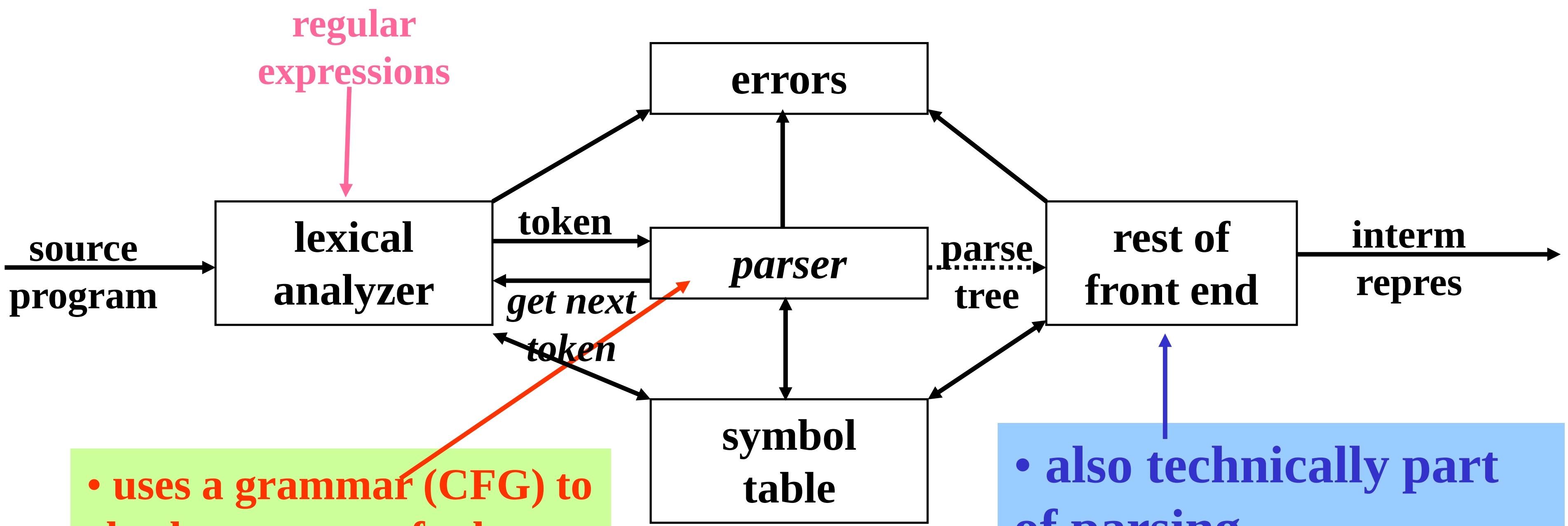
Syntax Analysis (Parsing)

- Error handling
- An overview of parsing
  - Functions & Responsibilities
- Context Free Grammars
  - Concepts & Terminology
- Writing and Designing Grammars
- Resolving Grammar Problems / Difficulties
- Top-Down Parsing
  - Recursive Descent & Predictive LL
- Bottom-Up Parsing
  - SLR、LR & LALR
- Concluding Remarks/Looking Ahead

Parsing During Compilation

- Parser works on a stream of tokens.
- The smallest item is a token.



- uses a grammar (CFG) to check structure of tokens
- produces a parse tree
- syntactic errors and recovery
- recognize correct syntax
- report errors

- also technically part of parsing
- includes augmenting info on tokens in source, type checking, semantic analysis

Error Handling

What are some Typical Errors ?

```
#include<stdio.h>
int f1(int v)
{       int i,j=0;
        for (i=1;i<5;i++)
           { j=v+f2(i) }
        return j; }
int f2(int u)
{       int j;
        j=u+f1(u*u);
        return j; }
int main()
{       int i,j=0;
        for (i=1;i<10;i++)
           { j=j+i*i  printf("%d\n",i); }
        printf("%d\n",f1(j$));
        return 0;
}
```

As reported by MS VC++

1. 'f2' undefined; assuming extern returning int
2. syntax error : missing ';' before '}'
3. syntax error : missing ';' before identifier 'printf'
4. j$ invalid id

Which are "easy" to recover from?
Which are "hard" ?

Error Handling

Error type	Example	Detector
Lexical	x#y=1	Lexer
Syntax	x=1 y=2	Parser
Semantic	int x; y=x(1)	Type checker
Correctness	Program can be compiled	Tester/user/static analysis/model-checker

Error Processing

- Detecting errors
- Finding position at which they occur
- Clear / accurate presentation
- **Recover (pass over) to continue and find later errors**
- Don’t impact compilation of “correct” programs

Error Recovery Strategies

Panic Mode– Discard tokens until a "synchronizing" token is found (end, ";", "}", etc.)

-- Decision of designer

E.g., (1 + + 2) * 3 skip +

-- Problems:

skip input ⇒miss declaration – causing more errors

⇒miss errors in skipped material

-- Advantages:

simple ⇒suited to 1 error per statement

Phrase Level – Local correction on input

-- "," **replace**";" – **Delete** "," – **insert** ";"……

-- Also decision of designer, Not suited to all situations

-- Used in conjunction with panic mode to allow less input to be skipped

E.g., x=1 ; y=2

Error Recovery Strategies – (2)

Error Productions:

-- Augment grammar with rules
-- Augment grammar used for parser construction / generation
-- example: add a rule for
:= in C assignment statements
Report error but continue compile
-- Self correction + diagnostic messages
Error -> ID := Expr

Global Correction:

-- Adding / deleting / replacing symbols is chancy – may <u>do many</u> changes !
-- Algorithms available to minimize changes costly - key issues
-- Rarely used in practice

Error Recovery Strategies – (3)

Past

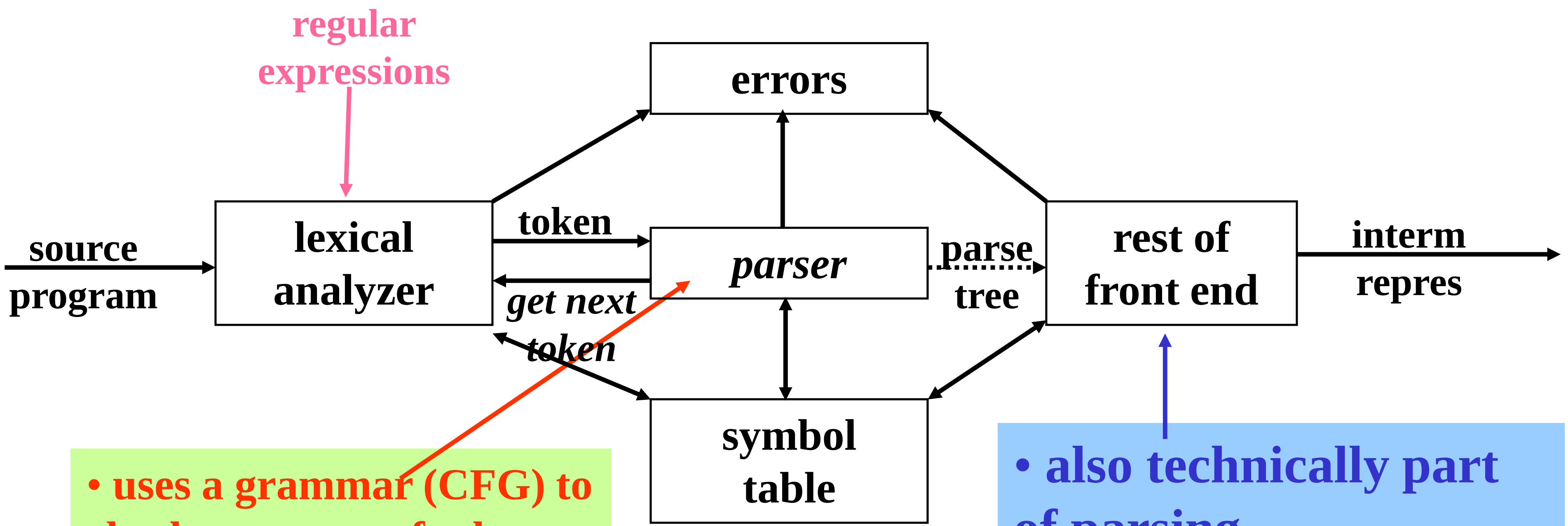
- Slow recompilation cycle (even once a day)
- Find as many errors in one cycle as possible
- Researchers could not let go of the topic

Present

- Quick recompilation cycle
- Users tend to correct one error/cycle
- Complex error recovery is less compelling
- Panic-mode seems enough

Parsing During Compilation

- Parser works on a stream of tokens.
- The smallest item is a token.



- uses a grammar (CFG) to check structure of tokens
- produces a parse tree
- syntactic errors and recovery
- recognize correct syntax
- report errors

- also technically part of parsing

- includes augmenting info on tokens in source, type checking, semantic analysis

Grammar >> language

Why are Grammars to formally describe Languages Important ?

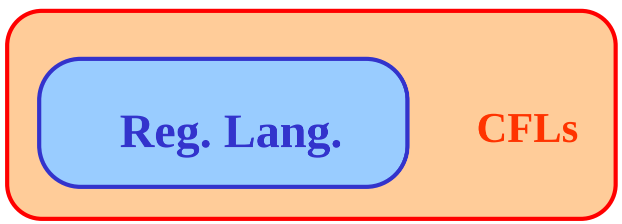
1. Precise, easy-to-understand representations
2. Compiler-writing tools can take grammar and generate a compiler (YACC, BISON)
3. allow language to be evolved (new statements, changes to statements, etc.) Languages are not static, but are constantly upgraded to add new features or fix “old” ones

C90/C89 -> C95 ->C99 - > C11 - > C18……

C++98 -> C++03 -> C++11 -> C++14 -> C++17-> C++20……

RE vs. CFG

- Regular Expressions
  - → Basis of lexical analysis
  - → Represent regular languages
- Context Free Grammars
  - → Basis of parsing
  - → Represent language constructs
  - → Characterize context free languages



A0 -> aA0 | bA0 | aA1

A1 -> bA2

A2 -> bA3

A3 -> ε

EXAMPLE: (a+b)*abb

EXAMPLE: $\{ (^i)^i \mid i>=0 \}$ Non-regular language (Why)

Context-Free Grammars

- Inherently **recursive** structures of a programming language are defined by a context-free grammar.
- In a context-free grammar, we have:
  - **A finite set of terminals (in our case, this will be the set of tokens)**
  - **A finite set of non-terminals (syntactic-variables)**
  - **A finite set of productions rules in the following form**
    - **$A \rightarrow \alpha$ where A is a non-terminal and α is a string of terminals and non-terminals (including the empty string)**
  - **A start symbol (one of the non-terminal symbol)**

- Example:

$E \rightarrow E + E \ | \ E - E \ | \ E * E \ | \ E / E \ | \ - E$

$E \rightarrow (E)$

$E \rightarrow id$

$E \rightarrow num$

Concepts & Terminology

A Context Free Grammar (CFG), is described by (T, NT, S, PR), where:

T: Terminals / tokens of the language

NT: Non-terminals, S: Start symbol, $S \in NT$

PR: Production rules to indicate how T and NT are combined to generate valid strings of the language.

$$\textbf{PR: } NT \rightarrow (T \mid NT)^*$$

E.g.: $E \rightarrow E + E$

$E \rightarrow num$

Like a Regular Expression / DFA / NFA, a Context Free Grammar is a mathematical model

Example Grammar

expr → *expr op expr*

expr → (*expr*)

expr → - *expr*

expr → *id*

op → +

op → -

op → *

op → /

Black : NT

Blue : T

expr* : *S

8 Production rules

Example Grammar

A fragment of Cool:

EXPR → if EXPR then EXPR else EXPR fi

| while EXPR loop EXPR pool

| id

Terminology (cont.)

- *A* ***sentence*** *of G* is a string of terminal symbols of G.

  ω is a sentence of L(G) if $S \overset{+}{\Rightarrow} \omega$ where S is the start symbol of G and ω is a string of terminals of G.
- L(G) is *the language of G* (the language generated by G) which is a set of sentences.

- A language that can be generated by a context-free grammar is said to be a **context-free language**
- Two grammars are *equivalent* if they produce the same language.
- $S \overset{*}{\Rightarrow} \alpha$ - it is called as a ***sentential form*** of G.

  - If α does not contain non-terminals, it is called as a ***sentence*** of G.

How does this relate to Languages?

Let G be a CFG with start symbol S. Then $S \stackrel{+}{\Rightarrow} W$ (where W has no non-terminals) represents the sentence generated by G, denoted L(G). So **$W \in L(G) \Leftrightarrow S \stackrel{+}{\Rightarrow} W$**.

W : is a sentence of G

EXAMPLE: id * id is a sentence

Here's the derivation:

$E \Rightarrow E \text{ op } E \Rightarrow E * E \Rightarrow id * E \Rightarrow id * id$

Sentential forms **Sentence**

$E \stackrel{*}{\Rightarrow} id * id$

Example Grammar – Terminology

Terminals: a,b,c,+,-,punc,0,1,…,9, blue strings

Non Terminals: A,B,C,S, black strings

T or NT: X,Y,Z

Strings of Terminals: u,v,…,z in T*

Strings of T / NT: α, β, γ in (T $\cup$ NT)*

Alternatives of production rules:

$A \to \alpha_1; A \to \alpha_2; \ldots; A \to \alpha_k; \Rightarrow A \to \alpha_1 \mid \alpha_2 \mid \ldots \mid \alpha_k$

First NT on LHS of 1st production rule is designated as start symbol !

$E \to E \text{ op } E \mid (E) \mid -E \mid \text{id}$

$\text{op} \to + \mid - \mid * \mid /$

Derivations

The central idea here is that a production is treated as a **rewriting rule** in which the non-terminal on the left is replaced by the string on the right side of the production.

$E \Rightarrow E+E$

- E+E derives from E
  - we can replace E by E+E
  - to able to do this, we have to have a production rule $E \rightarrow E+E$ in our grammar.

$E \Rightarrow E+E \Rightarrow id+E \Rightarrow id+id$

- A sequence of replacements of non-terminal symbols is called a **derivation** of id+id from E.

- In general a derivation step is

$\alpha_1 \Rightarrow \alpha_2 \Rightarrow ... \Rightarrow \alpha_n$ (α_n derives from α_1 or α_1 derives α_n)

Grammar Concepts

A step in a derivation is one action that replaces a NT with the RHS of a production rule.

EXAMPLE: $E \Rightarrow -E$ (the $\Rightarrow$ means "derives" in one step) using the production rule: $E \rightarrow -E$

EXAMPLE: $E \Rightarrow E \text{ op } E \Rightarrow E * E \Rightarrow E * (E)$

DEFINITION: $\Rightarrow$ derives in one step

$\overset{+}{\Rightarrow}$ derives in $\geq$ one step

$\overset{*}{\Rightarrow}$ derives in $\geq$ zero steps

EXAMPLES: $\alpha A \beta \Rightarrow \alpha \gamma \beta$ if $A \rightarrow \gamma$ is a production rule

$\alpha_1 \Rightarrow \alpha_2 \Rightarrow \ldots \Rightarrow \alpha_n$ then $\alpha_1 \overset{*}{\Rightarrow} \alpha_n$; $\alpha \overset{*}{\Rightarrow} \alpha$ for all α

If $\alpha \overset{*}{\Rightarrow} \beta$ and $\beta \rightarrow \gamma$ then $\alpha \overset{*}{\Rightarrow} \gamma$

Other Derivation Concepts

Example

$$E \Rightarrow -E \Rightarrow -(E) \Rightarrow -(E+E) \Rightarrow -(id+E) \Rightarrow -(id+id)$$

OR

$$E \Rightarrow -E \Rightarrow -(E) \Rightarrow -(E+E) \Rightarrow -(E+id) \Rightarrow -(id+id)$$

- At each derivation step, we can choose any of the non-terminal in the sentential form of G for the replacement.

- If we always choose the left-most non-terminal in each derivation step, this derivation is called as **left-most derivation**.

- If we always choose the right-most non-terminal in each derivation step, this derivation is called as **right-most derivation**.

Derivation Example

Leftmost: Replace the leftmost non-terminal symbol

$$E \underset{lm}{\Rightarrow} E \text{ op } E \underset{lm}{\Rightarrow} \text{id op } E \underset{lm}{\Rightarrow} \text{id} * E \underset{lm}{\Rightarrow} \text{id} * \text{id}$$

Rightmost: Replace the leftmost non-terminal symbol

$$E \underset{rm}{\Rightarrow} E \text{ op } E \underset{rm}{\Rightarrow} E \text{ op id} \underset{rm}{\Rightarrow} E * \text{id} \underset{rm}{\Rightarrow} \text{id} * \text{id}$$

Important Notes: $A \rightarrow \delta$

If $\beta A \gamma \underset{lm}{\Rightarrow} \beta \delta \gamma$ what's true about β ?

If $\beta A \gamma \underset{rm}{\Rightarrow} \beta \delta \gamma$ what's true about γ ?

Derivations: Actions to parse input can be represented pictorially in a parse tree.

Left-Most and Right-Most Derivations

Left-Most Derivation

$$E \underset{lm}{\Rightarrow} -E \underset{lm}{\Rightarrow} -(E) \underset{lm}{\Rightarrow} -(E+E) \underset{lm}{\Rightarrow} -(id+E) \underset{lm}{\Rightarrow} -(id+id) \qquad (4.4)$$

Right-Most Derivation (called *canonical derivation*)

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

- We will see that the ***top-down parsers*** try to find the ***left-most derivation*** of the given source program.

- We will see that the ***bottom-up parsers*** try to find the ***right-most derivation*** of the given source program in the reverse order.

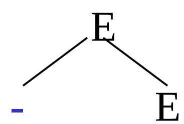
Parse Tree

- Inner nodes of a parse tree are non-terminal symbols.
- The leaves of a parse tree are terminal symbols.

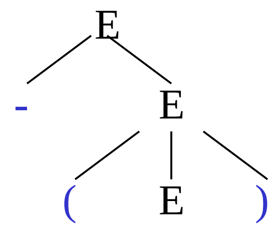
A parse tree can be seen as a graphical representation of a derivation.

EX. $E \Rightarrow -E \Rightarrow -(E) \Rightarrow -(E+E) \Rightarrow -(id+E) \Rightarrow -(id+id)$

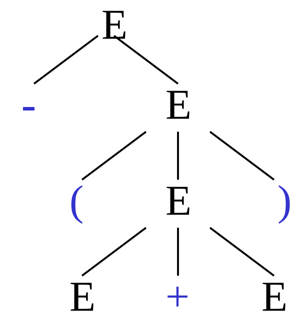
$E \Rightarrow -E$



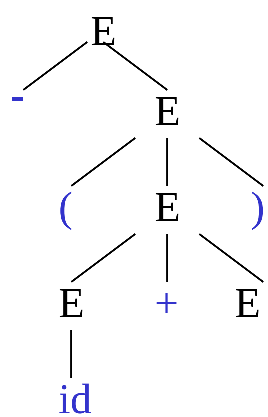
$\Rightarrow -(E)$



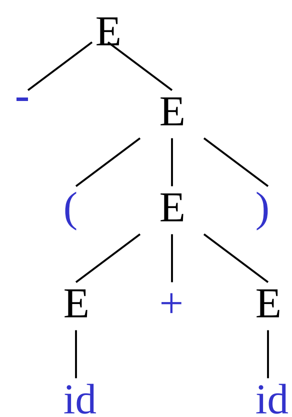
$\Rightarrow -(E+E)$



$\Rightarrow -(id+E)$



$\Rightarrow -(id+id)$



Examples of LM / RM Derivations

$E \rightarrow E \text{ op } E \mid (E) \mid -E \mid \text{id}$

$\text{op} \rightarrow + \mid - \mid * \mid /$

A leftmost derivation of : id + id * id

See latter slides

A rightmost derivation of : id + id * id

DO IT BY YOURSELF.

A leftmost derivation of : id + id * id

$E \rightarrow E \text{ op } E \mid (E) \mid -E \mid \text{id}$

$\text{op} \rightarrow + \mid - \mid * \mid /$

$E \Rightarrow E \text{ op } E$

$\Rightarrow \text{id op } E$

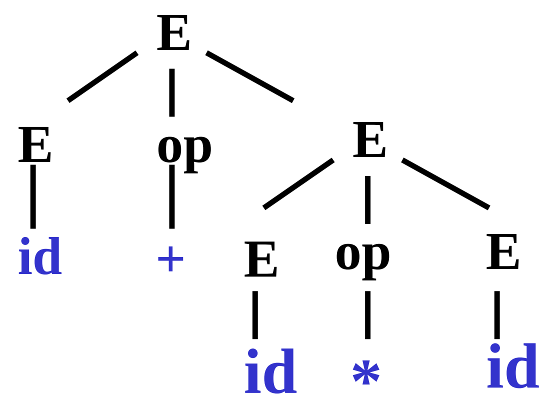
$\Rightarrow \text{id} + E$

$\Rightarrow \text{id} + E \text{ op } E$

$\Rightarrow \text{id} + \text{id op } E$

$\Rightarrow \text{id} + \text{id} * E$

$\Rightarrow \text{id} + \text{id} * \text{id}$

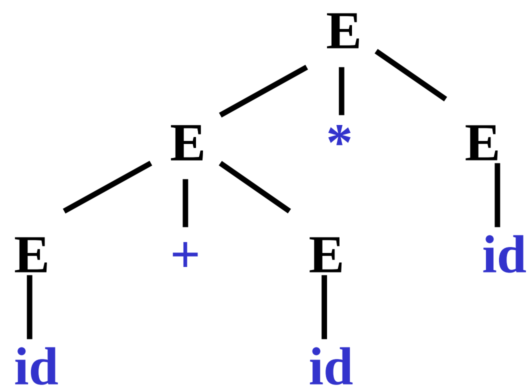


A rightmost derivation of : id + id * id

DO IT BY YOURSELF.

Alternative Parse Tree & Derivation

$$
\begin{aligned}
E &\Rightarrow E * E \\
&\Rightarrow E + E * E \\
&\Rightarrow id + E * E \\
&\Rightarrow id + id * E \\
&\Rightarrow id + id * id
\end{aligned}
$$



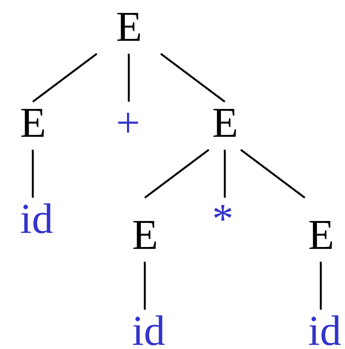
WHAT'S THE ISSUE HERE ?

Two distinct leftmost derivations!

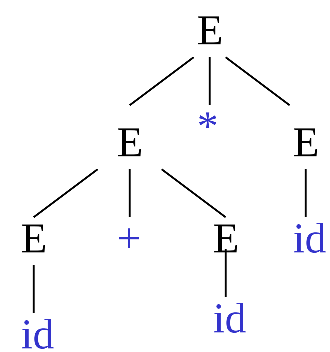
Ambiguity

- A grammar produces more than one parse tree for a sentence is called as an ***ambiguous*** grammar.

$$E \Rightarrow E+E \Rightarrow id+E \Rightarrow id+E*E$$
$$\Rightarrow id+id*E \Rightarrow id+id*id$$



$$E \Rightarrow E*E \Rightarrow E+E*E \Rightarrow id+E*E$$
$$\Rightarrow id+id*E \Rightarrow id+id*id$$



Two parse trees for id+id*id.

Ambiguity (cont.)

- For the most parsers, the grammar must be unambiguous.

- **unambiguous grammar**

  ➔ unique selection of the parse tree for a sentence

- We have to prefer one of the parse trees of a sentence (generated by an ambiguous grammar) to disambiguate that grammar to restrict to this choice by “throw away” undesirable parse trees.
- We should eliminate the ambiguity in the grammar during the design phase of the compiler.
  1. **Grammar rewritten** to eliminate the ambiguity
  2. Enforce **precedence** and **associativity**

Ambiguity: precedence and associativity

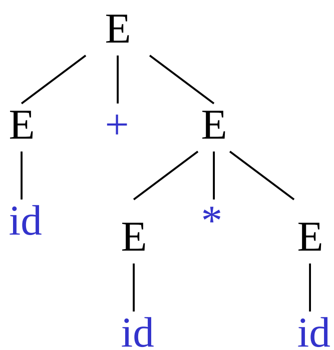
- Ambiguous grammars (because of ambiguous operators) can be disambiguated according to the **precedence** and **associativity** rules.

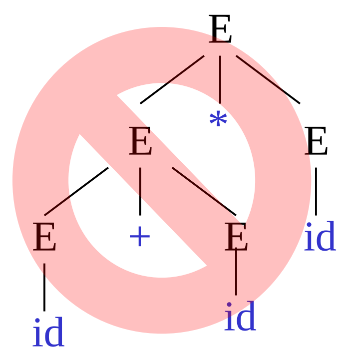
$E \rightarrow E+E \mid E*E \mid id \mid (E)$

disambiguate the grammar

precedence:
* (left associativity)
\+ (left associativity)

Ex. id + id * id





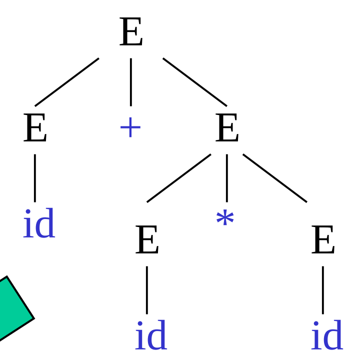
Ambiguity: Grammar rewritten

- Ambiguous grammars (because of ambiguous operators) can be disambiguated according to the **precedence** and **associativity** rules.

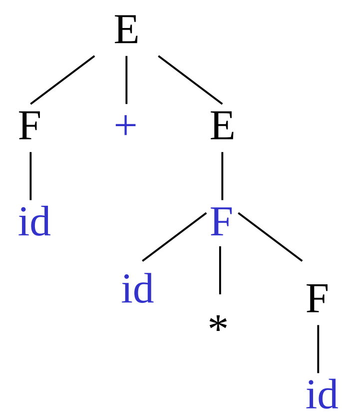
$$E \rightarrow E+E \mid E*E \mid id \mid (E)$$

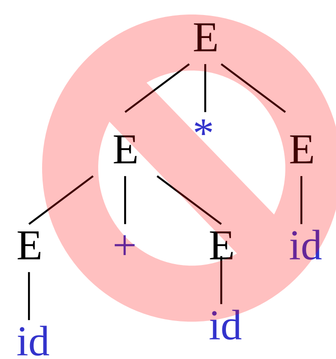
$$E \rightarrow F + E \mid F$$

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



Ex. id + id * id





Eliminating Ambiguity

Consider the following grammar segment:

$$\begin{aligned} stmt \rightarrow\ & \textbf{if}\ expr\ \textbf{then}\ stmt \\ |\ & \textbf{if}\ expr\ \textbf{then}\ stmt\ \textbf{else}\ stmt \\ |\ & \textbf{other}\ \ \textbf{(any other statement)} \end{aligned}$$

How is this parsed ?

$$\textbf{if}\ E_1\ \textbf{then}\ \ \textbf{if}\ E_2\ \textbf{then}\ S_1\ \textbf{else}\ S_2$$

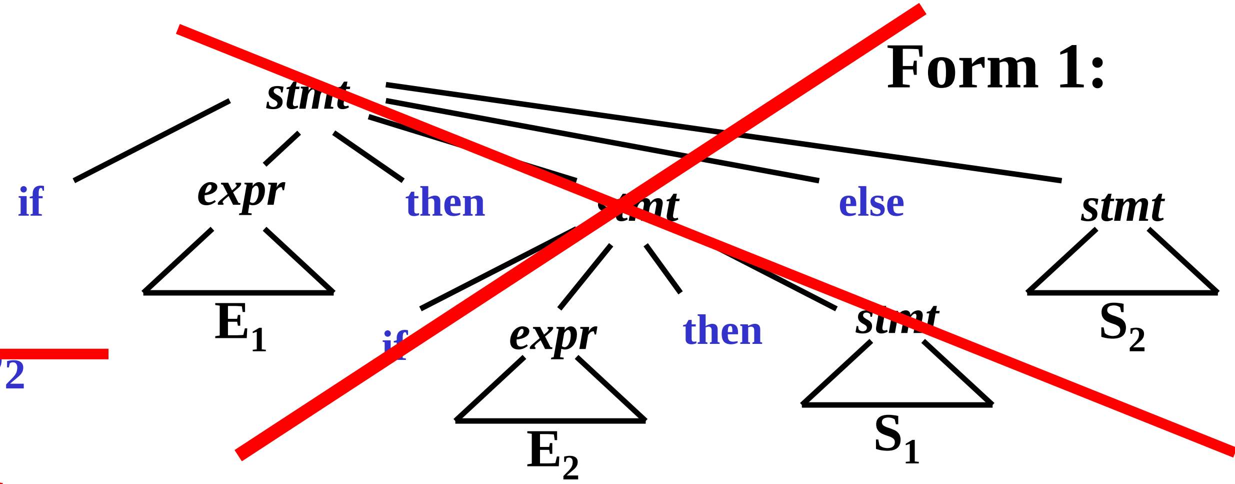
Parse Trees for Example

stmt → **if** *expr* **then** *stmt*

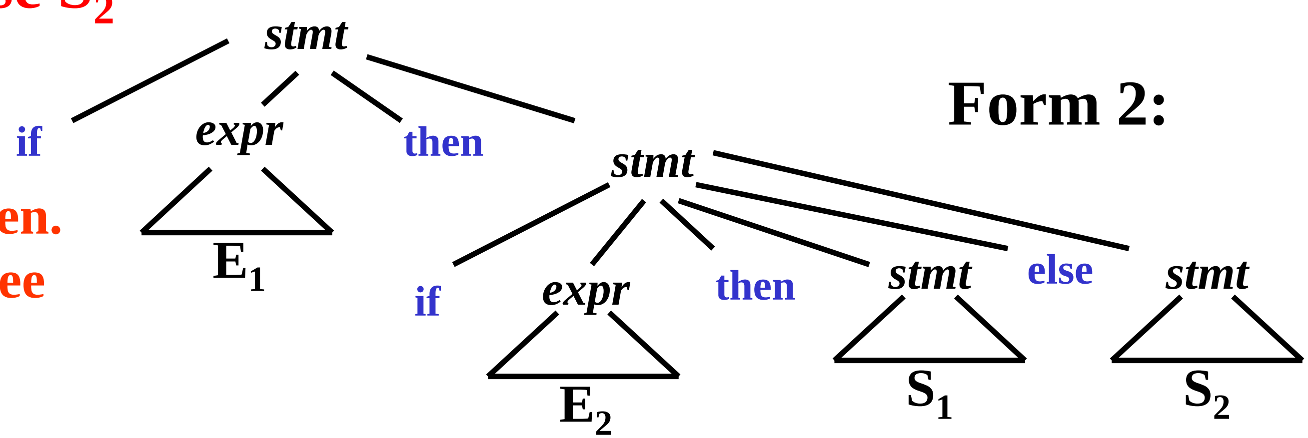
| **if** *expr* **then** *stmt* **else** *stmt*

| **other** (any other statement)

~~**if E$_1$ then if E$_2$ then S$_1$ else S$_2$**~~



if E$_1$ then if E$_2$ then S$_1$ else S$_2$



Else <u>must</u> match to previous then. Structure indicates parse subtree for expression.

Two parse trees for an ambiguous sentence.

Ambiguity (cont.)

- We prefer the second parse tree (else matches with closest then).
- So, we have to disambiguate our grammar to reflect this choice.

- The unambiguous grammar will be:

stmt → matchedstmt
| unmatchedstmt
matchedstmt → if expr then matchedstmt else matchedstmt
| otherstmts
unmatchedstmt → if expr then stmt
| if expr then matchedstmt else unmatchedstmt

The general rule is "match each **else** with the closest previous unmatched **then**."

Non-Context Free Language Constructs

- There are some language constructions in the programming languages which are not context-free. This means that, we cannot write a context-free grammar for these constructions.

Example

- L1 = { $\omega c \omega \mid \omega$ is in (a+b)*} is not context-free

➔ declaring an identifier and checking whether it is declared or not later. We cannot do this with a context-free language. We need semantic analyzer (which is not context-free).

Example

- L2 = $\{a^n b^m c^n d^m \mid n \geq 1 \text{ and } m \geq 1\}$ is not context-free

➔ declaring two functions (one with n parameters, the other one with m parameters), and then calling them with actual parameters.

Operations on Languages

OPERATION	
union of L and M writte $L \cup M$	
concatenation of L and M written LM	
Kleene closure of L written L*	$_{i=0}$ L* denotes “zero or more concatenations of L
positive closure of L written L^+	$L^+ = \bigcup_{i=1}^{\infty} L^i$ L^+ denotes “one or more concatenations of L
Intersection of L and M written $L \cap M$	$L \cap M$ = {s | s is in L and s is in M}

Does context-free languages closed under these operations?

Let's derive: *id := id + real – integer ;*

Left-most derivation:

assign_stmt

→ *id := expr ;*

→ *id := expr op term;*

→ *id := expr op term op term;*

→ *id := term op term op term;*

→ *id := id op term op term;*

→ *id := id + term op term;*

→ *id := id + real op term;*

→ *id := id + real - term;*

→ *id := id + real - integer;*

using production:

assign_stmt → id := expr ;

expr → expr op term

expr → expr op term

expr → term

term → id

op → +

term → real

op → -

term → integer

Right-most derivation and parse tree?

CFG2Parser

Top-Down Parsing

- The parse tree is created top to bottom (from root to leaves).
- **By always replacing the leftmost non-terminal symbol via a production rule, we are guaranteed of developing a parse tree in a left-to-right fashion that is consistent with scanning the input.**
- Top-down parser
  - **Recursive-Descent Parsing**
    - Backtracking is needed (If a choice of a production rule does not work, we backtrack to try other alternatives.)
    - It is a general parsing technique, but not widely used.
    - Not efficient

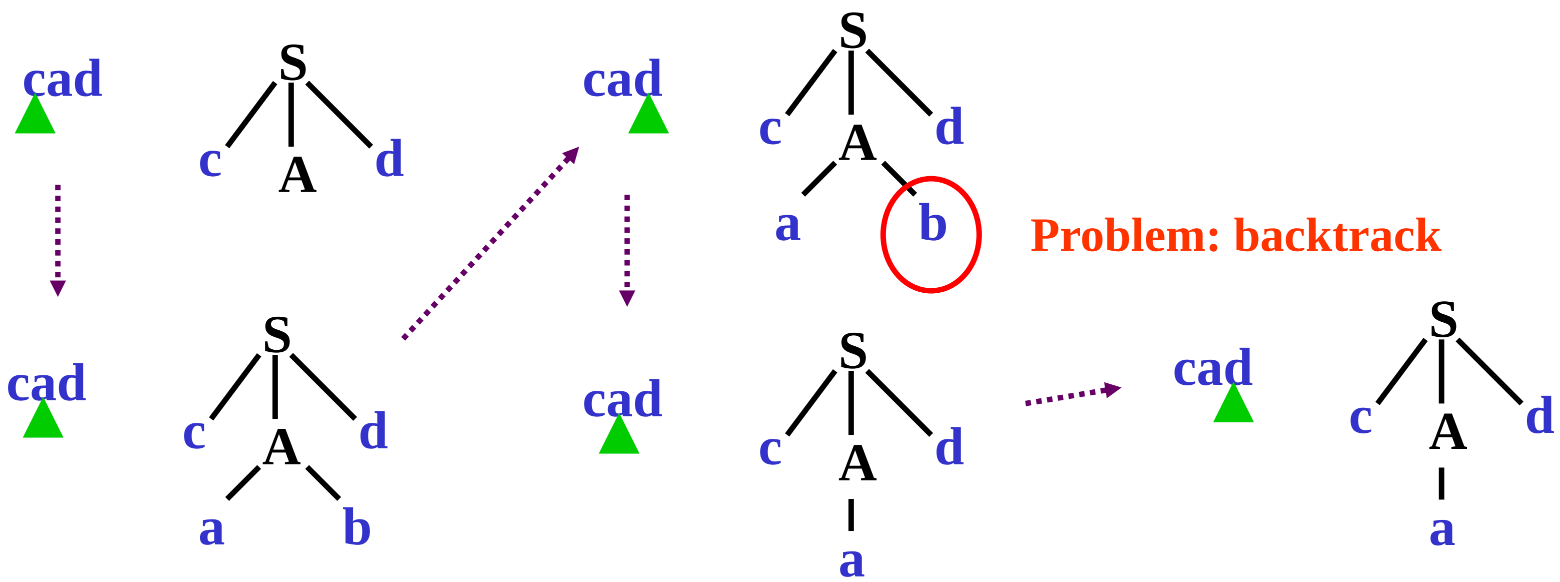
$A \Rightarrow aBc \Rightarrow adDc \Rightarrow adec$

(scan a, scan d, scan e, scan c - accept!)

  - **Predictive Parsing**
    - no backtracking, at each step, only one choices of production to use
    - efficient
    - needs a special form of grammars (**LL(k) grammars, k=1 in practice**).
    - **Recursive Predictive Parsing** is a special form of Recursive Descent parsing without backtracking.
    - **Non-Recursive (Table Driven) Predictive Parser** is also known as LL(k) parser.

Recursive-Descent Parsing (uses Backtracking)

- General category of Top-Down Parsing
- Choose production rule based on input symbol
- May require backtracking to correct a wrong choice.
- Example: $\left.\begin{array}{l} S \rightarrow c\,A\,d \\ A \rightarrow ab \mid a \end{array}\right\}$ **input: cad**



Steps in top-down parse

Implementation of Recursive-Descent

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

```
bool term(TOKEN tok) { return *next++ == tok; }

bool E1() { return T(); }

bool E2() { return T() && term(PLUS) && E(); }

bool E() {TOKEN *save = next;
        return (next = save, E1())||(next = save, E2()); }

bool T1() { return term(INT); }

bool T2() { return term(INT) && term(TIMES) && T(); }

bool T3() { return term(OPEN) && E() && term(CLOSE); }
bool T() { TOKEN *save = next; return (next = save, T1())
                                    ||(next = save, T2())
                                    ||(next = save, T3());}
```

When Recursive Descent Does Not Work ?

- Example: $S \rightarrow S\ a$
- Implementation:

```
bool S1() { return S() && term(a); }
bool S() { return S1(); }   infinite recursion
```

$S \rightarrow^{+} S\ a$ left-recursive grammar

we should remove left-recursive grammar

Left Recursion

- A grammar is ***left recursive*** if it has a non-terminal A such that there is a derivation $A \Rightarrow^{+} A\alpha$ for some string α.

- The left-recursion may appear in a single step of the derivation (***immediate left-recursion***), or may appear in more than one step of the derivation.
- Top-down parsing techniques **cannot** handle left-recursive grammars.

- So, we have to convert our left-recursive grammar into an equivalent grammar which is not left-recursive.

Immediate Left-Recursion

$A \rightarrow A\,\alpha \mid \beta$ where β does not start with A

$\Downarrow$ eliminate immediate left recursion

$A \rightarrow \beta\, A'$

$A' \rightarrow \alpha\, A' \mid \varepsilon$ an equivalent grammar: replaced by right-recursion

In general,

$A \rightarrow A\,\alpha_1 \mid \ldots \mid A\,\alpha_m \mid \beta_1 \mid \ldots \mid \beta_n$ where β_1 ... β_n do not start with A

$\Downarrow$ eliminate immediate left recursion

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

$A' \rightarrow \alpha_1\, A' \mid \ldots \mid \alpha_m\, A' \mid \varepsilon$ an equivalent grammar

Left-Recursion -- Problem

- A grammar cannot be immediately left-recursive, but it still can be left-recursive.
- By just eliminating the immediate left-recursion, we may not get a grammar which is not left-recursive.

$S \rightarrow Aa \mid b$

$A \rightarrow Sc \mid d$ This grammar is not immediately left-recursive, but it is still left-recursive.

$\underline{S} \Rightarrow Aa \Rightarrow \underline{S}ca$

or $\underline{A} \Rightarrow Sc \Rightarrow \underline{A}ac$ causes to a left-recursion

- **So, we have to eliminate all left-recursions from our grammar**

Elimination of Left-Recursion

Algorithm eliminating left recursion.

- Arrange non-terminals in some order: $A_1 \ldots A_n$
- **for** i **from** 1 **to** n **do** {
  - **for** j **from** 1 **to** i-1 **do** {

      replace each production

      $A_i \rightarrow A_j \gamma$ // $j < i$

      by

      $A_i \rightarrow \alpha_1 \gamma \mid \ldots \mid \alpha_k \gamma$

      where $A_j \rightarrow \alpha_1 \mid \ldots \mid \alpha_k$ are all the current A_j-productions.

    } // for all A_m at the begin of α, m>=i
  - **eliminate immediate left-recursions among A_i-productions**

    // for all A_m at the begin of α, m>i

}

Algorithm to eliminate left recursion from a grammar.

Example

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

- Order of non-terminals: S, A

for S:

- we do not enter the inner loop.
- there is no immediate left recursion in S.

for A:

- Replace $A \rightarrow Sd$ with $A \rightarrow Aad \mid bd$

  So, we will have $A \rightarrow Ac \mid Aad \mid bd \mid \varepsilon$
- Eliminate the immediate left-recursion in A

  $A \rightarrow bdA' \mid A'$
  $A' \rightarrow cA' \mid adA' \mid \varepsilon$

no immediately left-recursive

↓

immediate left-recursive

↓

eliminating immediate left-recursive

So, the resulting equivalent grammar which is not left-recursive is:

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

Reading materials

- This algorithm crucially depends on the ordering of the nonterminals
- The number of resulting production rules is large

[1] Moore, Robert C. (2000). Removing Left Recursion from Context-Free Grammars

1. A novel strategy for ordering the nonterminals
2. An alternative approach

Predictive Parser vs. Recursive Descent Parser

- In recursive-descent,
  - At each step, many choices of production to use
  - Backtracking used to undo bad choices
- In Predictive Parser (lookahead)
  - At each step, only one choice of production
  - That is
    - When a non-terminal A is leftmost in a derivation
    - The k input symbols are $a_1a_2\ldots a_k$
    - There is a unique production $A \longrightarrow \alpha$ to use
    - Or no production to use (an error state)
    - LL(k) is a recursive descent variant without backtracking

Predictive Parser (example)

stmt → **if** expr **then** stmt **else** stmt
|**while** expr **do** stmt
|**begin** stmt_list **end**
|**for** .....

- When we are trying to write the non-terminal *stmt*, if the current token is **`if`** we have to choose first production rule.
- When we are trying to write the non-terminal *stmt*, we can uniquely choose the production rule by just looking the current token.
- LL(1) grammar

Recursive Predictive Parsing

- Each non-terminal corresponds to a procedure (function).

Ex: $A \rightarrow aBb$ (This is only the production rule for A)

proc A() {

- match the current token with a, and move to the next token;
- call B();
- match the current token with b, and move to the next token;

}

Recursive Predictive Parsing (cont.)

$A \rightarrow aBb \mid bAB$

```
proc A() {
   case of the current token {
        'a':  - match the current token with a, and move to the next token;
              - call 'B';
              - match the current token with b, and move to the next token;
        'b':  - match the current token with b, and move to the next token;
              - call 'A';
              - call 'B';
   }
}
```

Recursive Predictive Parsing (cont.)

- When to apply ε-productions.

  $A \rightarrow aA \mid bB \mid \varepsilon$

- If all other productions fail, we should apply an ε-production. For example, if the current token is not a or b, we may apply the ε-production.
- Most correct choice: We should apply an ε-production for a non-terminal A when the current token is in the **follow** set of A (which terminals can follow A in the sentential forms).

Recursive Predictive Parsing (Example)

$A \to aBe \mid cBd \mid C$

$B \to bB \mid \varepsilon$

$C \to f$

```
proc A {
    case of the current token {
        a:  - match the current token with a,
              and move to the next token;
            - call B;
            - match the current token with e,
              and move to the next token;
        c:  - match the current token with c,
              and move to the next token;
            - call B;
            - match the current token with d,
              and move to the next token;
        f:  - call C
    }
}
```

first set of C/A

```
proc C {  match the current token with f,
          and move to the next token; }
```

```
proc B {
    case of the current token {
        b:- match the current token with b,
              and move to the next token;
            - call B
        e,d:  do nothing
    }
}
```

follow set of B

Predictive Parsing and Left Factoring

- Recall the grammar

  $E \rightarrow T + E \mid T$

  $T \rightarrow \text{int} \mid \text{int} * T \mid (E)$
- Hard to predict because
  - For T two productions start with int
  - For E it is not clear how to predict
- We need to left-factor the grammar

Left-Factoring

Problem : Uncertain which of 2 rules to choose:

$$stmt \to \text{if } expr \text{ then } stmt \text{ else } stmt$$
$$\mid \text{if } expr \text{ then } stmt$$

When do you know which one is valid ?

What's the general form of *stmt* ?

$A \to \alpha\beta_1 \mid \alpha\beta_2$

α : if *expr* then *stmt*

β_1: else *stmt* β_2 : ε

Transform to:

$A \to \alpha A'$

$A' \to \beta_1 \mid \beta_2$

EXAMPLE:

$stmt \to \text{if } expr \text{ then } stmt\ rest$

$rest \to \text{else } stmt \mid \varepsilon$

So, we can immediately expand A to αA'

Left-Factoring -- Algorithm

- For each non-terminal A with two or more alternatives (production rules) with a common non-empty prefix, let say

$$A \rightarrow \alpha\beta_1 \mid \ldots \mid \alpha\beta_n \mid \gamma_1 \mid \ldots \mid \gamma_m$$

convert it into

$$A \rightarrow \alpha A' \mid \gamma_1 \mid \ldots \mid \gamma_m$$
$$A' \rightarrow \beta_1 \mid \ldots \mid \beta_n$$

Left-Factoring – Example1

$A \to \underline{a}bB \mid \underline{a}B \mid cdg \mid cdeB \mid cdfB$

$\Downarrow$ step1

$A \to aA' \mid \underline{cd}g \mid \underline{cd}eB \mid \underline{cd}fB$

$A' \to bB \mid B$

$\Downarrow$ step 2

$A \to aA' \mid cdA''$

$A' \to bB \mid B$

$A'' \to g \mid eB \mid fB$

Predictive Parser

a grammar $\rightarrow$ eliminate left recursion $\rightarrow$ left factor a grammar suitable for predictive parsing (a LL(1) grammar) no %100 guarantee.

- When re-writing a non-terminal in a derivation step, a predictive parser can uniquely choose a production rule by just looking the **current symbol** in the input string.

current token

$$A \rightarrow \alpha_1 \mid \ldots \mid \alpha_n$$

input: ... a ....

current token

Implementation of Recursive-Descent

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

```
bool term(TOKEN tok) { return *next++ == tok; }

bool E1() { return T(); }

bool E2() { return T() && term(PLUS) && E(); }

bool E() {TOKEN *save = next;
        return (next = save, E1())||(next = save, E2()); }

bool T1() { return term(INT); }

bool T2() { return term(INT) && term(TIMES) && T(); }

bool T3() { return term(OPEN) && E() && term(CLOSE); }
bool T() { TOKEN *save = next; return (next = save, T1())
                                      ||(next = save, T2())
                                      ||(next = save, T3());}
```

Recursive Predictive Parsing (Example)

$A \rightarrow aBe \mid cBd \mid C$

$B \rightarrow bB \mid \varepsilon$

$C \rightarrow f$

```
proc C {  match the current token with f,
          and move to the next token; }

proc A {
   case of the current token {
      a:  - match the current token with a,
            and move to the next token;
          - call B;
          - match the current token with e,
            and move to the next token;
      c:  - match the current token with c,
            and move to the next token;
          - call B;
          - match the current token with d,
            and move to the next token;
      f:  - call C
   }
}

proc B {
   case of the current token {
      b:- match the current token with b,
          and move to the next token;
        - call B
      e,d:  do nothing
   }
}
```

follow set of B

first set of C/A

Non-Recursive Predictive Parsing -- LL(1)

- Non-Recursive predictive parsing is a **table-driven parser** which has an **input buffer**, a **stack**, a **parsing table**, and an **output stream**.
- It is also known as LL(1) Parser.

1. Left to right scan input
2. Find leftmost derivation

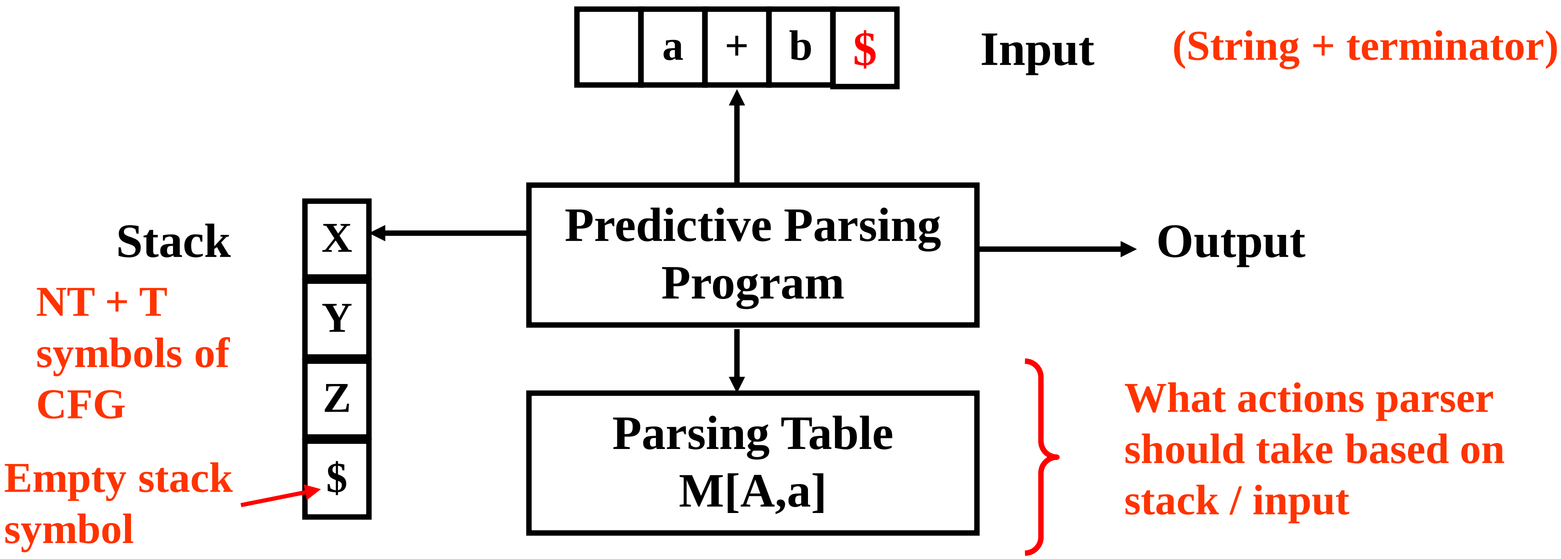
Grammar: $E \rightarrow TE'$

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

$T \rightarrow id$

Input : **id + id**

Non-Recursive / Table Driven



General parser behavior: X : top of stack   a : current input

1. When X=a = $ halt, accept, success
2. When X=a ≠ $, POP X off stack, advance input, go to 1.
3. When X is a non-terminal, examine M[X,a]

   if it is an error → call recovery routine

   if M[X,a] = {X → UVW}, POP X, PUSH W,V,U

   DO NOT expend any input

Notice the pushing order

Algorithm for Non-Recursive Parsing

Set *ip* to point to the first symbol of w$; push the start symbol and $ onto the stack

repeat

let X be the top stack symbol and *a* the symbol pointed to by *ip*;

if X is terminal or $ **then**

Input pointer

if X=a **then**

pop X from the stack and advance *ip*

else *error()*

else /* X is a non-terminal */

if $M[X,a] = X \rightarrow Y_1 Y_2 \ldots Y_k$ **then begin**

pop X from stack;

push Y_k, Y_{k-1}, … , Y_1 onto stack, with Y_1 on top

output the production $X \rightarrow Y_1 Y_2 \ldots Y_k$

May also execute other code based on the production used, such as creating parse tree.

end

else *error()*

until X=$ /* stack is empty */

LL(1) Parser – Example2

Example:

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

how to construct *id+id*id* ?

Table M

Non-terminal	INPUT SYMBOL					
	id	+	*	(	)	$
E	$E \rightarrow TE'$			$E \rightarrow TE'$		
E'		$E' \rightarrow +TE'$			$E' \rightarrow \epsilon$	$E' \rightarrow \epsilon$
T	$T \rightarrow FT'$			$T \rightarrow FT'$		
T'		$T' \rightarrow \epsilon$	$T' \rightarrow *FT'$		$T' \rightarrow \epsilon$	$T' \rightarrow \epsilon$
F	$F \rightarrow id$			$F \rightarrow (E)$		

Parsing table M for grammar

LL(1) Parser – Example2

stack	input	output	
$E	id+id*id$		
$E'T	id+id*id$	$E \rightarrow TE'$	
$E'T'F	id+id*id$	$T \rightarrow FT'$	
$ E'T'id	id+id*id$	$F \rightarrow id$	
$ E'T'	+id*id$		
$ E'	+id*id$	$T' \rightarrow \varepsilon$	
$ E'T+	+id*id$	$E' \rightarrow +TE'$	
$ E'T	id*id$		
$ E'T'F	id*id$	$T \rightarrow FT'$	
$ E'T'id	id*id$	$F \rightarrow id$	
$ E'T'	*id$		
$ E'T'F*	*id$	$T' \rightarrow *FT'$	
$ E'T'F	id$		
$ E'T'id	id$	$F \rightarrow id$	
$ E'T'	$		
$ E'	$	$T' \rightarrow \varepsilon$	
$	$	$E' \rightarrow \varepsilon$	accept

moves made by predictive parser on input id+id*id.

Leftmost Derivation for the Example

The leftmost derivation for the example is as follows:

$E \Rightarrow TE'$

$\Rightarrow FT'E'$

$\Rightarrow \mathbf{id}\ T'E'$

$\Rightarrow \mathbf{id}\ E'$

$\Rightarrow \mathbf{id} + TE'$

$\Rightarrow \mathbf{id} + FT'E'$

$\Rightarrow \mathbf{id} + \mathbf{id}\ T'E'$

$\Rightarrow \mathbf{id} + \mathbf{id} * FT'E'$

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

$\Rightarrow \mathbf{id} + \mathbf{id} * \mathbf{id}\ E'$

$\Rightarrow \mathbf{id} + \mathbf{id} * \mathbf{id}$

LL(1) Parser – Example

$S \rightarrow aBa$
$B \rightarrow bB \mid \varepsilon$

Sentence "**abba**" is correct or not **?**

	a	b	$
S	$S \rightarrow aBa$		
B	$B \rightarrow \varepsilon$	$B \rightarrow bB$	

LL(1) Parsing Table

stack	**input**	**output**
$S	**abba$**	$S \rightarrow aBa$
$aBa	**abba$**	
$aB	**bba$**	$B \rightarrow bB$
$aBb	**bba$**	
$aB	**ba$**	$B \rightarrow bB$
$aBb	**ba$**	
$aB	**a$**	$B \rightarrow \varepsilon$
$a	**a$**	
$	**$**	**accept, successful completion**

LL(1) Parser – Example (cont.)

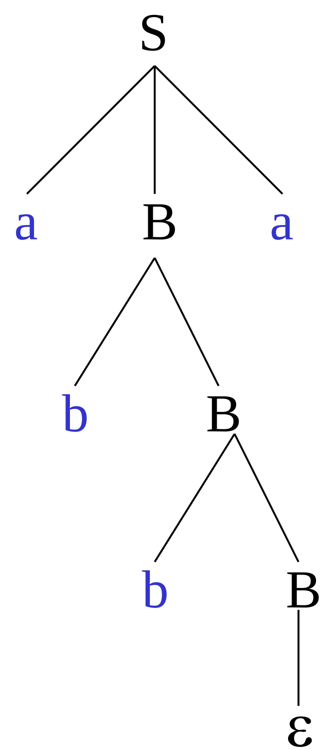
Outputs:

$S \rightarrow aBa \qquad B \rightarrow bB \qquad B \rightarrow bB \qquad B \rightarrow \varepsilon$

Derivation(left-most):

$S \Rightarrow aBa \Rightarrow abBa \Rightarrow abbBa \Rightarrow abba$

parse tree



Constructing LL(1) Parsing Tables

Constructing the Parsing Table M !

1st : Calculate FIRST & FOLLOW for Grammar

2nd: Apply Construction Algorithm for Parsing Table

(We'll see this shortly)

Basic Functions:

FIRST: Let α be a string of grammar symbols. FIRST(α) is the set that includes every **terminal** that appears leftmost in α or in any string originating from α.

NOTE: If $\alpha \Rightarrow \varepsilon$, then ε is FIRST(α).

FOLLOW: Let A be a non-terminal. FOLLOW(A) is the set of **terminals** a that can appear directly to the right of A in some sentential form. ($S \Rightarrow \alpha A a \beta$, for some α and β).

NOTE: If $S \Rightarrow \alpha A$, then \$ is FOLLOW(A).

Compute FIRST for Any String X

1. If X is a terminal, FIRST(X) = {X}
2. If $X \to \varepsilon$ is a production rule, add ε to FIRST (X)
3. If X is a non-terminal, and $X \to Y_1 Y_2 \ldots Y_k$ is a production rule

Place FIRST(Y_1) in FIRST (X)

if $Y_1 \Rightarrow \varepsilon$, Place FIRST ($Y_2$) in FIRST(X)

if $Y_2 \Rightarrow \varepsilon$, Place FIRST($Y_3$) in FIRST(X)

…

if $Y_{k-1} \Rightarrow \varepsilon$, Place FIRST($Y_k$) in FIRST(X)

NOTE: As soon as $Y_i \overset{*}{\not\Rightarrow} \varepsilon$, Stop.

Repeat above steps until no more elements are added to any FIRST( ) set.

Checking "$Y_i \Rightarrow \varepsilon$?" essentially amounts to checking whether ε belongs to FIRST(Y_i)

Computing FIRST(X) : All Grammar Symbols - continued

Informally, suppose we want to compute

$FIRST(X_1\ X_2 \ldots X_n) = FIRST(X_1)$

“+” $FIRST(X_2)$ if ε is in $FIRST(X_1)$

“+” $FIRST(X_3)$ if ε is in $FIRST(X_2)$

…

“+” $FIRST(X_n)$ if ε is in $FIRST(X_{n-1})$

Note 1: Only add ε to $FIRST(X_1\ X_2 \ldots X_n)$ if ε is in $FIRST(X_i)$ for all i

Note 2: For $FIRST(X_i)$, if $X_i \rightarrow Z_1\ Z_2 \ldots Z_m$, then we need to compute $FIRST(Z_1\ Z_2 \ldots Z_m)$!

FIRST Example

Example

$E \rightarrow TE'$

$E' \rightarrow +TE' \;|\; \varepsilon$

$T \rightarrow FT'$

$T' \rightarrow *FT' \;|\; \varepsilon$

$F \rightarrow (E) \;|\; id$

FIRST($\mathtt{F}$) = { $\mathtt{(}$, $\mathtt{id}$}

FIRST($\mathtt{T'}$) = {$\mathtt{*}$, ε}

FIRST($\mathtt{T}$) = { $\mathtt{(}$, $\mathtt{id}$}

FIRST($\mathtt{E'}$) = {$\mathtt{+}$, ε}

FIRST($\mathtt{E}$) = { $\mathtt{(}$, $\mathtt{id}$}

FIRST($\mathtt{*FT'}$) = {$\mathtt{*}$}

FIRST($\mathtt{(E)}$) = { $\mathtt{(}$}

FIRST($\mathtt{id}$) = {$\mathtt{id}$}

FIRST($\mathtt{+TE'}$) = {$\mathtt{+}$}

FIRST(ε) = {ε}

FIRST($\mathtt{TE'}$) = { $\mathtt{(}$, $\mathtt{id}$}

FIRST($\mathtt{FT'}$) = { $\mathtt{(}$, $\mathtt{id}$}

Alternative way to compute FIRST

Computing FIRST for:

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

FIRST(E) → **FIRST(TE')** → **FIRST(T) "+" FIRST(E')**

No FIRST(E') since $T \not\Rightarrow^{*} \varepsilon$

→ **FIRST(T)** → **FIRST(F) "+" FIRST(T')** → **FIRST(F)**

No FIRST(T') since $F \not\Rightarrow^{*} \varepsilon$

→ **FIRST((E)) "+" FIRST(id)** → "(" and "id"

Overall: FIRST(E) = FIRST(T) = FIRST(F) = { (, id }

FIRST(E') = { + , ε }   FIRST(T') = { * , ε }

Compute FOLLOW (for non-terminals)

1. If S is the start symbol ➔ **$** is in FOLLOW(S)   **Initially S$**

2. If $A \rightarrow \alpha B \beta$ is a production rule
➔ everything in **FIRST(β)** is FOLLOW(B) except ε

3. If ($A \rightarrow \alpha B$ is a production rule) or
($A \rightarrow \alpha B \beta$ is a production rule and ε is in FIRST(β))
➔ everything in **FOLLOW(A)** is in FOLLOW(B).
(Whatever followed A must follow B, since nothing follows B from the production rule)

We apply these rules until nothing more can be added to any FOLLOW set.

The Algorithm for FOLLOW – pseudocode

1. Initialize FOLLOW(X) for all non-terminals X to empty set.
   Place $ in FOLLOW(S), where S is the start NT.
2. Repeat the following step until no modifications are made to any Follow-set

   For any production $\mathbf{X \rightarrow X_1\ X_2\ \ldots\ X_j\ X_{j+1}\ \ldots\ X_m}$

   For j=1 to m,

   if $\mathbf{X_j}$ is a non-terminal then:

FOLLOW(X_j)=FOLLOW(X_j)$\cup$(FIRST(X_{j+1},…,X_m)-{ε});

If FIRST(X_{j+1},…,X_m) contains ε or X_{j+1},…,X_m= ε

then **FOLLOW(X_j)=FOLLOW(X_j)$\cup$** FOLLOW(X);

FOLLOW Example

Example :

$E \rightarrow TE'$

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

$T \rightarrow FT'$

$T' \rightarrow *FT' \mid \varepsilon$

$F \rightarrow (E) \mid id$

$$\mathbf{X \rightarrow X_1\ X_2 \ldots X_j\ X_{j+1} \ldots X_m}$$

For j=1 to m,

if $\mathbf{X_j}$ is a non-terminal then:

$\mathrm{FOLLOW}(X_j)=\mathrm{FOLLOW}(X_j) \cup (\mathrm{FIRST}(X_{j+1},\ldots,X_m)-\{\varepsilon\})$;

If $\mathrm{FIRST}(X_{j+1},\ldots,X_m)$ contains ε or $X_{j+1},\ldots,X_m = \varepsilon$

then **$\mathrm{FOLLOW}(X_j)=\mathrm{FOLLOW}(X_j) \cup$** $\mathrm{FOLLOW}(X)$;

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

FIRST(*FT') = {*}
FIRST((E)) = {(}
FIRST(id) = {id}
FIRST(+TE') = {+}
FIRST(ε) = {ε}
FIRST(TE') = {(,id}
FIRST(FT') = {(,id}

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

Constructing LL(1) Parsing Table

Algorithm:

1. Repeat Steps 2 & 3 for each rule $A \rightarrow \alpha$

   2. Terminal a in FIRST(α)? Add $A \rightarrow \alpha$ to M[A, a]

   3.1 ε in FIRST(α)? Add $A \rightarrow \alpha$ to M[A, b] for all terminals b in FOLLOW(A).

   3.2 ε in FIRST(α) and \$ in FOLLOW(A)? Add $A \rightarrow \alpha$ to M[A, \$]

4. All undefined entries are errors.

Constructing LL(1) Parsing Table -- Example

Example:

$E \rightarrow TE'$	FIRST(TE')={(,id}	➔ $E \rightarrow TE'$ into M[E,(] and M[E,id]
$E' \rightarrow +TE'$	FIRST(+TE')={+}	➔ $E' \rightarrow +TE'$ into M[E',+]
$E' \rightarrow \varepsilon$	FIRST(ε)={ε}	➔ none
	but since ε in FIRST(ε)	
	and FOLLOW(E')={$,)}	➔ $E' \rightarrow \varepsilon$ into M[E',$] and M[E',)]
$T \rightarrow FT'$	FIRST(FT')={(,id}	➔ $T \rightarrow FT'$ into M[T,(] and M[T,id]
$T' \rightarrow *FT'$	FIRST(*FT')={*}	➔ $T' \rightarrow *FT'$ into M[T',*]
$T' \rightarrow \varepsilon$	FIRST(ε)={ε}	➔ none
	but since ε in FIRST(ε)	
	and FOLLOW(T')={$,),+}	➔ $T' \rightarrow \varepsilon$ into M[T',$], M[T',)] and M[T',+]
$F \rightarrow (E)$	FIRST((E))={(}	➔ $F \rightarrow (E)$ into M[F,(]
$F \rightarrow id$	FIRST(id)={id}	➔ $F \rightarrow id$ into M[F,id]

Constructing LL(1) Parsing Table (cont.)

Non-terminal	Input symbol					
	id	**+**	*****	**(**	**)**	**$**
E	$E \rightarrow TE'$			$E \rightarrow TE'$		
E'		$E' \rightarrow +TE'$			$E' \rightarrow \varepsilon$	$E' \rightarrow \varepsilon$
T	$T \rightarrow FT'$			$T \rightarrow FT'$		
T'		$T' \rightarrow \varepsilon$	$T' \rightarrow *FT'$		$T' \rightarrow \varepsilon$	$T' \rightarrow \varepsilon$
F	$F \rightarrow id$			$F \rightarrow (E)$		

Parsing table M for grammar

LL(1) Grammars

L : Scan input from Left to Right

L : Construct a Leftmost Derivation

1 : Use “1” input symbol as lookahead in conjunction with stack to decide on the parsing action

LL(1) grammars == they have no multiply-defined entries in the parsing table.

Properties of LL(1) grammars:

- Grammar can’t be ambiguous or left recursive
- Grammar is LL(1) $\Leftrightarrow$ when $A \rightarrow \alpha \mid \beta$

1. α and β do not derive strings starting with the same terminal a
2. Either α or β can derive ε, but not both.
3. If β can derive to ε, then α cannot derive to any string starting with a terminal in FOLLOW(A).

Note: It may not be possible for a grammar to be manipulated into an LL(1) grammar

A Grammar which is not LL(1)

Example:

$S \rightarrow i\ C\ t\ S\ E \mid a$

$E \rightarrow e\ S \mid \varepsilon$

$C \rightarrow b$

FIRST(iCtSE) = {i}
FIRST(a) = {a}
FIRST(eS) = {e}
FIRST(ε) = {ε}
FIRST(b) = {b}

FOLLOW(S) = { $,e }
FOLLOW(E) = { $,e }
FOLLOW(C) = { t }

	a	b	e	i	t	$
S	$S \rightarrow a$			$S \rightarrow iCtSE$		
E			$E \rightarrow e\ S$ $E \rightarrow \varepsilon$			$E \rightarrow \varepsilon$
C		$C \rightarrow b$				

Problem ➔ ambiguity

two production rules for M[E,e]

A Grammar which is not LL(1) (cont.)

- A left recursive grammar cannot be a LL(1) grammar.
  - $A \rightarrow A\alpha \mid \beta$

    ➔ any terminal that appears in FIRST(β) also appears FIRST($A\alpha$) because $A\alpha \Rightarrow \beta\alpha$.

    ➔ If β is ε, any terminal that appears in FIRST(α) also appears in FIRST($A\alpha$) and FOLLOW(A).
- A grammar is not left factored, it cannot be a LL(1) grammar
  - $A \rightarrow \alpha\beta_1 \mid \alpha\beta_2$

    ➔any terminal that appears in FIRST($\alpha\beta_1$) also appears in FIRST($\alpha\beta_2$).
- An ambiguous grammar cannot be a LL(1) grammar.
- What do we have to do if the resulting parsing table contains multiply defined entries?
  - If we didn't eliminate left recursion, eliminate the left recursion in the grammar.
  - If the grammar is not left factored, we have to left factor the grammar.
  - If its (new grammar's) parsing table still contains multiply defined entries, that grammar is ambiguous or it is inherently not a LL(1) grammar.

Error Recovery in Predictive Parsing

- An error may occur in the predictive parsing (LL(1) parsing)
  - if the terminal symbol on the top of stack does not match with the current input symbol.
  - if the top of stack is a non-terminal A, the current input symbol is a, and the parsing table entry M[A,a] is empty.
- What should the parser do in an error case?
  - The parser should be able to give an error message (as much as possible meaningful error message).
  - It should be recover from that error case, and it should be able to continue the parsing with the rest of the input.

Error Recovery Techniques

- Panic-Mode Error Recovery
  - Skipping the input symbols until a synchronizing token is found.
- Phrase-Level Error Recovery
  - Each empty entry in the parsing table is filled with a pointer to a specific error routine to take care that error case.
- Error-Productions (used in GCC etc.)
  - If we have a good idea of the common errors that might be encountered, we can augment the grammar with productions that generate erroneous constructs.
  - When an error production is used by the parser, we can generate appropriate error diagnostics.
  - Since it is almost impossible to know all the errors that can be made by the programmers, this method is not practical.
- Global-Correction
  - Ideally, we would like a compiler to make as few change as possible in processing incorrect inputs.
  - We have to globally analyze the input to find the error.
  - This is an expensive method, and it is not in practice.

Panic-Mode Error Recovery in LL(1) Parsing

- In panic-mode error recovery, we skip all the input symbols until a synchronizing token is found or pop terminal from the stack.
- What is the synchronizing token?
  - All the terminal-symbols in the **follow set** of a non-terminal can be used as a synchronizing token set for that non-terminal.
- So, a simple panic-mode error recovery for the LL(1) parsing:
  - All the empty entries are marked as ***synch*** to indicate that the parser will skip all the input symbols until a symbol in the **follow set** of the non-terminal A which on the top of the stack. Then the parser will pop that non-terminal A from the stack. The parsing continues from that state.
  - To handle unmatched terminal symbols, the parser pops that unmatched terminal symbol from the stack and it issues an error message saying that unmatched terminal is inserted.

Panic-Mode Error Recovery - Example

$S \to AbS \mid e \mid \varepsilon$

$A \to a \mid cAd$

FOLLOW(S)={$}

FOLLOW(A)={b,d}

	a	b	c	d	e	$
S	$S \to AbS$	*sync*	$S \to AbS$	*sync*	$S \to e$	$S \to \varepsilon$
A	$A \to a$	*sync*	$A \to cAd$	*sync*	*sync*	*sync*

stack	input	output
$S	aab$	$S \to AbS$
$SbA	aab$	$A \to a$
$Sba	aab$	
$Sb	ab$	Error: missing b, inserted
$S	ab$	$S \to AbS$
$SbA	ab$	$A \to a$
$Sba	ab$	
$Sb	b$	
$S	$	$S \to \varepsilon$
$	$	accept

stack	input	output
$S	ceadb$	$S \to AbS$
$SbA	ceadb$	$A \to cAd$
$SbdAc	ceadb$	
$SbdA	eadb$	Error:unexpected e (illegal A)
(Remove all input tokens until first b or d, pop A)		
$Sbd	db$	
$Sb	b$	
$S	$	$S \to \varepsilon$
$	$	accept

Motivation Behind FIRST & FOLLOW

FIRST: Is used to help find the appropriate reduction to follow given the top-of-the-stack non-terminal and the current input symbol.

Example: If $A \rightarrow \alpha$, and a is in FIRST(α), then when a=input, replace A with α (in the stack).

(a is one of first symbols of α, so when A is on the stack and a is input, POP A and PUSH α.)

FOLLOW: Is used when FIRST has a conflict, to resolve choices, or when FIRST gives no suggestion. When $\alpha \rightarrow \varepsilon$ or $\alpha \stackrel{*}{\Rightarrow} \varepsilon$, then what follows A dictates the next choice to be made.

Example: If $A \rightarrow \alpha$, and b is in FOLLOW(A), then when $\alpha \stackrel{*}{\Rightarrow} \varepsilon$ and b is an input character, then we expand A with α , which will eventually expand to ε, of which b follows!

($\alpha \stackrel{*}{\Rightarrow} \varepsilon$: i.e., FIRST(α) contains ε.)

Compute FIRST for Any String X

1. If X is a terminal, FIRST(X) = {X}
2. If $X \rightarrow \varepsilon$ is a production rule, add ε to FIRST (X)
3. If X is a non-terminal, and $X \rightarrow Y_1 Y_2 \ldots Y_k$ is a production rule

Place FIRST(Y_1) in FIRST (X)

if $Y_1 \Rightarrow \varepsilon$, Place FIRST ($Y_2$) in FIRST(X)

if $Y_2 \Rightarrow \varepsilon$, Place FIRST($Y_3$) in FIRST(X)

…

if $Y_{k-1} \Rightarrow \varepsilon$, Place FIRST($Y_k$) in FIRST(X)

NOTE: As soon as $Y_i \overset{*}{\not\Rightarrow} \varepsilon$, Stop.

Repeat above steps until no more elements are added to any FIRST( ) set.

Checking "$Y_i \Rightarrow \varepsilon$?" essentially amounts to checking whether ε belongs to FIRST(Y_i)

Computing FIRST(X) : All Grammar Symbols - continued

Informally, suppose we want to compute

$$\text{FIRST}(X_1\ X_2 \ldots X_n) = \text{FIRST}(X_1)$$

$$\text{"+"}\ \text{FIRST}(X_2) \quad \text{if } \varepsilon \text{ is in FIRST}(X_1)$$

$$\text{"+"}\ \text{FIRST}(X_3) \quad \text{if } \varepsilon \text{ is in FIRST}(X_2)$$

$$\ldots$$

$$\text{"+"}\ \text{FIRST}(X_n) \quad \text{if } \varepsilon \text{ is in FIRST}(X_{n-1})$$

Note 1: Only add ε to FIRST $(X_1\ X_2 \ldots X_n)$ if ε is in FIRST (X_i) for all i

Note 2: For FIRST(X_i), if $X_i \rightarrow Z_1\ Z_2 \ldots Z_m$, then we need to compute FIRST($Z_1\ Z_2 \ldots Z_m$) !

Compute FOLLOW (for non-terminals)

1. If S is the start symbol ➔ **$** is in FOLLOW(S) **Initially S$**

2. If $A \rightarrow \alpha B \beta$ is a production rule
➔ everything in **FIRST(β)** is FOLLOW(B) except ε

3. If ($A \rightarrow \alpha B$ is a production rule) or
($A \rightarrow \alpha B \beta$ is a production rule and ε is in FIRST(β))
➔ everything in **FOLLOW(A)** is in FOLLOW(B).
(Whatever followed A must follow B, since nothing follows B from the production rule)

We apply these rules until nothing more can be added to any FOLLOW set.

Constructing LL(1) Parsing Table

Algorithm:

1. Repeat Steps 2 & 3 for each rule $A \rightarrow \alpha$

   2. Terminal a in FIRST(α)? Add $A \rightarrow \alpha$ to M[A, a]

   3.1 ε in FIRST(α)? Add $A \rightarrow \alpha$ to M[A, b] for all terminals b in FOLLOW(A).

   3.2 ε in FIRST(α) and \$ in FOLLOW(A)? Add $A \rightarrow \alpha$ to M[A, \$]

4. All undefined entries are errors.

Answer

1 lexp → atom
2 lexp → list
3 atom →num
4 atom →id

5 list → (lexp_seq)
6 lexp_seq → lexp lexp_seq'
7 lexp_seq' → lexp lexp_seq'
8 lexp_seq' → ε

	num	id	(	)	$
lexp	1	1	2		
atom	3	4			
list			5		
lexp_seq	6	6	6		
lexp_seq'	7	7	7	8	

No conflict，Grammar is LL(1)

Panic-Mode Error Recovery - Example

$S \rightarrow AbS \mid e \mid \varepsilon$
$A \rightarrow a \mid cAd$

FOLLOW(S)={$}
FOLLOW(A)={b,d}

	a	b	c	d	e	$
S	$S \rightarrow AbS$	***sync***	$S \rightarrow AbS$	***sync***	$S \rightarrow e$	$S \rightarrow \varepsilon$
A	$A \rightarrow a$	***sync***	$A \rightarrow cAd$	***sync***	***sync***	***sync***

<u>**stack**</u>	<u>**input**</u>	<u>**output**</u>
$S	aab$	$S \rightarrow AbS$
$SbA	aab$	$A \rightarrow a$
$Sba	aab$	
$Sb	ab$	Error: missing b, inserted
$S	ab$	$S \rightarrow AbS$
$SbA	ab$	$A \rightarrow a$
$Sba	ab$	
$Sb	b$	
$S	$	$S \rightarrow \varepsilon$
$	$	accept

<u>**stack**</u>	<u>**input**</u>	<u>**output**</u>
$S	ceadb$	$S \rightarrow AbS$
$SbA	ceadb$	$A \rightarrow cAd$
$SbdAc	ceadb$	
$SbdA	eadb$	Error:unexpected e (illegal A)
(Remove all input tokens until first b or d, pop A)		
$Sbd	db$	
$Sb	b$	
$S	$	$S \rightarrow \varepsilon$
$	$	accept

Phrase-Level Error Recovery

- Each empty entry in the parsing table is filled with a pointer to a special error routine which will take care that error case.
- These error routines may:
  - change, insert, or delete input symbols.
  - issue appropriate error messages
  - pop items from the stack.
- We should be careful when we design these error routines, because we may put the parser into an infinite loop.

Final Comments – Top-Down Parsing

So far,

- **We've examined grammars and language theory and its relationship to parsing**
- **Key concepts: Rewriting grammar into an acceptable form**
- **Examined Top-Down parsing:**

  Brute Force : Recursion and backtracking

  Elegant : Table driven

- **We've identified its shortcomings:**

  Not all grammars can be made LL(1) !

- **Bottom-Up Parsing - Future**

Bottom-Up Parsing

- Goal: creates the parse tree of the given input starting from leaves towards the root.
- How: construct the right-most derivation of the given input in the reverse order.

$$S \Rightarrow r_1 \Rightarrow \ldots \Rightarrow r_n \Rightarrow \omega \quad \text{(the right-most derivation of } \omega\text{)}$$

$$\leftarrow \quad \text{(finds the right-most derivation in the reverse order)}$$

- Techniques:
  - General technique: **shift-reduce parsing**
    - ✓Shift: pushes the current symbol in the input to a stack.
    - ✓Reduction: replaces the symbols $X_1X_2\ldots X_n$ at the top of the stack by A if A ➔ $X_1X_2\ldots X_n$.
  - LR parsers (SLR, LR, LALR)

Bottom-Up Parsing -- Example

$S \rightarrow aABb$
$A \rightarrow aA \mid a$
$B \rightarrow bB \mid b$

input string: `aaabb`

`aaAbb`

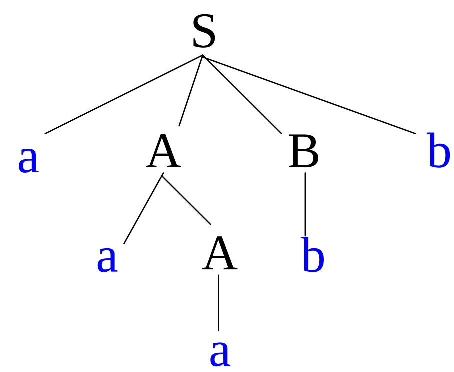
`aAbb` $\Downarrow$ reduction

`aABb`

`S`

$$S \underset{rm}{\Rightarrow} aABb \underset{rm}{\Rightarrow} aAbb \underset{rm}{\Rightarrow} aaAbb \underset{rm}{\Rightarrow} aaabb$$

Right Sentential Forms



Naive algorithm

$$S=\gamma_0 \underset{rm}{\Rightarrow} \gamma_1 \underset{rm}{\Rightarrow} \gamma_2 \underset{rm}{\Rightarrow} \dots \underset{rm}{\Rightarrow} \gamma_{n-1} \underset{rm}{\Rightarrow} \gamma_n = \omega$$

input string

Let ω = input string

repeat

pick a non-empty substring β of ω where $A \rightarrow \beta$ is a production

if (no such β)

backtrack

else

replace one β by A in ω

until ω = "S" (the start symbol) or all possibilities are exhausted

Questions

Let ω = input string

repeat

  pick a non-empty substring β of ω where $A \rightarrow \beta$ is a production

  if (no such β)

    backtrack

  else

    replace one β by A in ω

until ω = "S" (the start symbol) or all possibilities are exhausted

- Does this algorithm terminate?
- How fast is the algorithm?
- Does the algorithm handle all cases?
- How do we choose the substring to reduce at each step?

Important facts

- Important Fact #1
  - Let $\alpha\beta\omega$ be a step of a bottom-up parse
  - Assume the next reduction is by $A \rightarrow \beta$
  - Then ω must be a string of terminals

Why?

Because $\alpha A\omega \rightarrow \alpha\beta\omega$ is a step in a rightmost derivation

- Important Fact #2
  - Let $\alpha A\omega$ be a step of a bottom-up parse
  - β is replaced by A
  - The next reduction will not occur at left side of A

Why?

Because $\alpha A\omega \rightarrow \alpha\beta\omega$ is a step in a rightmost derivation

Handle

- Informally, a **handle of a string is a substring that matches the right side of a production rule**.
  - But not every substring matches the right side of a production rule is handle

- A **handle** of a right sentential form γ ($\equiv \alpha\beta\omega$) is (t, p)
  - t: a production $A \rightarrow \beta$
  - p: a position of γ

    where the string β may be found and replaced by A to produce the previous right-sentential form in a rightmost derivation of γ.

$$S \overset{rm}{\Rightarrow}^{*} \alpha \mathbf{A} \omega \overset{rm}{\Rightarrow} \alpha \boldsymbol{\beta} \omega$$

- If the grammar is unambiguous, then every right-sentential form of the grammar has exactly one handle.

Why?

Handle

If the grammar is unambiguous,

then every right-sentential form of the grammar has exactly one handle.

- **Proof idea:**

  The grammar is unambiguous

  $\Rightarrow$rightmost derivation is unique

  $\Rightarrow$a unique production $A \rightarrow \beta$ applied to take r_i to r_{i+1} at the position k

  $\Rightarrow$a unique handle $(A \rightarrow \beta, k)$

Handle Pruning

- A right-most derivation in reverse can be obtained by **handle-pruning**.

$$S=\gamma_0 \underset{rm}{\Rightarrow} \gamma_1 \underset{rm}{\Rightarrow} \gamma_2 \underset{rm}{\Rightarrow} \cdots \underset{rm}{\Rightarrow} \gamma_{n-1} \underset{rm}{\Rightarrow} \gamma_n = \omega$$

input string

handle-pruning

Let ω = input string

repeat

pick a non-empty substring β of ω where $A \rightarrow \beta$ is a production

if (no such β)

backtrack

else

replace one β by A in ω

until ω = "S" (the start symbol) or all possibilities are exhausted

handle ($A \rightarrow \beta$, pos)

Example

$E \rightarrow E+T \mid T$ x+y * z

$T \rightarrow T*F \mid F$ id+id*id

$F \rightarrow (E) \mid id$

Right-Most Sentential Form	Handle
$\underline{id}+id*id$	$F \rightarrow id$, 1
$\underline{F}+id*id$	$T \rightarrow F$, 1
$\underline{T}+id*id$	$E \rightarrow T$, 1
$E+\underline{id}*id$	$F \rightarrow id$, 3
$E+\underline{F}*id$	$T \rightarrow F$, 3
$E+T*\underline{id}$	$F \rightarrow id$, 5
$E+\underline{T*F}$	$T \rightarrow T*F$, 3
$\underline{E+T}$	$E \rightarrow E+T$, 1
E	

Shift-Reduce Parser

- Shift-reduce parsers require a stack and an input buffer
  - Initial stack just contains only the end-marker **$**
  - The end of the input string is marked by the end-marker **$**.

- There are four possible actions of a shift-parser action:

1. **Shift** : The next input symbol is shifted onto the top of the stack.
2. **Reduce**: Replace the handle on the top of the stack by the non-terminal.
3. **Accept**: Successful completion of parsing.
4. **Error**: Parser discovers a syntax error, and calls an error recovery routine.

Example

$E \rightarrow E+T \mid T$

$T \rightarrow T*F \mid F$

$F \rightarrow (E) \mid id$

x+y * z

id+id*id

Stack	Input	Action
$	id+id*id$	shift
$id	+id*id$	reduce by $F \rightarrow id$
$F	+id*id$	reduce by $T \rightarrow F$
$T	+id*id$	reduce by $E \rightarrow T$
$E	+id*id$	shift
$E+	id*id$	shift
$E+id	*id$	reduce by $F \rightarrow id$
$E+F	*id$	reduce by $T \rightarrow F$
$E+T	*id$	shift
$E+T*	id$	shift
$E+T*id	$	reduce by $F \rightarrow id$
$E+T*F	$	reduce by $T \rightarrow T*F$
$E+T	$	reduce by $E \rightarrow E+T$
$E	$	accept

Handle

$F \rightarrow id$, 1

$T \rightarrow F$, 1

$E \rightarrow T$, 1

$F \rightarrow id$, 3

$T \rightarrow F$, 3

$F \rightarrow id$, 5

$T \rightarrow T*F$, 3

$E \rightarrow E+T$, 1

F 6

id

$$E \Rightarrow E+T \Rightarrow E+T*F \Rightarrow E+T*id \Rightarrow E+F*id$$
$$\Rightarrow E+id*id \Rightarrow T+id*id \Rightarrow F+id*id \Rightarrow id+id*id$$

Question

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

Ex. id + id * id

Stack	**Input**	**Action**
?	?	?

Question

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

Ex. id + id * id

Stack	Input	Action
$	id+id*id$	shift
$id	+id*id$	reduce by E → id
$E	+id*id$	shift
$E+	id*id$	shift
$E+id	*id$	reduce by E → id
$E+E	*id$	reduce by E → E+E
$E	*id$	shift
$E*	id$	shift
$E*id	$	reduce by E → id
$E*E	$	reduce by E → E*E
$E	$	accept

Stack	Input	Action
$	id+id*id$	shift
$id	+id*id$	reduce by E → id
$E	+id*id$	shift
$E+	id*id$	shift
$E+id	*id$	reduce by E → id
$E+E	*id$	shift
$E+E*	*id$	shift
$E+E*id	id$	reduce by E → id
$E+E*E	$	reduce by E → E*E
$E+E	$	reduce by E → E+E
$E	$	accept

shift/reduce conflict

Question

$S \rightarrow aA \mid aB$, $A \rightarrow c$, $B \rightarrow c$

Ex. ac

Stack	Input	Action
\$	ac\$	shift
\$a	c\$	shift
\$ac	\$	reduce by which?

reduce/reduce conflict

When Shift-Reduce Parser fail

- There are no known efficient algorithms to recognize handles
- Stack contents and the next input symbol may not decide action:
  - **shift/reduce conflict**: Whether make a shift operation or a reduction.
    - ✓Option 1: modify the grammar to eliminate the conflict
    - ✓Option 2: resolve in favor of shifting

    Classic examples: "dangling else" ambiguity, insufficient associativity or precedence rules
  - **reduce/reduce conflict**: The parser cannot decide which of several reductions to make
    - ✓Often, no simple resolution
    - ✓Option 1: try to redesign grammar, perhaps with changes to language
    - ✓Option 2: use context information during parse (e.g., symbol table)

    Classic real example: call and subscript: id(id, id)

    When Stack = . . . id (id , input = id) . . .

    Reduce by expr → id, or Reduce by param → id

The role of precedence and associativity

- Precedence and associativity rules can be used to resolve shift/reduce conflicts in ambiguous grammars:
  - lookahead with higher precedence ⇒ shift
  - same precedence, left associative ⇒ reduce
- Alternative to encoding them in the grammar

Example

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

- ✓ * high precedence than +
- ✓ * and + left associativity

Stack	Input	Action
$	id+id*id$	shift
$id	+id*id$	reduce by E → id
$E	+id*id$	shift
$E+	id*id$	shift
$E+id	*id$	reduce by E → id
$E+E	*id$	reduce by E → E+E
$E	*id$	shift
$E*	id$	shift
$E*id	$	reduce by E → id
$E*E	$	reduce by E → E*E
$E	$	accept

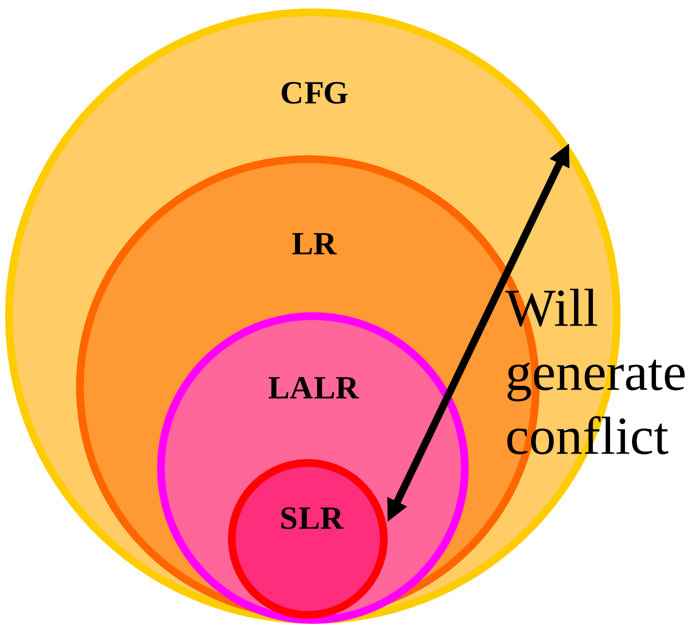
Stack	Input	Action
$	id+id*id$	shift
$id	+id*id$	reduce by E → id
$E	+id*id$	shift
$E+	id*id$	shift
$E+id	*id$	reduce by E → id
$E+E	*id$	shift
$E+E*	*id$	shift
$E+E*id	id$	reduce by E → id
$E+E*E	$	reduce by E → E*E
$E+E	$	reduce by E → E+E
$E	$	accept

shift/reduce conflict

Shift-Reduce Parsers

1. LR-Parsers

- covers wide range of grammars.
  - **SLR** – simple LR parser
  - **LR** – most general LR parser
  - **LALR** – intermediate LR parser (lookahead LR parser)
- SLR, LR and LALR work same, only their parsing tables are different.



LR Parsers

- The most powerful shift-reduce parsing (yet efficient) is:

LR(k) parsing.

left to right scanning

right-most derivation

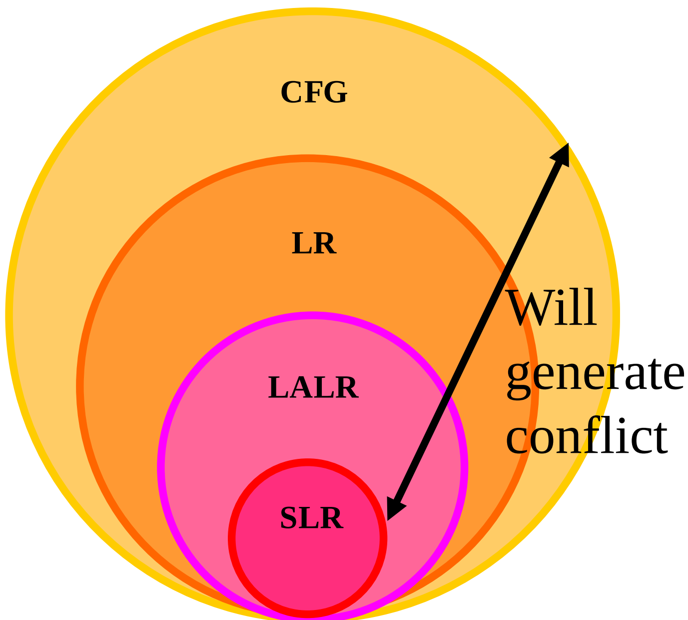
k lookahead (k is omitted ➔ it is 1)

- LR parsing is attractive because:
  - LR parsing is most general non-backtracking shift-reduce parsing, yet it is still efficient.
  - The class of grammars that can be parsed using LR methods is a proper superset of the class of grammars that can be parsed with predictive parsers.

    LL(1)-Grammars $\subset$ LR(1)-Grammars
  - An LR-parser can detect a syntactic error as soon as it is possible to do so a left-to-right scan of the input.

Shift-Reduce Parsers

1. **LR-Parsers**
   - covers wide range of grammars.
     - **SLR** – simple LR parser
     - **LR** – most general LR parser
     - **LALR** – intermediate LR parser (lookahead LR parser)
   - SLR, LR and LALR work same, only their parsing tables are different.



A Trivial Bottom-Up Parsing Algorithm

Let ω = input string

repeat

pick a non-empty substring β of ω where $A \rightarrow \beta$ is a production

if (no such β)

backtrack

else

replace one β by A in ω

until ω = "S" (the start symbol) or all possibilities are exhausted

We only want to reduce at handles

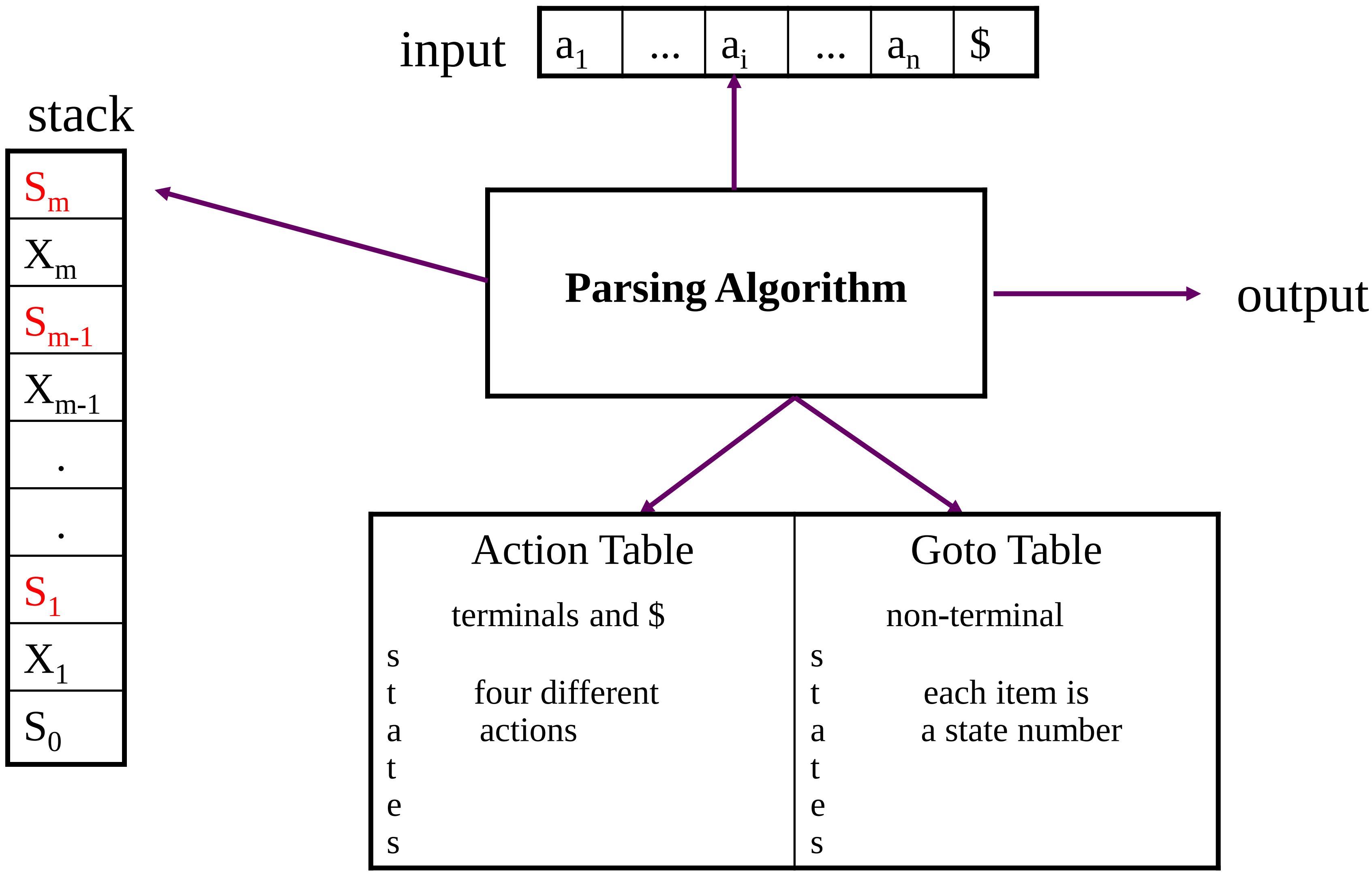
We have defined handles

Stack is: …T, input *…

But, how to detect handles ?

LR Parsers avoid backtrack

LR Parsing Algorithm Framework



Viable Prefix

- At each step the parser sees:

  The stack content must be a prefix of a right-sentential form
- $E \Rightarrow \ldots \Rightarrow F*id \Rightarrow (E)*\ id$
  — (, (E, (E) are prefix of (E)* id
- But, not all prefix can be the stack content

  — (E)* cannot be appear on the stack, it should be reduced by F->(E)

- A viable prefix is a prefix of a right-sentential form that can appear on the stack
- If the bottom-up parser enforces that the stack can only hold viable prefix, then, we do not backtrack

Observations on Viable Prefix

- A viable prefix does not extend past the right end of handle

  stack: $a_1a_2\ldots a_n$ input: $r_1r_2\omega$

  either $a_i\ldots a_n$ is a handle, or there is no handle

  Why?

  $a_1a_2\ldots a_{n-k}\, A \mid r_2\omega \Rightarrow a_1a_2a_{n-k}a_{n-k+1}\ldots a_n \mid r_1r_2\ \omega$ by $A \rightarrow a_{n-k+1}\ldots a_n\ r_1$

- A production A-> $\beta_1\beta_2$ is valid for viable prefix $\alpha\beta_1$ if

$$S \Rightarrow^* \alpha A\omega \Rightarrow \alpha\ \beta_1\beta_2\ \omega$$

- Reduce: if $\beta_2 = \varepsilon$
- Shift: if $\beta_2 \mathrel{!}= \varepsilon$
- Item: A-> $\beta_1 \bullet \beta_2$

LR(0) Items and LR(0) Automaton

- How to compute viable prefix and its corresponding valid productions?

Key point: the set of viable prefixes is a regular language

- We construct a finite state automaton recognizing the set of viable prefixes,
  - each state s represents a set of valid items if the initial state s_0 goes to the state s items after reading the viable prefix ω

LR(0) Items and LR(0) Automaton

- LR(0) Items: One production produces a set of items by placing . in to productions
- A production A-> $\beta_1\beta_2$ is valid for viable prefix $\alpha\beta_1$ if

$$S \Rightarrow^* \alpha A\omega \Rightarrow \alpha\, \beta_1\beta_2\, \omega$$

- Reduce: if $\beta_2 = \varepsilon$ Shift: if $\beta_2 \mathrel{!}= \varepsilon$
- Item: A-> $\beta_1 \bullet \beta_2$

E.g. The production T-> (E) gives items

- T-> .(E) // we have seen ε of T-> (E) , shift
- T->(. E) // we have seen (of T-> (E) , shift
- T->(E .) // we have seen (E of T-> (E), shift
- T->(E) . // we have seen (E) of T-> (E), reduce

The production T-> ε gives the item

- T-> .

LR(0) Items and LR(0) Automaton

- The stack may have many prefixes of productions

$$Prefix_1 Prefix_2 \ldots Prefix_{n-1} Prefix_n$$

Let $Prefix_n$ be a valid prefix of $A_n \to \alpha_n$

  - $Prefix_n$ will eventually reduce to A_n
  - $Prefix_{n-1} A_n$ is a valid prefix of $A_{n-1} \to Prefix_{n-1} A_n \beta$

- Finally, $Prefix_k \ldots Prefix_n$ eventually reduces to A_k
  - $Prefix_{k-1} A_k$ is a valid prefix of $A_{k-1} \to Prefix_{k-1} A_k \beta$

LR(0) Items and LR(0) Automaton

- Consider: (id * id) Stack: (id* input id)
  - id * is a viable prefix of T->id * T
  - ε is a viable prefix of E->T
  - (is a viable prefix of T->(E)

Example:

$E \rightarrow T + E \mid T$

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

A state (a set of items) {T->(.E), E-> .T, T->id *.T} says:

- We have seen (of T->(E)
- We have seen ε of E->T
- We have seen id * of T->id * T

Closure Operation: States

- If ***I*** is a set of LR(0) items for a grammar G, then ***closure(I)*** is the set of LR(0) items constructed from I by the two rules:

  1. Initially, every LR(0) item in I is added to closure(I).

  2. Repeat

  - **If $A \rightarrow \alpha.B\beta$ is in closure(I) and $B\rightarrow\gamma$ is a production in G;**
  - **then $B\rightarrow.\gamma$ will be in the closure(I).**

  Unitl no more new LR(0) items can be added to closure(I).

- The stack may have many prefixes of productions

$$\text{Prefix}_1\text{Prefix}_2\ldots\text{Prefix}_{n-1}\text{Prefix}_n$$

$\text{Prefix}_k\ldots\text{Prefix}_n$ eventually reduces to A_k

– $\text{Prefix}_{k-1}A_k$ is a valid prefix of $A_{k-1}\text{->}\ \text{Prefix}_{k-1}A_k\beta$

Closure Operation -- Example

Grammar	I_0: closure({E' → •E}) =
E' → E	{ E' → •E
E → E+T	E → •E+T
E → T	E → •T
T → T*F	T → •T*F
T → F	T → •F
F → (E)	F → •(E)
F → id	F → •id }

← kernel items

1. kernel items: the initial item and all items whose dots are not at the left end
2. Nonkernel items: otherwise

1. Initially, every LR(0) item in I is added to closure(I).

2. If A → α . Bβ is in closure(I) and B→γ is a production rule of G; then B→ . γ will be in the closure(I).

Closure Operation -- Example

$E' \rightarrow E$

$E \rightarrow E+T$

$E \rightarrow T$

$T \rightarrow T*F$

$T \rightarrow F$

$F \rightarrow (E)$

$F \rightarrow id$

I_0: closure($\{E \rightarrow E+\bullet T\}$) =

$\{\ E \rightarrow E+\bullet T$ ← kernel items

$T \rightarrow \bullet T*F$

$T \rightarrow \bullet F$

$F \rightarrow \bullet (E)$

$F \rightarrow \bullet id\ \}$

1. kernel items: the initial item and all items whose dots are not at the left end
2. Nonkernel items: otherwise

1. **Initially, every LR(0) item in I is added to closure(I).**
2. **If $A \rightarrow \alpha.B\beta$ is in closure(I) and $B \rightarrow \gamma$ is a production rule of G; then $B \rightarrow .\gamma$ will be in the closure(I).**

Goto Operation: Transitions

- If I is a set of LR(0) items and X is a grammar symbol (terminal or non-terminal), then **goto(I,X)** is defined as follows:
  - **If $A \rightarrow \alpha \bullet X\beta$ in I**
  - **then every item in closure($\{A \rightarrow \alpha X \bullet \beta\}$) will be in goto(I,X).**

Example:

$I = \{ E' \rightarrow \bullet E,\ \ E \rightarrow \bullet E+T,\ \ E \rightarrow \bullet T,\ \ T \rightarrow \bullet T*F,$
$\quad T \rightarrow \bullet F,\ \ F \rightarrow \bullet (E),\ \ F \rightarrow \bullet id \ \}$

$goto(I,E) = \{ E' \rightarrow E \bullet,\ E \rightarrow E \bullet +T \}$
$goto(I,T) = \{ E \rightarrow T \bullet,\ T \rightarrow T \bullet *F \}$
$goto(I,F) = \{ T \rightarrow F \bullet \}$
$goto(I,() = \{ F \rightarrow (\bullet E),\ E \rightarrow \bullet E+T,\ E \rightarrow \bullet T,\ T \rightarrow \bullet T*F,\ T \rightarrow \bullet F,$
$\quad F \rightarrow \bullet (E),\ F \rightarrow \bullet id \ \}$
$goto(I,id) = \{ F \rightarrow id \bullet \}$

LR(0) Automaton

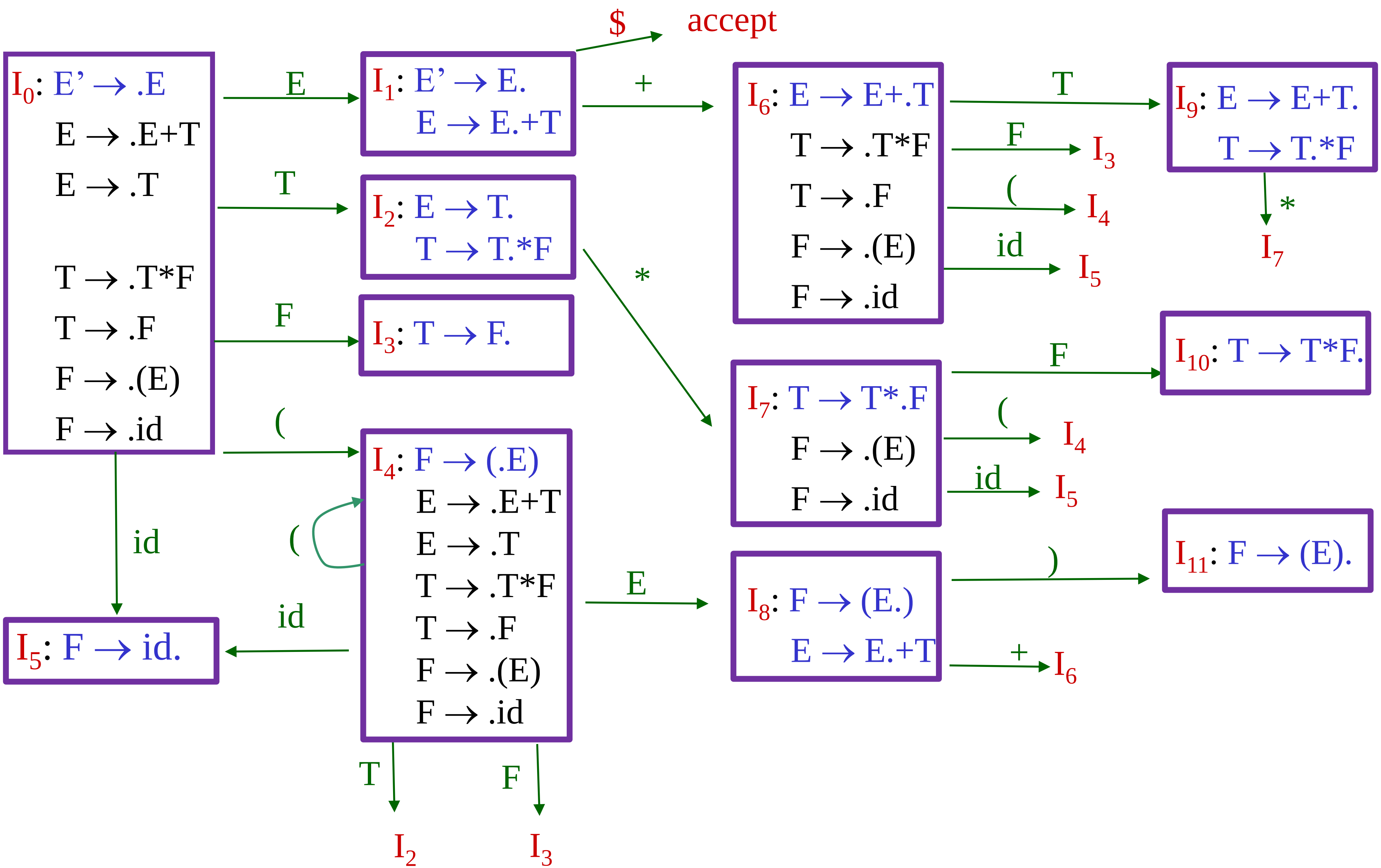
- Add a dummy production S'->S into G, get G'
- ***Algorithm***:

  C is { closure({S'→•S}) }
  repeat
      for each I in ***C*** and each grammar symbol X
          if goto(I,X) is not empty and not in ***C***
              add goto(I,X) to ***C***
  until no more set of LR(0) items can be added to ***C***.

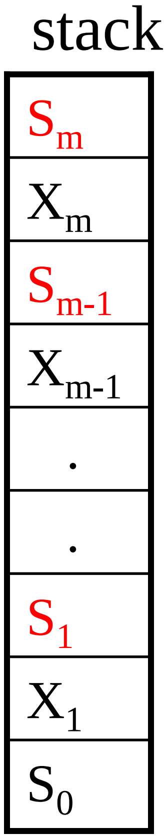
New item, corresponding to new state in DFA

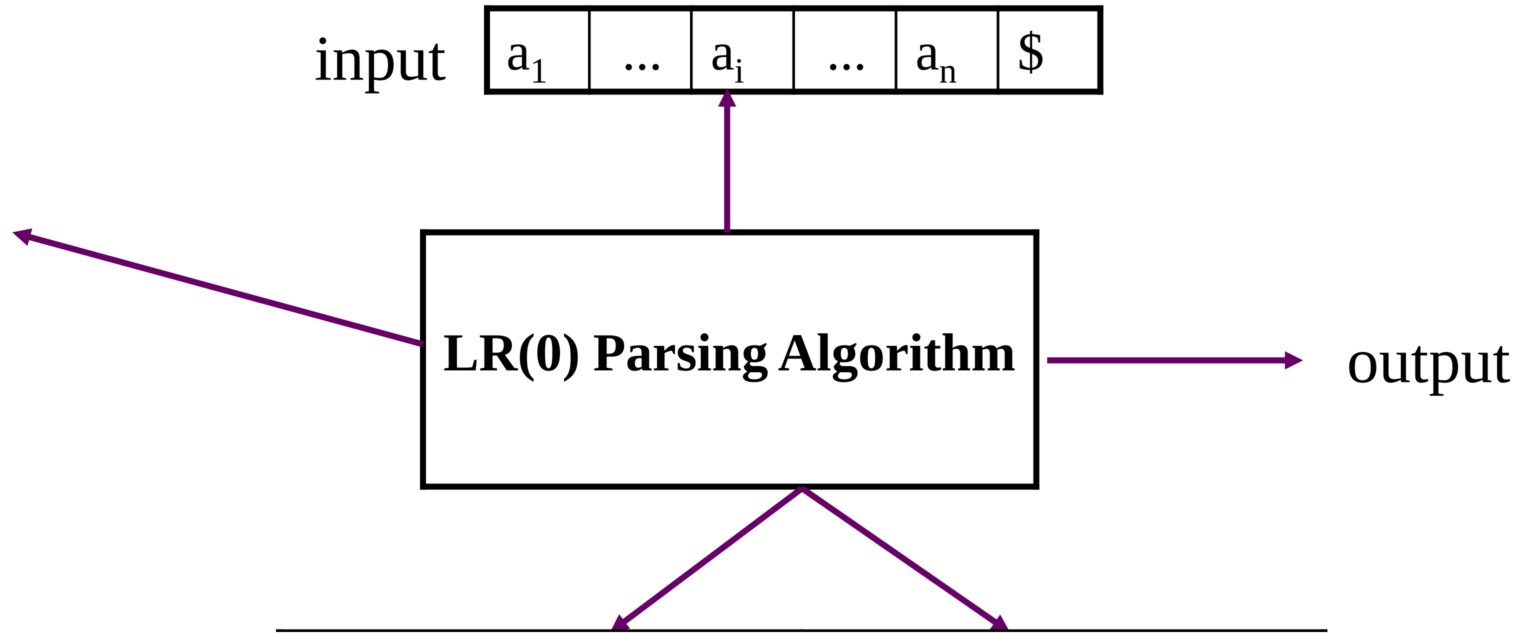
- goto function is the transition function of the DFA
- Initial state: closure({S'->S})

The Canonical LR(0) Collection -- Example



LR(0) Parsing Algorithm Implmentation





Idea: Assume: stack contains β and next input is a

DFA on input β terminates in state s

- Reduce by $X \rightarrow \beta$
  - if s contains item $X \rightarrow \beta\bullet$
- Shift if s contains item $X \rightarrow \beta\bullet a\omega$, i.e. DFA has (s, a, s')

How to update state?

A Configuration of LR(0) Parsing Algorithm

- A configuration of a LR(0) parsing is:

$$(\underbrace{S_o\ X_1\ S_1\ \dots\ X_m\ S_m}_{\text{Stack}},\ \underbrace{a_i\ a_{i+1}\ \dots\ a_n\ \$}_{\text{Rest of Input}})$$

- S_m and a_i decides the parser action by consulting the parsing action table. (***Initial Stack* contains just S_o**)

- A configuration of a LR(0) parsing represents the right sentential form:

$$X_1\ \dots\ X_m\ a_i\ a_{i+1}\ \dots\ a_n\ \$$$

Constructing LR(0) Parsing Table

(of an augumented grammar G')

1. Construct the canonical collection of sets of LR(0) items for G'.
   $C \leftarrow \{I_0, \ldots, I_n\}$

2. Create the parsing action table as follows:
   - If a is a terminal, $A \to \alpha . a\beta$ in I_i and $goto(I_i, a) = I_j$ then action[i,a] is ***shift j***.
   - If $A \to \alpha .$ is in I_i, then action[i,a] is ***reduce*** $\boldsymbol{A \to \alpha}$.
   - If $S' \to S .$ is in I_i, then action[i,\$] is ***accept***.
   - If any conflicting actions generated by these rules, the grammar is not LR(0).

3. Create the parsing goto table :
   - for all non-terminals A, if $goto(I_i, A) = I_j$ then goto[i,A]=j

4. All entries not defined by (2) and (3) are errors.

5. Initial state of the parser contains $S' \to .S$

Actions of A LR(0)-Parser

1. **shift s** -- shifts the next input symbol a_i and the state **S** onto the stack (s_m where m is a state number)

   $$(S_o\ X_1\ S_1 \ldots X_m\ S_m,\ a_i\ a_{i+1} \ldots a_n\ \$) \Rightarrow (S_o\ X_1\ S_1 \ldots X_m\ S_m\ a_i\ S,\ a_{i+1} \ldots a_n\ \$)$$

2. **reduce $A \rightarrow \beta$**
   - pop $2|\beta|$ (=r) items from the stack;
   - then push **A** and **S** where **S=goto[S_{m-r},A]**

   $$(S_o\ X_1\ S_1 \ldots X_m\ S_m,\ a_i\ a_{i+1} \ldots a_n\ \$) \Rightarrow (S_o\ X_1\ S_1 \ldots X_{m-r}\ S_{m-r}\ A\ S,\ a_i \ldots a_n\ \$)$$

   - Output is the reducing production reduce $A \rightarrow \beta$, (and others)

3. **Accept** – Parsing successfully completed

4. **Error** -- Parser detected an error (an empty entry in the action table)

Parsing Tables of Expression Grammar

1) $E \rightarrow E+T$
2) $E \rightarrow T$
3) $T \rightarrow T*F$
4) $T \rightarrow F$
5) $F \rightarrow (E)$
6) $F \rightarrow id$

LR(0) Action Table / Goto Table

state	id	+	*	(	)	$		E	T	F
0	s5			s4				1	2	3
1		s6				acc				
2	r2	r2	s7 r2	r2	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	r1	s7 r1	r1	r1	r1				
10		r3	r3		r3	r3				
11		r5	r5		r5	r5				

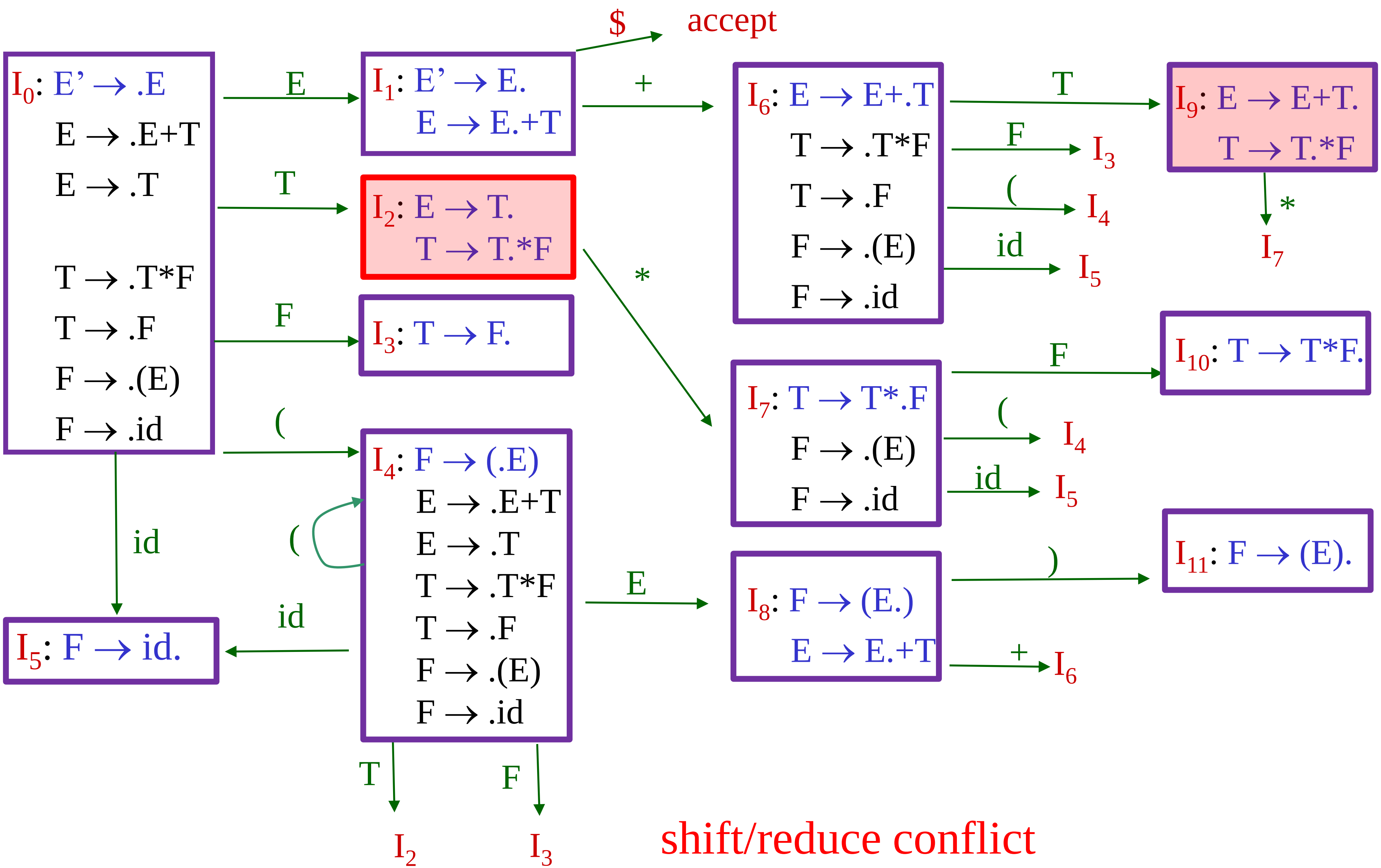
LR(0) Conflicts

- LR(0) has a reduce/reduce conflict if:
  - Any state has two reduce items:
  - E.g., $X \rightarrow \beta.$ and $Y \rightarrow \omega.$

- LR(0) has a shift/reduce conflict if:
  - Any state has a reduce item and a shift item:
  - E.g., $X \rightarrow \beta.$ and $Y \rightarrow \omega.t\delta$

G is in LR(0) Grammar if no conflict

LR(0) Conflicts



shift/reduce conflict

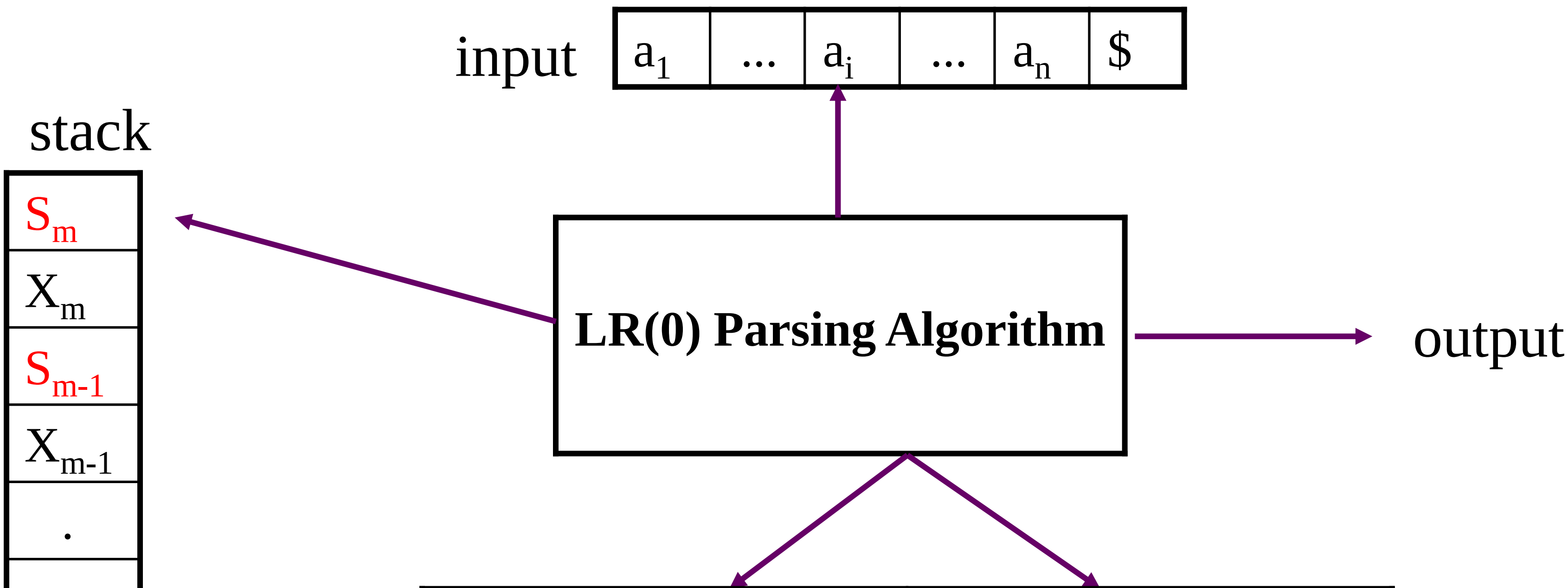
Parsing Tables of Expression Grammar

1) $E \rightarrow E+T$
2) $E \rightarrow T$
3) $T \rightarrow T*F$
4) $T \rightarrow F$
5) $F \rightarrow (E)$
6) $F \rightarrow id$

LR(0) Action Table / Goto Table

state	id	+	*	(	)	$		E	T	F
0	s5			s4				1	2	3
1		s6				acc				
2	r2	r2	s7 r2	r2	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	r1	s7 r1	r1	r1	r1				
10		r3	r3		r3	r3				
11		r5	r5		r5	r5				

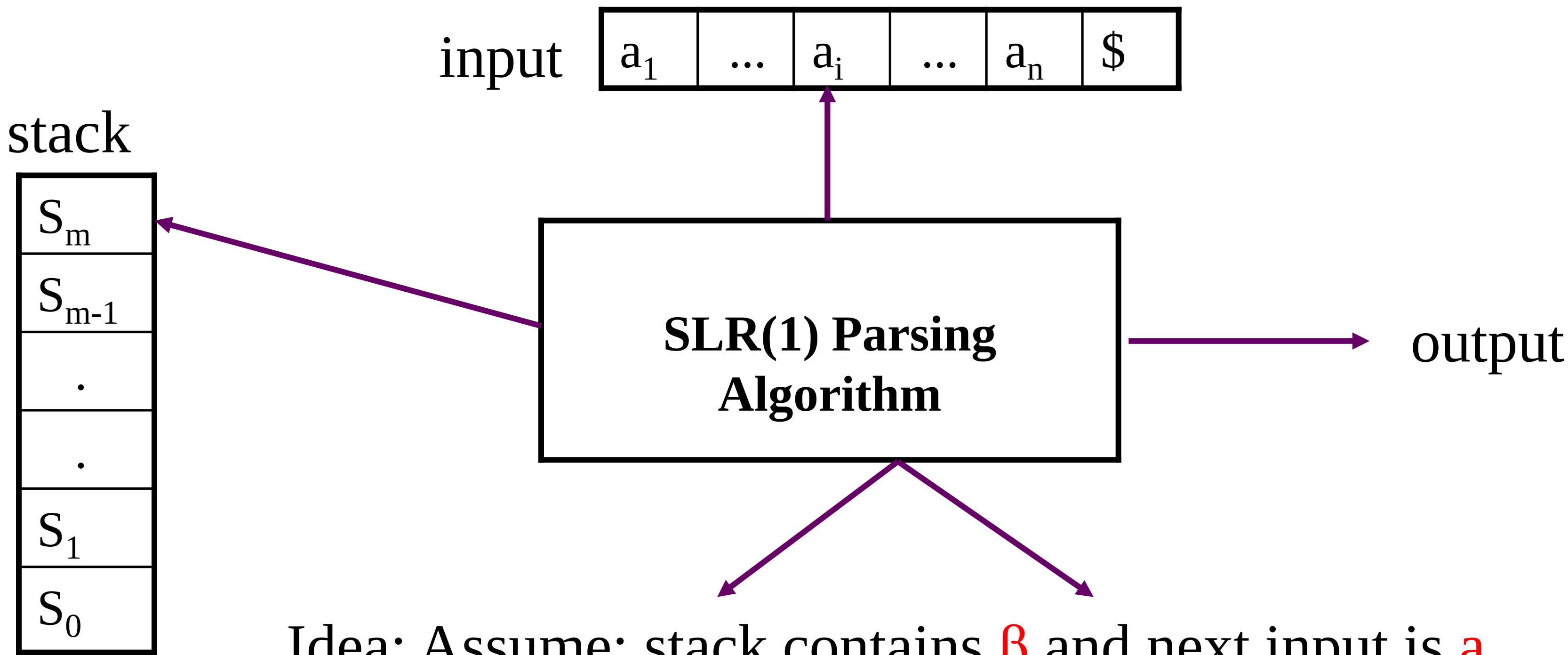
LR(0) Parsing Algorithm Implmentation



Idea: Assume: stack contains β and next input is a
DFA on input β terminates in state s

- Reduce by $X \rightarrow \beta$
  - if s contains item $X \rightarrow \beta\bullet$
- Shift if s contains item $X \rightarrow \beta\bullet a\bar{\omega}$, i.e. DFA has ($s$, a, s')

SLR(1) Parsing Algorithm



Idea: Assume: stack contains β and next input is a

DFA on input β terminates in state s

- Reduce by $X \rightarrow \beta$
  - if **a in FOLLOW(X)** (and $X \rightarrow \beta.$ in s)
- Shift if s contains item $X \rightarrow \beta.a\omega$,
  i.e. DFA has (s, a, s')

Constructing SLR(1) Parsing Table)

(of an augumented grammar G')

1. Construct the canonical collection of sets of LR(0) items for G'.
   $C \leftarrow \{I_0,...,I_n\}$

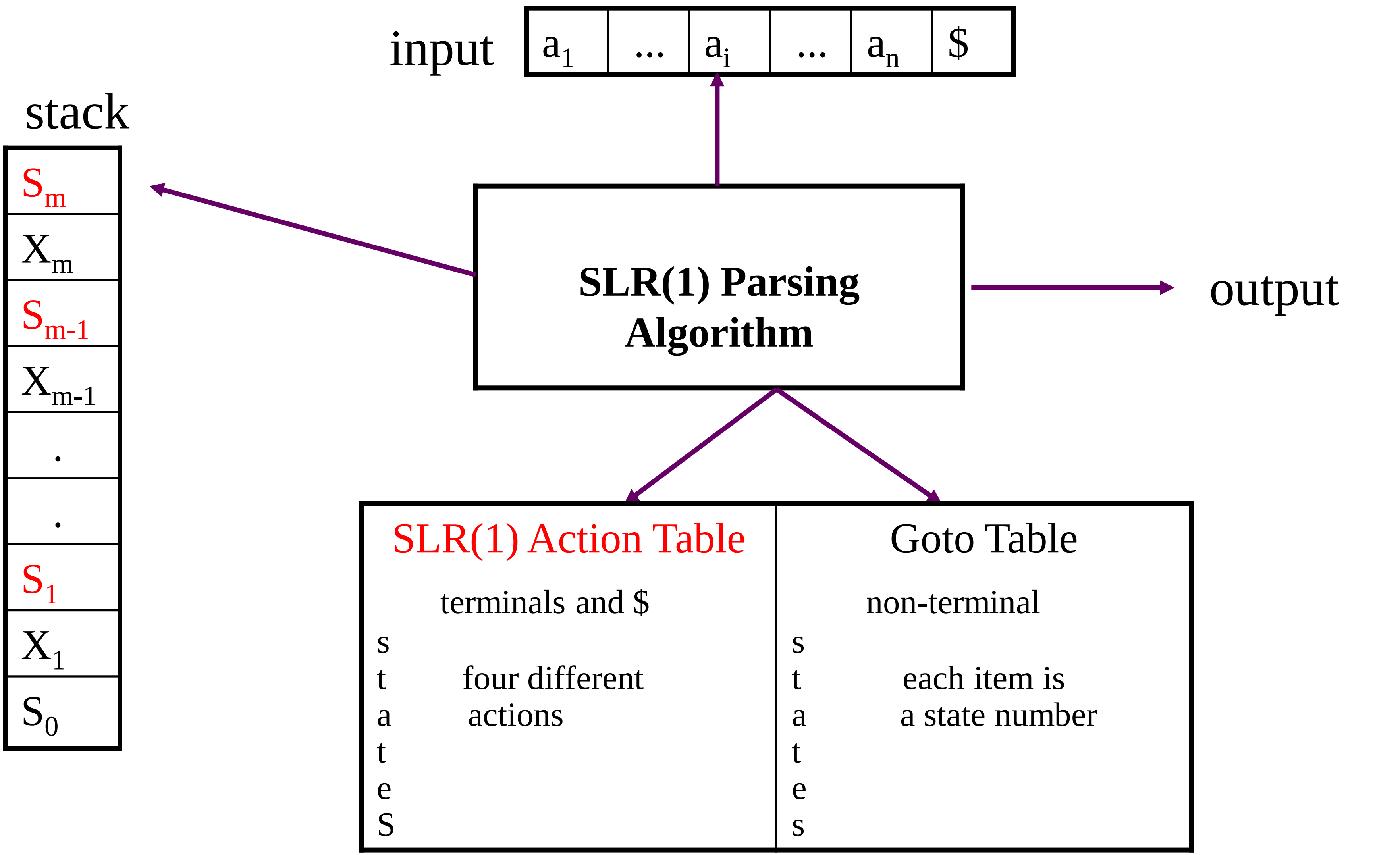
2. Create the parsing action table as follows:
   - If a is a terminal, $A \rightarrow \alpha . \mathbf{a} \beta$ in I_i and $goto(I_i,\mathbf{a})=I_j$ then action[i,a] is ***shift j***.
   - If $A \rightarrow \alpha .$ is in I_i, then action[i,a] is ***reduce A→α*** **for all a in FOLLOW(A) where A≠S'.**
   - If $S' \rightarrow S.$ is in I_i, then action[i,\$] is ***accept***.
   - If any conflicting actions generated by these rules, the grammar is not SLR(1).

3. Create the parsing goto table :
   - for all non-terminals A, if $goto(I_i,A)=I_j$ then goto[i,A]=j

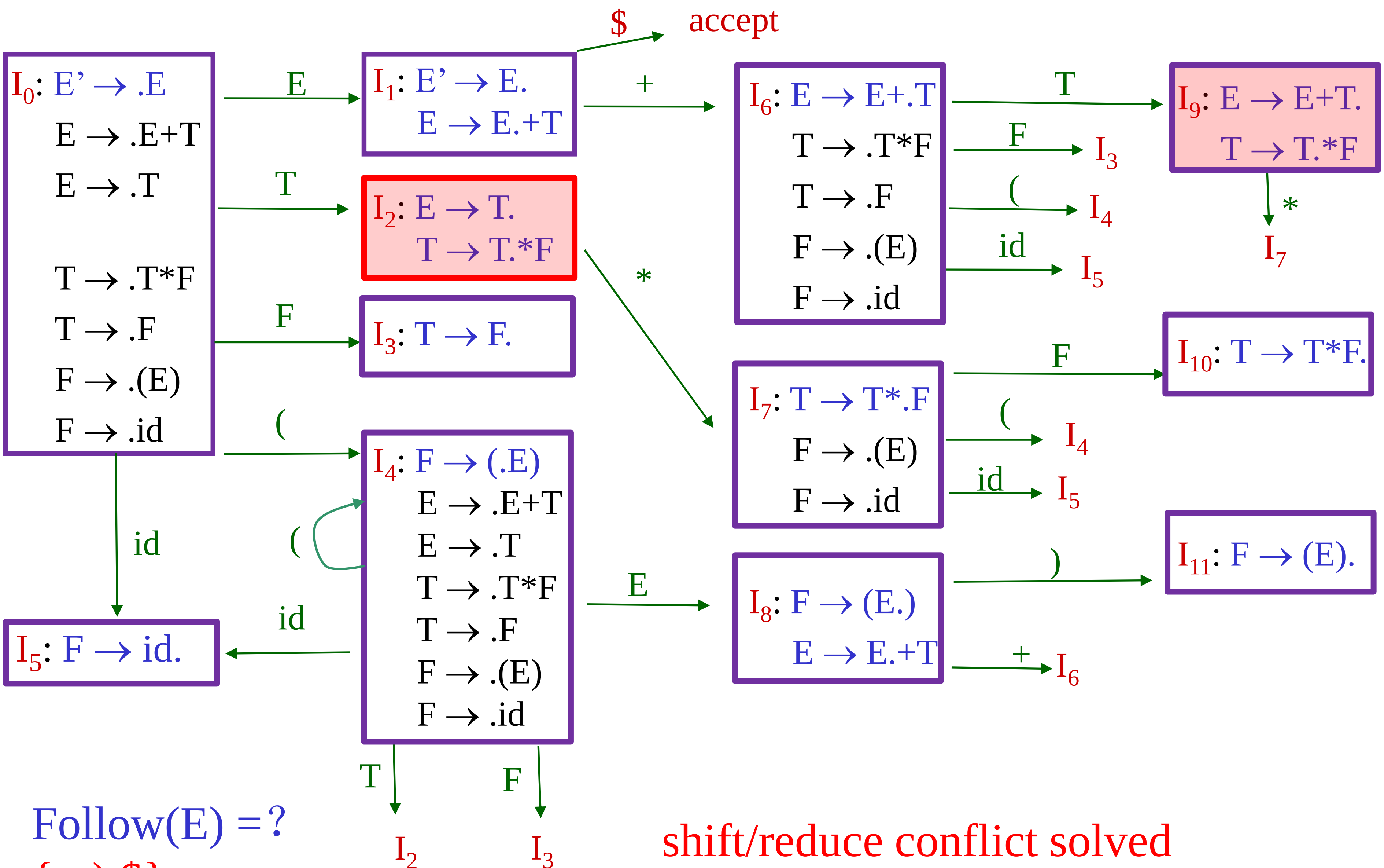
4. All entries not defined by (2) and (3) are errors.

5. Initial state of the parser contains $S' \rightarrow .S$

SLR(1) Parsing Algorithm Implmentation



LR(0) Conflicts



Follow(E) = ?
{+,),$}

shift/reduce conflict solved

Parsing Tables of Expression Grammar

1) $E \rightarrow E+T$
2) $E \rightarrow T$
3) $T \rightarrow T*F$
4) $T \rightarrow F$
5) $F \rightarrow (E)$
6) $F \rightarrow id$

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

SLR(1) Action Table | Goto Table

state	id	+	*	(	)	$		E	T	F
0	**s5**			**s4**				**1**	**2**	**3**
1		**s6**				**acc**				
2	~~**r2**~~	**r2**	**s7** ~~**r2**~~	~~**r2**~~	**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**~~	**r1**	**s7** ~~**r1**~~	~~**r1**~~	**r1**	**r1**				
10		**r3**	**r3**		**r3**	**r3**				
11		**r5**	**r5**		**r5**	**r5**				

Actions of A SLR(1) Parser – Example

stack	input	action	output
0	id*id+id$	shift 5	
0id5	*id+id$	reduce by F→id	F→id
0F3	*id+id$	reduce by T→F	T→F
0T2	*id+id$	shift 7	
0T2*7	id+id$	shift 5	
0T2*7id5	+id$	reduce by F→id	F→id
0T2*7F10	+id$	reduce by T→T*F	T→T*F
0T2	+id$	reduce by E→T	E→T
0E1	+id$	shift 6	
0E1+6	id$	shift 5	
0E1+6id5	$	reduce by F→id	F→id
0E1+6F3	$	reduce by T→F	T→F
0E1+6T9	$	reduce by E→E+T	E→E+T
0E1	$	accept	

SLR(1) Grammar

- An LR parser using SLR(1) parsing tables for a grammar G is called as the SLR(1) parser for G.
- If a grammar G has an SLR(1) parsing table, it is called SLR(1) grammar (or SLR grammar in short).
- Every SLR grammar is unambiguous, but not every unambiguous grammar is a SLR grammar.

Conflict Example

If $A \to \alpha.$ is in I_i, then action[i,a] is ***reduce $A \to \alpha$*** **for all a in FOLLOW(A)** **where $A \neq S'$.**

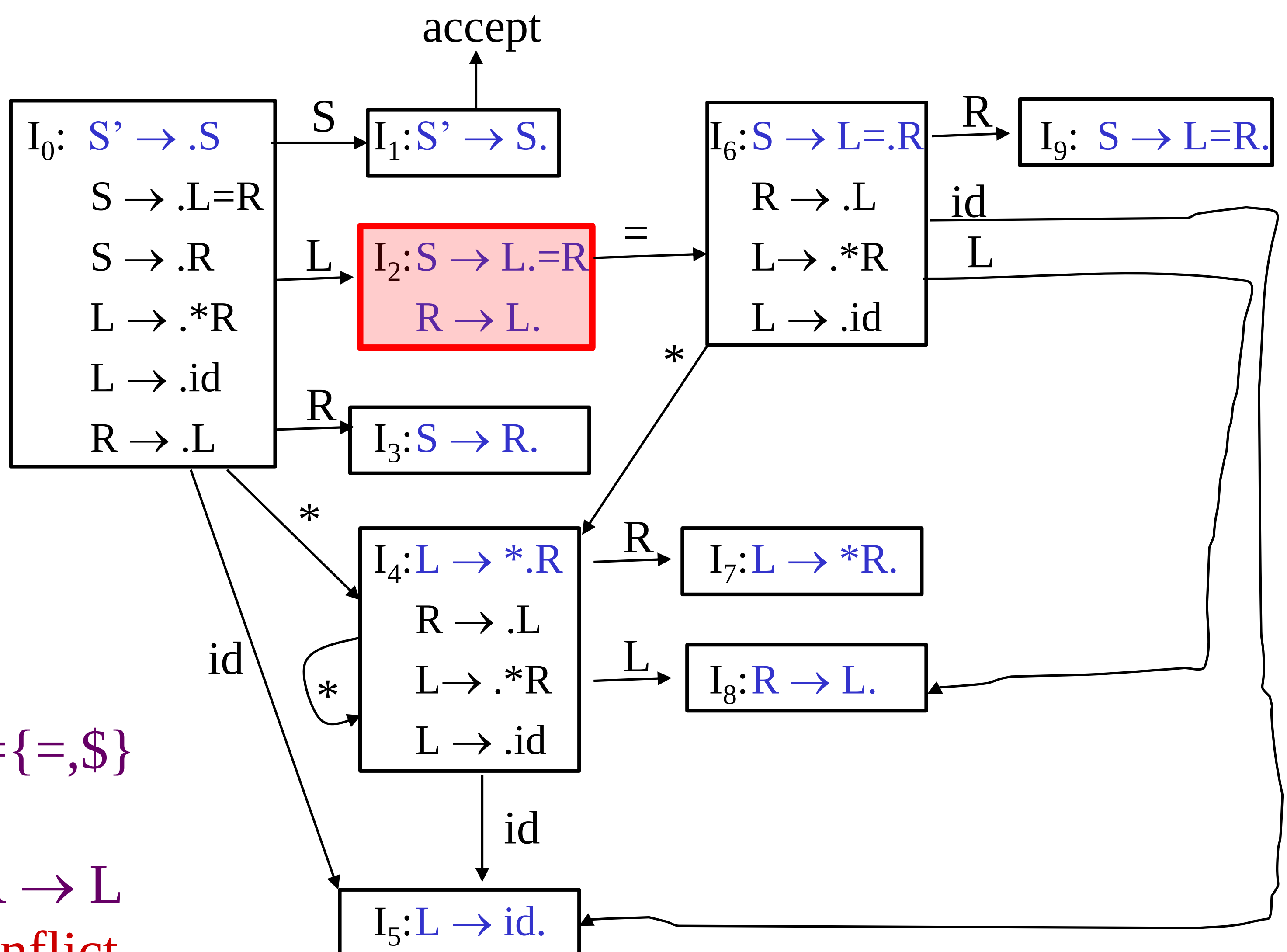
$S \to L=R$

$S \to R$

$L \to *R$

$L \to id$

$R \to L$



Problem ?

FOLLOW(R)={=,$}

1. shift 6
2. reduce by $R \to L$

shift/reduce conflict

shift/reduce and reduce/reduce conflicts

- If a state does not know whether it will make a shift operation or reduction for a terminal, we say that there is a **shift/reduce conflict**.

- If a state does not know whether it will make a reduction operation using the production rule `i` or `j` for a terminal, we say that there is a **reduce/reduce conflict**.

- If the SLR parsing table of a grammar G has a conflict, we say that that grammar is not SLR grammar.

SLR(1)

$S \to AaAb$
$S \to BbBa$
$A \to \varepsilon$
$B \to \varepsilon$

I_0: $S' \to .S$
$S \to .AaAb$
$S \to .BbBa$
$A \to .$
$B \to .$

Construct
SLR(1) action table

Problem
FOLLOW(A)={a,b}
FOLLOW(B)={a,b}

a → reduce by $A \to \varepsilon$
  → reduce by $B \to \varepsilon$
reduce/reduce conflict

b → reduce by $A \to \varepsilon$
  → reduce by $B \to \varepsilon$
reduce/reduce conflict

Constructing Canonical LR(1) Parsing Tables

- In SLR method, the state i makes a reduction by $A \rightarrow \alpha$ when the current token is a:
  - if $A \rightarrow \alpha.$ in the I_i and a is FOLLOW(A)
- In some situations, βA cannot be followed by the terminal a in a right-sentential form when $\beta\alpha$ and the state i are on the top stack. This means that making reduction in this case is not correct.

$S \rightarrow AaAb$

$S \rightarrow BbBa$

$A \rightarrow \varepsilon$

$B \rightarrow \varepsilon$

$S \Rightarrow AaAb \Rightarrow Aab \Rightarrow ab$

$AaAb \Rightarrow Aa\ \varepsilon\ b$

$Aab \Rightarrow \varepsilon\ ab$

$S \Rightarrow BbBa \Rightarrow Bba \Rightarrow ba$

$BbBa \Rightarrow Bb\ \varepsilon\ a$

$Bba \Rightarrow \varepsilon\ ba$

{ I_0: $S \rightarrow AaAb$, $S \rightarrow BbBa$, $A \rightarrow \varepsilon$, $B \rightarrow \varepsilon$ }: reduce/reduce conflict

FOLLOW(A)=FOLLOW(B)={a,b}

LR(1) Item

- To avoid some of invalid reductions, the states need to carry more information.
- Extra information is put into a state by including a terminal symbol as a second component in an item.

- A LR(0) item is:

$$A \rightarrow \alpha \bullet \beta$$

- A LR(1) item is:

$A \rightarrow \alpha \bullet \beta, a$ where **a** is the look-ahead of the LR(1) item

(**a** is a terminal or end-marker.)

LR(1) Item (cont.)

- When β (in the LR(1) item $A \rightarrow \alpha \bullet \beta, a$) is not empty, the look-ahead does not have any affect.
- When β is empty ($A \rightarrow \alpha \bullet, a$), we do the reduction by $A \rightarrow \alpha$ only if the next input symbol is **a** (not for any terminal in FOLLOW(A)).
- A state will contain

$$\begin{array}{c} A \rightarrow \alpha \bullet, a_1 \\ \ldots \\ A \rightarrow \alpha \bullet, a_n \end{array}$$

where $\{a_1,\ldots,a_n\} \subseteq \mathrm{FOLLOW}(A)$

LR(1) Automaton

- The states of LR(1) automaton are similar to the construction of the one for LR(0) automaton, except that *closure* and *goto* operations work a little bit different.

closure(I) is: (where I is a set of LR(1) items)

- every LR(1) item in I is in closure(I)
- if $A \to \alpha \bullet B\beta, a$ in closure(I) and $B \to \gamma$ is a production rule of G; then $B \to .\gamma$, **b** will be in the closure(I) for each terminal b in **FIRST(βa)** .

LR(0) automaton

- if $A \to \alpha \bullet B\beta$ in closure(I) and $B \to \gamma$ is a production rule of G; then $B \to .\gamma$, will be in the closure(I) .

The Algorithm for FOLLOW – pseudocode

1. Initialize FOLLOW(X) for all non-terminals X to empty set. Place $ in FOLLOW(S), where S is the start NT.
2. Repeat the following step until no modifications are made to any Follow-set

   For any production $\mathbf{X \rightarrow X_1\ X_2 \ldots X_j\ X_{j+1} \ldots X_m}$

   For j=1 to m,

   if $\mathbf{X_j}$ is a non-terminal then:

$\mathbf{FOLLOW(X_j)=FOLLOW(X_j) \cup (FIRST(X_{j+1},\ldots,X_m)-\{\varepsilon\})}$;

If FIRST($X_{j+1},\ldots,X_m$) contains ε or $X_{j+1},\ldots,X_m = \varepsilon$

then $\mathbf{FOLLOW(X_j)=FOLLOW(X_j)} \cup$ FOLLOW(X);

goto operation

- If I is a set of LR(1) items (i.e. state) and X is a grammar symbol (terminal or non-terminal), then goto(I,X) is defined as follows:
  - If $A \rightarrow \alpha.X\beta, a$ in I then every item in **closure({$A \rightarrow \alpha X.\beta, a$})** will be in goto(I,X).

Construction of LR(1) Automaton

- ***Algorithm***:

  C is { closure({S'→.S, $}) }

  repeat the followings until no more set of LR(1) items can be added to ***C***.

  for each I in ***C*** and each grammar symbol X

  if goto(I,X) is not empty and not in ***C***

  add goto(I,X) to ***C***

- goto function is a DFA on the sets in C.
- Initial state: closure({S'->S,$})

A Short Notation for The Sets of LR(1) Items

- A set of LR(1) items containing the following items

$$A \to \alpha \bullet \beta,\ a_1$$

$$\ldots$$

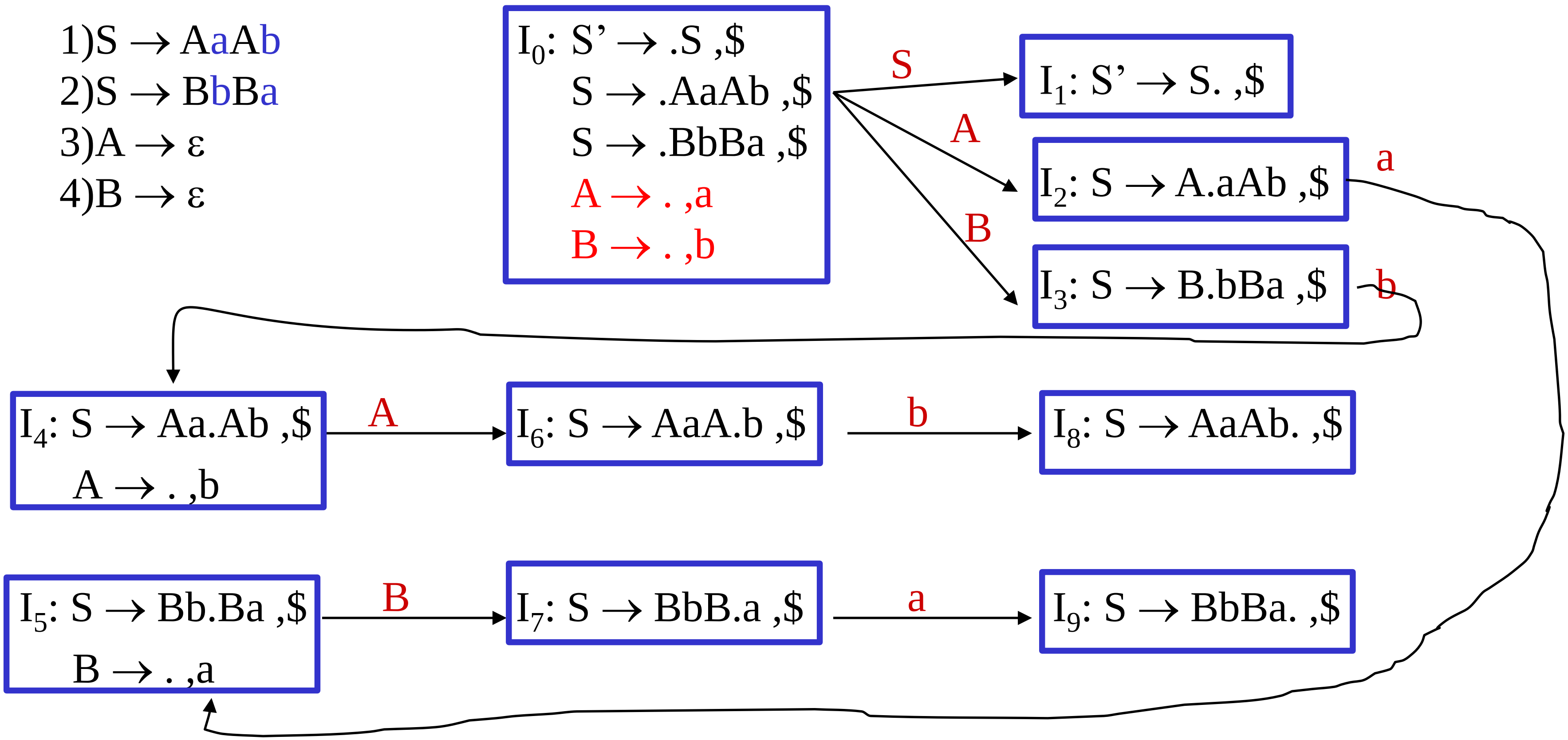
$$A \to \alpha \bullet \beta,\ a_n$$

can be written as

$$A \to \alpha \bullet \beta,\ a_1/a_2/\ldots/a_n$$

LR(1) Automaton example

1) $S \to AaAb$
2) $S \to BbBa$
3) $A \to \varepsilon$
4) $B \to \varepsilon$



if $S \to \mathbf{.}AaAb, \$$ in closure(I) and $A \to \varepsilon$ is a production rule of G; then $A \to ., \mathbf{a}$ will be in the closure(I) for each terminal a in **FIRST(AaAb$)** ={a}.

Construction of LR(1) Parsing Tables

1. Construct the canonical collection of sets of LR(1) items for G'.

   $C \leftarrow \{I_0,...,I_n\}$

2. Create the parsing action table as follows
   - If a is a terminal, **$A \rightarrow \alpha \bullet a\beta$, b in I_i** and **goto(I_i,a)=I_j** then action[i,a] is ***shift j***.
   - If **$A \rightarrow \alpha \bullet$, a** is in I_i, then **action[i,a]** is ***reduce $A \rightarrow \alpha$*** where $A \neq S'$.
   - If $S' \rightarrow S \bullet$, \$ is in I_i, then action[i,\$] is ***accept***.
   - If any conflicting actions generated by these rules, the grammar is not LR(1).

3. Create the parsing goto table
   - for all non-terminals A, if goto(I_i,A)=I_j then goto[i,A]=j

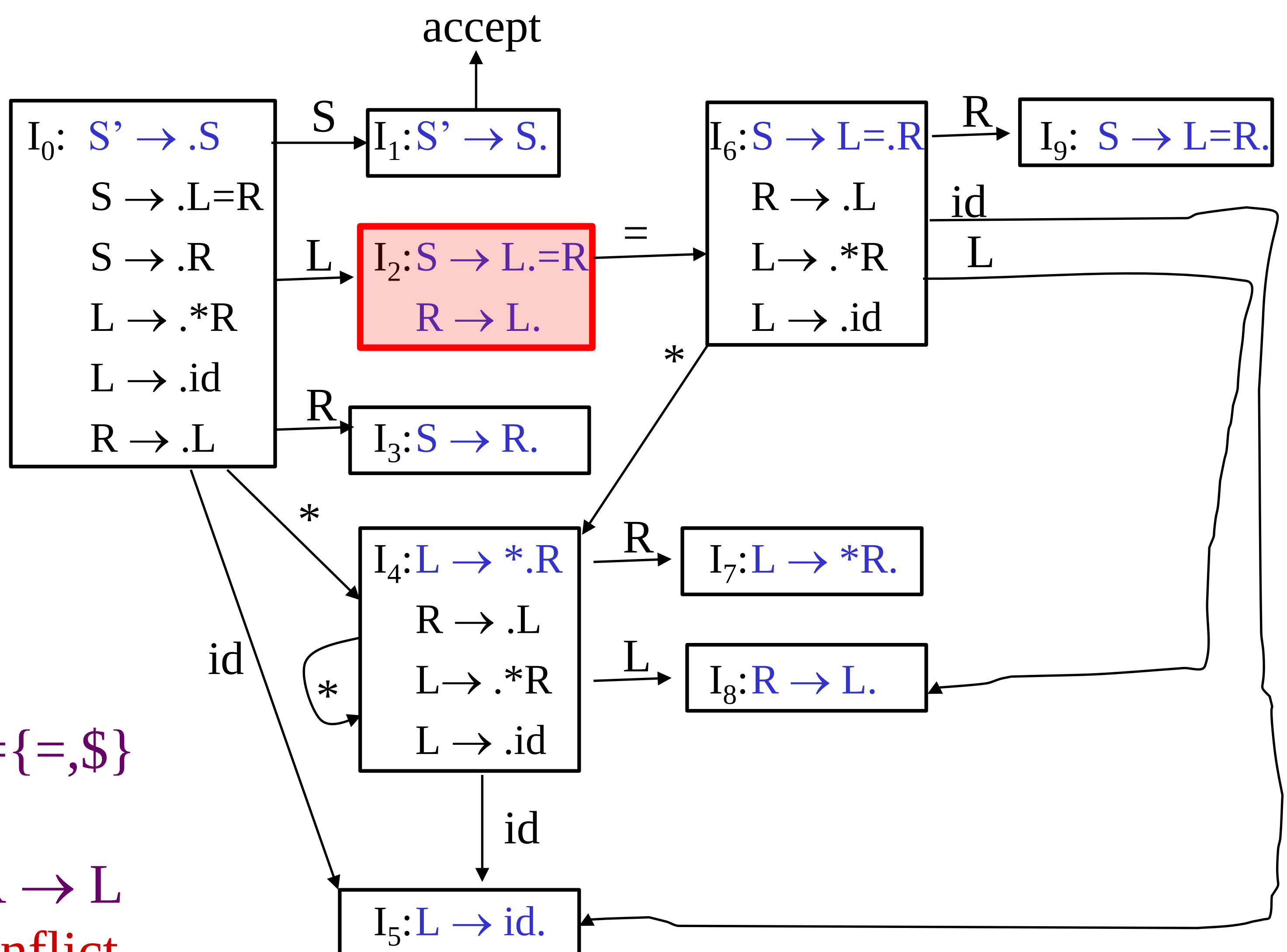
4. All entries not defined by (2) and (3) are errors.

5. Initial state of the parser contains $S' \rightarrow .S$, \$

Conflict Example SLR(1)

If $A \rightarrow \alpha.$ is in I_i , then action[i,a] is ***reduce*** $\boldsymbol{A \rightarrow \alpha}$ **for all a in FOLLOW(A) where A≠S'.**

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



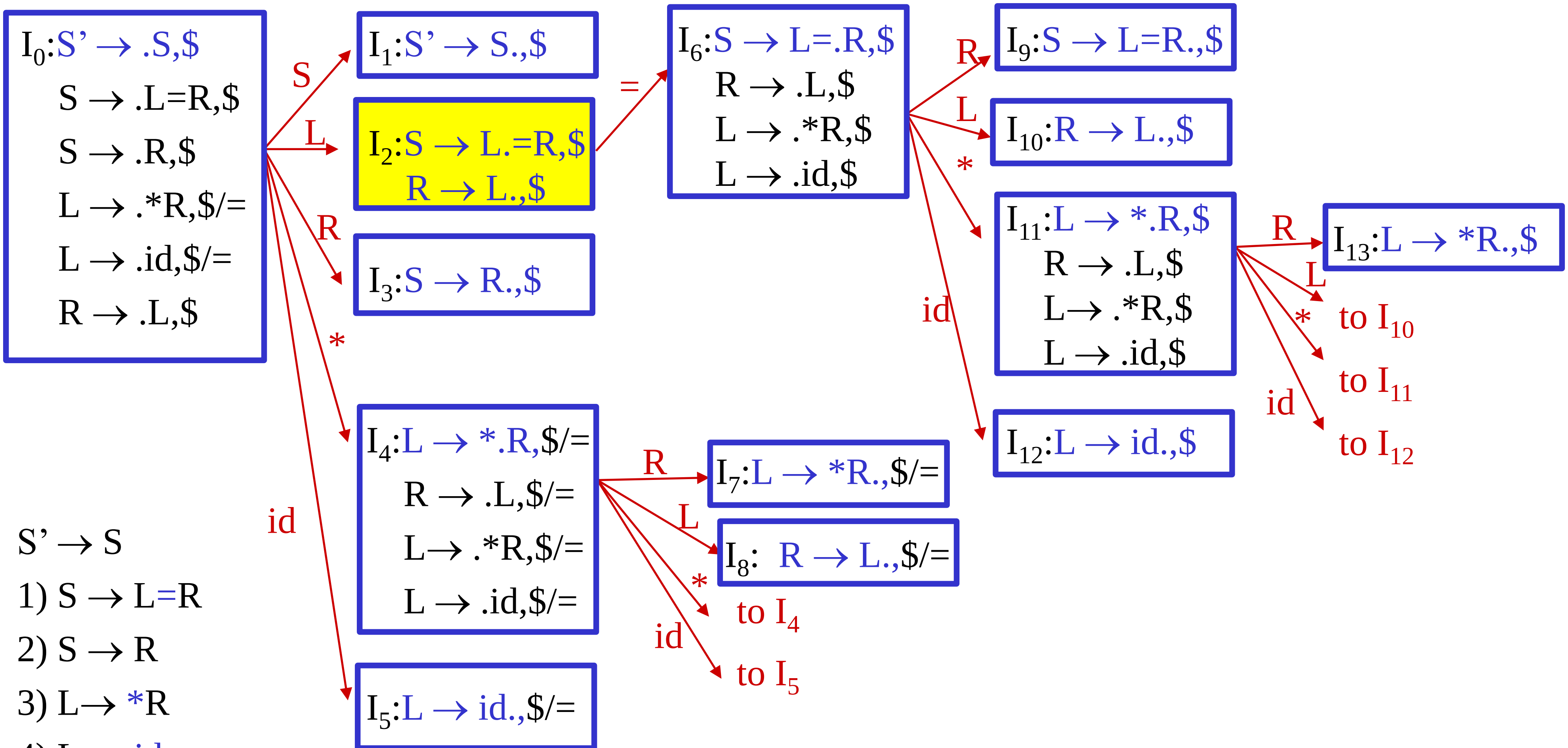
Problem ?

FOLLOW(R)={=,$}

1. shift 6
2. reduce by $R \rightarrow L$

shift/reduce conflict

Canonical LR(1) Collection – Example2



S' → S

1) S → L=R

2) S → R

3) L→ *R

4) L → id

5) R → L

Draw LR(1) automaton and LR(1) parsing table

LR(1) Parsing Tables – (for Example2)

	id	*	=	$	S	L	R
0	**s5**	**s4**			**1**	**2**	**3**
1				**acc**			
2			**s6**	**r5**			
3				**r2**			
4	**s5**	**s4**				**8**	**7**
5			**r4**	**r4**			
6	**s12**	**s11**				**10**	**9**
7			**r3**	**r3**			
8			**r5**	**r5**			
9				**r1**			
10				**r5**			
11	**s12**	**s11**				**10**	**13**
12				**r4**			
13				**r3**			

no shift/reduce or
no reduce/reduce conflict

$\Downarrow$

so, it is a LR(1) grammar

LALR Parsing Tables

- **LALR** stands for **LookAhead LR.**

- LALR parsers are often used in practice because LALR parsing tables are smaller than LR(1) parsing tables.
- The number of states in SLR and LALR parsing tables for a grammar G are equal.
- But LALR parsers recognize more grammars than SLR parsers.
- **YACC/Bison** creates a LALR parser for the given grammar.
- A state of LALR parser will be again a set of LR(1) items.

The Core of A Set of LR(1) Items

- The core of a set of LR(1) items is the set of its first component.

Ex: $S \rightarrow L \bullet = R, \$$ ➔ $S \rightarrow L \bullet = R$ ← **Core (LR(0) Item)**

$R \rightarrow L \bullet, \$$ $R \rightarrow L \bullet$

- We will find the states (sets of LR(1) items) in a canonical LR(1) parser with **same cores** . Then we will merge them as a single state.

$\boxed{I_1: L \rightarrow id \bullet, =}$ A new state: $I_{12}: L \rightarrow id \bullet, =$

➔ $L \rightarrow id \bullet, \$$

$\boxed{I_2: L \rightarrow id \bullet, \$}$ have same core, merge them or $\boxed{I_{12}: L \rightarrow id., =/\$}$

- We will do this for all states of a canonical LR(1) parser to get the states of the LALR parser.
- In fact, the number of the states of the LALR parser for a grammar will be equal to the number of states of the SLR parser for that grammar.

Creation of LALR Parsing Tables

- Create the canonical LR(1) collection of the sets of LR(1) items for the given grammar.
- Find each core; find all sets having that same core; replace those sets having same cores with a single set which is their union.

  $C=\{I_0,...,I_n\} \Rightarrow C'=\{J_1,...,J_m\}$ where $m \leq n$
- Create the parsing tables (action and goto tables) same as the construction of the parsing tables of LR(1) parser.
  - Note that: If $J=I_1 \cup ... \cup I_k$ since $I_1,...,I_k$ have same cores $\Rightarrow$ cores of $goto(I_1,X),...,goto(I_2,X)$ must be same.
  - So, $goto(J,X)=K$ where K is the union of all sets of items having same cores as $goto(I_1,X)$.

- **If no conflict is introduced, the grammar is LALR(1) grammar.** (We may only introduce reduce/reduce conflicts; we cannot introduce a shift/reduce conflict)

Creating LALR Parsing Tables

Canonical LR(1) Parser ➔ LALR Parser

shrink # of states

- This shrink process may introduce a **reduce/reduce** conflict in the resulting LALR parser (so the grammar is NOT LALR)

- But, this shrink process does not produce a **shift/reduce** conflict.

?

Reduce/Reduce Conflict

- But, we may introduce a reduce/reduce conflict during the shrink process for the creation of the states of a LALR parser.

$$\boxed{\begin{array}{l} I_1 : A \to \alpha \bullet, a \\ \quad\; B \to \beta \bullet, b \end{array}} \qquad \boxed{\begin{array}{l} I_2 : A \to \alpha \bullet, b \\ \quad\; B \to \beta \bullet, c \end{array}}$$

$$\Downarrow$$

$$\boxed{\begin{array}{l} I_{12} : A \to \alpha \bullet, a/b \\ \quad\;\;\; B \to \beta \bullet, b/c \end{array}}$$

➔ reduce/reduce conflict

Shift/Reduce Conflict

- We say that we cannot introduce a shift/reduce conflict during the shrink process for the creation of the states of a LALR parser.
- Assume that we can introduce a shift/reduce conflict. In this case, a state of LALR parser must have:

$$A \rightarrow \alpha\bullet, a \qquad \text{and} \qquad B \rightarrow \beta\bullet a\gamma, b$$

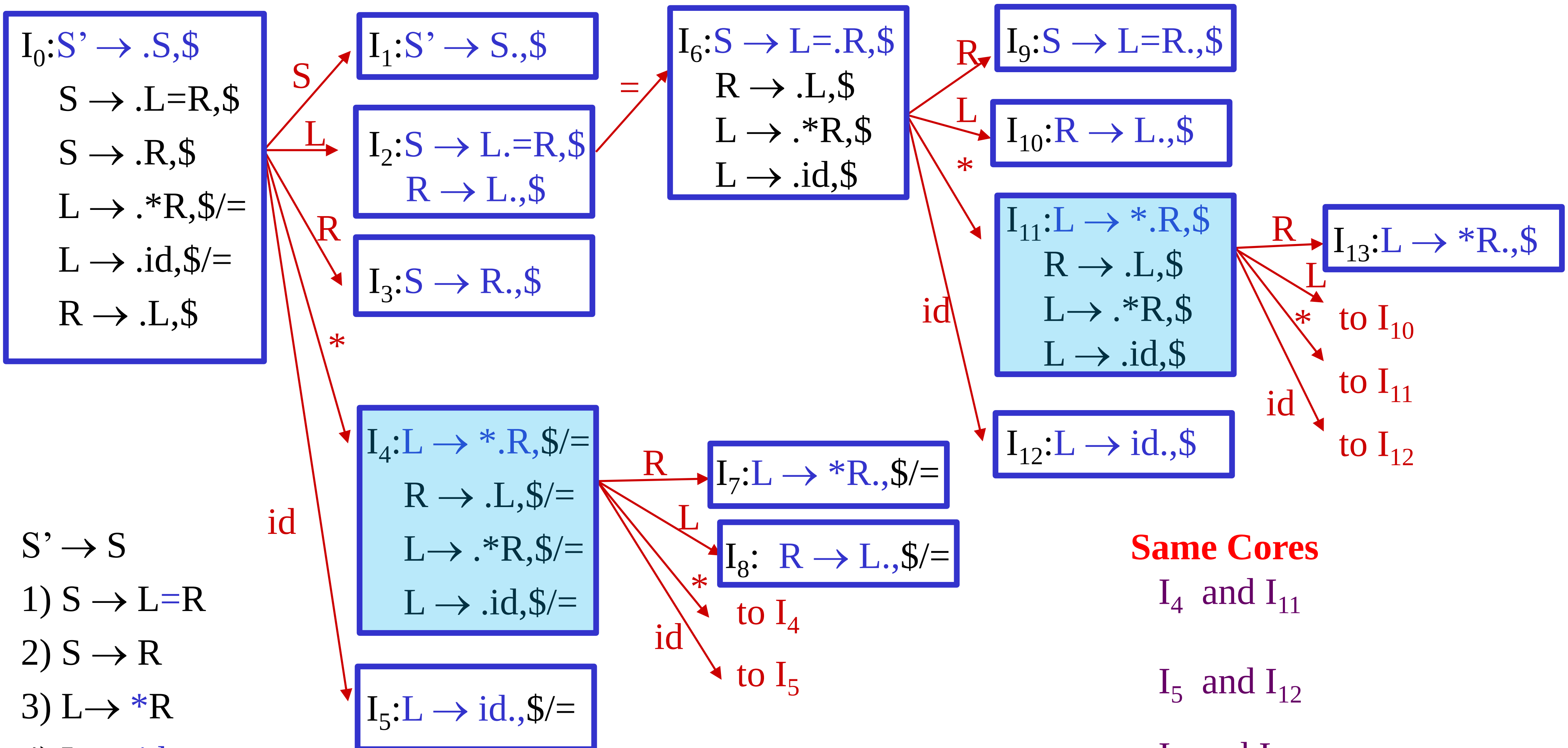
- This means that a state of the canonical LR(1) parser must have:

$$A \rightarrow \alpha\bullet, a \qquad \text{and} \qquad B \rightarrow \beta\bullet a\gamma, c$$

But, this state has also a shift/reduce conflict. i.e. the original canonical LR(1) parser has a conflict.

(Reason for this, the shift operation does not depend on lookaheads)

Canonical LR(1) Collection – Example2



S' → S
1) S → L=R
2) S → R
3) L→ *R
4) L → id
5) R → L

Same Cores

I_4 and I_{11}

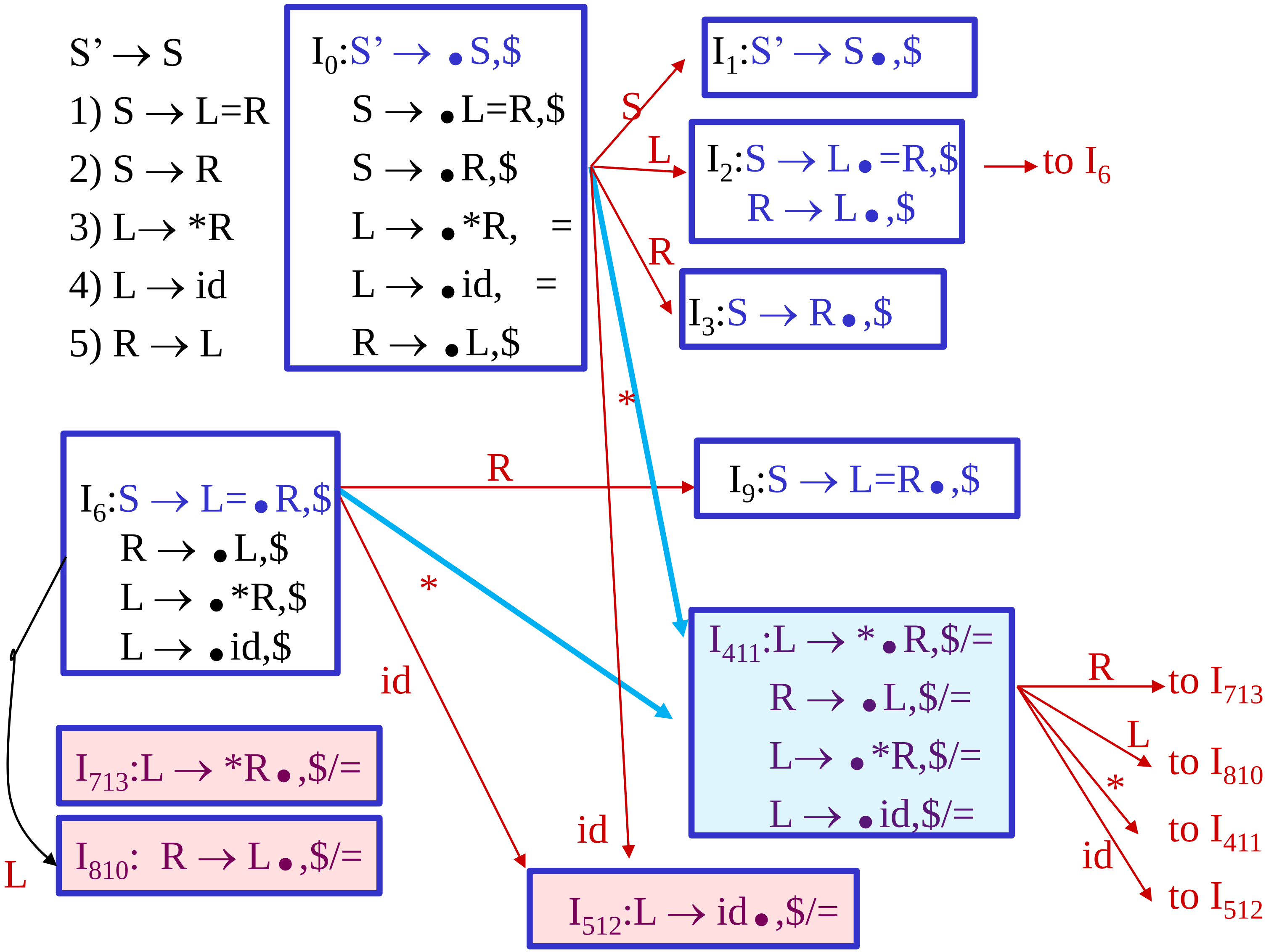
I_5 and I_{12}

I_7 and I_{13}

I_8 and I_{10}

Canonical LALR(1) Collection – Example2

S’ → S
1) S → L=R
2) S → R
3) L→ *R
4) L → id
5) R → L



Same Cores
I_4 and I_{11}

I_5 and I_{12}

I_7 and I_{13}

I_8 and I_{10}

LALR(1) Parsing Tables – (for Example2)

	id	*	=	$	S	L	R
0	s512	s411			1	2	3
1				acc			
2			s6	r5			
3				r2			
4	s512	s411				810	713
5			r4	r4			
6	s512	s411				810	9
7			r3	r3			
8			r5	r5			
9				r1			

	id	*	=	$	S	L	R
0	s5	s4			1	2	3
1				acc			
2			s6	r5			
3				r2			
4	s5	s4				8	7
5			r4	r4			
6	s12	s11				10	9
7			r3	r3			
8			r5	r5			
9				r1			
10				r5			
11	s12	s11				10	13
12				r4			
13				r3			

no shift/reduce or
no reduce/reduce conflict

$\Downarrow$

so, it is a LALR(1) grammar

Panic Mode Error Recovery in LR Parsing

- Discard zero or more input symbols until a symbol **a** is found that can legitimately follow A.
  - The symbol a is simply in FOLLOW(A), but this may not work for all situations.

- Scan down the stack until a state **s** with a goto on a particular nonterminal **A** is found. (Get rid of everything from the stack before this state s).

Phrase-Level Error Recovery in LR Parsing

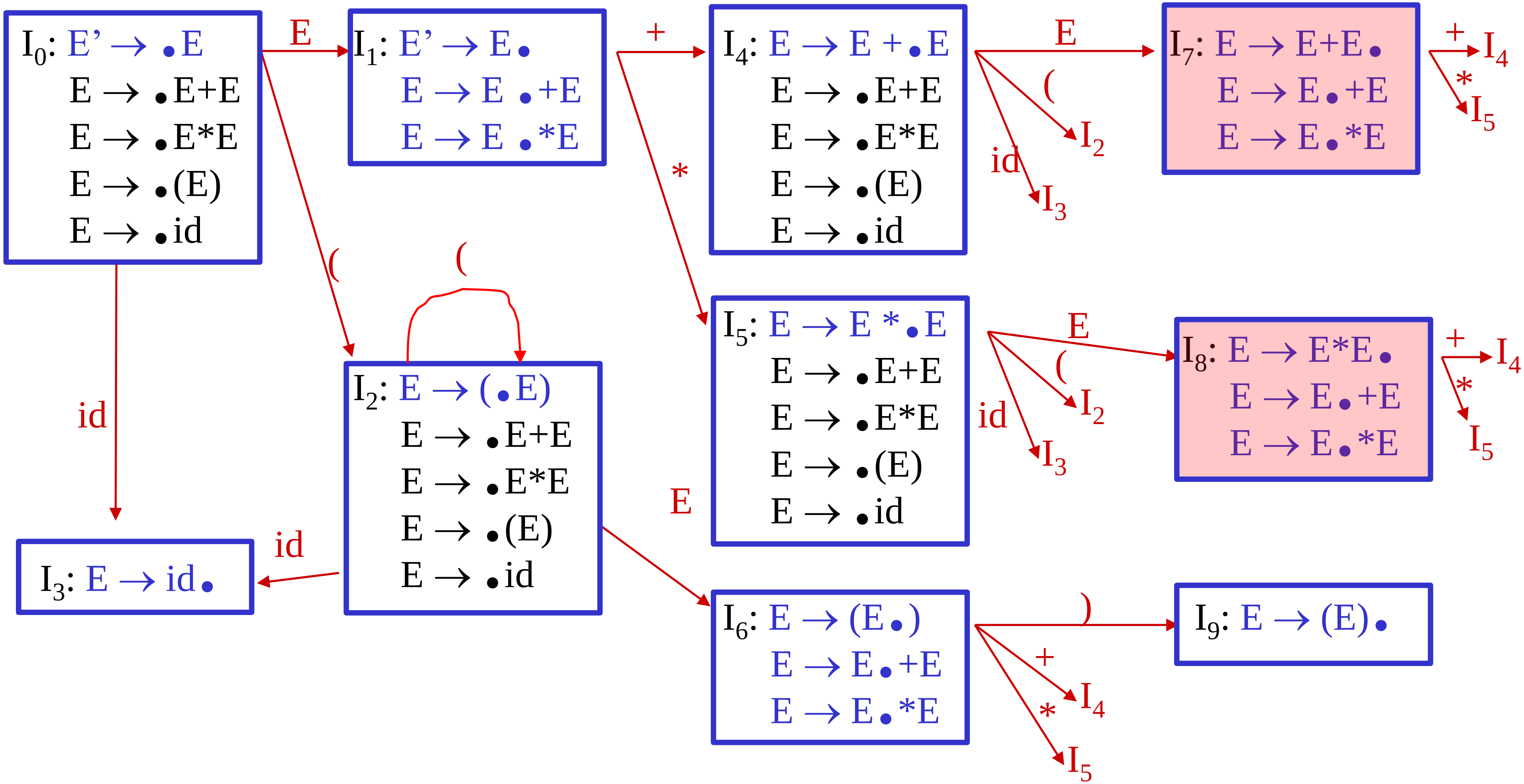
- Each empty entry in the action table is marked with a specific error routine.
- An error routine reflects the error that the user most likely will make in that case.
- An error routine inserts the symbols into the stack or the input (or it deletes the symbols from the stack and the input, or it can do both insertion and deletion).
  - missing operand
  - unbalanced right parenthesis

Using Ambiguous Grammars

- All grammars used in the construction of LR-parsing tables must be unambiguous.
- Can we create LR-parsing tables for ambiguous grammars ?
  - Yes, but they will have conflicts.
  - We can resolve these conflicts in favor of one of them to disambiguate the grammar.
  - At the end, we will have again an unambiguous grammar.
- Why we want to use an ambiguous grammar?
  - Some of the ambiguous grammars are **much natural**, and a corresponding unambiguous grammar can be very complex.
  - Usage of an ambiguous grammar may **eliminate unnecessary reductions**.

$$E \rightarrow E+E \mid E*E \mid (E) \mid id \quad \Rightarrow \quad \begin{array}{l} E \rightarrow E+T \mid T \\ T \rightarrow T*F \mid F \\ F \rightarrow (E) \mid id \end{array}$$

Sets of LR(0) Items for Ambiguous Grammar



shift/reduce conflicts for symbols + and *

SLR-Parsing Tables for Ambiguous Grammar

FOLLOW(E) = { $, +, *,) }

State I_7 has shift/reduce conflicts for symbols **+** and *****.

$$I_0 \xrightarrow{E} I_1 \xrightarrow{+} I_4 \xrightarrow{E} I_7$$

when current token is +
- shift ➔ + is right-associative
- reduce ➔ + is left-associative

when current token is *
- shift ➔ * has higher precedence than +
- reduce ➔ + has higher precedence than *

SLR-Parsing Tables for Ambiguous Grammar

FOLLOW(E) = { $, +, *,) }

State I_8 has shift/reduce conflicts for symbols **+** and *****.

$I_0 \xrightarrow{E} I_1 \xrightarrow{*} I_5 \xrightarrow{E} I_8$

when current token is *
shift ➔ * is right-associative
reduce ➔ * is left-associative

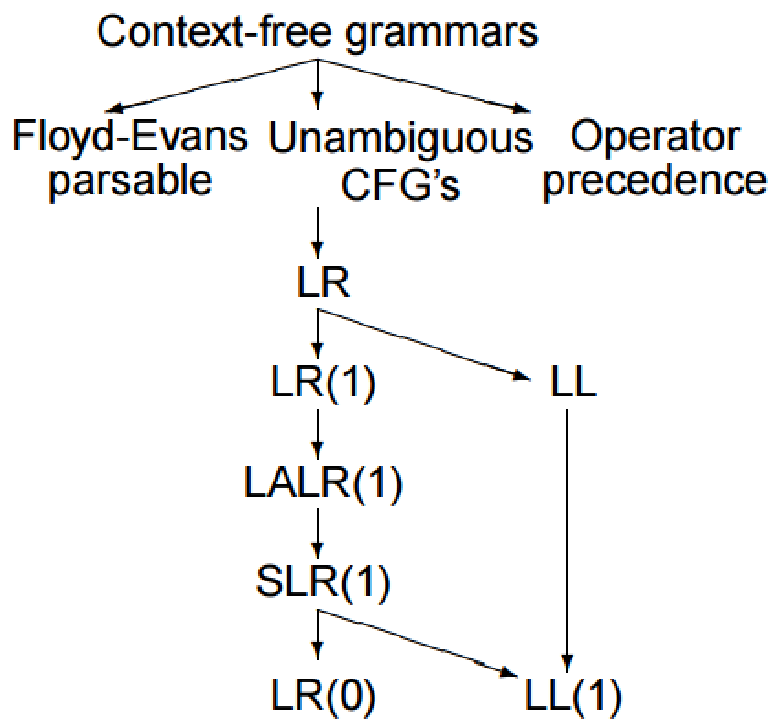
when current token is +
shift ➔ + has higher precedence than *
reduce ➔ * has higher precedence than +

SLR-Parsing Tables for Ambiguous Grammar

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

Summary

- Bottom-up parsing——shift-reduce parsing
- LR parsing——SLR、LR、LALR



Reading materials

[1] Frost, R.; R. Hafiz (2006). A New Top-Down Parsing Algorithm to Accommodate Ambiguity and Left Recursion in Polynomial Time.

[2] Frost, R.; R. Hafiz; P. Callaghan (2007). Modular and Efficient Top-Down Parsing for Ambiguous Left-Recursive Grammars.