(c)) {
        yylval = c-'0';
        return DIGIT;
    }
    return c;
}
```

If the character is a digit, the value of the digit is stored in yylval, and the token name DIGIT is returned. Otherwise, the character itself is returned as the token name.

Translation Rules

- Each rule consists of a grammar production and the associated semantic action. In Yacc,
  - Unquoted strings of letters and digits not declared to be tokens are taken to be nonterminals.
  - A quoted single character ‘c’ is
    - Taken to be the terminal symbol ‘c’, or
    - Taken to be the integer code for the token represented by that character.

Grammar productions: <head> → <body>$_1$ | <body>$_2$ | … | <body>$_n$

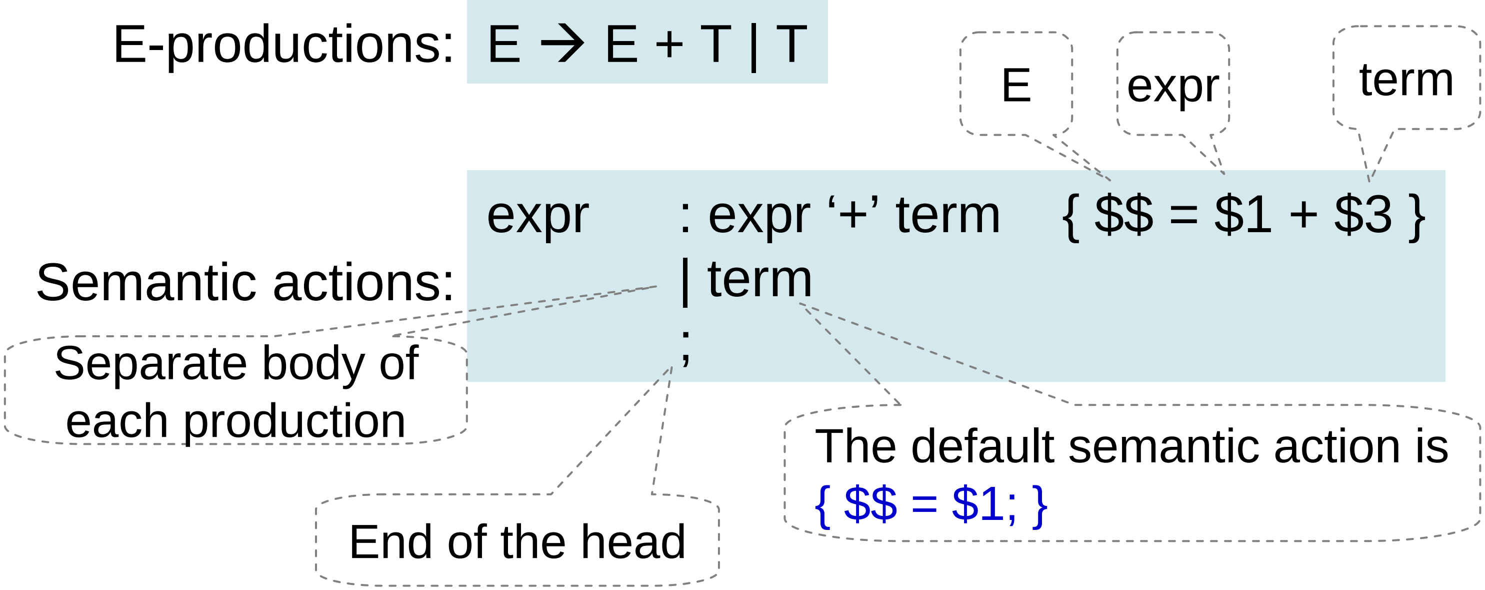
↓

Yacc productions:

<head> : <body>$_1$ { <semantic action>$_1$ }
| <body>$_2$ { <semantic action>$_2$ }
…
| <body>$_n$ { <semantic action>$_n$ }
;

Semantic Actions

- A Yacc semantic action is a sequence of C statements.
  - $$: refer to the attribute value associated with the nonterminal of the head.
  - $*i*: refer to the value associated with the *i*th grammar symbol of the body.



E-productions: E → E + T | T

Semantic actions:

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

The default semantic action is { $$ = $1; }

Semantic Actions (Cont.)

- Start symbol:

```
line      : expr '\n'        { printf("%d\n", $1); }
```

  - An input to the desk calculator is to be an expression followed by a newline character.
  - The semantic action associated with this production prints the decimal value of the expression followed by a newline character.

Supporting C-Routines

- A lexical analyzer by the name yylex() must be provided.
  - Using Lex to produce yylex() is a common choice.
  - The lexical analyzer yylex() produces tokens consisting of a token name and its associated attributed value.
    - The attributed value associated with a token is communicated to the parser through the Yacc-defined variable yylval.
  - If a token name such as DIGIT is returned, the token name must be declared in the first section of the Yacc specification.

Using Yacc with Ambiguous Grammars

- Yacc reports the number of parsing-action conflicts that are generated.
  - Invoke Yacc with a -v option to generate an additional file y.output that contains
    - 1. The kernels of the sets of items found for the grammar.
    - 2. A description of the parsing action conflicts generated by the LALR algorithm.
    - 3. A readable representation of the LR parsing table showing how the parsing actions conflicts were resolved.
- Yacc resolves all parsing action conflicts using two rules:
  - 1. A reduce/reduce conflict is resolved by choosing the conflicting production listed first in the Yacc specification.
  - 2. A shift/reduce conflict is resolved in favor of shift.
    - The dangling-else ambiguity problem can be resolved.

Advanced Desk Calculator

- Advanced desk calculator
  - Allow to evaluate a sequence of expressions, one to a line.
  - Allow blank lines between expressions.

```
lines      : lines expr '\n'   { printf("%g\n", $2); }
           | lines '\n'
           | /* empty */
           ;
```

ε

  - Enlarge the class of expressions
    - To include numbers instead of single digits and
    - To include the arithmetic operators +, - (both binary and unary), *, and /.

E→ E+E | E-E | E*E | E/E | -E | number (Ambiguous grammar)

The LALR algorithm will generate parsing-action conflicts.

Advanced Desk Calculator (Cont.)

```
%{
#include <stdio.h>
#include <ctype.h>
#define YYSTYPE double /* double type for Yacc stack */
%}
%token NUMBER

%left '+' '-'
%left '*' '/'
%right UMINUS
%%

lines       : lines expr '\n'         { printf("%g\n", $2); }
            | lines '\n'
            | /* empty */
            ;
expr        : expr '+' expr           { $$ = $1 + $3; }
            | expr '-' expr           { $$ = $1 - $3; }
            | expr '*' expr           { $$ = $1 * $3; }
            | expr '/' expr           { $$ = $1 / $3; }
            | '(' expr ')'            { $$ = $2; }
            | '-' expr %prec UMINUS {$$ = -$2; }
            | NUMBER
            ;
```

Make + and – the same precedence and left associative

Lowest priority first

Highest precedence

Force the production to be the highest precedence

```
%%
yylex(){
    int c;
    while( (c = getchar()) == ' ');
    if ( c == '.' || isdigit(c) ) {
        ungetc(c, stdin);
        scanf("%lf", &yylval);
        return NUMBER;
    }
    return c;
}
```

Skip spaces

Push the character back

Get an integer or floating value from input

Precedence and Associativity

- Tokens have higher priority if they are listed later.
- Make + and – be of the same precedence and be left associative
  - E.g., %left '+' ‘-'
- Declare an operator to be right associative:
  - E.g., %left ‘^'
- Force an operator to be a nonassociative binary operator (i.e., two occurrences of the operator cannot be combined at all)
  - E.g., %nonassoc ‘<'

Precedence and Associativity (Cont.)

- Each production or terminal involved in a shift/reduce conflict is attached with a precedence and associativity.
  - If Yacc needs to choose between shifting input symbol *a* and reducing by production $A \rightarrow \alpha$, it reduces
    - If the precedence of the production is greater than that of *a*, or
    - If the precedences are the same and the associativity of the production is left.
  - Otherwise, shift is the chosen action.

Precedence and Associativity (Cont.)

- The precedence of a production is taken to be the same as that of its rightmost terminal.
- E.g., Given production E→E+E (rightmost terminal is +)
  - Reduce E→E+E if the lookahead is +.
  - Shift if the lookahead is *.
- If the rightmost terminal of a production does not supply the proper precedence, we can force by appending to a production the tag "%prec <terminal>". Then
  - The precedence of this production is the same as that of this "terminal".
  - This "terminal" can be a placeholder that is not returned by the lexical analyzer.
- Yacc does not report shift/reduce conflicts that are resolved using this precedence and associativity mechanism.

Creating Yacc with Lex

- *Lex* was designed to produce lexical analyzers that could be used with *Yacc*.
- The Lex library ll provides a driver program named yylex() that is required by Yacc for lexical analysis.
  - If Lex is used to produe the lexical analyzer we replace the routine yylex() in the third part of the Yacc specification by the statement: #inculde “lex.yy.c”.
  - By using the #include “lex.yy.c” statement, the program yylex() has access to Yacc’s token names since the Lex output file is compiled as part of the Yacc output file “y.tab.c”.

Build the parser:

```
flex first.l
yacc second.y
gcc y.tab.c –ly
```

Creating Yacc with Lex

```
…
%%
yylex(){
    int c;
    while( (c = getchar()) == ' ');
    if ( c == '.' || isdigit(c) ) {
        ungetc(c, stdin);
        scanf("%lf", &yylval);
        return NUMBER;
    }
    return c;
}
```

second.y

```
number    [0-9]+\.?|[0-9]*\.[0-9]+
%%
[ ]                                 { /* skip blanks */ }
{number}  {sscanf(yytext, "%lf", &yylval); return NUMBER; }
\n|.                      {return yytext[0];}
%%
```

Any character

first.l

```
…
%%
int yywrap(void)
{
        return 1;
}
#include “lex.yy.c”
```

second.y

Error Recovery in Yacc

The reserved token generated when the lexical analysis from input encounters an error.

```
%{
#include <stdio.h>
#include <ctype.h>
/* double type for Yacc stack */
#define YYSTYPE double
%}
%token NUMBER

%left '+' '-'
%left '*' '/'
%right UMINUS
%%
```

```
lines       : lines expr '\n'          { printf("%g\n", $2); }
            | lines '\n‘
            | error '\n' {yyerror("reenter previous line:"); yyerrok; }
            | /* empty */
            ;
expr        : expr '+' expr            { $$ = $1 + $3; }
            | expr '-' expr            { $$ = $1 - $3; }
            | expr '*' expr            { $$ = $1 * $3; }
            | expr '/' expr            { $$ = $1 / $3; }
            | '(' expr ')'             { $$ = $2; }
            | '-' expr %prec UMINUS {$$ = -$2; }
            | NUMBER
            ;
%%
int yywrap(void)
{
        return 1;
}
#include “lex.yy.c”
```

Error production

Return normal operation

This function is needed by lex.yy.c

Error Recovery in Yacc (Cont.)

- In Yacc, error recovery uses a form of error productions.
  - Add to the grammar error productions of the form:
    A → error α
    - error is a Yacc reversed word.
    - A is a major nonterminal.
    - α is a string of grammar symbols.
  - The error productions are treated as ordinary productions.
- When the parser encounters an error,
  - It pops symbols from its stack until it finds the topmost state on its stack whose underlying set of items includes an item of the form **A→ · error** α.
  - Then shifts a fictitious token error onto the stack as though it saw the token error on its input.
  - If α is ε, a reduction to A occurs immediately and the semantic action associated with the production **A→ · error** invoked.
  - If α is not ε, Yacc skips ahead on the input looking for a substring that can be reduced to α **or get** α. Then reduce **error** α to **A**.

Project

- Revise the program in Slides 144 and 145 to add the following functions:
  - Add “tab” (i.e., “\t”) into the white space in addition to “ “.
  - Add the production **$E \rightarrow E_1\text{^}E_2$** to the grammar, where E = pow(E_1, E_2).
    - Note: remember to include “#include <math.h>” to declare the function pow().
  - Any conflict message during complication is not allowed.
- Cygwin: http://www.cygwin.com/ (remember to select “install all” during the installation.)
- Suppose the lex file is “first.l” and the yacc file is “second.y”. Build the project with the following commands under Cygwin:

```
flex first.l
yacc second.y
gcc y.tab.c –ly
```

- Requirements:
  - Send an email to me with two files:
    - calculator.l
    - calculator.y
  - Email title: [Compiler] Student ID, Name
  - Due: By noon of June 26

This is a bonus project with at most five points.