Syntax Analysis (Parsing)

- 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 or parsing
- includes augmenting info on tokens in source, type checking, semantic analysis
Lexical Analysis Review: RE2NFA

Regular expression $r ::= \varepsilon \mid a \mid r + s \mid rs \mid r^*$

RE to NFA

Symbols:
- $i$: Input symbol
- $f$: Final state
- $L(\varepsilon)$: Language of the empty string
- $L(a)$: Language of the symbol $a$
- $L(s) \cup L(t)$: Union of languages $L(s)$ and $L(t)$
- $L(s) L(t)$: Concatenation of languages $L(s)$ and $L(t)$
- $(L(s))^*$: Star of language $L(s)$
Lexical Analysis Review: NFA2DFA

put $\varepsilon$-closure($\{s_0\}$) as an unmarked state into the set of DFA (DStates)

while (there is one unmarked $S_1$ in DStates) do
  mark $S_1$
  for (each input symbol $a$)
    $S_2 \leftarrow \varepsilon$-closure($\text{move}(S_1,a)$)
    if ($S_2$ is not in DStates) then
      add $S_2$ into DStates as an unmarked state

transfunc[$S_1,a] \leftarrow S_2$

- the start state of DFA is $\varepsilon$-closure($\{s_0\}$)
- a state $S$ in DStates is an accepting state of DFA if a state in $S$ is an accepting state of NFA
Lexical Analysis Review: RE2DFA

• Create the syntax tree of (r) #
• Calculate the functions: followpos, firstpos, lastpos, nullable
• Put firstpos(root) into the states of DFA as an unmarked state.
• while (there is an unmarked state S in the states of DFA) do
  – mark S
  – for each input symbol a do
    • let s₁,...,sₙ are positions in S and symbols in those positions are a
    • S' \leftarrow \text{followpos}(s₁) \cup ... \cup \text{followpos}(sₙ)
    • move(S,a) \leftarrow S'
    • if (S' is not empty and not in the states of DFA)
      – put S' into the states of DFA as an unmarked state.

• the start state of DFA is firstpos(root)
• the accepting states of DFA are all states containing the position of #
Lexical Analysis Review: RE2DFA

\[
\begin{align*}
    s & \leftarrow s_0 \\
    c & \leftarrow \text{nextchar}; \\
    \text{while } c \neq \text{eof do} \\
    & \quad s \leftarrow \text{move}(s, c); \\
    & \quad c \leftarrow \text{nextchar}; \\
    \text{end}; \\
    \text{if } s \text{ is in } F \text{ then return "yes"} \\
    & \quad \text{else return "no"}
\end{align*}
\]

\[
\begin{align*}
    S & \leftarrow \varepsilon\text{-closure}({s_0}) \\
    c & \leftarrow \text{nextchar}; \\
    \text{while } c \neq \text{eof do} \\
    & \quad S \leftarrow \varepsilon\text{-closure}(\text{move}(S, c)); \\
    & \quad c \leftarrow \text{nextchar}; \\
    \text{end}; \\
    \text{if } S \cap F \neq \emptyset \text{ then return "yes"} \\
    & \quad \text{else return "no"}
\end{align*}
\]
Lexical Analysis Review: Minimizing DFA

• partition the set of states into two groups:
  – \( G_1 \): set of accepting states
  – \( G_2 \): set of non-accepting states

• For each new group \( G \)
  – partition \( G \) into subgroups such that states \( s_1 \) and \( s_2 \) are in the same group if for all input symbols \( a \), states \( s_1 \) and \( s_2 \) have transitions to states in the same group.

• Start state of the minimized DFA is the group containing the start state of the original DFA.

• Accepting states of the minimized DFA are the groups containing the accepting states of the original DFA.
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++
'f2' undefined; assuming extern returning int
syntax error : missing ';' before '}
 syntax error : missing ';' before identifier ‘printf’

Which are “easy” to recover from? Which are “hard”?
## Error Handling

<table>
<thead>
<tr>
<th>Error type</th>
<th>Example</th>
<th>Detector</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lexical</td>
<td>x#y=1</td>
<td>Lexer</td>
</tr>
<tr>
<td>Syntax</td>
<td>x=1 y=2</td>
<td>Parser</td>
</tr>
<tr>
<td>Semantic</td>
<td>int x; y=x(1)</td>
<td>Type checker</td>
</tr>
<tr>
<td>Correctness</td>
<td>Program can be compiled</td>
<td>Tester/user/model-checker</td>
</tr>
</tbody>
</table>
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
-- “,” ⇒ ”;” – 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 do many 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 or 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
C++98 -> C++03 -> C++11 -> C++14 -> C++17

How do grammars relate to parsing process?
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

EXAMPLE: (a+b)*abb

EXAMPLE: \{ (i)^i \mid i \geq 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 \( \alpha \) is a string of terminals and non-terminals (including the empty string)
  – A **start symbol** (one of the non-terminal symbol)

• Example:

  \[
  \begin{align*}
  E & \rightarrow \ E + \ E \mid \ E - \ E \mid \ E \ast \ E \mid \ E / \ E \mid - \ E \\
  E & \rightarrow ( \ E \ ) \\
  E & \rightarrow \text{id} \\
  E & \rightarrow \text{num}
  \end{align*}
  \]
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.

**PR**: \(NT \rightarrow (T \mid NT)^*\)

E.g.: \(E \rightarrow E + E\)

\(E \rightarrow \text{num}\)

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

\[ expr \rightarrow expr \ op \ expr \]
\[ expr \rightarrow ( \ expr \ ) \]
\[ expr \rightarrow - \ expr \]
\[ expr \rightarrow id \]
\[ op \rightarrow + \]
\[ op \rightarrow - \]
\[ op \rightarrow * \]
\[ op \rightarrow / \]

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.)

- $L(G)$ is the language of $G$ (the language generated by $G$) which is a set of sentences.
- A sentence of $L(G)$ is a string of terminal symbols of $G$.
- If $S$ is the start symbol of $G$ then
  - $\omega$ is a sentence of $L(G)$ if $S \Rightarrow^* \omega$ where $\omega$ is a string of terminals of $G$.

- 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.
- $S \Rightarrow^* \alpha$ - If $\alpha$ contains non-terminals, it is called as a sentential form of $G$.
  - If $\alpha$ does not contain non-terminals, it is called as a sentence of $G$.

EX. $E \Rightarrow E+E \Rightarrow \text{id}+E \Rightarrow \text{id}+\text{id}$
How does this relate to Languages?

Let G be a CFG with start symbol S. Then $S \Rightarrow^* W$ (where W has no non-terminals) represents the language generated by G, denoted $L(G)$. So $W \in L(G) \iff S \Rightarrow^* W$.

$W :$ is a sentence of G

When $S \Rightarrow^* \alpha$ (and $\alpha$ may have NTs) it is called a sentential form of G.

EXAMPLE: id * id is a sentence

Here’s the derivation:

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

$E \Rightarrow id * id$

Sentential forms

Sentence
Example Grammar – Terminology

Terminals: \texttt{a,b,c,+,-,punc,0,1,…,9}, blue strings

Non Terminals: \texttt{A,B,C,S}, black strings

T or NT: \texttt{X,Y,Z}

Strings of Terminals: \texttt{u,v,…,z} in T*

Strings of T / NT: \texttt{\alpha, \beta, \gamma} in (T \cup NT)*

Alternatives of production rules:

\[ A \rightarrow \alpha_1 ; A \rightarrow \alpha_2 ; \ldots ; A \rightarrow \alpha_k ; \Rightarrow A \rightarrow \alpha_1 | \alpha_2 | \ldots | \alpha_k \]

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

\[ E \rightarrow E \ op \ E \ | \ ( \ E \ ) \ | \ -E \ | \ id \]

\[ op \rightarrow + \ | \ - \ | \ * \ | \ / \ | \uparrow \]
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

\[ \alpha_1 \Rightarrow \alpha_2 \Rightarrow ... \Rightarrow \alpha_n \quad (\alpha_n \text{ derives from } \alpha_1 \text{ or } \alpha_1 \text{ derives } \alpha_n) \]
Grammar Concepts

A step in a derivation is zero or 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:** \( E \Rightarrow E \text{ op } E \Rightarrow E \ast E \Rightarrow E \ast (E) \)

**DEFINITION:** \( \Rightarrow \) derives in one step

\[ + \] derives in \( \geq \) one step

\[ * \] 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 \( \alpha_1 \Rightarrow^* \alpha_n \); \( \alpha \Rightarrow^* \alpha \) for all \( \alpha \)

If \( \alpha \Rightarrow^* \beta \) and \( \beta \rightarrow \gamma \) then \( \alpha \Rightarrow^* \gamma \)
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

\[ E \Rightarrow E \text{ op } E \Rightarrow id \text{ op } E \Rightarrow id \ast E \Rightarrow id \ast id \]

Rightmost: Replace the leftmost non-terminal symbol

\[ E \Rightarrow E \text{ op } E \Rightarrow E \text{ op } id \Rightarrow E \ast id \Rightarrow id \ast id \]

Important Notes: \[ A \rightarrow \delta \]

If \[ \beta A\gamma \Rightarrow \beta \delta \gamma \] what’s true about \( \beta \)?

If \[ \beta A\gamma \Rightarrow \beta \delta \gamma \] what’s true about \( \gamma \)?

Derivations: Actions to parse input can be represented pictorially in a parse tree.
Left-Most and Right-Most Derivations

Left-Most Derivation

\[ E \Rightarrow -E \Rightarrow -(E) \Rightarrow -(E+E) \Rightarrow -(id+E) \Rightarrow -(id+id) \]  

(4.4)

Right-Most Derivation (called *canonical derivation*)

\[ E \Rightarrow -E \Rightarrow -(E) \Rightarrow -(E+E) \Rightarrow -(E+id) \Rightarrow -(id+id) \]

- 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.

How mapping?
Derivation exercise 1 in class

**Productions:**

assign_stmt → id := expr ;

expr → expr op term

expr → term

term → id

term → real

term → integer

op → +

op → -

Let’s derive:

id := id + real − integer ;

Please use left-most derivation or right-most derivation.
Let’s derive: \( id := id + real – integer \);

**Left-most derivation:**

\[
\begin{align*}
assign\_stmt & \rightarrow id := expr ; \\
         & \rightarrow id := expr \ op \ term ; \\
         & \rightarrow id := expr \ op \ term \ op \ term ; \\
         & \rightarrow id := term \ op \ term \ op \ term ; \\
         & \rightarrow id := id \ op \ term \ op \ term ; \\
         & \rightarrow id := id + term \ op \ term ; \\
         & \rightarrow id := id + real \ op \ term ; \\
         & \rightarrow id := id + real - term ; \\
         & \rightarrow id := id + real - integer ;
\end{align*}
\]

**using production:**

\[
\begin{align*}
assign\_stmt & \rightarrow id := expr ; \\
expr & \rightarrow expr \ op \ term \\
expr & \rightarrow expr \ op \ term \\
expr & \rightarrow term \\
term & \rightarrow id \\
op & \rightarrow + \\
term & \rightarrow real \\
op & \rightarrow - \\
term & \rightarrow integer
\end{align*}
\]

**Right-most derivation?**
Example Grammar

A fragment of Cool:

Let’s derive

if while id loop id pool then id else id

EXPR → if EXPR then EXPR else EXPR fi

| while EXPR loop EXPR pool
| id
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
 \ E
 -
 E

⇒ -(E)
 \ E
 -
 (E)

⇒ -(E+E)
 \ E
 -
 (E)

⇒ -(id+E)
 \ E
 -
 (E)

⇒ -(id+id)
 \ E
 -
 (E)
```

```
Examples of LM / RM Derivations

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

\[
op \rightarrow + \mid - \mid * \mid /
\]

A leftmost derivation of : \( \text{id} + \text{id} * \text{id} \)

See latter slides

A rightmost derivation of : \( \text{id} + \text{id} * \text{id} \)

DO IT BY YOURSELF.
E ⇒ E op E

⇒ id op E

⇒ id + E

⇒ id + E op E

⇒ id + id op E

⇒ id + id * E

⇒ id + id * id

A rightmost derivation of: id + id * id

DO IT BY YOURSELF.
Parse Trees and Derivations

Consider the expression grammar:

\[ E \rightarrow E + E \mid E \ast E \mid (E) \mid -E \mid id \]

Leftmost derivations of \( id + id \ast id \):

1. \( E \Rightarrow E + E \)
2. \( E \Rightarrow id + E \)
3. \( id + E \Rightarrow id + E \ast E \)
 Parse Tree & Derivations (cont.)

\[ \text{id} + E \ast E \Rightarrow \text{id} + \text{id} \ast E \]

\[ \text{id} + \text{id} \ast E \Rightarrow \text{id} + \text{id} \ast \text{id} \]
Alternative Parse Tree & Derivation

E ⇒ E * E
⇒ E + E * E
⇒ id + E * E
⇒ id + id * E
⇒ id + id * id

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.

\[ E \Rightarrow E + E \Rightarrow id + E \Rightarrow id + E \ast E \]
\[ \Rightarrow id + id \ast E \Rightarrow id + id \ast id \]

_E_  \( \Rightarrow \)  _E_  \( \Rightarrow \)  _E_  \( + \)  _E_  \( \Rightarrow \)  _id_  \( + \)  _E_  \( \Rightarrow \)  _id_  \( + \)  _E_  \( \ast \)  _E_  \( \Rightarrow \)  _id_  \( + \)  _id_  \( \ast \)  _E_  \( \Rightarrow \)  _id_  \( + \)  _id_  \( \ast \)  _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.

\[
E \rightarrow E + E \mid E * E \mid \text{id} \mid (E)
\]

Disambiguate the grammar

\text{precedence:}\quad * \quad \text{(left associativity)}
\quad + \quad \text{(left associativity)}

Ex. \text{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:

\[
stmt \rightarrow \text{if } expr \text{ then } stmt \\
| \text{if } expr \text{ then } stmt \text{ else } stmt \\
| \text{other } (\text{any other statement})
\]

How is this parsed?

\[
\text{if } E_1 \text{ then } \text{if } E_2 \text{ then } S_1 \text{ else } S_2
\]
**Parse Trees for Example**

\[
stmt \rightarrow \text{if expr then stmt} \\
| \text{if expr then stmt else stmt} \\
| \text{other (any other statement)}
\]

**Form 1:**

If \( E_1 \) then if \( E_2 \) then \( S_1 \) else \( S_2 \)

Else must match to previous then. Structure indicates parse subtree for expression.

**Form 2:**

Two parse trees for an ambiguous sentence.
Ambiguity (cont.)

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

• The unambiguous grammar will be:

\[
\begin{align*}
\text{stmt} & \rightarrow \text{matchedstmt} \\
& \quad | \text{unmatchedstmt} \\
\text{matchedstmt} & \rightarrow \text{if} \ \text{expr} \ \text{then} \ \text{matchedstmt} \ \text{else} \ \text{matchedstmt} \\
& \quad | \ \text{otherstmts} \\
\text{unmatchedstmt} & \rightarrow \text{if} \ \text{expr} \ \text{then} \ \text{stmt} \\
& \quad | \ \text{if} \ \text{expr} \ \text{then} \ \text{matchedstmt} \ \text{else} \ \text{unmatchedstmt}
\end{align*}
\]

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
• \( L_1 = \{ \omega \omega \mid \omega \text{ is in } (a+b)^* \} \) 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
• \( L_2 = \{ a^n b^m c^n d^m \mid n \geq 1 \text{ and } m \geq 1 \} \) is not context-free
  ➔ declaring two functions (one with \( n \) parameters, the other one with \( m \) parameters), and then calling them with actual parameters.
Review

- Context-free grammar
  - Definition
  - Why use CFG for parsing
  - Derivations: left-most vs. right-most, id + id * id
  - Parse tree

- Context-free language vs. regular language, \{ (i)^i \mid i>0 \}
  - Closure property:
    - closed under union
    - Nonclosure under intersection, complement and difference
    - E.g., \{ a^n b^m c^k \mid n=m>0 \} \cap \{ a^n b^m c^k \mid m=k>0 \} = \{ a^n b^n c^n \mid n>0 \}

- Ambiguous grammar to unambiguous grammar
  - Grammar rewritten to eliminate the ambiguity
  - Enforce precedence and associativitiy
Let’s derive: \( id := id + \text{real} - \text{integer} ; \)

Left-most derivation:

\[
\begin{align*}
\text{assign_stmt} & \\
\to id := expr ; & \text{using production:} \\
\to id := expr \ op \ term ; & assign\_stmt \to id := expr ; \\
\to id := expr \ op \ term \ op \ term ; & expr \to expr \ op \ term \\
\to id := \text{term} \ op \ term \ op \ term ; & expr \to term \\
\to id := id \ op \ term \ op \ term ; & term \to id \\
\to id := id + \text{term} \ op \ term ; & op \to + \\
\to id := id + \text{real} \ op \ term ; & term \to real \\
\to id := id + \text{real} - \text{term} ; & op \to - \\
\to id := id + \text{real} - \text{integer} ; & term \to integer
\end{align*}
\]

Right-most derivation and parse tree?
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

  – **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:

\[
S \rightarrow c \ A \ d \\
A \rightarrow ab \mid a
\]

input: cad

Steps in top-down parse
Implementation of Recursive-Descent

E → T | T + E
T → int | int * T | ( 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:
  ```c
  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 $\alpha$.

• 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

\[ A \rightarrow A \alpha \mid \beta \quad \text{where } \beta \text{ does not start with } A \]
\[ \quad \downarrow \quad \text{eliminate immediate left recursion} \]
\[ A \rightarrow \beta A' \]
\[ A' \rightarrow \alpha A' \mid \varepsilon \quad \text{an equivalent grammar: replaced by right-recursion} \]

In general,
\[ A \rightarrow A \alpha_1 \mid \ldots \mid A \alpha_m \mid \beta_1 \mid \ldots \mid \beta_n \quad \text{where } \beta_1 \ldots \beta_n \text{ do not start with } A \]
\[ \quad \downarrow \quad \text{eliminate immediate left recursion} \]
\[ A \rightarrow \beta_1 A' \mid \ldots \mid \beta_n A' \]
\[ A' \rightarrow \alpha_1 A' \mid \ldots \mid \alpha_m A' \mid \varepsilon \quad \text{an equivalent grammar} \]
Eliminate immediate left recursion exercise 1 in class

Example

\[ E \rightarrow E + T \mid T \]
\[ T \rightarrow T * F \mid F \]
\[ F \rightarrow \text{id} \mid (E) \]

\[ E \rightarrow T E' \]
\[ E' \rightarrow +T E' \mid \varepsilon \]
\[ T \rightarrow F T' \]
\[ T' \rightarrow *F T' \mid \varepsilon \]
\[ F \rightarrow \text{id} \mid (E) \]
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.

\[
S \Rightarrow Aa \Rightarrow Sca \\
\text{or } A \Rightarrow Sc \Rightarrow Aac
\]

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  
    \[ A_i \rightarrow A_j \gamma \]  
    by  
    \[ A_i \rightarrow \alpha_1 \gamma | \ldots | \alpha_k \gamma \]  
    where \( A_j \rightarrow \alpha_1 | \ldots | \alpha_k \) are all the current \( A_j \)-productions.  
  }  
  - **eliminate immediate left-recursions among \( A_i \)-productions**
}

Algorithm to eliminate left recursion from a grammar.
Example

Example
S → Aa | b
A → Ac | Sd | ε

- 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 → Sd with A → Aad | bd
  So, we will have A → Ac | Aad | bd | ε
- Eliminate the immediate left-recursion in A
  A → bdA’ | A’
  A’ → cA’ | adA’ | ε

So, the resulting equivalent grammar which is not left-recursive is:
S → Aa | b
A → bdA’ | A’
A’ → cA’ | adA’ | ε
Reading materials

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


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
  – 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 $A \rightarrow \alpha$ 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 $A$)

```bash
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 → aBb | 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 $\varepsilon$-productions.

\[ A \rightarrow aA \mid bB \mid \varepsilon \]

• If all other productions fail, we should apply an $\varepsilon$-production. For example, if the current token is not a or b, we may apply the $\varepsilon$-production.

• Most correct choice: We should apply an $\varepsilon$-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 → aBe | cBd | C
B → bB | ε
C → 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;
        - call B;
        - match the current token with d, and move to the next token;
        - call B;
        f: - call C
    }
}

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

first set of C/A
Predictive Parsing and Left Factoring

• Recall the grammar
  \[ E \rightarrow T + E \mid T \]
  \[ T \rightarrow \text{int} \mid \text{int} \ast T \mid (E) \]

• 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:

\[ stmt \rightarrow \text{if expr then } stmt \text{ else } stmt \]
\[ \mid \text{if expr then } stmt \]

When do you know which one is valid?

What’s the general form of \( stmt \)?

\[ A \rightarrow \alpha \beta_1 \mid \alpha \beta_2 \]

\[ \alpha : \text{if expr then } stmt \]
\[ \beta_1 : \text{else stmt} \]
\[ \beta_2 : \epsilon \]

Transform to:

\[ A \rightarrow \alpha A' \]
\[ A' \rightarrow \beta_1 \mid \beta_2 \]

EXAMPLE:

\[ stmt \rightarrow \text{if expr then } stmt \text{ rest} \]
\[ rest \rightarrow \text{else stmt} \mid \epsilon \]

so, we can immediately expand \( A \) to \( \alpha A' \)
Left-Factoring -- Algorithm

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

\[ A \rightarrow \alpha \beta_1 | ... | \alpha \beta_n | \gamma_1 | ... | \gamma_m \]

convert it into

\[ A \rightarrow \alpha A' | \gamma_1 | ... | \gamma_m \]
\[ A' \rightarrow \beta_1 | ... | \beta_n \]
Left-Factoring – Example 1

\[ A \rightarrow abB \mid aB \mid cdg \mid cdeB \mid cdfB \]
\[ \downarrow \text{step 1} \]
\[ A \rightarrow aA' \mid cdg \mid cdeB \mid cdfB \]
\[ A' \rightarrow bB \mid B \]
\[ \downarrow \text{step 2} \]
\[ A \rightarrow aA' \mid cdA'' \]
\[ A' \rightarrow bB \mid B \]
\[ A'' \rightarrow g \mid eB \mid fB \]
Left-Factoring exercise 1 in class

\[ A \rightarrow ad \mid a \mid ab \mid abc \mid b \]

\[
\begin{align*}
A & \rightarrow ad \mid a \mid ab \mid abc \mid b \\
& \downarrow \text{step 1} \\
A & \rightarrow aA' \mid b \\
A' & \rightarrow d \mid \epsilon \mid b \mid bc \\
& \downarrow \text{step 2} \\
A & \rightarrow aA' \mid b \\
A' & \rightarrow d \mid \epsilon \mid bA'' \\
A'' & \rightarrow \epsilon \mid c
\end{align*}
\]

Normal form:

\[
\begin{align*}
A & \rightarrow aA' \mid b \\
A' & \rightarrow bA'' \mid d \mid \epsilon \\
A'' & \rightarrow c \mid \epsilon
\end{align*}
\]
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.

A $\rightarrow$ $\alpha_1$ | ... | $\alpha_n$

input: ... 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: \texttt{id + id}
Questions

• Compiler justification: correctness, usability, efficiency, more functionality
• Compiler update: less on frond-end, more on back-end
• Performance (speed, security, etc.) vary from programming languages
  • Naïve or horrible
  • Just-in-time (JIT) compilation
• Compiler and security (post-optimization): stack/heap canaries, CFI
• Optimization, more advanced techniques from research: graduate course
• Projects independent from class
• Notations in slides
• Why do we use flex/lex and bison/Yacc for projects
• Project assignments not clear
• Why latex, not markdown or word?
Review

Context-free grammar

Top-down parsing
  • recursive-descent parsing
  • recursive predicate parsing

Recursive-descent parser
  • backtracking
  • eliminating left-recursion

\[ B \rightarrow \text{begin DL SL end} \]
\[ \text{DL} \rightarrow \text{DL}; \text{d} | \text{d} \]
\[ \text{SL} \rightarrow \text{SL}; \text{s} | \text{s} \]

Write the recursive-descent parser (d,s, begin and end are terminals, suppose term(Token tok) return True iff tok==next)
Implementation of Recursive-Descent

\[
E \rightarrow T \mid T + E \\
T \rightarrow \text{int} \mid \text{int} \ast T \mid (E)
\]

```c
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());
```
bool B(){ return term(begin) && DL() && SL() && term(end) }
bool DL(){ return term(d) && DL’() }
bool DL’(){ TOKEN* save=next; return (next=save, DL’1()) || (next=save, DL’2()) }
bool DL’1(){ return (term(;) && term(d) && DL’) }
bool DL’2(){ return True }
bool SL(){ return term(s) && SL’() }
bool SL’(){ TOKEN* save=next; return (next=save, SL’1()) || (next=save, SL’2()) }
bool SL’1(){ return (term(;) && term(s) && SL’) }
bool SL’2(){ return True }
Review

Context-free grammar
Top-down parsing
  • recursive-descent parsing
  • recursive predicate parsing
Recursive-descent parser
  • **backtracking**
  • eliminating left-recursion (all CFG)
Recursive predicate parsing
  • **no backtracking**
  • left-factoring (subclass of CFG)

![Diagram](image_url)

A grammar $\Rightarrow$ elimination of left recursion $\Rightarrow$ a grammar suitable for predictive parsing (a LL(1) grammar) with no $100\%$ guarantee.
Recursive Predictive Parsing (Example)

A → aBe | cBd | C
B → bB | ε
C → 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;
    b: - match the current token with b,  
        and move to the next token;
    c: - match the current token with c,  
        and move to the next token;
    d: - call B;
    e: - 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;
    e, d: do nothing
  }
}

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

follow set of B

first set of C/A
B → begin DL SL end
DL → DL; d | d
SL → SL; s | s
Write the recursive predicate parser (d,s, begin and end are terminals, lookahead 1 symbol)

B → begin DL SL end
DL → d DL’
DL’ → ;d DL’ | ε
SL → s SL’
SL’ → ;s DL’ | ε
Review

Proc B{
    case of the current token{
        begin:  -match the current token with begin and move to the next token
                  -call DL
                  -call SL
                  -match the current token with end and move to the next token
        default: ErrorHandler1()   } }
Proc DL/SL{
    case of the current token{
        d:   -match the current token with d/s and move to the next token
                   -call DL’/SL’
        default: ErrorHandler2()   } }
Proc DL’/SL’{
    case of the current token{
        ;:   -match the current token with ; and move to the next token
                   -match the current token with d/s and move to the next token
                   -call DL’/SL’
        s/end:  - do nothing       // s in follow set of DL’, end in follow set of SL’
        default: ErrorHandler3()   } }
Non-Recursive / Table Driven

General parser behavior:  \( X : \) top of stack \( a : \) current input

1. When \( X=a = $ \) halt, accept, success

2. When \( X=a \neq $ \), 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 \( \rightarrow \) call recovery routine
   - if \( M[X,a] = \{X \rightarrow UVW\} \), POP \( X \), PUSH \( W,V,U \)
     Notice the pushing order
   - DO NOT expend any input
Algorithm for Non-Recursive Parsing

Set $ip$ to point to the first symbol of w$;$

repeat

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

if $X$ is terminal or $\$$ then

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 $Y_k, Y_{k-1}, \ldots, Y_1$ onto stack, with $Y_1$ on top

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

end

else  $error()$

until $X=\$$ /*  stack is empty  */

May also execute other code based on the production used, such as creating parse tree.
LL(1) Parser – Example

S → aBa
B → bB | ε

Sentence "abba" is correct or not?

<table>
<thead>
<tr>
<th>stack</th>
<th>input</th>
<th>output</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S</td>
<td>abba$</td>
<td>S → aBa</td>
</tr>
<tr>
<td>$aBa</td>
<td>abba$</td>
<td></td>
</tr>
<tr>
<td>$aB</td>
<td>bba$</td>
<td>B → bB</td>
</tr>
<tr>
<td>$aBb</td>
<td>bba$</td>
<td></td>
</tr>
<tr>
<td>$aB</td>
<td>ba$</td>
<td>B → bB</td>
</tr>
<tr>
<td>$aBb</td>
<td>ba$</td>
<td></td>
</tr>
<tr>
<td>$aB</td>
<td>a$</td>
<td>B → ε</td>
</tr>
<tr>
<td>$a</td>
<td>a$</td>
<td></td>
</tr>
<tr>
<td>$</td>
<td>$</td>
<td>accept, successful completion</td>
</tr>
</tbody>
</table>
LL(1) Parser – Example (cont.)

Outputs:

\[ S \rightarrow aBa \quad B \rightarrow bB \quad B \rightarrow bB \quad B \rightarrow \epsilon \]

Derivation (left-most):

\[ S \Rightarrow aBa \Rightarrow abBa \Rightarrow abbBa \Rightarrow abba \]
LL(1) Parser – Example 2

Example:

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

Table M

<table>
<thead>
<tr>
<th>Non-terminal</th>
<th>INPUT SYMBOL</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>id</td>
</tr>
<tr>
<td>E</td>
<td>E→TE'</td>
</tr>
<tr>
<td>E'</td>
<td>E'→+TE'</td>
</tr>
<tr>
<td>T</td>
<td>T→FT'</td>
</tr>
<tr>
<td>T'</td>
<td>T'→ε</td>
</tr>
<tr>
<td>F</td>
<td>F→id</td>
</tr>
</tbody>
</table>

how to construct \texttt{id+id*id} ?

Parsing table M for grammar
## LL(1) Parser – Example 2

<table>
<thead>
<tr>
<th>stack</th>
<th>input</th>
<th>output</th>
</tr>
</thead>
<tbody>
<tr>
<td>$E$</td>
<td>id+id*id$</td>
<td></td>
</tr>
<tr>
<td>$E'T$</td>
<td>id+id*id$</td>
<td>E → TE'</td>
</tr>
<tr>
<td>$E'T'F$</td>
<td>id+id*id$</td>
<td>T → FT'</td>
</tr>
<tr>
<td>$E'T'id$</td>
<td>id+id*id$</td>
<td>F → id</td>
</tr>
<tr>
<td>$E'T'$</td>
<td>+id*id$</td>
<td>T' → ε</td>
</tr>
<tr>
<td>$E'$</td>
<td>+id*id$</td>
<td>E' → +TE'</td>
</tr>
<tr>
<td>$E'T'$</td>
<td>id*id$</td>
<td></td>
</tr>
<tr>
<td>$E'T'F$</td>
<td>id*id$</td>
<td>T → FT'</td>
</tr>
<tr>
<td>$E'T'id$</td>
<td>id*id$</td>
<td>F → id</td>
</tr>
<tr>
<td>$E'T'$</td>
<td>*id$</td>
<td>T' → *FT'</td>
</tr>
<tr>
<td>$E'T'F*$</td>
<td>*id$</td>
<td></td>
</tr>
<tr>
<td>$E'T'F$</td>
<td>id$</td>
<td>F → id</td>
</tr>
<tr>
<td>$E'T'id$</td>
<td>id$</td>
<td></td>
</tr>
<tr>
<td>$E'T'$</td>
<td>$</td>
<td>T' → ε</td>
</tr>
<tr>
<td>$E'$</td>
<td>$</td>
<td>E' → ε</td>
</tr>
<tr>
<td>$</td>
<td>$</td>
<td>accept</td>
</tr>
</tbody>
</table>

The moves made by the predictive parser on input `id+id*id`.
Leftmost Derivation for the Example

The leftmost derivation for the example is as follows:

\[
\begin{align*}
E & \Rightarrow TE' \\
& \Rightarrow FT'E' \\
& \Rightarrow id T'E' \\
& \Rightarrow id E' \\
& \Rightarrow id + TE' \\
& \Rightarrow id + FT'E'
\end{align*}
\Rightarrow id + id T'E'
\Rightarrow id + id * FT'E'
\Rightarrow id + id * id T'E'
\Rightarrow id + id * id E'
\Rightarrow id + id * id E'
\Rightarrow id + id * id
\]
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 \( \alpha \) be a string of grammar symbols. \( \text{FIRST}(\alpha) \) is the set that includes every terminal that appears leftmost in \( \alpha \) or in any string originating from \( \alpha \).

NOTE: If \( \alpha \Rightarrow \varepsilon \), then \( \varepsilon \) is \( \text{FIRST}(\alpha) \).

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

NOTE: If \( S \Rightarrow \alpha A \), then $ is \( \text{FOLLOW}(A) \).
Compute FIRST for Any String X

1. If X is a terminal, FIRST(X) = {X}

2. If X→ ε is a production rule, add ε to FIRST(X)

3. If X is a non-terminal, and X→ Y₁Y₂…Yₖ is a production rule
   
   Place FIRST(Y₁) in FIRST(X)
   
   if Y₁⇒ ε, Place FIRST(Y₂) in FIRST(X)
   
   if Y₂⇒ ε, Place FIRST(Y₃) in FIRST(X)
   
   …
   
   if Yₖ₋₁⇒ ε, Place FIRST(Yₖ) in FIRST(X)
   
   NOTE: As soon as Yᵢ⇒ ε, Stop.

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

   Checking “Yᵢ⇒ ε?” essentially amounts to checking whether ε belongs to FIRST(Yᵢ)
Computing FIRST(X) :
All Grammar Symbols - continued

Informally, suppose we want to compute

\[ \text{FIRST}(X_1 \, X_2 \, \ldots \, X_n) = \text{FIRST} (X_1) \]

"+" FIRST \( (X_2) \) if \( \epsilon \) is in FIRST \( (X_1) \)

"+" FIRST \( (X_3) \) if \( \epsilon \) is in FIRST \( (X_2) \)

...\n
"+" FIRST \( (X_n) \) if \( \epsilon \) is in FIRST \( (X_{n-1}) \)

Note 1: Only add \( \epsilon \) to FIRST \( (X_1 \, X_2 \, \ldots \, X_n) \) if \( \epsilon \) 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 → TE’
E’ → +TE’ | ε
T → FT’
T’ → *FT’ | ε
F → (E) | id

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) }
Alternative way to compute FIRST

Computing FIRST for:

\[ E \rightarrow TE' \]
\[ E' \rightarrow + TE' | \epsilon \]
\[ T \rightarrow FT' \]
\[ T' \rightarrow * FT' | \epsilon \]
\[ F \rightarrow (E) | id \]

\text{FIRST}(E) \quad \text{FIRST}(T) \quad \text{FIRST}(F) \quad \text{FIRST}(E') \quad \text{FIRST}(T')

Overall:

\text{FIRST}(E) = \text{FIRST}(T) = \text{FIRST}(F) = \{ (, \text{id}) \}
\text{FIRST}(E') = \{ +, \epsilon \}
\text{FIRST}(T') = \{ *, \epsilon \}
Compute FOLLOW (for non-terminals)

1. If $S$ is the start symbol $\Rightarrow$ \$ is in FOLLOW($S$) \hspace{1cm} \text{Initially S\$}

2. If $A \rightarrow \alpha B \beta$ is a production rule $\Rightarrow$ everything in FIRST($\beta$) is FOLLOW($B$) except $\varepsilon$

3. If ( $A \rightarrow \alpha B$ is a production rule ) or ( $A \rightarrow \alpha B \beta$ is a production rule and $\varepsilon$ is in FIRST($\beta$) ) $\Rightarrow$ 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 $X \rightarrow X_1 X_2 \ldots X_j X_{j+1} \ldots X_m$
   For j=1 to m,
     if $X_j$ is a non-terminal then:

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

   If FIRST($X_{j+1}, \ldots, X_m$) contains \(\varepsilon\) or $X_{j+1}, \ldots, X_m = \varepsilon$
     then FOLLOW($X_j$) = FOLLOW($X_j$) \cup FOLLOW($X$);
FOLLOW Example

Example:

\[ X \rightarrow X_1 X_2 \ldots X_j X_{j+1} \ldots X_m \]

For \( j=1 \) to \( m \),

if \( X_j \) is a non-terminal then:

\[ \text{FOLLOW}(X_j) = \text{FOLLOW}(X_j) \cup (\text{FIRST}(X_{j+1}, \ldots, X_m) - \{\varepsilon\}) \]

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

then \( \text{FOLLOW}(X_j) = \text{FOLLOW}(X_j) \cup \text{FOLLOW}(X) \)

\[
\begin{align*}
\text{FIRST}(F) &= \{ (, \text{id}) \} \\
\text{FIRST}(T') &= \{ *, \varepsilon \} \\
\text{FIRST}(T) &= \{ (, \text{id}) \} \\
\text{FIRST}(E') &= \{ +, \varepsilon \} \\
\text{FIRST}(E) &= \{ (, \text{id}) \} \\
\text{FIRST}(\ast FT') &= \{ \ast \} \\
\text{FIRST}((E)) &= \{ ( ) \} \\
\text{FIRST}(\text{id}) &= \{ \text{id} \} \\
\text{FIRST}(+ TE') &= \{ + \} \\
\text{FIRST}(\varepsilon) &= \{ \varepsilon \} \\
\text{FIRST}(TE') &= \{ (, \text{id}) \} \\
\text{FIRST}(FT') &= \{ (, \text{id}) \}
\end{align*}
\]

\[
\begin{align*}
\text{FOLLOW}(E) &= \{ \ ), \$, \} \\
\text{FOLLOW}(E') &= \{ \ ), \$, \} \\
\text{FOLLOW}(T) &= \{ +, \), \$, \} \\
\text{FOLLOW}(T') &= \{ +, \), \$, \} \\
\text{FOLLOW}(F) &= \{ \ast, +, \), \$, \}
\end{align*}
\]
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($\alpha$), then when $a$=input, replace $A$ with $\alpha$ (in the stack).

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

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

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

($\alpha \Rightarrow^* \varepsilon$: i.e., FIRST($\alpha$) contains $\varepsilon$.)
Review: 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 \neq $ , 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 $\rightarrow$ call recovery routine
   - if $M[X,a] = \{X \rightarrow UVW\}$, POP $X$, PUSH $W,V,U$
     DO NOT expend any input

Notice the pushing order
**Review: 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($\alpha$), then when $a =$ input, replace $A$ with $\alpha$ (in the stack).

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

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

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

($\alpha \Rightarrow^* \varepsilon$: i.e., FIRST($\alpha$) contains $\varepsilon$.)
Review: Compute FIRST for Any String X

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

   Place FIRST(Y_1) in FIRST(X)

   if Y_1 ⇒ ε, Place FIRST(Y_2) in FIRST(X)

   if Y_2 ⇒ ε, Place FIRST(Y_3) in FIRST(X)

   …

   if Y_{k-1} ⇒ ε, Place FIRST(Y_k) in FIRST(X)

NOTE: As soon as Y_i ⇒ ε, Stop.

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

Checking “Y_i ⇒ ε ?” essentially amounts to checking whether ε belongs to FIRST(Y_i)
Review: Compute FOLLOW (for non-terminals)

1. If $S$ is the start symbol, $\Rightarrow$ $\$ \text{ is in FOLLOW}(S)$  \hspace{1cm} \text{Initially } S\$ \\

2. If $A \rightarrow \alpha B\beta$ is a production rule, $\Rightarrow$ everything in $\text{FIRST}(\beta)$ is FOLLOW(B) except $\epsilon$

3. If ( $A \rightarrow \alpha B$ is a production rule ) or ( $A \rightarrow \alpha B\beta$ is a production rule and $\epsilon$ is in $\text{FIRST}(\beta)$ ) $\Rightarrow$ everything in $\text{FOLLOW}(A)$ 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.
Exercises

• Given the following grammar:
• \( S \rightarrow aAb | Sc | \varepsilon \)
  \( A \rightarrow aAb | \varepsilon \)

• Please eliminate left-recursive,
• Then compute FIRST() and FOLLOW() for each non-terminal.
• $S \rightarrow aAb | Sc | \varepsilon$
  $A \rightarrow aAb | \varepsilon$

  Eliminating left-recursive

  $S \rightarrow aAbS' \mid S'$
  $S' \rightarrow cS' | \varepsilon$
  $A \rightarrow aAb | \varepsilon$

  **FIRST**($S$) = \{a, c, \varepsilon\}  \hspace{1cm} \text{FOLLOW}(S) = \{$$
  \text{FIRST}(S') = \{c, \varepsilon\}  \hspace{1cm} \text{FOLLOW}(S') = \{$$
  \text{FIRST}(A) = \{a, \varepsilon\}  \hspace{1cm} \text{FOLLOW}(A) = \{b\}
  \text{FIRST}(aAbS') = \{a, c, \varepsilon\}
  \text{FIRST}(cS') = \{c, \varepsilon\}
  \text{FIRST}(aAb) = \{a, \varepsilon\}$
Constructing LL(1) Parsing Table

Algorithm:

1. Repeat Steps 2 & 3 for each rule A → α
2. Terminal a in FIRST(α)? Add A → α to M[A, a ]
3.1 ε in FIRST(α)? Add A → α to M[A, b ] for all terminals b in FOLLOW(A).
3.2 ε in FIRST(α) and $ in FOLLOW(A)? Add A → α to M[A, $ ]
4. All undefined entries are errors.
FOLLOW Example

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

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

FIRST(F) = {(, id)
FIRST(T') = {*, ε}
FIRST(T) = {(, id)
FIRST(E') = {+, ε}
FIRST(E) = {(, id)
FIRST(TE') ={(, id)
FIRST(+TE') = { +}
FIRST(ε) = { ε}
FIRST(FT') = { (, id)
FIRST(*FT') = { *}
FIRST((E)) = { ()
FIRST(id) = { id}
Constructing LL(1) Parsing Table -- Example

<table>
<thead>
<tr>
<th>Rule</th>
<th>FIRST(Left-hand Side)</th>
<th>Transformation</th>
</tr>
</thead>
<tbody>
<tr>
<td>E → TE'</td>
<td>{ (, id}</td>
<td>E → TE’ into M[E,()] and M[E,id]</td>
</tr>
<tr>
<td>E' → +TE'</td>
<td>{ +}</td>
<td>E’ → +TE’ into M[E’,+]</td>
</tr>
<tr>
<td>E’ → ε</td>
<td>{ ε}</td>
<td>none</td>
</tr>
<tr>
<td></td>
<td>but since ε in FIRST(ε) and FOLLOW(E’)={$,})</td>
<td>E’ → ε into M[E’,$] and M[E’,)]</td>
</tr>
<tr>
<td>T → FT’</td>
<td>{ (, id}</td>
<td>T → FT’ into M[T,()] and M[T,id]</td>
</tr>
<tr>
<td>T’ → *FT’</td>
<td>{ *}</td>
<td>T’ → <em>FT’ into M[T’,</em>]</td>
</tr>
<tr>
<td>T’ → ε</td>
<td>{ ε}</td>
<td>none</td>
</tr>
<tr>
<td></td>
<td>but since ε in FIRST(ε) and FOLLOW(T’)={$,},+}</td>
<td>T’ → ε into M[T’,$], M[T’,)] and M[T’,+]</td>
</tr>
<tr>
<td>F → (E)</td>
<td>{ ()}</td>
<td>F → (E) into M[F,()]</td>
</tr>
<tr>
<td>F → id</td>
<td>{ id}</td>
<td>F → id into M[F,id]</td>
</tr>
</tbody>
</table>
## Constructing LL(1) Parsing Table (cont.)

<table>
<thead>
<tr>
<th>Non-terminal</th>
<th>Input symbol</th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>id</td>
<td>+</td>
<td>*</td>
<td>(</td>
<td>)</td>
</tr>
<tr>
<td>E</td>
<td>E → TE’</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>E’</td>
<td>E’ → +TE’</td>
<td>E’ → ε</td>
<td>E’ → ε</td>
<td></td>
<td></td>
</tr>
<tr>
<td>T</td>
<td>T → FT’</td>
<td>T → FT’</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>T’</td>
<td>T’ → ε</td>
<td>T’ → *FT’</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>F</td>
<td>F → id</td>
<td>F → (E)</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Parsing table M for grammar
LL(1) Grammars

L : Scan input from Left to Right
L : Construct a Leftmost Derivation
I : 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) ⇔ when \[ A \rightarrow \alpha | \beta \]
  1. \( \alpha \) and \( \beta \) do not derive strings starting with the same terminal \( a \)
  2. Either \( \alpha \) or \( \beta \) can derive \( \varepsilon \), but not both.
  3. If \( \beta \) can derive to \( \varepsilon \), then \( \alpha \) 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(\varepsilon) = \{\varepsilon\}
FIRST(b) = \{b\}

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

Problem \rightarrow ambiguity

two production rules for M[E,e]
Exercise in class

• Grammar:
  \( \text{lexp} \rightarrow \text{atom} | \text{list} \)
  \( \text{atom} \rightarrow \text{num} | \text{id} \)
  \( \text{list} \rightarrow (\text{lexp}_{\text{seq}}) \)
  \( \text{lexp}_{\text{seq}} \rightarrow \text{lexp}_{\text{seq}} \text{lexp} | \text{lexp} \)

Please calculate FIRST(), FOLLOW(), then construct parsing table and see whether the grammar is LL(1).
Eliminating left recursive

\[\text{lexp} \rightarrow \text{atom} \mid \text{list}\]
\[\text{atom} \rightarrow \text{numr} \mid \text{id}\]
\[\text{list} \rightarrow ( \text{lexp-seq } )\]
\[\text{lexp-seq} \rightarrow \text{lexp } \text{lexp-seq}'\]
\[\text{lexp-seq}' \rightarrow \text{lexp } \text{lexp-seq}' \mid \varepsilon\]
Answer

lexp → atom | list
atom → num | id
list → ( lexp-seq )
lexp-seq → lexp lexp-seq’
lexp-seq’ → lexp lexp-seq’ | ε

FIRST(lexp)={ (, num, id }
FIRST(atom)={ num, id }
FIRST(list)={ ( }
FIRST(lexp-seq)={ (, num, id }
FIRST(lexpseq’)={ (,num,id,ε}

FOLLOW(lexp-seq’)={}
Answer

1. $\text{lexp} \rightarrow \text{atom}$
2. $\text{lexp} \rightarrow \text{list}$
3. $\text{atom} \rightarrow \text{num}$
4. $\text{atom} \rightarrow \text{id}$
5. $\text{list} \rightarrow (\text{lexp_seq})$
6. $\text{lexp_seq} \rightarrow \text{lexp} \text{lexp_seq}'$
7. $\text{lexp_seq}' \rightarrow \text{lexp} \text{lexp_seq}'$
8. $\text{lexp_seq}' \rightarrow \varepsilon$

<table>
<thead>
<tr>
<th></th>
<th>num</th>
<th>id</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>lexp</strong></td>
<td>1</td>
<td>1</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>atom</strong></td>
<td>3</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>list</strong></td>
<td></td>
<td></td>
<td>5</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>lexp_seq</strong></td>
<td>6</td>
<td>6</td>
<td>6</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>lexp_seq’</strong></td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>8</td>
<td></td>
</tr>
</tbody>
</table>

No conflict, Grammar is LL(1)
A Grammar which is not LL(1) (cont.)

- A left recursive grammar cannot be a LL(1) grammar.
  - A → Aα | β
    - any terminal that appears in FIRST(β) also appears FIRST(Aα) because Aα ⇒ βα.
    - If β is ε, any terminal that appears in FIRST(α) also appears in FIRST(Aα) and FOLLOW(A).

- A grammar is not left factored, it cannot be a LL(1) grammar
  - A → αβ₁ | αβ₂
    - any terminal that appears in FIRST(αβ₁) also appears in FIRST(αβ₂).

- 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 that unmatched terminal is inserted.
Panic-Mode Error Recovery - Example

S → AbS | e | ε
A → a | cAd

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

\[
\begin{array}{|c|c|c|c|c|c|c|}
\hline
& a & b & c & d & e & $ \\
\hline
S & S \rightarrow AbS & sync & S \rightarrow AbS & sync & S \rightarrow e & S \rightarrow \varepsilon \\
A & A \rightarrow a & sync & A \rightarrow cAd & sync & sync & sync \\
\hline
\end{array}
\]

Stack | Input | Output
--- | --- | ---
$S$ | aab$ | S → AbS
$SbA$ | aab$ | A → a
$Sba$ | aab$ |
$Sb$ | ab$ | Error: missing b, inserted
$S$ | ab$ | S → AbS
$SbA$ | ab$ | A → a
$Sba$ | ab$ |
$Sb$ | b$ |
$S$ | $ | S → ε
$ | $ | accept

Stack | Input | Output
--- | --- | ---
$S$ | ceadb$ | S → AbS
$SbA$ | ceadb$ | A → cAd
$Sba$ | 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 → ε
$ | $ | 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.

  \[ S \Rightarrow r_1 \Rightarrow \ldots \Rightarrow r_n \Rightarrow \omega \]  
  (the right-most derivation of \( \omega \))  
  \[ \leftarrow \]  
  (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 \( A \rightarrow X_1X_2\ldots X_n \).
  - Operator precedence parsers
  - LR parsers (SLR, LR, LALR)
Bottom-Up Parsing -- Example

S → aABb
A → aA | a
B → bB | b

input string: aaabb

S

\[ S \Rightarrow aABb \Rightarrow aAbb \Rightarrow aaAbb \Rightarrow aaabb \]

Right Sentential Forms
Naive algorithm

\[ S = γ_0 \xrightarrow{rm} γ_1 \xrightarrow{rm} γ_2 \xrightarrow{rm} \ldots \xrightarrow{rm} γ_{n-1} \xrightarrow{rm} γ_n = \omega \]

Let \( \omega = \) input string
repeat
    pick a non-empty substring \( β \) of \( \omega \) where \( A \rightarrow β \) is a production
    if (no such \( β \))
        backtrack
    else
        replace one \( β \) by \( A \) in \( \omega \)
until \( \omega = \) “S” (the start symbol) or all possibilities are exhausted
Questions

Let $\omega = \text{input string}$
repeat
    pick a non-empty substring $\beta$ of $\omega$ where $A \rightarrow \beta$ is a production
    if (no such $\beta$)
        backtrack
    else
        replace one $\beta$ by $A$ in $\omega$
until $\omega = \text{"S"}$ (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 $\omega$ 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
  – $\beta$ 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 \( \gamma (\equiv \alpha \beta \omega) \) is \((t, p)\)
  – \(t\): a production \(A \rightarrow \beta\)
  – \(p\): a position of \(\gamma\)
    where the string \(\beta\) may be found and replaced by \(A\) to produce
    the previous right-sentential form in a rightmost derivation of \(\gamma\).
    \[
    S \xrightarrow{\text{rm}}^* \alpha A \omega \xrightarrow{\text{rm}} \alpha \beta \omega
    \]

• 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
  ⇒ rightmost derivation is unique
  ⇒ a unique production $A \rightarrow \beta$ applied to take $r_i$ to $r_{i+1}$ at the position $k$
  ⇒ a unique handle $(A \rightarrow \beta, k)$
Handle Pruning

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

\[ S = \gamma_0 \mapsto \gamma_1 \mapsto \gamma_2 \mapsto \ldots \mapsto \gamma_{n-1} \mapsto \gamma_n = \omega \]

Let \( \omega = \text{input string} \)

repeat

pick a non-empty substring \( \beta \) of \( \omega \) where \( A \rightarrow \beta \) is a production

if (no such \( \beta \))

backtrack

else

replace one \( \beta \) by \( A \) in \( \omega \)

until \( \omega = \text{“S”} \) (the start symbol) or all possibilities are exhausted
Review: Bottom-up Parsing and Handle

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

\[ S = \gamma_0 \xrightarrow{\text{rm}} \gamma_1 \xrightarrow{\text{rm}} \gamma_2 \xrightarrow{\text{rm}} \ldots \xrightarrow{\text{rm}} \gamma_{n-1} \xrightarrow{\text{rm}} \gamma_n = \omega \]

Let \( \omega = \text{input string} \)

repeat

- pick a non-empty substring \( \beta \) of \( \omega \) where \( A \rightarrow \beta \) is a production
- if (no such \( \beta \))
  - backtrack
- else
  - replace one \( \beta \) by \( A \) in \( \omega \)

until \( \omega = \text{“S”} \) (the start symbol) or all possibilities are exhausted
Example

E → E+T  |  T  x+y * z
T → T*F  |  F  id+id*id
F → (E)  |  id

Right-Most Sentential Form

id+id*id  
F+id*id  
T+id*id  
E+id*id  
E+F*id  
E+T*id  
E+T*F  
E+T  
E

Handle

F → id, 1  
T → F, 1  
E → T, 1  
F → id, 3  
T → F, 3  
F → id, 5  
T → T*F, 3  
E → E+T, 1
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

<table>
<thead>
<tr>
<th>Stack</th>
<th>Input</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>$</td>
<td>id+id*id$</td>
<td>shift</td>
</tr>
<tr>
<td>$id</td>
<td>+id*id$</td>
<td>reduce by F → id</td>
</tr>
<tr>
<td>$F</td>
<td>+id*id$</td>
<td>reduce by T → F</td>
</tr>
<tr>
<td>$T</td>
<td>+id*id$</td>
<td>reduce by E → T</td>
</tr>
<tr>
<td>$E</td>
<td>+id*id$</td>
<td>shift</td>
</tr>
<tr>
<td>$E+$</td>
<td>id*id$</td>
<td>shift</td>
</tr>
<tr>
<td>$E+id$</td>
<td>*id$</td>
<td>reduce by F → id</td>
</tr>
<tr>
<td>$E+F$</td>
<td>*id$</td>
<td>reduce by T → F</td>
</tr>
<tr>
<td>$E+T$</td>
<td>*id$</td>
<td>shift</td>
</tr>
<tr>
<td>$E+T*$</td>
<td>id$</td>
<td>shift</td>
</tr>
<tr>
<td>$E+T*id$</td>
<td>$</td>
<td>reduce by F → id</td>
</tr>
<tr>
<td>$E+T*F$</td>
<td>$</td>
<td>reduce by T → T*F</td>
</tr>
<tr>
<td>$E+T$</td>
<td>$</td>
<td>reduce by E → E+T</td>
</tr>
<tr>
<td>$E$</td>
<td>$</td>
<td>accept</td>
</tr>
</tbody>
</table>

E → E+T | T
T → T*F | F
F → (E) | id

Handle
F → id, 1
T → F, 1
E → T, 1
F → id, 3
T → F, 3
F → id, 5
T → T*F, 3
E → E+T, 1

x+y * z
id+id*id

E ⇒ E+T ⇒ E+T*F ⇒ E+T*id ⇒ E+F*id
⇒ E+id*id ⇒ T+id*id ⇒ F+id*id ⇒ id+id*id
**Question**

E → E+E | E*E | id | (E)

Ex. id + id * id

<table>
<thead>
<tr>
<th>Stack</th>
<th>Input</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>?</td>
<td>?</td>
<td>?</td>
</tr>
</tbody>
</table>
**Question**

\[ E \rightarrow E + E \mid E \cdot E \mid \text{id} \mid (E) \]

Ex. id + id * id

<table>
<thead>
<tr>
<th>Stack</th>
<th>Input</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>$</td>
<td>id + id * id$</td>
<td>shift</td>
</tr>
<tr>
<td>$id</td>
<td>+ id * id$</td>
<td>reduce by E → id</td>
</tr>
<tr>
<td>$E</td>
<td>+ id * id$</td>
<td>shift</td>
</tr>
<tr>
<td>$E +</td>
<td>id * id$</td>
<td>shift</td>
</tr>
<tr>
<td>$E + id$</td>
<td>* id$</td>
<td>reduce by E → id</td>
</tr>
<tr>
<td>$E + E$</td>
<td>* id$</td>
<td>reduce by E → E + E</td>
</tr>
<tr>
<td>$E</td>
<td>* id$</td>
<td>shift</td>
</tr>
<tr>
<td>$E *</td>
<td>id$</td>
<td>shift</td>
</tr>
<tr>
<td>$E * id$</td>
<td>$</td>
<td>reduce by E → id</td>
</tr>
<tr>
<td>$E * E$</td>
<td>$</td>
<td>reduce by E → E * E</td>
</tr>
<tr>
<td>$E</td>
<td>$</td>
<td>accept</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Stack</th>
<th>Input</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>$</td>
<td>id + id * id$</td>
<td>shift</td>
</tr>
<tr>
<td>$id</td>
<td>+ id * id$</td>
<td>reduce by E → id</td>
</tr>
<tr>
<td>$E</td>
<td>+ id * id$</td>
<td>shift</td>
</tr>
<tr>
<td>$E +</td>
<td>id * id$</td>
<td>shift</td>
</tr>
<tr>
<td>$E + id$</td>
<td>* id$</td>
<td>reduce by E → id</td>
</tr>
<tr>
<td>$E + E$</td>
<td>* id$</td>
<td>shift</td>
</tr>
<tr>
<td>$E + E *</td>
<td>* id$</td>
<td>shift</td>
</tr>
<tr>
<td>$E + E *id</td>
<td>id$</td>
<td>reduce by E → id</td>
</tr>
<tr>
<td>$E + E * E</td>
<td>$</td>
<td>reduce by E → E * E</td>
</tr>
<tr>
<td>$E + E</td>
<td>$</td>
<td>reduce by E → E + E</td>
</tr>
<tr>
<td>$E</td>
<td>$</td>
<td>accept</td>
</tr>
</tbody>
</table>

*shift/reduce conflict*
Question

S → aA | aB , A → c, B → c

Ex. ac

<table>
<thead>
<tr>
<th>Stack</th>
<th>Input</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>$</td>
<td>ac$</td>
<td>shift</td>
</tr>
<tr>
<td>$a</td>
<td>c$</td>
<td>shift</td>
</tr>
<tr>
<td>$ac</td>
<td>$</td>
<td>reduce by which?</td>
</tr>
</tbody>
</table>
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 → E+E | E*E | id | (E)

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

<table>
<thead>
<tr>
<th>Stack</th>
<th>Input</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>$</td>
<td>id+id*id$</td>
<td>shift</td>
</tr>
<tr>
<td>$id</td>
<td>+id*id$</td>
<td>reduce by E → id</td>
</tr>
<tr>
<td>$E</td>
<td>+id*id$</td>
<td>shift</td>
</tr>
<tr>
<td>$E+</td>
<td>id*id$</td>
<td>shift</td>
</tr>
<tr>
<td>$E+id</td>
<td>*id$</td>
<td>reduce by E → id</td>
</tr>
<tr>
<td>$E+E</td>
<td>*id$</td>
<td>reduce by E → E+E</td>
</tr>
<tr>
<td>$E</td>
<td>*id$</td>
<td>shift</td>
</tr>
<tr>
<td>$E*</td>
<td>id$</td>
<td>shift</td>
</tr>
<tr>
<td>$E*id</td>
<td>$</td>
<td>reduce by E → id</td>
</tr>
<tr>
<td>$E*E</td>
<td>$</td>
<td>reduce by E → E*E</td>
</tr>
<tr>
<td>$E</td>
<td>$</td>
<td>accept</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Stack</th>
<th>Input</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>$</td>
<td>id+id*id$</td>
<td>shift</td>
</tr>
<tr>
<td>$id</td>
<td>+id*id$</td>
<td>reduce by E → id</td>
</tr>
<tr>
<td>$E</td>
<td>+id*id$</td>
<td>shift</td>
</tr>
<tr>
<td>$E+</td>
<td>id*id$</td>
<td>shift</td>
</tr>
<tr>
<td>$E+id</td>
<td>*id$</td>
<td>reduce by E → id</td>
</tr>
<tr>
<td>$E+E</td>
<td>*id$</td>
<td>reduce by E → id</td>
</tr>
<tr>
<td>$E+E*</td>
<td>*id$</td>
<td>shift</td>
</tr>
<tr>
<td>$E+E*id</td>
<td>id$</td>
<td>reduce by E → id</td>
</tr>
<tr>
<td>$E+E*E</td>
<td>$</td>
<td>reduce by E → E*E</td>
</tr>
<tr>
<td>$E+E</td>
<td>$</td>
<td>reduce by E → E+E</td>
</tr>
<tr>
<td>$E</td>
<td>$</td>
<td>accept</td>
</tr>
</tbody>
</table>

shift/reduce conflict
Operator-Precedence Parser

• **Operator grammar**
  – small, but an important class of grammars
  – we may have an efficient operator precedence parser (a shift-reduce parser) for an operator grammar.

• In an *operator grammar*, **no production** rule can have:
  – \( \varepsilon \) at the right side
  – two adjacent non-terminals at the right side.

Ex:

\[
\begin{align*}
E \rightarrow & \ AB & E \rightarrow & \ EOE & E \rightarrow & \ E+E \\
A \rightarrow & \ a & E \rightarrow & \ id & E*E \rightarrow & \\
B \rightarrow & \ b & O \rightarrow & \ + \ | \ * \ | \ / & E/E \rightarrow & \ id \\
\end{align*}
\]

not operator grammar not operator grammar operator grammar
Precedence Relations

• In operator-precedence parsing, we define three disjoint precedence relations between certain pairs of terminals.

  \[ a < \cdot b \]  b has higher precedence than a
  \[ a = \cdot b \]  b has same precedence as a
  \[ a \cdot > b \]  b has lower precedence than a

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

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

• In our input string \(a_1a_2...a_n\), we insert the precedence relation between the pairs of terminals (the precedence relation holds between the terminals in that pair).
Using Operator -Precedence Relations

\[ E \rightarrow E+E \mid E-E \mid E*E \mid E/E \mid E^E \mid (E) \mid -E \mid \text{id} \]

The partial operator-precedence table for this grammar:

<table>
<thead>
<tr>
<th></th>
<th>id</th>
<th>+</th>
<th>*</th>
<th>$</th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td>.&gt;</td>
<td>.&gt;</td>
<td>.&gt;</td>
<td></td>
</tr>
<tr>
<td>+</td>
<td>&lt;</td>
<td>.&gt;</td>
<td>&lt;</td>
<td>.&gt;</td>
</tr>
<tr>
<td>*</td>
<td>&lt;</td>
<td>.&gt;</td>
<td>&gt;</td>
<td>.&gt;</td>
</tr>
<tr>
<td>$</td>
<td>&lt;</td>
<td>&lt;</td>
<td>&lt;</td>
<td></td>
</tr>
</tbody>
</table>

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

\[ $ \prec id \succ + \prec id \succ * \prec id \succ $ \]
To Find The Handles

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

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

$ id + id * id $  
$ E + id * id $  
$ E + E * id $  
$ E + E * E $  
$ E + E $  
$ E $
Operator-Precedence Parsing Algorithm

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

set $p$ to point to the first symbol of $w\$ ;

repeat forever
  if ($\$ is on top of the stack and $p$ points to $\$ ) then return
  else {
    let $a$ be the topmost terminal symbol on the stack and let $b$ be the symbol pointed to by $p$;
    if ($a < b$ or $a = b$) then {
      /* SHIFT */
      push $b$ onto the stack;
      advance $p$ to the next input symbol;
    }
    else if ($a > b$) then /* REDUCE */
      repeat pop stack
      until ( the top of stack terminal is related by $<$ to the terminal most recently popped );
    else error();
  }
}
### Operator-Precedence Parsing Algorithm -- Example

<table>
<thead>
<tr>
<th>stack</th>
<th>input</th>
<th>action</th>
</tr>
</thead>
<tbody>
<tr>
<td>$</td>
<td>id+id*id$</td>
<td>$ \leftarrow \text{id} \quad \text{shift} $</td>
</tr>
<tr>
<td>$id$</td>
<td>+id*id$</td>
<td>id \rightarrow + \quad \text{reduce} \quad E \rightarrow \text{id}</td>
</tr>
<tr>
<td>$</td>
<td>+id*id$</td>
<td>shift</td>
</tr>
<tr>
<td>$+$</td>
<td>id*id$</td>
<td>shift</td>
</tr>
<tr>
<td>$+$id</td>
<td>*id$</td>
<td>id \rightarrow * \quad \text{reduce} \quad E \rightarrow \text{id}</td>
</tr>
<tr>
<td>$+$</td>
<td>*id$</td>
<td>shift</td>
</tr>
<tr>
<td>$+$*</td>
<td>id$</td>
<td>shift</td>
</tr>
<tr>
<td>$+$*id</td>
<td>$</td>
<td>id \rightarrow $ \quad \text{reduce} \quad E \rightarrow \text{id}</td>
</tr>
<tr>
<td>$+$*</td>
<td>$</td>
<td>* \rightarrow $ \quad \text{reduce} \quad E \rightarrow E*E</td>
</tr>
<tr>
<td>$+$</td>
<td>$</td>
<td>+ \rightarrow $ \quad \text{reduce} \quad E \rightarrow E+E</td>
</tr>
<tr>
<td>$</td>
<td>$</td>
<td>accept</td>
</tr>
</tbody>
</table>
How to Create Operator-Precedence Relations

- We use associativity and precedence relations among operators.

1. If operator $O_1$ has higher precedence than operator $O_2$,
   $$O_1 \rightarrow O_2 \quad \text{and} \quad O_2 \leftarrow O_1$$

2. If operator $O_1$ and operator $O_2$ have equal precedence,
   they are left-associative
   $$O_1 \rightarrow O_2 \quad \text{and} \quad O_2 \rightarrow O_1$$
   they are right-associative
   $$O_1 \leftarrow O_2 \quad \text{and} \quad O_2 \leftarrow O_1$$

3. For all operators $O$,
   $$O \leftarrow \text{id}, \quad \text{id} \rightarrow O, \quad O \leftarrow (, (\leftarrow O, O \rightarrow ), ) \rightarrow O, \quad O \rightarrow $, \quad $ \leftarrow O$$

4. Also, let
   $$(\rightarrow \cdot ) \quad $ \leftarrow ( \quad \text{id} \rightarrow ) \quad ) \rightarrow$$
   $$(\leftarrow \cdot ( \quad $ \leftarrow \text{id} \quad \text{id} \rightarrow $ \quad ) \rightarrow )$$
   $$(\leftarrow \text{id}$$
## Operator-Precedence Relations

<table>
<thead>
<tr>
<th></th>
<th>+</th>
<th>-</th>
<th>*</th>
<th>/</th>
<th>^</th>
<th>id</th>
<th>(</th>
<th>)</th>
<th>$</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>&gt;</td>
<td>&gt;</td>
<td>&lt;</td>
<td>&lt;</td>
<td>&lt;</td>
<td>&lt;</td>
<td>&lt;</td>
<td>&gt;</td>
<td>&gt;</td>
</tr>
<tr>
<td>-</td>
<td>&gt;</td>
<td>&gt;</td>
<td>&lt;</td>
<td>&lt;</td>
<td>&lt;</td>
<td>&lt;</td>
<td>&lt;</td>
<td>&gt;</td>
<td>&gt;</td>
</tr>
<tr>
<td>*</td>
<td>&gt;</td>
<td>&gt;</td>
<td>&gt;</td>
<td>&gt;</td>
<td>&lt;</td>
<td>&lt;</td>
<td>&lt;</td>
<td>&gt;</td>
<td>&gt;</td>
</tr>
<tr>
<td>/</td>
<td>&gt;</td>
<td>&gt;</td>
<td>&gt;</td>
<td>&gt;</td>
<td>&lt;</td>
<td>&lt;</td>
<td>&lt;</td>
<td>&gt;</td>
<td>&gt;</td>
</tr>
<tr>
<td>^</td>
<td>&gt;</td>
<td>&gt;</td>
<td>&gt;</td>
<td>&gt;</td>
<td>&lt;</td>
<td>&lt;</td>
<td>&lt;</td>
<td>&gt;</td>
<td>&gt;</td>
</tr>
<tr>
<td>id</td>
<td>&gt;</td>
<td>&gt;</td>
<td>&gt;</td>
<td>&gt;</td>
<td>&gt;</td>
<td>&gt;</td>
<td>&gt;</td>
<td>&gt;</td>
<td>&gt;</td>
</tr>
<tr>
<td>(</td>
<td>&lt;</td>
<td>&lt;</td>
<td>&lt;</td>
<td>&lt;</td>
<td>&lt;</td>
<td>&lt;</td>
<td>&lt;</td>
<td>=</td>
<td>&gt;</td>
</tr>
<tr>
<td>)</td>
<td>&gt;</td>
<td>&gt;</td>
<td>&gt;</td>
<td>&gt;</td>
<td>&gt;</td>
<td>&gt;</td>
<td>&gt;</td>
<td>&gt;</td>
<td>&gt;</td>
</tr>
<tr>
<td>$</td>
<td>&lt;</td>
<td>&lt;</td>
<td>&lt;</td>
<td>&lt;</td>
<td>&lt;</td>
<td>&lt;</td>
<td>&lt;</td>
<td>&lt;</td>
<td>&lt;</td>
</tr>
</tbody>
</table>
Exercise

• id+(id-id*id)

• $ <\cdot id > + <\cdot( <\cdot id > - <\cdot id > * <\cdot id > ) > $
• $ <\cdot+ <\cdot( <\cdot id > - <\cdot id > * <\cdot id > ) > $
• $ <\cdot+ <\cdot( <\cdot id > * <\cdot id > ) > $
• $ <\cdot+ <\cdot( <\cdot id > * <\cdot id > ) > $
• $ <\cdot+ <\cdot( <\cdot id > * <\cdot id > ) > $
• $ <\cdot+ <\cdot( <\cdot id > ) > $
• $ <\cdot+ <\cdot( <\cdot id > ) > $
• $ <\cdot+ <\cdot( <\cdot id > ) > $

Using relation symbols or stack
Using relation symbols or \textbf{stack}
Handling Unary Minus

• Operator-Precedence parsing cannot handle the unary minus when we also have the binary minus in our grammar.

• The best approach to solve this problem, let the lexical analyzer handle this problem.
  – The lexical analyzer will return two different operators for the unary minus and the binary minus.
  – The lexical analyzer will need a lookahead to distinguish the binary minus from the unary minus.

• Then, we make
  
  \[
  O \prec \text{unary-minus} \quad \text{for any operator}
  \]
  \[
  \text{unary-minus} \succ O \quad \text{if unary-minus has higher precedence than } O
  \]
  \[
  \text{unary-minus} \prec O \quad \text{if unary-minus has lower (or equal) precedence than } O
  \]
Precedence Functions

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

- \[ f(a) < g(b) \quad \text{whenever} \quad a \ll b \]
- \[ f(a) = g(b) \quad \text{whenever} \quad a \equiv b \]
- \[ f(a) > g(b) \quad \text{whenever} \quad a \gg b \]
Disadvantages of Operator Precedence Parsing

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

• Advantages:
  – simple
  – powerful enough for expressions in programming languages
Error Recovery in Operator-Precedence Parsing

Error Cases:

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

Error Recovery:

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

1. Top-down parsers
   – Shirt-Reduce
   – Handle
   – Operator-Precedence Parsing

We only defined handles, but

How to discover hands?

• There are no known efficient algorithms to recognize handles
• Solution: use heuristics to guess which stacks are handles
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:

\[
\text{LR}(k) \text{ parsing.}
\]

left to right scanning \quad \text{right-most derivation} \quad k \text{ lookahead}

\(k\) is omitted \(\Rightarrow\) 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.
    \[
    \text{LL(1)-Grammars} \subset \text{LR(1)-Grammars}
    \]
  – 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.
LR Parsing Algorithm Framework

input

\[ a_1 \ldots a_i \ldots a_n \] $ \]

Parsing Algorithm

stack

\[ S_m \]
\[ X_m \]
\[ S_{m-1} \]
\[ X_{m-1} \]
\[ \ldots \]
\[ S_1 \]
\[ X_1 \]
\[ S_0 \]

Action Table

- terminals and $
- four different actions

Goto Table

- non-terminal
- each item is a state number
A Trivial Bottom-Up Parsing Algorithm

Let $\omega = \text{input string}$

repeat

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

   if (no such $\beta$) We only want to reduce at handles
      backtrack
   else
      replace one $\beta$ by $A$ in $\omega$

until $\omega = "$S"$ (the start symbol) or all possibilities are exhausted

We have defined handles
Stack is: ...T, input *...

LR Parsers avoid backtrack But, how to detect handles?
Viable Prefix

• At each step the parser sees:

  The stack content must be a prefix of a right-sentential form

• $E \Rightarrow \ldots \Rightarrow F^{*}id \Rightarrow (E)^{*} id$

  — $(, (E, (E) \text{ are prefix of } (E)^{*} id$

• But, not all prefix can be the stack content

  — $(E)^{*}$ cannot be appear on the stack, it should be reduced by $F \rightarrow (E)$

• 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: $a_1a_2...a_n$  
  input: $r_1r_2\omega$
  
  either $a_i...a_n$ is a handle, or there is no handle

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

• It is always possible to shift symbol from input onto the stack and obtain a new viable prefix

• A production $A \rightarrow \beta_1\beta_2$ 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 $\beta_2 \neq \varepsilon$

• Item: $A \rightarrow \beta_1\cdot\beta_2$
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 $\omega$

- LR(0) Items: One production produces a set of items by placing . in to productions

  E.g. The production $T \rightarrow (E)$ gives items
  
  - $T \rightarrow .(E)$ // we have seen $\epsilon$ of $T \rightarrow (E)$ , shift
  - $T \rightarrow .(E)$ // we have seen ( of $T \rightarrow (E)$ , shift
  - $T \rightarrow (E.)$ // we have seen (E of $T \rightarrow (E)$, shift
  - $T \rightarrow (E).$ // we have seen (E) of $T \rightarrow (E)$, reduce

  The production $T \rightarrow \epsilon$ gives the item
  
  - $T \rightarrow .$


LR(0) Items and LR(0) Automaton

• The stack may have many prefixes of productions
  \[\text{Prefix}_1\text{Prefix}_2\ldots\text{Prefix}_{n-1}\text{Prefix}_n\]

Let \(\text{Prefix}_n\) be a valid prefix of \(A_n \rightarrow \alpha_n\)
  - \(\text{Prefix}_n\) will eventually reduce to \(A_n\)
  - \(\text{Prefix}_{n-1} A_n\) is a valid prefix of \(A_{n-1} \rightarrow \text{Prefix}_{n-1} A_n \beta\)

• Finally, \(\text{Prefix}_k\ldots\text{Prefix}_n\) eventually reduces to \(A_k\)
  - \(\text{Prefix}_{k-1} A_k\) is a valid prefix of \(A_{k-1} \rightarrow \text{Prefix}_{k-1} A_k \beta\)
LR(0) Items and LR(0) Automaton

- Consider: \((\text{id} \ast \text{id})\)  
  Stack: \((\text{id} \ast \text{ input id})\)
  - \((\) is a viable prefix of \(T \rightarrow (E)\)
  - \(\epsilon\) is a viable prefix of \(E \rightarrow T\)
  - \(\text{id} \ast\) is a viable prefix of \(T \rightarrow \text{id} \ast T\)

Example:

\[
\begin{align*}
  &E \rightarrow T + E \mid T \\
  &T \rightarrow \text{id} \ast T \mid \text{id} \mid (E)
\end{align*}
\]

A state (a set of items) \(\{T \rightarrow (E), E \rightarrow .T, T \rightarrow \text{id} \ast .T\}\) says:
  - We have seen (of \(T \rightarrow (E)\))
  - We have seen \(\epsilon\) of \(E \rightarrow T\)
  - We have seen \(\text{id} \ast\) of \(T \rightarrow \text{id} \ast T\)
Closure Operation: States

- If $I$ is a set of LR(0) items for a grammar $G$, then $\text{closure}(I)$ 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 $\text{closure}(I)$.
  2. Repeat
     - If $A \rightarrow \alpha.B\beta$ is in $\text{closure}(I)$ and $B \rightarrow \gamma$ is a production in $G$;
     - then $B \rightarrow \gamma$ will be in the $\text{closure}(I)$.

   Until no more new LR(0) items can be added to $\text{closure}(I)$.

- 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 $A_{k-1} \rightarrow \text{Prefix}_{k-1}A_k\beta$
Closure Operation -- Example

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

\[ \text{I}_0: \text{closure} \{ \text{E'} → \cdot \text{E} \} = \]

{ E' → \cdot \text{E} \leftarrow \]
kernels: the initial item
and all items whose dots are not at the left end

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’ → E  I₀: closure(\{E → E+ \cdot T\}) =
E → E+T  \{ E → E+ \cdot T \} ← kernel items
E → T  T → \cdot T*F
T → T*F  T → \cdot F
T → F  F → \cdot (E)
F → (E)  F → \cdot id  }  2. Nonkernel items: otherwise
F → id

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→\cdot γ 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 $\text{goto}(I, X)$ is defined as follows:
  - If $A \rightarrow \alpha \cdot X \beta$ in $I$
  - then every item in $\text{closure}\left(\{A \rightarrow \alpha X \cdot \beta\}\right)$ will be in $\text{goto}(I, X)$.

Example:
$I = \{ E' \rightarrow \bullet E, \ E \rightarrow \bullet E + T, \ E \rightarrow \bullet T, \ T \rightarrow \bullet T*F,$
\[ T \rightarrow \bullet F, \ F \rightarrow \bullet (E), \ F \rightarrow \bullet \text{id} \}$

$\text{goto}(I, E) = \{ E' \rightarrow E \bullet, \ E \rightarrow E \bullet + T \}$
$\text{goto}(I, T) = \{ E \rightarrow T \bullet, \ T \rightarrow T \bullet *F \}$
$\text{goto}(I, F) = \{ T \rightarrow F \bullet \}$
$\text{goto}(I,()) = \{ F \rightarrow (\bullet E), \ E \rightarrow \bullet E + T, \ E \rightarrow \bullet T, \ T \rightarrow \bullet T*F, \ T \rightarrow \bullet F,$
\[ F \rightarrow \bullet (E), \ F \rightarrow \bullet \text{id} \}$
$\text{goto}(I, \text{id}) = \{ F \rightarrow \text{id} \bullet \}$
LR(0) Automaton

- Add a dummy production $S' \rightarrow S$ into $G$, get $G'$
- **Algorithm:**
  
  $C$ is \{ closure(\{\(S' \rightarrow \bullet 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 $C$.

- goto function is the transition function of the DFA
- Initial state: closure(\{\(S' \rightarrow S\})\)
- Every state is an accepting state

New item, corresponding to new state in DFA
The Canonical LR(0) Collection -- Example

I₀: E’ → E
  E → .E+T
  E → .T
  T → .T*F
  T → .F
  F → .(E)
  F → .id

I₁: E’ → E.
  E → E+.T

I₂: E → T.
  T → T.*F

I₃: T → F.

I₄: F → (.E)
  E → .E+T
  E → .T
  T → .T*F
  T → .F
  F → .(E)
  F → .id

I₅: F → id.

I₆: E → E+.T
  T → .T*F
  T → .F
  F → .(E)
  F → .id

I₇: T → T*.F
  F → .(E)
  F → .id

I₈: F → (E.)
  E → E+.T

I₉: E → E+T.
  T → T.*F

I₁₀: T → T*F.

I₁¹: F → (E).
LR(0) Parsing Algorithm Implementation

**Input**: \[ a_1 \ldots a_i \ldots a_n \$ \]

**Stack**: 
- \( S_m \)
- \( X_m \)
- \( S_{m-1} \)
- \( X_{m-1} \)
- \( \ldots \)
- \( S_1 \)
- \( X_1 \)
- \( S_0 \)

**LR(0) Parsing Algorithm**

Idea: Assume: stack contains \( \beta \) and next input is \( a \)

- **Reduce by** \( X \rightarrow \beta \)
  - if \( s \) contains item \( X \rightarrow \beta \).
- **Shift if** \( s \) contains item \( X \rightarrow \beta \cdot a \omega \), 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:

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

Stack \hspace{2cm} 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:

\[
X_1 \ldots X_m a_i a_{i+1} \ldots a_n $
\]
Constructing LR(0) Parsing Table
(of an augmented 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, \( A \rightarrow \alpha \cdot \alpha\beta \) in \( I_i \) and \( \text{goto}(I_i, a) = I_j \) then \( \text{action}[i, a] \) is \( \text{shift } j \).
   - If \( A \rightarrow \alpha \cdot \) is in \( I_i \), then \( \text{action}[i, a] \) is \( \text{reduce } A \rightarrow \alpha \).
   - If \( S’ \rightarrow S \cdot \) is in \( I_i \), then \( \text{action}[i, \$] \) is \( \text{accept} \).
   - 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 \( \text{goto}(I_i, A) = I_j \) then \( \text{goto}[i, A] = j \)

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

5. Initial state of the parser contains \( S’ \rightarrow .S \)
Actions of A LR(0)-Parser

1. **Shift s** -- shifts the next input symbol $a_i$ and the state $s$ onto the stack ($s_n$ where $n$ is a state number)

   \[
   (S_0 \, X_1 \, S_1 \ldots \, X_m \, S_m, \, a_i \, a_{i+1} \ldots \, a_n \, \$) \rightarrow (S_0 \, X_1 \, S_1 \ldots \, X_m \, S_m \, a_i \, s, \, a_{i+1} \ldots \, a_n \, \$)
   \]

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

   \[
   (S_0 \, X_1 \, S_1 \ldots \, X_m \, S_m, \, a_i \, a_{i+1} \ldots \, a_n \, \$) \rightarrow (S_0 \, X_1 \, S_1 \ldots \, X_{m-r} \, S_{m-r} \, A \, s, \, a_i \ldots \, a_n \, \$)
   \]
   - 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)
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 \( Y \rightarrow \omega \cdot \delta \)

\( G \) is in LR(0) Grammar if no conflict
LR(0) Conflicts

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

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

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

$I_3$: $T \rightarrow F$

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

$I_5$: $F \rightarrow id$

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

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

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

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

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

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

shift/reduce conflict
## Parsing Tables of Expression Grammar

### LR(0) Action Table

<table>
<thead>
<tr>
<th>State</th>
<th>id</th>
<th>+</th>
<th>*</th>
<th>(</th>
<th>)</th>
<th>$</th>
<th>E</th>
<th>T</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>s5</td>
<td></td>
<td>s4</td>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>1</td>
<td>s6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>acc</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>r2</td>
<td>r2</td>
<td>s7</td>
<td>r2</td>
<td>r2</td>
<td>r2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>r4</td>
<td>r4</td>
<td>r4</td>
<td>r4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>s5</td>
<td></td>
<td>s4</td>
<td></td>
<td></td>
<td></td>
<td>8</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td>r6</td>
<td>r6</td>
<td>r6</td>
<td>r6</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>s5</td>
<td></td>
<td>s4</td>
<td></td>
<td></td>
<td></td>
<td>9</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>s5</td>
<td></td>
<td>s4</td>
<td></td>
<td></td>
<td></td>
<td>10</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td>s6</td>
<td></td>
<td></td>
<td></td>
<td>s11</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>r1</td>
<td>r1</td>
<td>s7</td>
<td>r1</td>
<td>r1</td>
<td>r1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
<td>r3</td>
<td>r3</td>
<td>r3</td>
<td>r3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td></td>
<td>r5</td>
<td>r5</td>
<td>r5</td>
<td>r5</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Goto Table

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

Idea: Assume: stack contains $\beta$ and next input is $a$

- DFA on input $\beta$ terminates in state $s$
- Reduce by $X \rightarrow \beta$
  - if $a$ in FOLLOW($X$) (details refer to next slide)
- Shift if $s$ contains item $X \rightarrow \beta.a\omega$, i.e. DFA has $(s, a, s')$
Constructing SLR(1) Parsing Table
(of an augmented 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, \( A \rightarrow \alpha \cdot a \beta \) in \( I_i \) and goto\((I_i, a) = I_j \) then action\([i, a] \) is \textit{shift} \( j \).
   - If \( A \rightarrow \alpha \cdot \) is in \( I_i \), then action\([i, a] \) is \textit{reduce} \( A \rightarrow \alpha \) for all \( a \) in FOLLOW\((A) \) where \( A \neq S' \).
   - If \( S' \rightarrow S. \) is in \( I_i \), then action\([i, \$] \) is \textit{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 Implementation

SLR(1) Parsing Algorithm

Input: \( a_1, \ldots, a_i, \ldots, a_n, \$ \)

Stack:
- \( S_m \)
- \( X_m \)
- \( S_{m-1} \)
- \( X_{m-1} \)
- \( \ldots \)
- \( S_1 \)
- \( X_1 \)
- \( S_0 \)

Output:

SLR(1) Action Table
- terminals and \( \$ \)
- \( s \) four different actions
- \( t \)
- \( a \)
- \( e \)
- \( S \)

Goto Table
- non-terminal
- \( s \) each item is
- \( t \)
- \( a \)
- \( e \)
- \( S \)
### 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($E$) = ? 
{+, ), $}  

<table>
<thead>
<tr>
<th>state</th>
<th>id</th>
<th>+</th>
<th>*</th>
<th>(</th>
<th>)</th>
<th>$</th>
<th>E</th>
<th>T</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>s5</td>
<td></td>
<td>s4</td>
<td></td>
<td></td>
<td>$</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>1</td>
<td>s6</td>
<td></td>
<td></td>
<td></td>
<td>acc</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>r2</td>
<td>r2</td>
<td>s7</td>
<td>r2</td>
<td>r2</td>
<td>r2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>r4</td>
<td>r4</td>
<td>r4</td>
<td>r4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>s5</td>
<td>s4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>8</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>5</td>
<td>r6</td>
<td>r6</td>
<td>r6</td>
<td>r6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>s5</td>
<td>s4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>9</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>s5</td>
<td>s4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>10</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>s6</td>
<td></td>
<td></td>
<td>s11</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>r1</td>
<td>r1</td>
<td>s7</td>
<td>r1</td>
<td>r1</td>
<td>r1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>r3</td>
<td>r3</td>
<td>r3</td>
<td>r3</td>
<td>r3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>r5</td>
<td>r5</td>
<td>r5</td>
<td>r5</td>
<td>r5</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
LR(0) Conflicts

I₀: E' → E
   E → E+.E+T
   E → E
   T → T*.F
   T → T
   F → (.E)
   F → .id

I₁: E' → E.
   E → E+.E
   E → E+.T

I₂: E → T.
   T → T*.F

I₃: T → F.

I₄: F → (.E)
   E → E+.E+T
   E → E
   T → T*.F
   T → T
   F → (.E)
   F → .id

I₅: F → id.

I₆: E → E+.T
   T → .T*.F
   T → T
   F → (.E)
   F → .id

I₇: T → T*.F.
   F → (.E)
   F → .id

I₈: F → (E.)
   E → E+.E+T

I₉: E → E+.T
   T → T*.F

I₁₀: T → T*.F.

I₁₁: F → (E).

shift/reduce conflict solved

accept
<table>
<thead>
<tr>
<th>stack</th>
<th>input</th>
<th>action</th>
<th>output</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>id*id+id$</td>
<td>shift 5</td>
<td></td>
</tr>
<tr>
<td>0id5</td>
<td>*id+id$</td>
<td>reduce by F→id</td>
<td>F→id</td>
</tr>
<tr>
<td>0F3</td>
<td>*id+id$</td>
<td>reduce by T→F</td>
<td>T→F</td>
</tr>
<tr>
<td>0T2</td>
<td>*id+id$</td>
<td>shift 7</td>
<td></td>
</tr>
<tr>
<td>0T2*7</td>
<td>id+id$</td>
<td>shift 5</td>
<td></td>
</tr>
<tr>
<td>0T2*7id5</td>
<td>+id$</td>
<td>reduce by F→id</td>
<td>F→id</td>
</tr>
<tr>
<td>0T2*7F10</td>
<td>+id$</td>
<td>reduce by T→T*F</td>
<td>T→T*F</td>
</tr>
<tr>
<td>0T2</td>
<td>+id$</td>
<td>reduce by E→T</td>
<td>E→T</td>
</tr>
<tr>
<td>0E1</td>
<td>+id$</td>
<td>shift 6</td>
<td></td>
</tr>
<tr>
<td>0E1+6</td>
<td>id$</td>
<td>shift 5</td>
<td></td>
</tr>
<tr>
<td>0E1+6id5</td>
<td>$</td>
<td>reduce by F→id</td>
<td>F→id</td>
</tr>
<tr>
<td>0E1+6F3</td>
<td>$</td>
<td>reduce by T→F</td>
<td>T→F</td>
</tr>
<tr>
<td>0E1+6T9</td>
<td>$</td>
<td>reduce by E→E+T</td>
<td>E→E+T</td>
</tr>
<tr>
<td>0E1</td>
<td>$</td>
<td>accept</td>
<td></td>
</tr>
</tbody>
</table>
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.
Exercise

S → L=R
S → R
L → *R
L → id
R → L

Construction LR(0) automaton
Conflict Example

I₀: \( S' \rightarrow S \)
- \( S \rightarrow L=R \)
- \( S \rightarrow R \)
- \( S \rightarrow .L=R \)
- \( S \rightarrow .R \)
- \( L \rightarrow .*R \)
- \( L \rightarrow .id \)
- \( R \rightarrow .L \)

I₁: \( S' \rightarrow S \)

I₂: \( S \rightarrow L=R \)
- \( L \rightarrow .L \)
- \( R \rightarrow .L \)

I₃: \( S \rightarrow R \)
- \( R \rightarrow .L \)

I₄: \( L \rightarrow .R \)
- \( R \rightarrow .L \)
- \( L \rightarrow .*R \)
- \( L \rightarrow .id \)

I₅: \( L \rightarrow .id \)

I₆: \( S \rightarrow L=R \)
- \( R \rightarrow .L \)
- \( L \rightarrow .*R \)
- \( L \rightarrow .id \)

I₇: \( L \rightarrow .id \)

I₈: \( R \rightarrow .L \)

I₉: \( S \rightarrow L=R \)
Conflict Example

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

\[
\begin{align*}
S & \rightarrow L=R \\
S & \rightarrow R \\
L & \rightarrow \ast R \\
L & \rightarrow \text{id} \\
R & \rightarrow L
\end{align*}
\]

\[
\begin{align*}
I_0: & \quad S' \rightarrow .S \\
\quad & \quad \begin{align*}
S & \rightarrow .L=R \\
S & \rightarrow .R \\
L & \rightarrow \ast R \\
L & \rightarrow \text{id} \\
R & \rightarrow .L
\end{align*} \\
I_1: & \quad S' \rightarrow S. \\
I_2: & \quad S \rightarrow L.=R \\
\quad & \quad \begin{align*}
R & \rightarrow L. \\
L & \rightarrow \ast R \\
L & \rightarrow \text{id} \\
R & \rightarrow L.
\end{align*}
I_3: & \quad S \rightarrow R. \\
I_4: & \quad L \rightarrow \ast .R \\
\quad & \quad \begin{align*}
R & \rightarrow .L \\
L & \rightarrow \ast R \\
L & \rightarrow \text{id}
\end{align*}
I_5: & \quad L \rightarrow \text{id.} \\
I_6: & \quad S \rightarrow L.=R \\
\quad & \quad \begin{align*}
R & \rightarrow .L \\
L & \rightarrow \ast R \\
L & \rightarrow \text{id}
\end{align*}
I_7: & \quad L \rightarrow \ast R. \\
I_8: & \quad R \rightarrow L.
\end{align*}
\]

Problem ?

FOLLOW(R)=\{=,\$\}

1. shift 6
2. reduce by \( R \rightarrow L \)

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.
Review

• LR(0) Items and LR(0) automaton
  – Closure and goto function
• LR(0) Parser
  – Action table and goto table
• Reduce/reduce conflict
• Reduce/shift conflict
• SLR(1) Parser
  – Action table and goto table
Exercise-Review LL(1)

Production rules
1. $S \rightarrow L = R$
2. $S \rightarrow R$
3. $L \rightarrow * R$
4. $L \rightarrow id$
5. $R \rightarrow L$

Construct
LL(1) Table

<table>
<thead>
<tr>
<th></th>
<th>FIRST</th>
<th>FOLLOW</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S$</td>
<td>*, id</td>
<td>$$$</td>
</tr>
<tr>
<td>$L$</td>
<td>*, id</td>
<td>$$, =</td>
</tr>
<tr>
<td>$R$</td>
<td>*, id</td>
<td>$$, =</td>
</tr>
</tbody>
</table>

LL(1) Table

<table>
<thead>
<tr>
<th></th>
<th>*</th>
<th>id</th>
<th>$$$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S$</td>
<td>1, 2</td>
<td>1, 2</td>
<td></td>
</tr>
<tr>
<td>$L$</td>
<td>3</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>$R$</td>
<td>5</td>
<td>5</td>
<td></td>
</tr>
</tbody>
</table>

Is this grammar in LL(1)? No, Conflict
**Conflict Example**

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

**Problem**

FOLLOW($R$) = {$=$,$\$}$

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

shift/reduce conflict
Exercise - review LL(1)

Productions
1. \( S \rightarrow AaAb \)
2. \( S \rightarrow BbBa \)
3. \( A \rightarrow \varepsilon \)
4. \( B \rightarrow \varepsilon \)

Construct

LL(1) Table

<table>
<thead>
<tr>
<th></th>
<th>FIRST</th>
<th>FOLLOW</th>
</tr>
</thead>
<tbody>
<tr>
<td>( S )</td>
<td>a, b</td>
<td>$</td>
</tr>
<tr>
<td>( A )</td>
<td>( \varepsilon )</td>
<td>a, b</td>
</tr>
<tr>
<td>( B )</td>
<td>( \varepsilon )</td>
<td>a, b</td>
</tr>
</tbody>
</table>

\[
\begin{array}{|c|c|c|}
\hline
& a & b & $
\hline
S & 1 & 2 & \\
A & 3 & 3 & \\
B & 4 & 4 & \\
\hline
\end{array}
\]

This grammar in LL(1)?  Yes, no Conflict
Construct SLR(1) action table

**Problem**

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

- a \rightarrow \text{reduce by } A \rightarrow \varepsilon
- b \rightarrow \text{reduce by } A \rightarrow \varepsilon
- a \rightarrow \text{reduce by } B \rightarrow \varepsilon
- b \rightarrow \text{reduce by } B \rightarrow \varepsilon
- reduce/reduce conflict

**SLR(1)**

- S \rightarrow AaAb
- S \rightarrow BbBa
- A \rightarrow \varepsilon
- B \rightarrow \varepsilon

**I_0:**
- S' \rightarrow .S
- S \rightarrow .AaAb
- S \rightarrow .BbBa
- A \rightarrow .
- B \rightarrow .
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 $A \rightarrow \alpha$ is in the $I_i$ and $a$ is FOLLOW($A$)

- In some situations, $\beta A$ 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.

\[
S \rightarrow AaAb \\
s \Rightarrow AaAb \Rightarrow Aab \Rightarrow ab \\
S \Rightarrow BbBa \Rightarrow Bba \Rightarrow ba
\]

\[
S \rightarrow BbBa \\
A \rightarrow \epsilon \\
AaAb \Rightarrow Aa \epsilon b \\
B \rightarrow \epsilon \\
Aab \Rightarrow \epsilon ab
\]

\[
BbBa \Rightarrow Bb \epsilon a \\
Bba \Rightarrow \epsilon ba
\]

\{ $I_0$: $S \rightarrow AaAb$, $S \rightarrow BbBa$, $A \rightarrow \epsilon$, $B \rightarrow \epsilon$ \}: 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:

\[ A \rightarrow \alpha \cdot \beta \]

• A LR(1) item is:

\[ A \rightarrow \alpha \cdot \beta, a \]

where \( a \) is the look-ahead of the LR(1) item

(\( a \) is a terminal or end-marker.)
LR(1) Item (cont.)

- When \( \beta \) (in the LR(1) item \( A \rightarrow \alpha \cdot \beta, a \)) is not empty, the look-ahead does not have any affect.

- When \( \beta \) is empty (\( A \rightarrow \alpha \cdot, a \)), we do the reduction by \( A \rightarrow \alpha \) only if the next input symbol is \( a \) (not for any terminal in FOLLOW(\( A \))).

\[
\begin{align*}
A & \rightarrow \alpha \cdot, a_1 \\
    & \ldots \\
A & \rightarrow \alpha \cdot, a_n
\end{align*}
\]

- A state will contain where \( \{a_1, \ldots, a_n\} \subseteq \text{FOLLOW}(A) \).
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.

\textbf{closure(I)} is: (where I is a set of LR(1) items)

\begin{itemize}
  \item every LR(1) item in I is in closure(I)
  \item if \( A \rightarrow \alpha \cdot B \beta, a \) in closure(I) and \( B \rightarrow \gamma \) is a production rule of G; then \( B \rightarrow \gamma, b \) will be in the closure(I) for each terminal \( b \) in FIRST(\( \beta a \)).
\end{itemize}

LR(0) automaton

\begin{itemize}
  \item if \( A \rightarrow \alpha \cdot B \beta \) in closure(I) and \( B \rightarrow \gamma \) is a production rule of G; then \( B \rightarrow \gamma, \) will be in the closure(I).\end{itemize}
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 $\text{closure}({A \rightarrow \alpha X.\beta, a})$ will be in goto(I,X).
Construction of LR(1) Automaton

• **Algorithm:**
  
  \[ C \text{ is } \{ \text{closure}(\{S' \rightarrow S, \$\}) \} \]

  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' \rightarrow S,\$\})
• Every state is an accepting state
A Short Notation for The Sets of LR(1) Items

- A set of LR(1) items containing the following items
  
  \[ A \rightarrow \alpha \cdot \beta, a_1 \]
  
  ...

  \[ A \rightarrow \alpha \cdot \beta, a_n \]

  can be written as

  \[ A \rightarrow \alpha \cdot \beta, a_1/a_2/.../a_n \]
LR(1) Automaton example

1) \( S \rightarrow AaAb \)
2) \( S \rightarrow BbBa \)
3) \( A \rightarrow \varepsilon \)
4) \( B \rightarrow \varepsilon \)

if \( S \rightarrow \cdot AaAb, \$ \) in closure(I) and \( A \rightarrow \varepsilon \) is a production rule of G; then \( A \rightarrow \cdot, a \) will be in the closure(I) for each terminal \( a \) in \( \text{FIRST}(AaAb\$) = \{a\} \).
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, \( A \rightarrow \alpha \cdot a \beta \), b in \( I_i \) and \( \text{goto}(I_i,a) = I_j \) then action\([i,a]\) is \text{shift} j.
   - If \( A \rightarrow \alpha \cdot \), a is in \( I_i \), then action\([i,a]\) is \text{reduce} \( A \rightarrow \alpha \) where \( A \neq S' \).
   - If \( S' \rightarrow S \cdot \), $ is in \( I_i \), then action\([i,\$]\) is \text{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 \( S' \rightarrow .S, \$ \)
LR(1) Parsing Tables

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

I₀: $S' \rightarrow S , \$, $S \rightarrow AaAb , \$, $S \rightarrow BbBa , \$,
$A \rightarrow . , a$
$B \rightarrow . , b$

I₄: $S \rightarrow Aa.Ab , \$
A → . , b

I₅: $S \rightarrow Bb.Ba , \$
B → . , a

I₁: $S' \rightarrow S , \$, $S \rightarrow . AaAb , \$
A → . , a

I₂: $S \rightarrow A.aAb , \$

I₃: $S \rightarrow B.bBa , \$

I₆: $S \rightarrow AaA.b , \$

I₇: $S \rightarrow BbB.a , \$

A → . , a

I₈: $S \rightarrow AaAb , \$

I₉: $S \rightarrow BbBa , \$

A → . , a

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>$</th>
<th>S</th>
<th>L</th>
<th>R</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>r3</td>
<td>r4</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>acc</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>s4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>s5</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>r3</td>
<td></td>
<td>6</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>r4</td>
<td></td>
<td></td>
<td>7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td>s8</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>s9</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>r1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>r2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

no shift/reduce or
no reduce/reduce conflict

↓

so, it is a LR(1) grammar
If \( A \rightarrow \alpha \) is in \( I_i \), then action\([i,a]\) is \textit{reduce} \( A \rightarrow \alpha \) for all \( a \) in \( \text{FOLLOW}(A) \) where \( A \neq S' \).

**Problem**

\( \text{FOLLOW}(R) = \{=,\$\} \)

1. shift 6
2. reduce by \( R \rightarrow L \)

shift/reduce conflict
Draw LR(1) automaton and LR(1) parsing table
LR(1) Parsing Tables – (for Example2)

<table>
<thead>
<tr>
<th></th>
<th>id</th>
<th>*</th>
<th>=</th>
<th>$</th>
<th>S</th>
<th>L</th>
<th>R</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>s5</td>
<td>s4</td>
<td></td>
<td></td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>acc</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>s6</td>
<td>r5</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td>r2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>s5</td>
<td>s4</td>
<td></td>
<td></td>
<td>8</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td>r4</td>
<td>r4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>s12</td>
<td>s11</td>
<td></td>
<td></td>
<td>10</td>
<td>9</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td>r3</td>
<td>r3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td>r5</td>
<td>r5</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
<td>r1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
<td>r5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>s12</td>
<td>s11</td>
<td></td>
<td></td>
<td>10</td>
<td>13</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td></td>
<td></td>
<td>r4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>13</td>
<td></td>
<td></td>
<td>r3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

no shift/reduce or no reduce/reduce conflict

↓

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: \[ S \rightarrow L \bullet=R, \] \[ \Rightarrow \quad S \rightarrow L \bullet=R \]
\[ R \rightarrow L \bullet, \] \[ \Rightarrow \quad R \rightarrow L \bullet. \]

• 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.

I_1: \[ L \rightarrow \text{id} \bullet, = \]
A new state: \[ I_{12}: L \rightarrow \text{id} \bullet, = \]
\[ \Rightarrow \quad L \rightarrow \text{id} \bullet, \$
I_2: \[ L \rightarrow \text{id} \bullet, \$ \]

have same core, merge them

or \[ I_{12}: L \rightarrow \text{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.
  \[ C = \{I_0, ..., I_n\} \implies C' = \{J_1, ..., J_m\} \text{ where } m \leq n \]

• 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 \( \implies \) cores of goto\((I_1, X), ..., \text{goto}(I_2, X)\) 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 \(\rightarrow\) 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.
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:
  \[ A \rightarrow \alpha \cdot, a \quad \text{and} \quad B \rightarrow \beta \cdot a\gamma, b \]
- This means that a state of the canonical LR(1) parser must have:
  \[ A \rightarrow \alpha \cdot, a \quad \text{and} \quad B \rightarrow \beta \cdot a\gamma, c \]
- 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)
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.

\[ I_1: A \rightarrow \alpha \cdot, a \]
\[ B \rightarrow \beta \cdot, b \]

\[ I_2: A \rightarrow \alpha \cdot, b \]
\[ B \rightarrow \beta \cdot, c \]

\[ I_{12}: A \rightarrow \alpha \cdot, a/b \]
\[ B \rightarrow \beta \cdot, b/c \]

\[ \rightarrow \text{reduce/reduce conflict} \]
Canonical LR(1) Collection – Example2

I₀: $S' \rightarrow S, S$  
S → .L=R,$  
S → .R,$  
L → .*R,$/=  
L → .id,$/=  
R → .L,$

I₁: $S' \rightarrow S,$  
S → .L=$R,$  
S → .R,$  
L → .*R,$/=  
L → .id,$/=  
R → .L,$

I₂: $S \rightarrow .L=R,$  
R → .L,$  
L → .*R,$/=  
L → .id,$/=  
R → .L,$

I₃: $S \rightarrow .R,$  
S → .L=$R,$  
L → .*R,$/=  
L → .id,$/=  
R → .L,$

I₄: $L \rightarrow .*R,$/=  
R → .L,$/=  
L → .*R,$/=  
L → .id,$/=  
R → .L,$

I₅: $L \rightarrow .id,$/=  
S → .L=$R,$  
S → .R,$  
L → .*R,$/=  
L → .id,$/=  
R → .L,$

I₆: $S \rightarrow L=R,$  
R → .L,$  
L → .*R,$/=  
L → .id,$/=  
R → .L,$

I₇: $L \rightarrow .*R,$/=  
R → .L,$  
L → .*R,$/=  
L → .id,$/=  
R → .L,$

I₈: $R \rightarrow .L,$/=  
S → .L=$R,$  
S → .R,$  
L → .*R,$/=  
L → .id,$/=  
R → .L,$

I₉: $S \rightarrow L=R,$  
R → .L,$  
L → .*R,$/=  
L → .id,$/=  
R → .L,$

I₁₀: $R \rightarrow .L,$  
S → .L=$R,$  
S → .R,$  
L → .*R,$/=  
L → .id,$/=  
R → .L,$

I₁¹: $L \rightarrow .*R,$  
R → .L,$  
L → .*R,$/=  
L → .id,$/=  
R → .L,$

I₁²: $L \rightarrow .id,$  
S → .L=$R,$  
S → .R,$  
L → .*R,$/=  
L → .id,$/=  
R → .L,$

I₁³: $L \rightarrow .id,$  
S → .L=$R,$  
S → .R,$  
L → .*R,$/=  
L → .id,$/=  
R → .L,$

I₁₄: $S \rightarrow .L,$  
R → .L,$  
L → .*R,$/=  
L → .id,$/=  
R → .L,$

I₁₅: $L \rightarrow .id,$  
S → .L=$R,$  
S → .R,$  
L → .*R,$/=  
L → .id,$/=  
R → .L,$

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

Same Cores  
I₄ and I₁¹  
I₅ and I₁²  
I₇ and I₁³  
I₈ and I₁₀
Canonical LALR(1) Collection – Example2

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

I₀: S’ → S,$
I₁: S’ → S . ,$
I₂: S → L . =R ,$
I₃: S → R . ,$
I₄₁₁: L → * . R ,$/=
I₅₁₂: L → id . ,$/=
I₆: S → L . = R ,$
I₇₁₃: L → * . R ,$/=
I₈₁₀: R → L . ,$/=

Same Cores
I₄ and I₁₁
I₅ and I₁₂
I₇ and I₁₃
I₈ and I₁₀
LALR(1) Parsing Tables – (for Example2)

<table>
<thead>
<tr>
<th>id</th>
<th>*</th>
<th>=</th>
<th>$</th>
<th>S</th>
<th>L</th>
<th>R</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>s512</td>
<td>s411</td>
<td></td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td>acc</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td>s6</td>
<td>r5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td>r2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>s512</td>
<td>s411</td>
<td></td>
<td>810</td>
<td>713</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td>r4</td>
<td>r4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>s512</td>
<td>s411</td>
<td></td>
<td>810</td>
<td>9</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td>r3</td>
<td>r3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td>r5</td>
<td>r5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
<td></td>
<td></td>
<td>r1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

no shift/reduce or
no reduce/reduce conflict
so, it is a LALR(1) grammar

<table>
<thead>
<tr>
<th></th>
<th>id</th>
<th>*</th>
<th>=</th>
<th>$</th>
<th>S</th>
<th>L</th>
<th>R</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>s5</td>
<td>s4</td>
<td></td>
<td>$</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td>acc</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td>s6</td>
<td>r5</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td>r2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>s5</td>
<td>s4</td>
<td></td>
<td></td>
<td>8</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td>r4</td>
<td>r4</td>
<td>10</td>
<td>9</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>s12</td>
<td>s11</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td>r3</td>
<td>r3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td>r5</td>
<td>r5</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
<td></td>
<td></td>
<td>r1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td>r5</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>s12</td>
<td>s11</td>
<td></td>
<td></td>
<td>10</td>
<td>13</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td></td>
<td></td>
<td></td>
<td>r4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>13</td>
<td></td>
<td></td>
<td></td>
<td>r3</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Panic Mode Error Recovery in LR Parsing

• 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 \( s \)).

• 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.
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
Shift-Reduce Parsers

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.

1. Efficient Construction of LALR parsing table
2. Compaction of LR parsing table
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.

\[
E \rightarrow E + E \mid E \ast E \mid (E) \mid \text{id} \quad \Rightarrow \quad E \rightarrow E + T \mid T
\]

\[
T \rightarrow T \ast F \mid F
\]

\[
F \rightarrow (E) \mid \text{id}
\]
Sets of LR(0) Items for Ambiguous Grammar

I₀: $E' \rightarrow \cdot E$
    $E \rightarrow \cdot E + E$
    $E \rightarrow \cdot E * E$
    $E \rightarrow \cdot (E)$
    $E \rightarrow \cdot id$

I₁: $E' \rightarrow E \cdot$
    $E \rightarrow E \cdot + E$
    $E \rightarrow E \cdot * E$

I₂: $E \rightarrow (\cdot E)$
    $E \rightarrow \cdot E + E$
    $E \rightarrow \cdot E * E$
    $E \rightarrow \cdot (E)$
    $E \rightarrow \cdot id$

I₃: $E \rightarrow id \cdot$

I₄: $E \rightarrow E + \cdot E$
    $E \rightarrow \cdot E + E$
    $E \rightarrow \cdot E * E$
    $E \rightarrow \cdot (E)$
    $E \rightarrow \cdot id$

I₅: $E \rightarrow E * \cdot E$
    $E \rightarrow \cdot E + E$
    $E \rightarrow \cdot E * E$
    $E \rightarrow \cdot (E)$
    $E \rightarrow \cdot id$

I₆: $E \rightarrow (E \cdot)$
    $E \rightarrow E \cdot + E$
    $E \rightarrow E \cdot * E$

I₇: $E \rightarrow E + E \cdot$
    $E \rightarrow E \cdot + E$
    $E \rightarrow E \cdot * E$

I₈: $E \rightarrow E * E \cdot$
    $E \rightarrow E \cdot + E$
    $E \rightarrow E \cdot * E$

I₉: $E \rightarrow (E) \cdot$

shift/reduce conflicts for symbols + and *
**SLR-Parsing Tables for Ambiguous Grammar**

\[
\text{FOLLOW}(E) = \{ \$, +, *, ) \}
\]

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

\[
\begin{align*}
I_0 & \xrightarrow{E} I_1 \xrightarrow{+} I_4 \xrightarrow{E} I_7 \\
\text{when current token is +} \quad & \\
\text{shift} & \quad \Rightarrow + \text{ is right-associative} \\
\text{reduce} & \quad \Rightarrow + \text{ is left-associative} \\
\text{when current token is *} \quad & \\
\text{shift} & \quad \Rightarrow * \text{ has higher precedence than +} \\
\text{reduce} & \quad \Rightarrow + \text{ has higher precedence than *}
\end{align*}
\]
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 \rightarrow * is right-associative
    reduce \rightarrow * is left-associative

when current token is +
    shift \rightarrow + has higher precedence than *
    reduce \rightarrow * has higher precedence than +
### SLR-Parsing Tables for Ambiguous Grammar

<table>
<thead>
<tr>
<th>id</th>
<th>Action</th>
<th>Goto</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>s3</td>
<td>s2</td>
</tr>
<tr>
<td>1</td>
<td>s4</td>
<td>s5</td>
</tr>
<tr>
<td>2</td>
<td>s3</td>
<td>s2</td>
</tr>
<tr>
<td>3</td>
<td>r4</td>
<td>r4</td>
</tr>
<tr>
<td>4</td>
<td>s3</td>
<td>s2</td>
</tr>
<tr>
<td>5</td>
<td>s3</td>
<td>s2</td>
</tr>
<tr>
<td>6</td>
<td>s4</td>
<td>s5</td>
</tr>
<tr>
<td>7</td>
<td>r1</td>
<td>s5</td>
</tr>
<tr>
<td>8</td>
<td>r2</td>
<td>r2</td>
</tr>
<tr>
<td>9</td>
<td>r3</td>
<td>r3</td>
</tr>
</tbody>
</table>
Summary

• Bottom-up parsing — — shift-reduce parsing
• Operator precedence
• LR parsing — — SLR, LR, LALR
Reading materials
