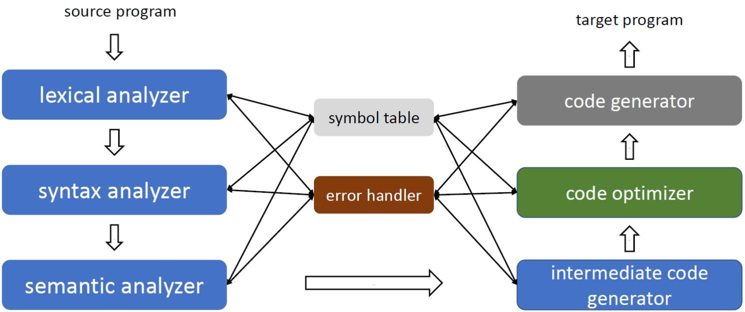
Parsing

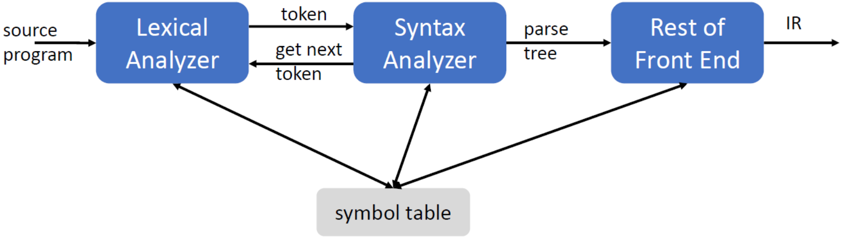
Tapas Kumar Mishra
Sumanta Pyne

Department Of Computer Science and Engineering,
National Institute of Technology, Rourkela

Overview of the compilation process



Parser link


source
program
Lexical
Analyzer
token
get next
token
Syntax
Analyzer
parse
tree
Rest of
Front End
IR
symbol table

Role of a parser

Role of a parser

- Parser obtains a string of tokens from Lexical analyzer.

Role of a parser

- Parser obtains a string of tokens from Lexical analyzer.
- It verifies whether the string of tokens can be generated by the *grammar* of the source language .

Role of a parser

- Parser obtains a string of tokens from Lexical analyzer.
- It verifies whether the string of tokens can be generated by the *grammar* of the source language .
- Outputs a parse tree that will be used by the rest of the phases.

Role of a parser

- Parser obtains a string of tokens from Lexical analyzer.
- It verifies whether the string of tokens can be generated by the *grammar* of the source language .
- Outputs a parse tree that will be used by the rest of the phases.
- Reports syntax errors (Tries to recover in a intelligent fashion).

Types of Parsing

Types of Parsing

- Topdown Parsing -

Types of Parsing

- Topdown Parsing -*Parse tree* is created starting at the root moving towards the leafs .

Types of Parsing

- Topdown Parsing -*Parse tree* is created starting at the root moving towards the leafs .
- Bottomup Parsing -

Types of Parsing

- Topdown Parsing -*Parse tree* is created starting at the root moving towards the leafs .
- Bottomup Parsing -*Parse tree* is created starting at the leafs moving upwards towards the root .

Types of Parsing

- Topdown Parsing -*Parse tree* is created starting at the root moving towards the leafs .
- Bottomup Parsing -*Parse tree* is created starting at the leafs moving upwards towards the root .
- Universal Parsing -

Types of Parsing

- Topdown Parsing -*Parse tree* is created starting at the root moving towards the leafs .
- Bottomup Parsing -*Parse tree* is created starting at the leafs moving upwards towards the root .
- Universal Parsing -CYK algorithm.

Grammars

$$G = (V, T, R, S)$$

- V: Set of variables.
- T: Set of terminals.
- R: Set of Production rules.
- S: Start variable.

Example 1.1

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

$V = \{E, T, F\}$, $T = \{+, *, (,), id\}$, $S = E$,
$R = \{E \rightarrow E + T, E \rightarrow T, \ldots\}$

Context-Free Grammars

$$G = (V, T, R, S)$$

- V: Set of variables. T: Set of terminals. S: Start variable.
- R: Set of Production rules of the form $A \rightarrow \alpha$, where $A \in V$, $\alpha \in \{V \cup T\}^*$.

Example 1.2 (CFG)

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

Example 1.3 (NOT CFG)

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

Derivations

$$G = (V, T, R, S)$$

Let $A \to \alpha$ be a production. Then, $\beta A \gamma \implies \beta \alpha \gamma$ means LHS derives RHS in one step.

Derivations

$$G = (V, T, R, S)$$

Let $A \to \alpha$ be a production. Then, $\beta A \gamma \implies \beta \alpha \gamma$ means LHS derives RHS in one step.
Given a sequence of derivations $\alpha_1 \implies \alpha_2 \implies \ldots \implies \alpha_n$, α_1 derives α_n in $n-1$ steps.

Derivations

$$G = (V, T, R, S)$$

Let $A \to \alpha$ be a production. Then, $\beta A \gamma \implies \beta \alpha \gamma$ means LHS derives RHS in one step.
Given a sequence of derivations $\alpha_1 \implies \alpha_2 \implies \ldots \implies \alpha_n$, α_1 derives α_n in $n-1$ steps.
Shorthand: $\alpha_1 \overset{*}{\implies} \alpha_n$.

Derivations

$$G = (V, T, R, S)$$

Let $A \to \alpha$ be a production. Then, $\beta A \gamma \implies \beta \alpha \gamma$ means LHS derives RHS in one step.
Given a sequence of derivations $\alpha_1 \implies \alpha_2 \implies \ldots \implies \alpha_n$, α_1 derives α_n in $n - 1$ steps.
Shorthand: $\alpha_1 \overset{*}{\implies} \alpha_n$.
If the start symbol $S \overset{*}{\implies} \alpha$, α is said to be a *sentential form* of G.

Derivations

$$G = (V, T, R, S)$$

Let $A \to \alpha$ be a production. Then, $\beta A \gamma \implies \beta \alpha \gamma$ means LHS derives RHS in one step.

Given a sequence of derivations $\alpha_1 \implies \alpha_2 \implies \ldots \implies \alpha_n$, α_1 derives α_n in $n-1$ steps.

Shorthand: $\alpha_1 \overset{*}{\implies} \alpha_n$.

If the start symbol $S \overset{*}{\implies} \alpha$, α is said to be a *sentential form* of G.

A sentence of G is a sentential form without any variables.

Derivations

Grammar

$E \rightarrow E + T | T$

$T \rightarrow T * F | F$

$F \rightarrow (E) | id$

String to be derived

$id + id$

$$
\begin{aligned}
E &\implies E + T \\
&\implies E + F \\
&\implies T + F \\
&\implies F + F \\
&\implies F + id \implies id + id
\end{aligned}
$$

At each step of the production, we have two choices to make:

- Which variable to choose
- Which production to choose for corresponding variable

Derivations

Leftmost Derivation: At each step, the leftmost variable is chosen for production, denoted as $\alpha \underset{lm}{\Longrightarrow} \beta$.

Grammar

$E \rightarrow E + T | T$

$T \rightarrow T * F | F$

$F \rightarrow (E) | id$

String to be derived

$id + id$

$$\begin{aligned} E &\Longrightarrow E + T \\ &\Longrightarrow T + T \\ &\Longrightarrow F + T \\ &\Longrightarrow id + T \\ &\Longrightarrow id + F \Longrightarrow id + id \end{aligned}$$

Derivations

Leftmost Derivation: At each step, the leftmost variable is chosen for production, denoted as $\alpha \underset{lm}{\Longrightarrow} \beta$.

Grammar

$E \rightarrow E + T | T$

$T \rightarrow T * F | F$

$F \rightarrow (E) | id$

String to be derived

$id + id$

$$\begin{aligned} E &\Longrightarrow E + T \\ &\Longrightarrow T + T \\ &\Longrightarrow F + T \\ &\Longrightarrow id + T \\ &\Longrightarrow id + F \Longrightarrow id + id \end{aligned}$$

$$E \underset{lm}{\overset{*}{\Longrightarrow}} id + id.$$

Parse Tree

A parse tree is a graphical representation of derivation.

Parse Tree

A parse tree is a graphical representation of derivation.

- The start symbol S is the root of the tree.

Parse Tree

A parse tree is a graphical representation of derivation.

- The start symbol S is the root of the tree.
- Each internal node is a variable - they represent application of productions.

Parse Tree

A parse tree is a graphical representation of derivation.

- The start symbol S is the root of the tree.
- Each internal node is a variable - they represent application of productions.
- leaves are labelled with terminals (or may be with variables in case of a partial parse tree)

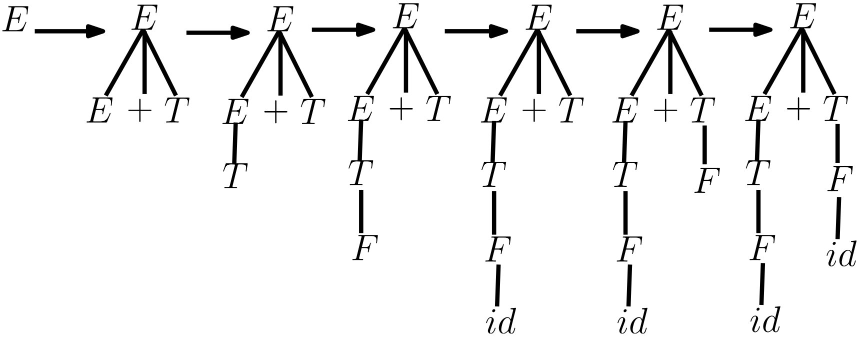
Parse Tree

A parse tree is a graphical representation of derivation.

- The start symbol S is the root of the tree.
- Each internal node is a variable - they represent application of productions.
- leaves are labelled with terminals (or may be with variables in case of a partial parse tree)
- leaves of a parse tree chosen in left to right order produces a sentential form - also known as Yield of the tree.

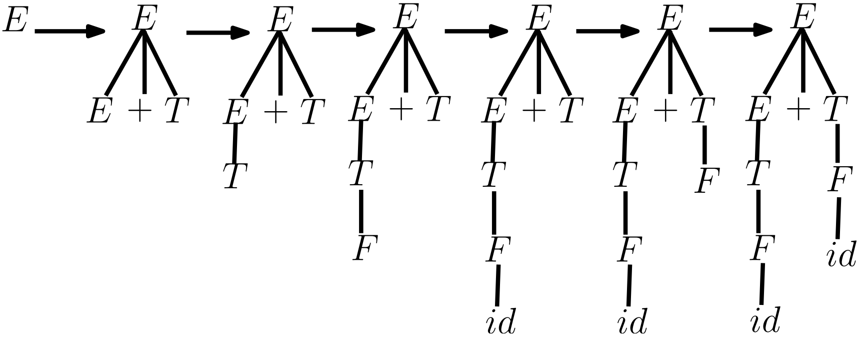
Parse Tree

$E \implies E + T \implies T + T \implies F + T \implies id + T \implies$
$id + F \implies id + id$



Parse Tree

$$E \implies E + T \implies T + T \implies F + T \implies id + T \implies id + F \implies id + id$$



The role of parsing is: Given a grammar $G(V, T, R, S)$ and a string w of terminals, can w be generated from S using the set of rules given in R - if so, generate the corresponding parse tree.

Expressive power: CFG vs NFA

For every NFA, there exists a CFG that generates the language accpeted by the NFA.

- For each state i in the NFA, create a variable A_i in the grammar.
- If state i has a transition to state j on input a, add the rule $A_i \rightarrow aA_j$.
- If state i is accepting state, add the rule $A_i \rightarrow \epsilon$.
- If state i is start state, make A_i the start variable.

However, every language generated by a CFG may not have an equivalent NFA (see with $a^n b^n$).

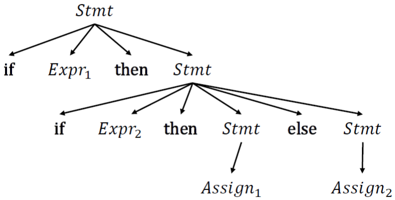
Can the problem of balanced parenthesis be solved with NFAs?

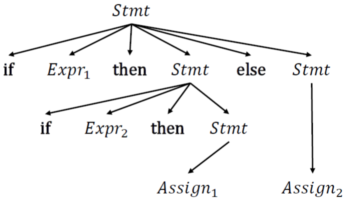
Ambiguous Grammar - Dangling else

A grammar $G(V, T, R, S)$ is said to be ambiguous if certain sentence can be derived from S using rules in R using two distinct left most (or right most) derivations. There are multiple parse trees yielding the same sentence.

$$
\begin{aligned}
Stmt \rightarrow\ & \textbf{if}\ Expr\ \textbf{then}\ Stmt \\
\mid\ & \textbf{if}\ Expr\ \textbf{then}\ Stmt\ \textbf{else}\ Stmt \\
\mid\ & Assign
\end{aligned}
$$

$$\textbf{if}\ Expr_1\ \textbf{then}\ \textbf{if}\ Expr_2\ \textbf{then}\ Assign_1\ \textbf{else}\ Assign_2$$





Ambiguous Grammar - Fixing Dangling else

In all programming languages, an else is matched with the closest unmatched if-then.

Modified grammar:

$$
\begin{aligned}
\mathit{Stmt} &\rightarrow \textbf{if } \mathit{Expr} \textbf{ then } \mathit{Stmt} \\
&\quad\mid \textbf{if } \mathit{Expr} \textbf{ then } \mathit{ThenStmt} \textbf{ else } \mathit{Stmt} \\
&\quad\mid \mathit{Assign} \\
\mathit{ThenStmt} &\rightarrow \textbf{if } \mathit{Expr} \textbf{ then } \mathit{ThenStmt} \textbf{ else } \mathit{ThenStmt} \\
&\quad\mid \mathit{Assign}
\end{aligned}
$$

$$\textbf{if } \mathit{Expr}_1 \textbf{ then if } \mathit{Expr}_2 \textbf{ then } \mathit{Assign}_1 \textbf{ else } \mathit{Assign}_2$$

$$\Downarrow$$

$$
\begin{aligned}
\mathit{Stmt} &\rightarrow \textbf{if } \mathit{Expr} \textbf{ then } \mathit{Stmt} \\
&\rightarrow \textbf{if } \mathit{Expr} \textbf{ then if } \mathit{Expr} \textbf{ then } \mathit{ThenStmt} \textbf{ else } \mathit{Stmt} \\
&\rightarrow \textbf{if } \mathit{Expr} \textbf{ then if } \mathit{Expr} \textbf{ then } \mathit{ThenStmt} \textbf{ else } \mathit{Assign} \\
&\rightarrow \textbf{if } \mathit{Expr} \textbf{ then if } \mathit{Expr} \textbf{ then } \mathit{Assign} \textbf{ else } \mathit{Assign}
\end{aligned}
$$

Ambiguous Grammar

Consider the grammar. $E \rightarrow E + E | E * E | id$. There are two leftmost derivations of the string $id + id * id$.

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

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

To remove ambiguity, encode precedence in the grammar.

$$\begin{gathered} E \rightarrow E + T | T \\ T \rightarrow T * F | F \\ F \rightarrow id \end{gathered}$$

Increasing order of Priority

$$\Downarrow$$

Ambiguous Grammar: Precedence encoding

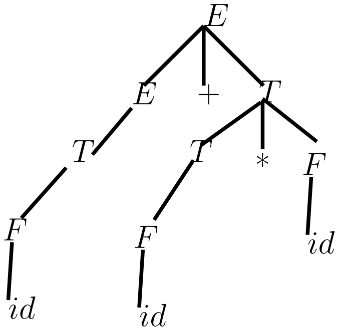
$$E \rightarrow E + T | T$$
$$T \rightarrow T * F | F$$
$$F \rightarrow id$$

Increasing order of Priority

$\Downarrow$

Corresponding Leftmost derivation:

$$\begin{aligned} E &\Longrightarrow E + T \\ &\Longrightarrow T + T \\ &\Longrightarrow F + T \\ &\Longrightarrow id + T \\ &\Longrightarrow id + T * F \\ &\Longrightarrow id + F * F \\ &\Longrightarrow id + id * F \\ &\Longrightarrow id + id * id \end{aligned}$$



Left Recursion

A grammar is left recursive if there is a variable A such that there is a derivation $A \stackrel{+}{\Longrightarrow} A\alpha$ for some string α.

Left Recursion

A grammar is left recursive if there is a variable A such that there is a derivation $A \stackrel{+}{\Longrightarrow} A\alpha$ for some string α.

- Direct left recursion: there is a production of the form $A \Longrightarrow A\alpha$.

Left Recursion

A grammar is left recursive if there is a variable A such that there is a derivation $A \stackrel{+}{\Longrightarrow} A\alpha$ for some string α.

- Direct left recursion: there is a production of the form $A \Longrightarrow A\alpha$.
- Indirect left recursion: there is a production of the form $A \Longrightarrow S\alpha$ and $S \stackrel{*}{\Longrightarrow} A\beta$.

Left Recursion

A grammar is left recursive if there is a variable A such that there is a derivation $A \stackrel{+}{\Longrightarrow} A\alpha$ for some string α.

- Direct left recursion: there is a production of the form $A \Longrightarrow A\alpha$.
- Indirect left recursion: there is a production of the form $A \Longrightarrow S\alpha$ and $S \stackrel{*}{\Longrightarrow} A\beta$.

Left recursion introduces problems during Topdown parsing - it can be avoided by rewriting the rules.

Direct Left Recursion Elimination

$$A \rightarrow A\alpha_1 | A\alpha_2 | \ldots | A\alpha_m | \beta_1 | \ldots | \beta_n$$

$$A \rightarrow \beta_1 A' | \beta_2 A' | \ldots | \beta_n A'$$
$$A' \rightarrow \alpha_1 A' | \alpha_2 A' | \ldots | \alpha_m A' | \epsilon$$

Direct Left Recursion Elimination example

Example 1.4

$$
\begin{aligned}
E &\rightarrow E + T | T \\
T &\rightarrow T * F | F \\
F &\rightarrow id.
\end{aligned}
$$

Direct Left Recursion Elimination example

Example 1.4

$$E \to E + T | T$$
$$T \to T * F | F$$
$$F \to id.$$

After removal of Left recursion,

$E \to TE'$
$E' \to +TE' | \epsilon$
$T \to FT'$
$T' \to *FT' | \epsilon$
$F \to id.$

Indirect Left Recursion

$$S \rightarrow Aa|b$$
$$A \rightarrow Sc|\epsilon$$

There is indirect left recursion since $S \implies Aa \implies Sda$.

Algorithm for Left Recursion Elimination

Input: CFG G with no cycles or ϵ production
Output: Equivalent CFG with no left recursion

1.

Algorithm for Left Recursion Elimination

Input: CFG G with no cycles or ϵ production
Output: Equivalent CFG with no left recursion

1. Arrange the variables in a order $A_1, \ldots, A_n$.
2.

Algorithm for Left Recursion Elimination

Input: CFG G with no cycles or ϵ production
Output: Equivalent CFG with no left recursion

1. Arrange the variables in a order $A_1, \ldots, A_n$.
2. foreach (i in 1 to n):

Algorithm for Left Recursion Elimination

Input: CFG G with no cycles or ϵ production
Output: Equivalent CFG with no left recursion

1. Arrange the variables in a order $A_1, \ldots, A_n$.
2. foreach (i in 1 to n):

    foreach (j in 1 to $i-1$):

        Replace each production of the form $A_i \implies A_j\alpha$ by the production $A_i \implies \delta_1\alpha|\delta_2\alpha|\ldots|\delta_k\alpha$
        where $A_j \implies \delta_1|\delta_2|\ldots|\delta_k$ are productions of A_j.

Algorithm for Left Recursion Elimination

Input: CFG G with no cycles or ϵ production
Output: Equivalent CFG with no left recursion

1. Arrange the variables in a order $A_1, \ldots, A_n$.
2. foreach (i in 1 to n):
   - foreach (j in 1 to $i-1$):
     - Replace each production of the form $A_i \implies A_j\alpha$ by the production $A_i \implies \delta_1\alpha|\delta_2\alpha|\ldots|\delta_k\alpha$ where $A_j \implies \delta_1|\delta_2|\ldots|\delta_k$ are productions of A_j.
   - Eliminate immediate left recursion among all A_i productions.

Algorithm for Left Recursion Elimination

Input: CFG G with no cycles or ϵ production
Output: Equivalent CFG with no left recursion

1. Arrange the variables in a order $A_1, \ldots, A_n$.
2. foreach (i in 1 to n):
   foreach (j in 1 to $i - 1$):
   Replace each production of the form $A_i \implies A_j\alpha$ by the production $A_i \implies \delta_1\alpha|\delta_2\alpha|\ldots|\delta_k\alpha$
   where $A_j \implies \delta_1|\delta_2|\ldots|\delta_k$ are productions of A_j.
   Eliminate immediate left recursion among all A_i productions.

$$S \rightarrow Aa|b$$
$$A \rightarrow Sc|\epsilon$$

Algorithm for Left Recursion Elimination

Input: CFG G with no cycles or ϵ production
Output: Equivalent CFG with no left recursion

1. Arrange the variables in a order $A_1, \ldots, A_n$.
2. foreach (i in 1 to n):
   foreach (j in 1 to $i-1$):
   Replace each production of the form $A_i \implies A_j\alpha$ by the production $A_i \implies \delta_1\alpha|\delta_2\alpha|\ldots|\delta_k\alpha$
   where $A_j \implies \delta_1|\delta_2|\ldots|\delta_k$ are productions of A_j.
   Eliminate immediate left recursion among all A_i productions.

$S \to Aa|b$
$A \to Sc|\epsilon$

When $i = 2$, $A \to Sc$ is replaced with $A \to Aac|bc|\epsilon$. Removing immediate LR, the grammar becomes
$S \to Aa|b$
$A \to bcA'|A'$
$A' \to acA'|\epsilon$

Left factoring

In Topdown parsing, when choice between two A-productions are not clear, it is desirable to defer the decision until enough of the input is seen to make the correct branching decision.

Left factoring

In Topdown parsing, when choice between two A-productions are not clear, it is desirable to defer the decision until enough of the input is seen to make the correct branching decision.

$$A \rightarrow \alpha B | \alpha C$$

$$A \rightarrow \alpha A'$$
$$A' \rightarrow B | C$$

Left factoring

In Topdown parsing, when choice between two A-productions are not clear, it is desirable to defer the decision until enough of the input is seen to make the correct branching decision.

$$A \rightarrow \alpha B | \alpha C$$

$$A \rightarrow \alpha A'$$
$$A' \rightarrow B | C$$

$$S \rightarrow iEtS | iEtSeS | a$$
$$E \rightarrow b\ .$$

Left factoring

In Topdown parsing, when choice between two A-productions are not clear, it is desirable to defer the decision until enough of the input is seen to make the correct branching decision.

$$A \rightarrow \alpha B | \alpha C$$

$$A \rightarrow \alpha A'$$
$$A' \rightarrow B | C$$

$$S \rightarrow iEtS | iEtSeS | a$$
$$E \rightarrow b \;.$$

$$S \rightarrow iEtSS' | a$$
$$S' \rightarrow eS | \epsilon$$
$$E \rightarrow b \;.$$

Top-Down Parsing

Parse tree is created starting at the root creating the nodes of the tree in preorder(depth first).

Top-Down Parsing

Parse tree is created starting at the root creating the nodes of the tree in preorder(depth first).
Equivalent to finding the leftmost derivation of the input string.

Top-Down Parsing

Parse tree is created starting at the root creating the nodes of the tree in preorder(depth first).

Equivalent to finding the leftmost derivation of the input string.

The Key problem is to choose the correct production from the set of rules.

Example 1.5

$$
\begin{gathered}
E \rightarrow E + T | T \\
T \rightarrow T * F | F \\
F \rightarrow id.
\end{gathered}
$$

Top-Down Parsing

Parse tree is created starting at the root creating the nodes of the tree in preorder(depth first).

Equivalent to finding the leftmost derivation of the input string.

The Key problem is to choose the correct production from the set of rules.

Example 1.5

$$E \rightarrow E + T | T$$
$$T \rightarrow T * F | F$$
$$F \rightarrow id.$$

After removal of Left recursion,

$E \rightarrow TE'$

$E' \rightarrow +TE' | \epsilon$

$T \rightarrow FT'$

$T' \rightarrow *FT' | \epsilon$

$F \rightarrow id.$

Top-Down Parsing

Parse tree is created starting at the root creating the nodes of the tree in preorder(depth first).

Equivalent to finding the leftmost derivation of the input string.

The Key problem is to choose the correct production from the set of rules.

Example 1.5

$$E \to E + T | T$$
$$T \to T * F | F$$
$$F \to id.$$

After removal of Left recursion,

$E \to TE'$

$E' \to +TE' | \epsilon$

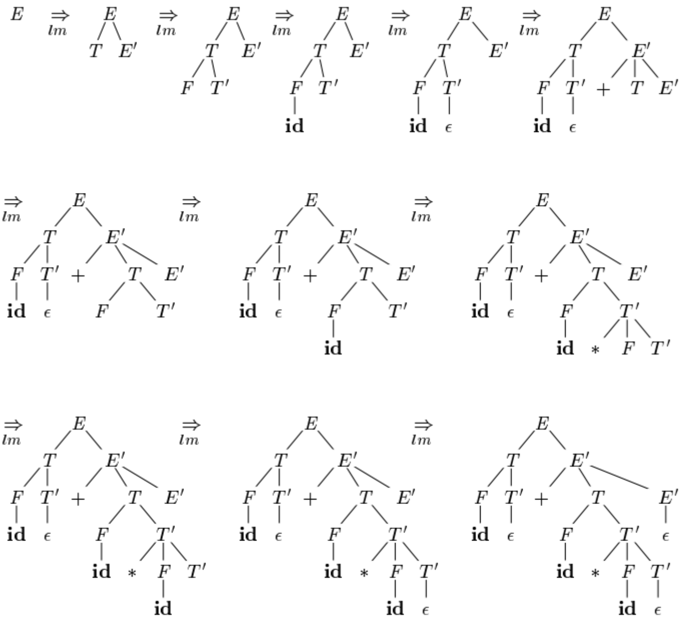
$T \to FT'$

$T' \to *FT' | \epsilon$

$F \to id.$

The derivation of the string $id + id * id$ according to Top-Down parse given in the following figure.

Top-Down Parse



Recursive Descent Parsing

A Recursive Descent parsing program consists of a set of procedures, one for each variable.
Execution begins at the start symbol procedure $S()$.
Success if it scans the entire input and halts.

```
void A() {
        Choose an A-production, A → X1 X2 ··· Xk;
        for ( i = 1 to k ) {
                if ( Xi is a nonterminal )
                        call procedure Xi();
                else if ( Xi equals the current input symbol a )
                        advance the input to the next symbol;
                else /* an error has occurred */;
        }
}
```

Process above is nondeterministic - which A-production to choose.

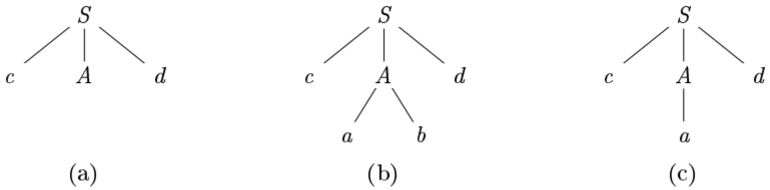
Recursive Descent - Backtracking

$$S \rightarrow cAd$$
$$A \rightarrow ab|a$$

Recursive Descent - Backtracking

$$S \to cAd$$
$$A \to ab|a$$

We want to construct the parse tree for the derivation of string $w = cad$.



Left recursion may cause infinite loop.

$FIRST(X)$

Definition 1.6 ($FIRST(X)$)

The set of terminals that begin strings derived from X.

$FIRST(X)$

Definition 1.6 ($FIRST(X)$)

The set of terminals that begin strings derived from X.

If $A \stackrel{*}{\Longrightarrow} a\beta$, then $a \in FIRST(A)$.

$FIRST(X)$

Definition 1.6 ($FIRST(X)$)

The set of terminals that begin strings derived from X.

If $A \stackrel{*}{\Longrightarrow} a\beta$, then $a \in FIRST(A)$.
Why this is helpful:
Let $A \rightarrow \alpha|\beta$ and $FIRST(\alpha) \cap FIRST(\beta) = \phi$. Then by looking at the next input symbol a, we can say for sure which A-production to choose next.

$FOLLOW(X)$

Definition 1.7 ($FOLLOW(X)$)

The set of terminals a such that there exists a production of the form $S \stackrel{*}{\Longrightarrow} \alpha X a \beta$. If X is the rightmost symbol in some sentential form, then \$ is added to $FOLLOW(X)$. (\$ is a special endmarker symbol)

Computing $FIRST(X)$

Computing $FIRST(X)$

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

Computing $FIRST(X)$

1. If X is a terminal, $FIRST(X) = \{X\}$.
2. If $X \rightarrow X_1 X_2 \ldots X_k$ is a production, $a \in FIRST(X)$ if $X_1 X_2 \ldots X_{l-1} \stackrel{*}{\Longrightarrow} \epsilon$ and $a \in FIRST(X_l)$.

Computing $FIRST(X)$

1. If X is a terminal, $FIRST(X) = \{X\}$.
2. If $X \rightarrow X_1 X_2 \ldots X_k$ is a production, $a \in FIRST(X)$ if $X_1 X_2 \ldots X_{l-1} \stackrel{*}{\Longrightarrow} \epsilon$ and $a \in FIRST(X_l)$.
3. If $X \rightarrow \epsilon$, then add ϵ to $FIRST(X)$.

Computing $FOLLOW(X)$

Computing $FOLLOW(X)$

1. Place \$ in $FOLLOW(S)$ - S is the start symbol.

Computing $FOLLOW(X)$

1. Place \$ in $FOLLOW(S)$ - S is the start symbol.
2. If $A \rightarrow \alpha X \beta$, then everything in $FIRST(\beta)$ except ϵ is added to $FOLLOW(X)$

Computing $FOLLOW(X)$

1. Place \$ in $FOLLOW(S)$ - S is the start symbol.
2. If $A \rightarrow \alpha X \beta$, then everything in $FIRST(\beta)$ except ϵ is added to $FOLLOW(X)$
3. If $A \rightarrow \alpha X$ or $A \rightarrow \alpha X \beta$ and $FIRST(\beta)$ contains ϵ, then everything in $FOLLOW(A)$ is added to $FOLLOW(X)$.

Example

$$E \to TE' \ E' \to +TE'|\epsilon$$
$$T \to FT' \ T' \to *FT'|\epsilon$$
$$F \to (E)|id.$$

Example

$$E \rightarrow TE' \; E' \rightarrow +TE'|\epsilon$$
$$T \rightarrow FT' \; T' \rightarrow *FT'|\epsilon$$
$$F \rightarrow (E)|id.$$

1. If X is a terminal, $FIRST(X) = \{X\}$.
2. If $X \rightarrow X_1X_2 \ldots X_k$ is a production, $a \in FIRST(X)$ if $X_1X_2 \ldots X_{l-1} \stackrel{*}{\Longrightarrow} \epsilon$ and $a \in FIRST(X_l)$.
3. If $X \rightarrow \epsilon$, then add ϵ to $FIRST(X)$.

Example

$$E \rightarrow TE' \; E' \rightarrow +TE'|\epsilon$$
$$T \rightarrow FT' \; T' \rightarrow *FT'|\epsilon$$
$$F \rightarrow (E)|id.$$

1. If X is a terminal, $FIRST(X) = \{X\}$.
2. If $X \rightarrow X_1X_2 \ldots X_k$ is a production, $a \in FIRST(X)$ if $X_1X_2 \ldots X_{l-1} \stackrel{*}{\Longrightarrow} \epsilon$ and $a \in FIRST(X_l)$.
3. If $X \rightarrow \epsilon$, then add ϵ to $FIRST(X)$.

Example

$$E \to TE' \; E' \to +TE'|\epsilon$$
$$T \to FT' \; T' \to *FT'|\epsilon$$
$$F \to (E)|id.$$

1. If X is a terminal, $FIRST(X) = \{X\}$.
2. If $X \to X_1X_2 \ldots X_k$ is a production, $a \in FIRST(X)$ if $X_1X_2 \ldots X_{l-1} \overset{*}{\Longrightarrow} \epsilon$ and $a \in FIRST(X_l)$.
3. If $X \to \epsilon$, then add ϵ to $FIRST(X)$.

1. Place \$ in $FOLLOW(S)$ - S is the start symbol.
2. If $A \to \alpha X\beta$, then everything in $FIRST(\beta)$ except ϵ is added to $FOLLOW(X)$
3. If $A \to \alpha X$ or $A \to \alpha X\beta$ and $FIRST(\beta)$ contains ϵ, then everything in $FOLLOW(A)$ is added to $FOLLOW(X)$.

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

Example

$$E \rightarrow TE' \; E' \rightarrow +TE'|\epsilon$$
$$T \rightarrow FT' \; T' \rightarrow *FT'|\epsilon$$
$$F \rightarrow (E)|id.$$

1. If X is a terminal, $FIRST(X) = \{X\}$.
2. If $X \rightarrow X_1X_2 \ldots X_k$ is a production, $a \in FIRST(X)$ if $X_1X_2 \ldots X_{l-1} \overset{*}{\Longrightarrow} \epsilon$ and $a \in FIRST(X_l)$.
3. If $X \rightarrow \epsilon$, then add ϵ to $FIRST(X)$.

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

1. Place \$ in $FOLLOW(S)$ - S is the start symbol.
2. If $A \rightarrow \alpha X \beta$, then everything in $FIRST(\beta)$ except ϵ is added to $FOLLOW(X)$
3. If $A \rightarrow \alpha X$ or $A \rightarrow \alpha X \beta$ and $FIRST(\beta)$ contains ϵ, then everything in $FOLLOW(A)$ is added to $FOLLOW(X)$.

Example

$$E \rightarrow TE' \; E' \rightarrow +TE' | \epsilon$$
$$T \rightarrow FT' \; T' \rightarrow *FT' | \epsilon$$
$$F \rightarrow (E) | id.$$

1. If X is a terminal, $FIRST(X) = \{X\}$.
2. If $X \rightarrow X_1 X_2 \ldots X_k$ is a production, $a \in FIRST(X)$ if $X_1 X_2 \ldots X_{l-1} \overset{*}{\Longrightarrow} \epsilon$ and $a \in FIRST(X_l)$.
3. If $X \rightarrow \epsilon$, then add ϵ to $FIRST(X)$.

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

1. Place \$ in $FOLLOW(S)$ - S is the start symbol.
2. If $A \rightarrow \alpha X \beta$, then everything in $FIRST(\beta)$ except ϵ is added to $FOLLOW(X)$
3. If $A \rightarrow \alpha X$ or $A \rightarrow \alpha X \beta$ and $FIRST(\beta)$ contains ϵ, then everything in $FOLLOW(A)$ is added to $FOLLOW(X)$.

Example

$$E \rightarrow TE' \; E' \rightarrow +TE' | \epsilon$$
$$T \rightarrow FT' \; T' \rightarrow *FT' | \epsilon$$
$$F \rightarrow (E) | id.$$

1. If X is a terminal, $FIRST(X) = \{X\}$.
2. If $X \rightarrow X_1 X_2 \ldots X_k$ is a production, $a \in FIRST(X)$ if $X_1 X_2 \ldots X_{l-1} \overset{*}{\Longrightarrow} \epsilon$ and $a \in FIRST(X_l)$.
3. If $X \rightarrow \epsilon$, then add ϵ to $FIRST(X)$.

1. Place \$ in $FOLLOW(S)$ - S is the start symbol.
2. If $A \rightarrow \alpha X \beta$, then everything in $FIRST(\beta)$ except ϵ is added to $FOLLOW(X)$
3. If $A \rightarrow \alpha X$ or $A \rightarrow \alpha X \beta$ and $FIRST(\beta)$ contains ϵ, then everything in $FOLLOW(A)$ is added to $FOLLOW(X)$.

$$FIRST(F) = \{(, id\}$$
$$FIRST(T) = FIRST(F)$$
$$FIRST(E) = FIRST(T).$$
$$FIRST(T') = \{*, \epsilon\}$$

Example

$$E \to TE' \; E' \to +TE'|\epsilon$$
$$T \to FT' \; T' \to *FT'|\epsilon$$
$$F \to (E)|id.$$

1. If X is a terminal, $FIRST(X) = \{X\}$.
2. If $X \to X_1 X_2 \dots X_k$ is a production, $a \in FIRST(X)$ if $X_1 X_2 \dots X_{l-1} \stackrel{*}{\Longrightarrow} \epsilon$ and $a \in FIRST(X_l)$.
3. If $X \to \epsilon$, then add ϵ to $FIRST(X)$.

1. Place \$ in $FOLLOW(S)$ - S is the start symbol.
2. If $A \to \alpha X \beta$, then everything in $FIRST(\beta)$ except ϵ is added to $FOLLOW(X)$
3. If $A \to \alpha X$ or $A \to \alpha X \beta$ and $FIRST(\beta)$ contains ϵ, then everything in $FOLLOW(A)$ is added to $FOLLOW(X)$.

$$FIRST(F) = \{(, id\}$$
$$FIRST(T) = FIRST(F)$$
$$FIRST(E) = FIRST(T).$$
$$FIRST(T') = \{*, \epsilon\}$$
$$FIRST(E') = \{+, \epsilon\}$$

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

Example

$$E \rightarrow TE' \; E' \rightarrow +TE'|\epsilon$$
$$T \rightarrow FT' \; T' \rightarrow *FT'|\epsilon$$
$$F \rightarrow (E)|id.$$

1. If X is a terminal, $FIRST(X) = \{X\}$.
2. If $X \rightarrow X_1X_2 \ldots X_k$ is a production, $a \in FIRST(X)$ if $X_1X_2 \ldots X_{l-1} \overset{*}{\Longrightarrow} \epsilon$ and $a \in FIRST(X_l)$.
3. If $X \rightarrow \epsilon$, then add ϵ to $FIRST(X)$.

$$FIRST(F) = \{(, id\}$$
$$FIRST(T) = FIRST(F)$$
$$FIRST(E) = FIRST(T).$$
$$FIRST(T') = \{*, \epsilon\}$$
$$FIRST(E') = \{+, \epsilon\}$$

1. Place \$ in $FOLLOW(S)$ - S is the start symbol.
2. If $A \rightarrow \alpha X \beta$, then everything in $FIRST(\beta)$ except ϵ is added to $FOLLOW(X)$
3. If $A \rightarrow \alpha X$ or $A \rightarrow \alpha X \beta$ and $FIRST(\beta)$ contains ϵ, then everything in $FOLLOW(A)$ is added to $FOLLOW(X)$.

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

Example

$$E \to TE' \quad E' \to +TE'|\epsilon$$
$$T \to FT' \quad T' \to *FT'|\epsilon$$
$$F \to (E)|id.$$

1. If X is a terminal, $FIRST(X) = \{X\}$.
2. If $X \to X_1 X_2 \ldots X_k$ is a production, $a \in FIRST(X)$ if $X_1 X_2 \ldots X_{l-1} \overset{*}{\Longrightarrow} \epsilon$ and $a \in FIRST(X_l)$.
3. If $X \to \epsilon$, then add ϵ to $FIRST(X)$.

1. Place \$ in $FOLLOW(S)$ - S is the start symbol.
2. If $A \to \alpha X \beta$, then everything in $FIRST(\beta)$ except ϵ is added to $FOLLOW(X)$
3. If $A \to \alpha X$ or $A \to \alpha X \beta$ and $FIRST(\beta)$ contains ϵ, then everything in $FOLLOW(A)$ is added to $FOLLOW(X)$.

$$FIRST(F) = \{(, id\}$$
$$FIRST(T) = FIRST(F)$$
$$FIRST(E) = FIRST(T).$$
$$FIRST(T') = \{*, \epsilon\}$$
$$FIRST(E') = \{+, \epsilon\}$$

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

Example

$$E \to TE' \; E' \to +TE'|\epsilon$$
$$T \to FT' \; T' \to *FT'|\epsilon$$
$$F \to (E)|id.$$

1. If X is a terminal, $FIRST(X) = \{X\}$.
2. If $X \to X_1X_2 \ldots X_k$ is a production, $a \in FIRST(X)$ if $X_1X_2 \ldots X_{l-1} \overset{*}{\Longrightarrow} \epsilon$ and $a \in FIRST(X_l)$.
3. If $X \to \epsilon$, then add ϵ to $FIRST(X)$.

1. Place \$ in $FOLLOW(S)$ - S is the start symbol.
2. If $A \to \alpha X \beta$, then everything in $FIRST(\beta)$ except ϵ is added to $FOLLOW(X)$
3. If $A \to \alpha X$ or $A \to \alpha X \beta$ and $FIRST(\beta)$ contains ϵ, then everything in $FOLLOW(A)$ is added to $FOLLOW(X)$.

$$FIRST(F) = \{(, id\}$$
$$FIRST(T) = FIRST(F)$$
$$FIRST(E) = FIRST(T).$$
$$FIRST(T') = \{*, \epsilon\}$$
$$FIRST(E') = \{+, \epsilon\}$$

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

Example

$$E \to TE' \; E' \to +TE'|\epsilon$$
$$T \to FT' \; T' \to *FT'|\epsilon$$
$$F \to (E)|id.$$

1. If X is a terminal, $FIRST(X) = \{X\}$.
2. If $X \to X_1 X_2 \ldots X_k$ is a production, $a \in FIRST(X)$ if $X_1 X_2 \ldots X_{l-1} \stackrel{*}{\Longrightarrow} \epsilon$ and $a \in FIRST(X_l)$.
3. If $X \to \epsilon$, then add ϵ to $FIRST(X)$.

1. Place \$ in $FOLLOW(S)$ - S is the start symbol.
2. If $A \to \alpha X \beta$, then everything in $FIRST(\beta)$ except ϵ is added to $FOLLOW(X)$
3. If $A \to \alpha X$ or $A \to \alpha X \beta$ and $FIRST(\beta)$ contains ϵ, then everything in $FOLLOW(A)$ is added to $FOLLOW(X)$.

$$FIRST(F) = \{(, id\}$$
$$FIRST(T) = FIRST(F)$$
$$FIRST(E) = FIRST(T).$$
$$FIRST(T') = \{*, \epsilon\}$$
$$FIRST(E') = \{+, \epsilon\}$$

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

$LL(1)$ Grammars

Recursive Descent parsers without backtracking can be constructed for a class of grammars known as $LL(1)$ grammars.

Leftmost derivation

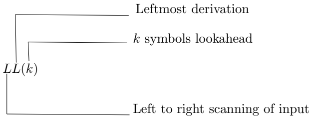
k symbols lookahead

$LL(k)$

Left to right scanning of input

$LL(1)$ Grammars

Recursive Descent parsers without backtracking can be constructed for a class of grammars known as $LL(1)$ grammars.



Definition 1.8 ($LL(1)$ Grammar)

Whenever $A \rightarrow \alpha|\beta$ are two distinct productions in the grammar,

1. For no terminal a do both α and β derive strings starting with a.
2. Atmost one of α or β derives ϵ.
3. If $\beta \stackrel{*}{\Longrightarrow} \epsilon$, then α does not derive any string starting with a terminal in $FOLLOW(A)$.

$LL(1)$ Grammars

Whenever $A \rightarrow \alpha|\beta$ are two distinct productions in the grammar,

1. For no terminal *a* do both α and β derive strings starting with *a*.
2. Atmost one of α or β derives ϵ.
3. If $\beta \stackrel{*}{\Longrightarrow} \epsilon$, then α does not derive any string starting with a terminal in $FOLLOW(A)$.

(1) and (2) ensure that $FIRST(\alpha) \cap FIRST(\beta) = \phi$. (3) ensures that if $\beta \stackrel{*}{\Longrightarrow} \epsilon$, then $FIRST(\alpha) \cap FOLLOW(A) = \phi$.

Algorithm for predictive parsing table

Input: CFG G
Output: Parsing table M Method: For each production $A \rightarrow \alpha$, do the following.

Algorithm for predictive parsing table

Input: CFG G
Output: Parsing table M Method: For each production $A \to \alpha$, do the following.

1. For each terminal $a \in FIRST(\alpha)$, add $A \to \alpha$ to $M[A, a]$.

Algorithm for predictive parsing table

Input: CFG G

Output: Parsing table M Method: For each production $A \rightarrow \alpha$, do the following.

1. For each terminal $a \in FIRST(\alpha)$, add $A \rightarrow \alpha$ to $M[A, a]$.
2. If $\epsilon \in FIRST(\alpha)$, then for each terminal $b \in FOLLOW(A)$, add $A \rightarrow \alpha$ to $M[A, b]$. If $\epsilon \in FIRST(\alpha)$ and $\$ \in FOLLOW(A)$, add $A \rightarrow \alpha$ to $M[A, \$]$.

Empty entries in the table are errors.

Example 1

$$E \to TE' \; E' \to +TE'|\epsilon$$
$$T \to FT' \; T' \to *FT'|\epsilon$$
$$F \to (E)|id.$$

Example 1

$$E \rightarrow TE' \; E' \rightarrow +TE'|\epsilon$$
$$T \rightarrow FT' \; T' \rightarrow *FT'|\epsilon$$
$$F \rightarrow (E)|id.$$

$FIRST(F) = \{(, id\}$ $FIRST(T) = FIRST(F)$
$FIRST(E) = FIRST(T)$. $FIRST(T') = \{*, \epsilon\}$
$FIRST(E') = \{+, \epsilon\}$

Example 1

$$E \rightarrow TE' \; E' \rightarrow +TE'|\epsilon$$
$$T \rightarrow FT' \; T' \rightarrow *FT'|\epsilon$$
$$F \rightarrow (E)|id.$$

$FIRST(F) = \{(, id\}$ $FIRST(T) = FIRST(F)$
$FIRST(E) = FIRST(T)$. $FIRST(T') = \{*, \epsilon\}$
$FIRST(E') = \{+, \epsilon\}$

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

NON - TERMINAL	INPUT SYMBOL					
	id	+	*	(	)	\$
E	$E \rightarrow TE'$			$E \rightarrow TE'$		
E'		$E' \rightarrow +TE'$			$E' \rightarrow \epsilon$	$E' \rightarrow \epsilon$
T	$T \rightarrow FT'$			$T \rightarrow FT'$		
T'		$T' \rightarrow \epsilon$	$T' \rightarrow *FT'$		$T' \rightarrow \epsilon$	$T' \rightarrow \epsilon$
F	$F \rightarrow \mathbf{id}$			$F \rightarrow (E)$		

Example 2

$$S \to iEtSS'|a$$
$$S' \to eS|\epsilon \; E \to b.$$

Example 2

$$S \to iEtSS'|a$$
$$S' \to eS|\epsilon \; E \to b.$$

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

Example 2

$$S \to iEtSS'|a$$
$$S' \to eS|\epsilon \; E \to b.$$

$$FIRST(E) = \{b\}$$
$$FIRST(S') = \{e, \epsilon\}$$

Example 2

$$S \to iEtSS'|a$$
$$S' \to eS|\epsilon \; E \to b.$$

$$FIRST(E) = \{b\}$$
$$FIRST(S') = \{e, \epsilon\}$$
$$FIRST(S) = \{i, a\}.$$

Example 2

$$S \rightarrow iEtSS'|a$$
$$S' \rightarrow eS|\epsilon \; E \rightarrow b.$$

$$FIRST(E) = \{b\}$$
$$FIRST(S') = \{e, \epsilon\}$$
$$FIRST(S) = \{i, a\}.$$

$$FOLLOW(E) = \{t\}$$

Example 2

$$S \rightarrow iEtSS'|a$$
$$S' \rightarrow eS|\epsilon \ E \rightarrow b.$$

$$FIRST(E) = \{b\}$$
$$FIRST(S') = \{e, \epsilon\}$$
$$FIRST(S) = \{i, a\}.$$

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

Example 2

$$S \rightarrow iEtSS'|a$$
$$S' \rightarrow eS|\epsilon \; E \rightarrow b.$$

$$FIRST(E) = \{b\}$$
$$FIRST(S') = \{e, \epsilon\}$$
$$FIRST(S) = \{i, a\}.$$

$$FOLLOW(E) = \{t\}$$
$$FOLLOW(S) = \{\$, e\}$$
$$FOLLOW(S') = \{\$, e\}$$

Example 2

$$S \to iEtSS'|a$$
$$S' \to eS|\epsilon \; E \to b.$$

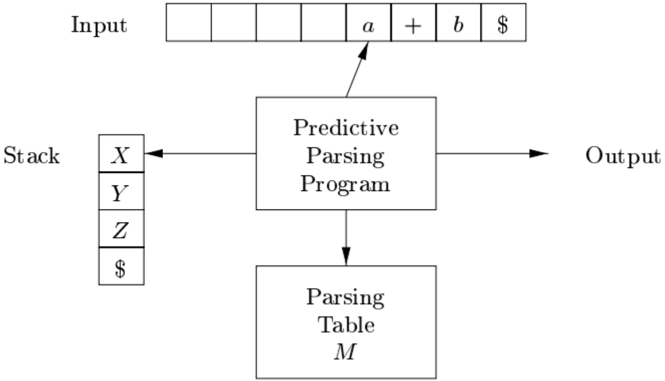
$$FIRST(E) = \{b\}$$
$$FIRST(S') = \{e, \epsilon\}$$
$$FIRST(S) = \{i, a\}.$$

$$FOLLOW(E) = \{t\}$$
$$FOLLOW(S) = \{\$, e\}$$
$$FOLLOW(S') = \{\$, e\}$$

NON - TERMINAL	INPUT SYMBOL					
	a	b	e	i	t	$\$$
S	$S \to a$			$S \to iEtSS'$		
S'			$S' \to \epsilon$ $S' \to eS$			$S' \to \epsilon$
E		$E \to b$				

Nonrecursive (Table driven) Predictive Parser

Maintains a stack explicitly rather than implicitly via recursive calls.
If $S \xRightarrow[lm]{*} w\alpha$, and w is the matched input, then the stack contains α.



Algorithm for Nonrecursive Predictive parser

Input: A parsing table derived from the grammar G and input string w

Output: If $w \in L(G)$, a leftmost derivation of w; else raise an error

Algorithm for Nonrecursive Predictive parser

Input: A parsing table derived from the grammar G and input string w

Output: If $w \in L(G)$, a leftmost derivation of w; else raise an error

Method: Initially, the stack contains $S\$$ with S at the top; the input pointer points to the leftmost symbol of w.

let a be the first symbol of w;
let X be the top stack symbol;
while ($X \neq \$$) { /* stack is not empty */
    if ($X = a$) pop the stack and **let** a be the next symbol of w;
    else if (X is a terminal) *error*();
    else if ($M[X, a]$ is an error entry) *error*();
    else if ($M[X, a] = X \rightarrow Y_1 Y_2 \cdots Y_k$) {
        output the production $X \rightarrow Y_1 Y_2 \cdots Y_k$;
        pop the stack;
        push $Y_k, Y_{k-1}, \ldots, Y_1$ onto the stack, with Y_1 on top;
    }
    let X be the top stack symbol;
}

Example 1

$E \rightarrow TE'$ $E' \rightarrow +TE'|\epsilon$
$T \rightarrow FT'$ $T' \rightarrow *FT'|\epsilon$

$F \rightarrow (E)|id.$

The sequence of moves made by the parser in derivation of $id + id * id$.

Example 1

$E \to TE'$ $E' \to +TE'|\epsilon$
$T \to FT'$ $T' \to *FT'|\epsilon$

$F \to (E)|id.$

The sequence of moves made by the parser in derivation of $id + id * id$.

MATCHED	STACK	INPUT	ACTIONS
	$E\$$	$id + id * id\$$	

Example 1

$E \to TE'$ $E' \to +TE'|\epsilon$
$T \to FT'$ $T' \to *FT'|\epsilon$

$F \to (E)|id.$

The sequence of moves made by the parser in derivation of $id + id * id$.

MATCHED	STACK	INPUT	ACTIONS
	$E\$$	$id + id * id\$$	
	$TE'\$$	$id + id * id\$$	Using $E \to TE'$

Example 1

$E \rightarrow TE'$ $E' \rightarrow +TE'|\epsilon$
$T \rightarrow FT'$ $T' \rightarrow *FT'|\epsilon$

$F \rightarrow (E)|id.$

The sequence of moves made by the parser in derivation of $id + id * id$.

MATCHED	STACK	INPUT	ACTIONS
	$E\$$	$id + id * id\$$	
	$TE'\$$	$id + id * id\$$	Using $E \rightarrow TE'$
	$FT'E'\$$	$id + id * id\$$	Using $T \rightarrow FT'$

Example 1

$E \rightarrow TE'$ $E' \rightarrow +TE'|\epsilon$
$T \rightarrow FT'$ $T' \rightarrow *FT'|\epsilon$

$F \rightarrow (E)|id.$

The sequence of moves made by the parser in derivation of $id + id * id$.

MATCHED	STACK	INPUT	ACTIONS
	$E\$$	$id + id * id\$$	
	$TE'\$$	$id + id * id\$$	Using $E \rightarrow TE'$
	$FT'E'\$$	$id + id * id\$$	Using $T \rightarrow FT'$
	$idT'E'\$$	$id + id * id\$$	Using $F \rightarrow id$

Example 1

$E \rightarrow TE'$ $E' \rightarrow +TE'|\epsilon$
$T \rightarrow FT'$ $T' \rightarrow *FT'|\epsilon$

$F \rightarrow (E)|id.$

The sequence of moves made by the parser in derivation of $id + id * id$.

MATCHED	STACK	INPUT	ACTIONS
	$E\$$	$id + id * id\$$	
	$TE'\$$	$id + id * id\$$	Using $E \rightarrow TE'$
	$FT'E'\$$	$id + id * id\$$	Using $T \rightarrow FT'$
	$idT'E'\$$	$id + id * id\$$	Using $F \rightarrow id$
id	$T'E'\$$	$+id * id\$$	Match id

Example 1

$E \to TE' \quad E' \to +TE'|\epsilon$
$T \to FT' \quad T' \to *FT'|\epsilon$

$F \to (E)|id.$

The sequence of moves made by the parser in derivation of $id + id * id$.

MATCHED	STACK	INPUT	ACTIONS
	$E\$$	$id + id * id\$$	
	$TE'\$$	$id + id * id\$$	Using $E \to TE'$
	$FT'E'\$$	$id + id * id\$$	Using $T \to FT'$
	$idT'E'\$$	$id + id * id\$$	Using $F \to id$
id	$T'E'\$$	$+id * id\$$	Match id
id	$E'\$$	$+id * id\$$	Using $T' \to \epsilon$

Example 1

$E \to TE'$ $E' \to +TE'|\epsilon$
$T \to FT'$ $T' \to *FT'|\epsilon$

$F \to (E)|id.$

The sequence of moves made by the parser in derivation of $id + id * id$.

MATCHED	STACK	INPUT	ACTIONS
	$E\$$	$id + id * id\$$	
	$TE'\$$	$id + id * id\$$	Using $E \to TE'$
	$FT'E'\$$	$id + id * id\$$	Using $T \to FT'$
	$idT'E'\$$	$id + id * id\$$	Using $F \to id$
id	$T'E'\$$	$+id * id\$$	Match id
id	$E'\$$	$+id * id\$$	Using $T' \to \epsilon$
id	$+TE'\$$	$+id * id\$$	Using $E' \to +TE'$

Example 1

$E \to TE'$ $E' \to +TE'|\epsilon$
$T \to FT'$ $T' \to *FT'|\epsilon$

$F \to (E)|id.$

The sequence of moves made by the parser in derivation of $id + id * id$.

MATCHED	STACK	INPUT	ACTIONS
	$E\$$	$id + id * id\$$	
	$TE'\$$	$id + id * id\$$	Using $E \to TE'$
	$FT'E'\$$	$id + id * id\$$	Using $T \to FT'$
	$idT'E'\$$	$id + id * id\$$	Using $F \to id$
id	$T'E'\$$	$+id * id\$$	Match id
id	$E'\$$	$+id * id\$$	Using $T' \to \epsilon$
id	$+TE'\$$	$+id * id\$$	Using $E' \to +TE'$
id+	$TE'\$$	$id * id\$$	Match $+$

Example 1

$E \to TE'$ $E' \to +TE'|\epsilon$
$T \to FT'$ $T' \to *FT'|\epsilon$

$F \to (E)|id.$

The sequence of moves made by the parser in derivation of $id + id * id$.

MATCHED	**STACK**	**INPUT**	**ACTIONS**
	$E\$$	$id + id * id\$$	
	$TE'\$$	$id + id * id\$$	Using $E \to TE'$
	$FT'E'\$$	$id + id * id\$$	Using $T \to FT'$
	$idT'E'\$$	$id + id * id\$$	Using $F \to id$
id	$T'E'\$$	$+id * id\$$	Match id
id	$E'\$$	$+id * id\$$	Using $T' \to \epsilon$
id	$+TE'\$$	$+id * id\$$	Using $E' \to +TE'$
id+	$TE'\$$	$id * id\$$	Match $+$
id+	$FT'E'\$$	$id * id\$$	Using $T \to FT'$

Example 1

$E \rightarrow TE' \quad E' \rightarrow +TE'|\epsilon$
$T \rightarrow FT' \quad T' \rightarrow *FT'|\epsilon$

$F \rightarrow (E)|id.$

The sequence of moves made by the parser in derivation of $id + id * id$.

MATCHED	STACK	INPUT	ACTIONS
	$E\$$	$id + id * id\$$	
	$TE'\$$	$id + id * id\$$	Using $E \rightarrow TE'$
	$FT'E'\$$	$id + id * id\$$	Using $T \rightarrow FT'$
	$idT'E'\$$	$id + id * id\$$	Using $F \rightarrow id$
id	$T'E'\$$	$+id * id\$$	Match id
id	$E'\$$	$+id * id\$$	Using $T' \rightarrow \epsilon$
id	$+TE'\$$	$+id * id\$$	Using $E' \rightarrow +TE'$
id+	$TE'\$$	$id * id\$$	Match $+$
id+	$FT'E'\$$	$id * id\$$	Using $T \rightarrow FT'$
id+	$idT'E'\$$	$id * id\$$	Using $F \rightarrow id$

Example 1

$E \to TE'$ $E' \to +TE'|\epsilon$
$T \to FT'$ $T' \to *FT'|\epsilon$

$F \to (E)|id.$

The sequence of moves made by the parser in derivation of $id + id * id$.

MATCHED	STACK	INPUT	ACTIONS
	$E\$$	$id + id * id\$$	
	$TE'\$$	$id + id * id\$$	Using $E \to TE'$
	$FT'E'\$$	$id + id * id\$$	Using $T \to FT'$
	$idT'E'\$$	$id + id * id\$$	Using $F \to id$
id	$T'E'\$$	$+id * id\$$	Match id
id	$E'\$$	$+id * id\$$	Using $T' \to \epsilon$
id	$+TE'\$$	$+id * id\$$	Using $E' \to +TE'$
id+	$TE'\$$	$id * id\$$	Match $+$
id+	$FT'E'\$$	$id * id\$$	Using $T \to FT'$
id+	$idT'E'\$$	$id * id\$$	Using $F \to id$
id+id	$T'E'\$$	$*id\$$	Match id

MATCHED	STACK	INPUT	ACTIONS
	$E\$$	$id + id * id\$$	
	$TE'\$$	$id + id * id\$$	Using $E \rightarrow TE'$
	$FT'E'\$$	$id + id * id\$$	Using $T \rightarrow FT'$
	$idT'E'\$$	$id + id * id\$$	Using $F \rightarrow id$
id	$T'E'\$$	$+id * id\$$	Match id
id	$E'\$$	$+id * id\$$	Using $T' \rightarrow \epsilon$
id	$+TE'\$$	$+id * id\$$	Using $E' \rightarrow +TE'$
id+	$TE'\$$	$id * id\$$	Match $+$
id+	$FT'E'\$$	$id * id\$$	Using $T \rightarrow FT'$
id+	$idT'E'\$$	$id * id\$$	Using $F \rightarrow id$
id+id	$T'E'\$$	$*id\$$	Match id

MATCHED	STACK	INPUT	ACTIONS
	$E\$$	$id + id * id\$$	
	$TE'\$$	$id + id * id\$$	Using $E \rightarrow TE'$
	$FT'E'\$$	$id + id * id\$$	Using $T \rightarrow FT'$
	$idT'E'\$$	$id + id * id\$$	Using $F \rightarrow id$
id	$T'E'\$$	$+id * id\$$	Match id
id	$E'\$$	$+id * id\$$	Using $T' \rightarrow \epsilon$
id	$+TE'\$$	$+id * id\$$	Using $E' \rightarrow +TE'$
id+	$TE'\$$	$id * id\$$	Match $+$
id+	$FT'E'\$$	$id * id\$$	Using $T \rightarrow FT'$
id+	$idT'E'\$$	$id * id\$$	Using $F \rightarrow id$
id+id	$T'E'\$$	$*id\$$	Match id
id+id	$*FT'E'\$$	$*id\$$	Using $T' \rightarrow *FT'$

MATCHED	STACK	INPUT	ACTIONS
	$E\$$	$id + id * id\$$	
	$TE'\$$	$id + id * id\$$	Using $E \rightarrow TE'$
	$FT'E'\$$	$id + id * id\$$	Using $T \rightarrow FT'$
	$idT'E'\$$	$id + id * id\$$	Using $F \rightarrow id$
id	$T'E'\$$	$+id * id\$$	Match id
id	$E'\$$	$+id * id\$$	Using $T' \rightarrow \epsilon$
id	$+TE'\$$	$+id * id\$$	Using $E' \rightarrow +TE'$
id+	$TE'\$$	$id * id\$$	Match $+$
id+	$FT'E'\$$	$id * id\$$	Using $T \rightarrow FT'$
id+	$idT'E'\$$	$id * id\$$	Using $F \rightarrow id$
id+id	$T'E'\$$	$*id\$$	Match id
id+id	$*FT'E'\$$	$*id\$$	Using $T' \rightarrow *FT'$
id+id*	$FT'E'\$$	$id\$$	Match $*$

MATCHED	STACK	INPUT	ACTIONS
	$E\$$	$id + id * id\$$	
	$TE'\$$	$id + id * id\$$	Using $E \to TE'$
	$FT'E'\$$	$id + id * id\$$	Using $T \to FT'$
	$idT'E'\$$	$id + id * id\$$	Using $F \to id$
id	$T'E'\$$	$+id * id\$$	Match id
id	$E'\$$	$+id * id\$$	Using $T' \to \epsilon$
id	$+TE'\$$	$+id * id\$$	Using $E' \to +TE'$
id+	$TE'\$$	$id * id\$$	Match $+$
id+	$FT'E'\$$	$id * id\$$	Using $T \to FT'$
id+	$idT'E'\$$	$id * id\$$	Using $F \to id$
id+id	$T'E'\$$	$*id\$$	Match id
id+id	$*FT'E'\$$	$*id\$$	Using $T' \to *FT'$
id+id*	$FT'E'\$$	$id\$$	Match $*$
id+id*	$idT'E'\$$	$id\$$	Using $F \to id$

MATCHED	STACK	INPUT	ACTIONS
	$E\$$	$id + id * id\$$	
	$TE'\$$	$id + id * id\$$	Using $E \rightarrow TE'$
	$FT'E'\$$	$id + id * id\$$	Using $T \rightarrow FT'$
	$idT'E'\$$	$id + id * id\$$	Using $F \rightarrow id$
id	$T'E'\$$	$+id * id\$$	Match id
id	$E'\$$	$+id * id\$$	Using $T' \rightarrow \epsilon$
id	$+TE'\$$	$+id * id\$$	Using $E' \rightarrow +TE'$
id+	$TE'\$$	$id * id\$$	Match $+$
id+	$FT'E'\$$	$id * id\$$	Using $T \rightarrow FT'$
id+	$idT'E'\$$	$id * id\$$	Using $F \rightarrow id$
id+id	$T'E'\$$	$*id\$$	Match id
id+id	$*FT'E'\$$	$*id\$$	Using $T' \rightarrow *FT'$
id+id*	$FT'E'\$$	$id\$$	Match $*$
id+id*	$idT'E'\$$	$id\$$	Using $F \rightarrow id$
id+id*id	$T'E'\$$	$\$$	Match id

MATCHED	STACK	INPUT	ACTIONS
	$E\$$	$id + id * id\$$	
	$TE'\$$	$id + id * id\$$	Using $E \rightarrow TE'$
	$FT'E'\$$	$id + id * id\$$	Using $T \rightarrow FT'$
	$idT'E'\$$	$id + id * id\$$	Using $F \rightarrow id$
id	$T'E'\$$	$+id * id\$$	Match id
id	$E'\$$	$+id * id\$$	Using $T' \rightarrow \epsilon$
id	$+TE'\$$	$+id * id\$$	Using $E' \rightarrow +TE'$
id+	$TE'\$$	$id * id\$$	Match $+$
id+	$FT'E'\$$	$id * id\$$	Using $T \rightarrow FT'$
id+	$idT'E'\$$	$id * id\$$	Using $F \rightarrow id$
id+id	$T'E'\$$	$*id\$$	Match id
id+id	$*FT'E'\$$	$*id\$$	Using $T' \rightarrow *FT'$
id+id*	$FT'E'\$$	$id\$$	Match $*$
id+id*	$idT'E'\$$	$id\$$	Using $F \rightarrow id$
id+id*id	$T'E'\$$	$\$$	Match id
id+id*id	$E'\$$	$\$$	Using $T' \rightarrow \epsilon$

MATCHED	STACK	INPUT	ACTIONS
	$E\$$	$id + id * id\$$	
	$TE'\$$	$id + id * id\$$	Using $E \rightarrow TE'$
	$FT'E'\$$	$id + id * id\$$	Using $T \rightarrow FT'$
	$idT'E'\$$	$id + id * id\$$	Using $F \rightarrow id$
id	$T'E'\$$	$+id * id\$$	Match id
id	$E'\$$	$+id * id\$$	Using $T' \rightarrow \epsilon$
id	$+TE'\$$	$+id * id\$$	Using $E' \rightarrow +TE'$
id+	$TE'\$$	$id * id\$$	Match $+$
id+	$FT'E'\$$	$id * id\$$	Using $T \rightarrow FT'$
id+	$idT'E'\$$	$id * id\$$	Using $F \rightarrow id$
id+id	$T'E'\$$	$*id\$$	Match id
id+id	$*FT'E'\$$	$*id\$$	Using $T' \rightarrow *FT'$
id+id*	$FT'E'\$$	$id\$$	Match $*$
id+id*	$idT'E'\$$	$id\$$	Using $F \rightarrow id$
id+id*id	$T'E'\$$	$\$$	Match id
id+id*id	$E'\$$	$\$$	Using $T' \rightarrow \epsilon$
id+id*id	$\$$	$\$$	Using $E' \rightarrow \epsilon$

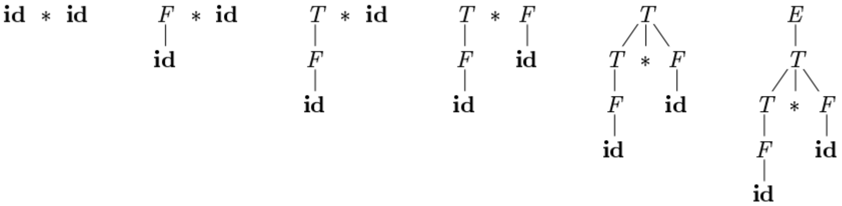
Bottom-Up Parsing

Construction of the parse tree for a input string beginning at the leafs working up towards the root.

$E \rightarrow E + T | T \;\; T \rightarrow T * F | F \;\; F \rightarrow id$.

Bottom-Up Parsing

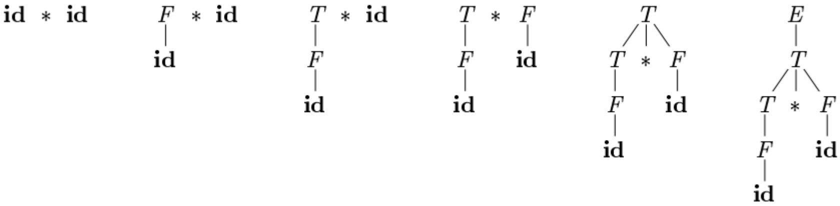
Construction of the parse tree for a input string beginning at the leafs working up towards the root.
$E \rightarrow E + T|T \;\; T \rightarrow T * F|F \;\; F \rightarrow id$.



Bottom-Up Parsing

Construction of the parse tree for a input string beginning at the leafs working up towards the root.

$E \rightarrow E + T | T \;\; T \rightarrow T * F | F \;\; F \rightarrow id.$

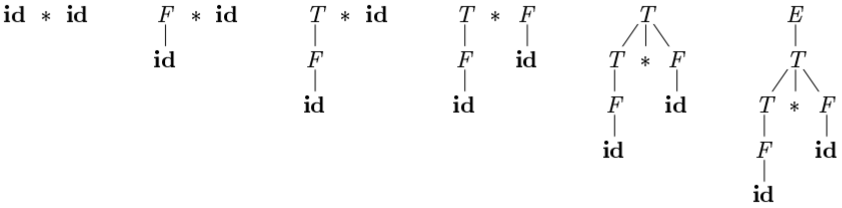


Consider the rightmost derivation of $id * id$ from E.

$E \rightarrow$

Bottom-Up Parsing

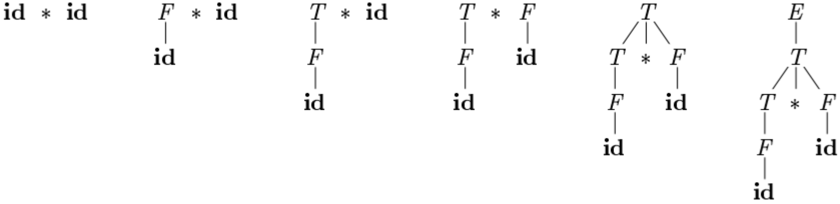
Construction of the parse tree for a input string beginning at the leafs working up towards the root.
$E \to E + T | T \;\; T \to T * F | F \;\; F \to id.$



Consider the rightmost derivation of $id * id$ from E.
$E \to T \to$

Bottom-Up Parsing

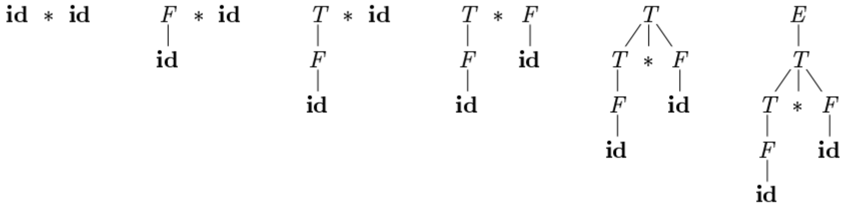
Construction of the parse tree for a input string beginning at the leafs working up towards the root.
$E \rightarrow E + T | T \;\; T \rightarrow T * F | F \;\; F \rightarrow id.$



Consider the rightmost derivation of $id * id$ from E.
$E \rightarrow T \rightarrow T * F \rightarrow$

Bottom-Up Parsing

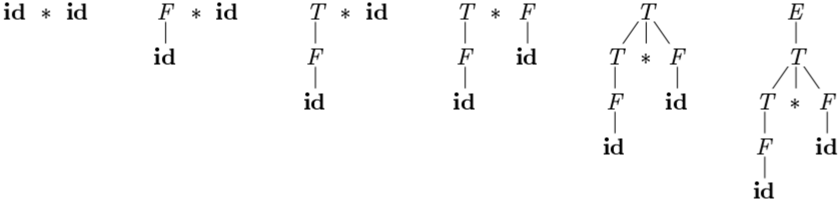
Construction of the parse tree for a input string beginning at the leafs working up towards the root.
$E \to E + T|T \;\; T \to T * F|F \;\; F \to id.$



Consider the rightmost derivation of $id * id$ from E.
$E \to T \to T * F \to T * id \to$

Bottom-Up Parsing

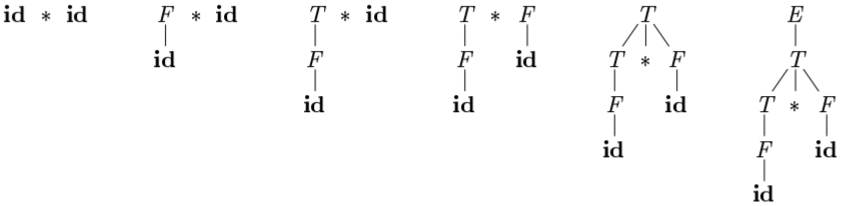
Construction of the parse tree for a input string beginning at the leafs working up towards the root.
$E \rightarrow E + T | T \;\; T \rightarrow T * F | F \;\; F \rightarrow id.$



Consider the rightmost derivation of $id * id$ from E.
$E \rightarrow T \rightarrow T * F \rightarrow T * id \rightarrow F * id \rightarrow$

Bottom-Up Parsing

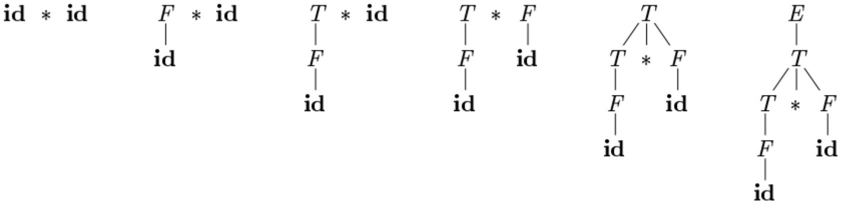
Construction of the parse tree for a input string beginning at the leafs working up towards the root.
$E \rightarrow E + T|T \;\; T \rightarrow T * F|F \;\; F \rightarrow id.$



Consider the rightmost derivation of $id * id$ from E.
$E \rightarrow T \rightarrow T * F \rightarrow T * id \rightarrow F * id \rightarrow id * id.$

Bottom-Up Parsing

Construction of the parse tree for a input string beginning at the leafs working up towards the root.
$E \rightarrow E + T | T \;\; T \rightarrow T * F | F \;\; F \rightarrow id.$



Consider the rightmost derivation of $id * id$ from E.
$E \rightarrow T \rightarrow T * F \rightarrow T * id \rightarrow F * id \rightarrow id * id.$
Can you see any connection between the parse tree construction and derivation?

Reductions

Reductions

- Bottom-up parsing is the process of reduction of some string w to the start symbol S(if at all possible)

Reductions

- Bottom-up parsing is the process of reduction of some string w to the start symbol S(if at all possible)
- At each reduction step, a specific substring is replaced by a variable at the head of some production.

Reductions

- Bottom-up parsing is the process of reduction of some string w to the start symbol S(if at all possible)
- At each reduction step, a specific substring is replaced by a variable at the head of some production.
- Key decisions are when to reduce and what production to apply.

Reductions

- Bottom-up parsing is the process of reduction of some string w to the start symbol S(if at all possible)
- At each reduction step, a specific substring is replaced by a variable at the head of some production.
- Key decisions are when to reduce and what production to apply.
- Goal of BU parsing is to create the derivation tree in reverse.

Handle

Definition 1.9

A handle is a substring that matches the body of a production, whose reduction represents one step along the reverse of the rightmost derivation.

RIGHT SENTENTIAL FORM	HANDLE	REDUCING PRODUCTION
$\mathbf{id}_1 * \mathbf{id}_2$	$\mathbf{id}_1$	$F \to \mathbf{id}$
$F * \mathbf{id}_2$	F	$T \to F$
$T * \mathbf{id}_2$	$\mathbf{id}_2$	$F \to \mathbf{id}$
$T * F$	$T * F$	$T \to T * F$
T	T	$E \to T$

Handle and Pruning

Definition 1.10

Formally, if $S \underset{rm}{\overset{*}{\rightarrow}} \alpha A \gamma \rightarrow \alpha\beta\gamma$, then $A \rightarrow \beta$ is the handle of $\alpha\beta\gamma$.

Handle and Pruning

Definition 1.10

Formally, if $S \xrightarrow[rm]{*} \alpha A\gamma \to \alpha\beta\gamma$, then $A \to \beta$ is the handle of $\alpha\beta\gamma$.

Consider the derivation $S \to r_0 \to r_1 \to \cdots \to r_{n-1} \to r_n = w$.

Handle and Pruning

Definition 1.10

Formally, if $S \overset{*}{\underset{rm}{\rightarrow}} \alpha A\gamma \rightarrow \alpha\beta\gamma$, then $A \rightarrow \beta$ is the handle of $\alpha\beta\gamma$.

Consider the derivation $S \rightarrow r_0 \rightarrow r_1 \rightarrow \cdots \rightarrow r_{n-1} \rightarrow r_n = w$.
Objective: Construct the derivation tree in reverse order.

Handle and Pruning

Definition 1.10

Formally, if $S \overset{*}{\underset{rm}{\to}} \alpha A \gamma \to \alpha\beta\gamma$, then $A \to \beta$ is the handle of $\alpha\beta\gamma$.

Consider the derivation $S \to r_0 \to r_1 \to \cdots \to r_{n-1} \to r_n = w$.
Objective: Construct the derivation tree in reverse order.
How achieved: via Handle Pruning -

Handle and Pruning

Definition 1.10

Formally, if $S \xrightarrow[rm]{*} \alpha A \gamma \rightarrow \alpha\beta\gamma$, then $A \rightarrow \beta$ is the handle of $\alpha\beta\gamma$.

Consider the derivation $S \rightarrow r_0 \rightarrow r_1 \rightarrow \cdots \rightarrow r_{n-1} \rightarrow r_n = w$.
Objective: Construct the derivation tree in reverse order.
How achieved: via Handle Pruning -

1. Locate the handle β_n in r_n

Handle and Pruning

Definition 1.10

Formally, if $S \overset{*}{\underset{rm}{\rightarrow}} \alpha A \gamma \rightarrow \alpha\beta\gamma$, then $A \rightarrow \beta$ is the handle of $\alpha\beta\gamma$.

Consider the derivation $S \rightarrow r_0 \rightarrow r_1 \rightarrow \cdots \rightarrow r_{n-1} \rightarrow r_n = w$.
Objective: Construct the derivation tree in reverse order.
How achieved: via Handle Pruning -

1. Locate the handle β_n in r_n
2. Replace β_n with the head of the production $A_n \rightarrow \beta_n$ to obtain r_{n-1}

Handle and Pruning

Definition 1.10

Formally, if $S \underset{rm}{\overset{*}{\rightarrow}} \alpha A \gamma \rightarrow \alpha\beta\gamma$, then $A \rightarrow \beta$ is the handle of $\alpha\beta\gamma$.

Consider the derivation $S \rightarrow r_0 \rightarrow r_1 \rightarrow \cdots \rightarrow r_{n-1} \rightarrow r_n = w$.
Objective: Construct the derivation tree in reverse order.
How achieved: via Handle Pruning -

1. Locate the handle β_n in r_n
2. Replace β_n with the head of the production $A_n \rightarrow \beta_n$ to obtain r_{n-1}
3. Repeat until S is reached.

Shift-Reduce Parsing

STACK	INPUT	ACTIONS

Shift-Reduce Parsing

STACK	INPUT	ACTIONS
$	$id_1 * id_2\$$	shift

Shift-Reduce Parsing

STACK	INPUT	ACTIONS
$\$$	$id_1 * id_2\$$	shift
$\$id_1$	$*id_2\$$	reduce by $F \rightarrow id$

Shift-Reduce Parsing

STACK	INPUT	ACTIONS
$\$$	$id_1 * id_2\$$	shift
$\$id_1$	$*id_2\$$	reduce by $F \rightarrow id$
$\$F$	$*id_2\$$	reduce by $T \rightarrow F$

Shift-Reduce Parsing

STACK	INPUT	ACTIONS
$\$$	$id_1 * id_2\$$	shift
$\$id_1$	$*id_2\$$	reduce by $F \to id$
$\$F$	$*id_2\$$	reduce by $T \to F$
$\$T$	$*id_2\$$	shift

Shift-Reduce Parsing

STACK	INPUT	ACTIONS
$\$$	$id_1 * id_2\$$	shift
$\$id_1$	$*id_2\$$	reduce by $F \rightarrow id$
$\$F$	$*id_2\$$	reduce by $T \rightarrow F$
$\$T$	$*id_2\$$	shift
$\$T*$	$id_2\$$	shift

Shift-Reduce Parsing

STACK	INPUT	ACTIONS
$\$$	$id_1 * id_2\$$	shift
$\$id_1$	$*id_2\$$	reduce by $F \to id$
$\$F$	$*id_2\$$	reduce by $T \to F$
$\$T$	$*id_2\$$	shift
$\$T*$	$id_2\$$	shift
$\$T * id_2$	$\$$	reduce by $F \to id$

Shift-Reduce Parsing

STACK	INPUT	ACTIONS
$\$$	$id_1 * id_2\$$	shift
$\$id_1$	$*id_2\$$	reduce by $F \to id$
$\$F$	$*id_2\$$	reduce by $T \to F$
$\$T$	$*id_2\$$	shift
$\$T*$	$id_2\$$	shift
$\$T * id_2$	$\$$	reduce by $F \to id$
$\$T * F$	$\$$	reduce by $T \to T * F$

Shift-Reduce Parsing

STACK	INPUT	ACTIONS
$\$$	$id_1 * id_2\$$	shift
$\$id_1$	$*id_2\$$	reduce by $F \rightarrow id$
$\$F$	$*id_2\$$	reduce by $T \rightarrow F$
$\$T$	$*id_2\$$	shift
$\$T*$	$id_2\$$	shift
$\$T * id_2$	$\$$	reduce by $F \rightarrow id$
$\$T * F$	$\$$	reduce by $T \rightarrow T * F$
$\$T$	$\$$	reduce by $E \rightarrow T$

Shift-Reduce Parsing

STACK	INPUT	ACTIONS
$\$$	$id_1 * id_2\$$	shift
$\$id_1$	$*id_2\$$	reduce by $F \rightarrow id$
$\$F$	$*id_2\$$	reduce by $T \rightarrow F$
$\$T$	$*id_2\$$	shift
$\$T*$	$id_2\$$	shift
$\$T * id_2$	$\$$	reduce by $F \rightarrow id$
$\$T * F$	$\$$	reduce by $T \rightarrow T * F$
$\$T$	$\$$	reduce by $E \rightarrow T$
$\$E$	$\$$	accept

Shift-Reduce Parsing

Stack is useful here due to the following fact - The handle will always appear at the top of the stack, never inside WHY?. Parser sometimes

Shift-Reduce Parsing

Stack is useful here due to the following fact - The handle will always appear at the top of the stack, never inside WHY?.
Parser sometimes

- cannot decide whether to shift or reduce - shift-reduce conflict.

Shift-Reduce Parsing

Stack is useful here due to the following fact - The handle will always appear at the top of the stack, never inside WHY?.
Parser sometimes

- cannot decide whether to shift or reduce - shift-reduce conflict.
- cannot decide among several reductions - reduce-reduce conflict.

Shift-Reduce Parsing

Stack is useful here due to the following fact - The handle will always appear at the top of the stack, never inside WHY?.

Parser sometimes

- cannot decide whether to shift or reduce - shift-reduce conflict.
- cannot decide among several reductions - reduce-reduce conflict.

Example 1.11 ($S \rightarrow iEtS | iEtSeS | \ldots$)

Stack - $iEtS$ Remaining input - $e \ldots \$$

Here, we cannot decide whether to reduce $iEtS$ by $S \rightarrow iEtS$ or to shift e.

$LR(k)$ Parsers

L - Left to right scanning of input

$LR(k)$ Parsers

L - Left to right scanning of input
R - Rightmost derivation

$LR(k)$ Parsers

L - Left to right scanning of input
R - Rightmost derivation
k - Lookahead used for parsing decisions.

$LR(k)$ Parsers

L - Left to right scanning of input
R - Rightmost derivation
k - Lookahead used for parsing decisions.
LR parsers are table-driven much like the nonrecursive predictive parser.

$LR(k)$ Parsers

L - Left to right scanning of input

R - Rightmost derivation

k - Lookahead used for parsing decisions.

LR parsers are table-driven much like the nonrecursive predictive parser.

Can recognize most programming language constructs.

$LR(k)$ Parsers

L - Left to right scanning of input
R - Rightmost derivation
k - Lookahead used for parsing decisions.
LR parsers are table-driven much like the nonrecursive predictive parser.
Can recognize most programming language constructs.
Non backtracking shift-reduce parser

$LR(k)$ Parsers

L - Left to right scanning of input
R - Rightmost derivation
k - Lookahead used for parsing decisions.
LR parsers are table-driven much like the nonrecursive predictive parser.
Can recognize most programming language constructs.
Non backtracking shift-reduce parser
The class of grammars for LR methods are superset of LL methods.

$LR(k)$ Parser - LR(0) Items

The LR parser makes shift-reduce decisions by maintaining states to keep track of where we are in the current parse.

$LR(k)$ Parser - LR(0) Items

The LR parser makes shift-reduce decisions by maintaining states to keep track of where we are in the current parse.
States are a set of items.

$LR(k)$ Parser - LR(0) Items

The LR parser makes shift-reduce decisions by maintaining states to keep track of where we are in the current parse.

States are a set of items.

An $LR(0)$ item of a grammar is a production of G with a dot . at some position of the body.

$LR(k)$ Parser - LR(0) Items

The LR parser makes shift-reduce decisions by maintaining states to keep track of where we are in the current parse.

States are a set of items.

An $LR(0)$ item of a grammar is a production of G with a dot . at some position of the body.

Example 1.12 ($A \rightarrow XYZ$ yields 4 items)

$A \rightarrow .XYZ$
$A \rightarrow X.YZ$
$A \rightarrow XY.Z$
$A \rightarrow XYZ.$

$LR(k)$ Parser - LR(0) Items

The LR parser makes shift-reduce decisions by maintaining states to keep track of where we are in the current parse.
States are a set of items.
An $LR(0)$ item of a grammar is a production of G with a dot . at some position of the body.

Example 1.12 ($A \rightarrow XYZ$ yields 4 items)

$A \rightarrow .XYZ$
$A \rightarrow X.YZ$
$A \rightarrow XY.Z$
$A \rightarrow XYZ.$
$A \rightarrow \epsilon$ yields $A \rightarrow .$

$LR(k)$ Parser - LR(0) Items

The LR parser makes shift-reduce decisions by maintaining states to keep track of where we are in the current parse.
States are a set of items.
An $LR(0)$ item of a grammar is a production of G with a dot . at some position of the body.

Example 1.12 ($A \to XYZ$ yields 4 items)

$A \to .XYZ$
$A \to X.YZ$
$A \to XY.Z$
$A \to XYZ.$
$A \to \epsilon$ yields $A \to .$

$A \to .XYZ$ - We expect to see string derived from XYZ.

$LR(k)$ Parser - LR(0) Items

The LR parser makes shift-reduce decisions by maintaining states to keep track of where we are in the current parse.
States are a set of items.
An $LR(0)$ item of a grammar is a production of G with a dot . at some position of the body.

Example 1.12 ($A \rightarrow XYZ$ yields 4 items)

$A \rightarrow .XYZ$
$A \rightarrow X.YZ$
$A \rightarrow XY.Z$
$A \rightarrow XYZ.$
$A \rightarrow \epsilon$ yields $A \rightarrow .$

$A \rightarrow .XYZ$ - We expect to see string derived from XYZ.
$A \rightarrow X.YZ$ - We have seen string derived from X; we expect to see string derived from YZ.

$LR(k)$ Parser - LR(0) Items

The LR parser makes shift-reduce decisions by maintaining states to keep track of where we are in the current parse.
States are a set of items.
An $LR(0)$ item of a grammar is a production of G with a dot . at some position of the body.

Example 1.12 ($A \rightarrow XYZ$ yields 4 items)

$A \rightarrow .XYZ$
$A \rightarrow X.YZ$
$A \rightarrow XY.Z$
$A \rightarrow XYZ.$
$A \rightarrow \epsilon$ yields $A \rightarrow .$

$A \rightarrow .XYZ$ - We expect to see string derived from XYZ.
$A \rightarrow X.YZ$ - We have seen string derived from X; we expect to see string derived from YZ.
$A \rightarrow XYZ.$ - We have seen string derived from XYZ; It is time to reduce XYZ to A.

Cannonical $LR(0)$ Collection of items: It is the collection of a set of $LR(0)$ items used for creating a DFA that is used for parsing decisions.

Cannonical $LR(0)$ Collection of items: It is the collection of a set of $LR(0)$ items used for creating a DFA that is used for parsing decisions.
The automaton is called $LR(0)$ automaton.

Definition 1.13 (Augmented Grammar)

G is the grammar with start symbol S. G' is constructed from G by

- adding a new start symbol S'

Cannonical $LR(0)$ Collection of items: It is the collection of a set of $LR(0)$ items used for creating a DFA that is used for parsing decisions.
The automaton is called $LR(0)$ automaton.

Definition 1.13 (Augmented Grammar)

G is the grammar with start symbol S. G' is constructed from G by

- adding a new start symbol S'
- adding the production $S' \to S$.

Cannonical $LR(0)$ Collection of items: It is the collection of a set of $LR(0)$ items used for creating a DFA that is used for parsing decisions.
The automaton is called $LR(0)$ automaton.

Definition 1.13 (Augmented Grammar)

G is the grammar with start symbol S. G' is constructed from G by

- adding a new start symbol S'
- adding the production $S' \to S$.

this ensures the start symbol never lying in the body of any production.

Cannonical $LR(0)$ Collection of items: It is the collection of a set of $LR(0)$ items used for creating a DFA that is used for parsing decisions.
The automaton is called $LR(0)$ automaton.

Definition 1.13 (Augmented Grammar)

G is the grammar with start symbol S. G' is constructed from G by

- adding a new start symbol S'
- adding the production $S' \rightarrow S$.

this ensures the start symbol never lying in the body of any production.
Acceptance occurs when the parser is about to reduce by $S' \rightarrow S$..

Definition 1.14 (CLOSURE of itemsets)

For item I, $CLOSURE(I)$ is constructed using two rules.

Definition 1.14 (CLOSURE of itemsets)

For item I, $CLOSURE(I)$ is constructed using two rules.

1. Add every item in I to $CLOSURE(I)$.

Definition 1.14 (CLOSURE of itemsets)

For item I, $CLOSURE(I)$ is constructed using two rules.

1. Add every item in I to $CLOSURE(I)$.
2. If $A \rightarrow \alpha.B\beta$ is in $CLOSURE(I)$, and $B \rightarrow \gamma$ is a production in G, then add $B \rightarrow .\gamma$ to $CLOSURE(I)$ (if not already present). Repeat until no new production is added to $CLOSURE(I)$.

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

Definition 1.14 (CLOSURE of itemsets)

For item I, $CLOSURE(I)$ is constructed using two rules.

1. Add every item in I to $CLOSURE(I)$.
2. If $A \rightarrow \alpha.B\beta$ is in $CLOSURE(I)$, and $B \rightarrow \gamma$ is a production in G, then add $B \rightarrow .\gamma$ to $CLOSURE(I)$ (if not already present). Repeat until no new production is added to $CLOSURE(I)$.

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

Let I contains $[E' \rightarrow .E]$, then $CLOSURE(I)$ contains the following items.

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

Two categories of items exist.

Two categories of items exist.

- Kernel items: $S' \to S$ and all productions where . is not in the leftmost position in the production body.

Two categories of items exist.

- Kernel items: $S' \rightarrow S$ and all productions where . is not in the leftmost position in the production body.
- NonKernel items: All items with . at the leftmost position except $S' \rightarrow S$.

GOTO function

If $[A \to \alpha.X\beta] \in I$, $GOTO(I, X)$ contains $CLOSURE([A \to \alpha X.\beta])$.

Example 1.15 (I contains two items: $E' \to E$ and $E \to E. + T$)

$GOTO(I, +)$ contains the following items:

GOTO function

If $[A \to \alpha.X\beta] \in I$, $GOTO(I, X)$ contains $CLOSURE([A \to \alpha X.\beta])$.

Example 1.15 (I contains two items: $E' \to E$ and $E \to E. + T$)

$GOTO(I, +)$ contains the following items:
$E \to E + .T$

GOTO function

If $[A \to \alpha.X\beta] \in I$, $GOTO(I, X)$ contains $CLOSURE([A \to \alpha X.\beta])$.

Example 1.15 (I contains two items: $E' \to E$ and $E \to E. + T$)

$GOTO(I, +)$ contains the following items:
$E \to E + .T$
$T \to .T * F$

$GOTO$ function

If $[A \to \alpha.X\beta] \in I$, $GOTO(I, X)$ contains $CLOSURE([A \to \alpha X.\beta])$.

Example 1.15 (I contains two items: $E' \to E$ and $E \to E. + T$)

$GOTO(I, +)$ contains the following items:
$E \to E + .T$
$T \to .T * F$
$T \to .F$

GOTO function

If $[A \to \alpha.X\beta] \in I$, $GOTO(I, X)$ contains $CLOSURE([A \to \alpha X.\beta])$.

Example 1.15 (I contains two items: $E' \to E$ and $E \to E. + T$)

$GOTO(I, +)$ contains the following items:
$E \to E + .T$
$T \to .T * F$
$T \to .F$
$F \to .(E)$

GOTO function

If $[A \rightarrow \alpha.X\beta] \in I$, $GOTO(I, X)$ contains $CLOSURE([A \rightarrow \alpha X.\beta])$.

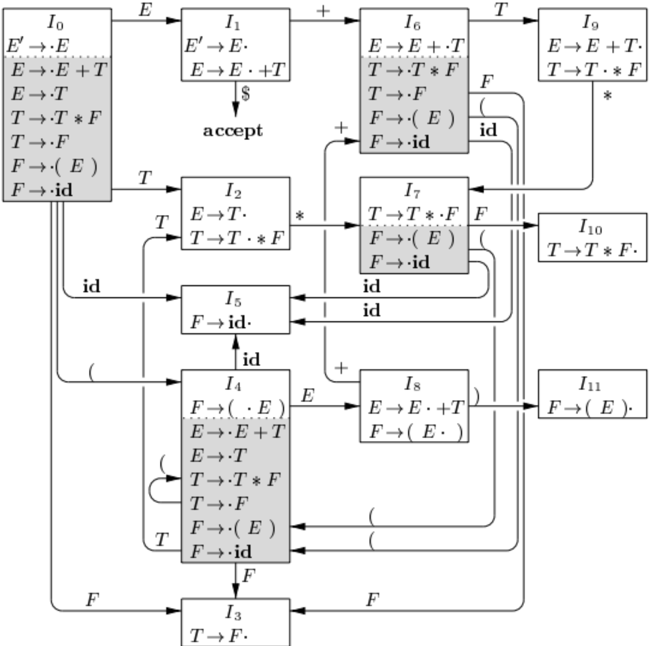
Example 1.15 (I contains two items: $E' \rightarrow E$ and $E \rightarrow E. + T$)

$GOTO(I, +)$ contains the following items:
$E \rightarrow E + .T$
$T \rightarrow .T * F$
$T \rightarrow .F$
$F \rightarrow .(E)$
$F \rightarrow .id$

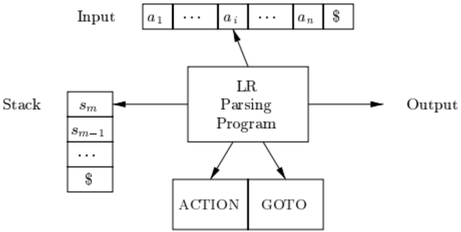
Algorithm for $LR(0)$ automaton construction

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

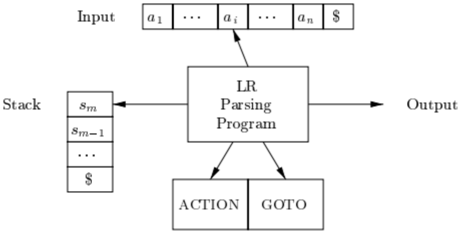
The $LR(0)$ automaton



LR parsing algorithm

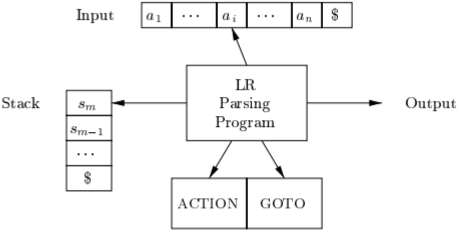


LR parsing algorithm



Shifts states rather than symbols on to the stack.

LR parsing algorithm



Shifts states rather than symbols on to the stack.
Each state except the start state has a grammar symbol associated with it. (HOW?)

LR parsing algorithm

Structure of the Parsing table

Definition 1.16

$ACTION[i, a]$

LR parsing algorithm

Structure of the Parsing table

Definition 1.16

$ACTION[i, a]$

1. Shift j: pushes state j on to stack as representative of a.

LR parsing algorithm

Structure of the Parsing table

Definition 1.16

$ACTION[i, a]$

1. Shift j: pushes state j on to stack as representative of a.
2. Reduce $A \rightarrow \beta$: β on the top of the stack is reduced to A.

LR parsing algorithm

Structure of the Parsing table

Definition 1.16

$ACTION[i, a]$

1. Shift j: pushes state j on to stack as representative of a.
2. Reduce $A \rightarrow \beta$: β on the top of the stack is reduced to A.
3. Accept: S' at the top, \$ in input - Parser accepts and halts.

LR parsing algorithm

Structure of the Parsing table

Definition 1.16

$ACTION[i, a]$

1. Shift j: pushes state j on to stack as representative of a.
2. Reduce $A \to \beta$: β on the top of the stack is reduced to A.
3. Accept: S' at the top, \$ in input - Parser accepts and halts.
4. Error:

Definition 1.17

If $GOTO[I_i, A] = I_j$, GOTO maps a state i and variable A to state j.

LR parsing algorithm

Parser Configuration

It is a pair $(s_0 s_1 \ldots s_m, a_i a_{i+1} \ldots a_n)$ -
represents the right sentential form $X_1 X_2 \ldots X_m a_i a_{i+1} \ldots a_n$, where X_i is associated with s_i.

LR algorithm- Parser Behaviour

Let parser configuration be $(s_0 s_1 \ldots s_m, a_i a_{i+1} \ldots a_n)$

LR algorithm- Parser Behaviour

Let parser configuration be $(s_0 s_1 \ldots s_m, a_i a_{i+1} \ldots a_n)$

1. $ACTION[s_m, a_i] =$ shift s - Parser enters the configuration $(s_0 s_1 \ldots s_m s, a_{i+1} \ldots a_n)$

LR algorithm- Parser Behaviour

Let parser configuration be $(s_0 s_1 \ldots s_m, a_i a_{i+1} \ldots a_n)$

1. $ACTION[s_m, a_i] =$ shift s - Parser enters the configuration $(s_0 s_1 \ldots s_m s, a_{i+1} \ldots a_n)$
2. $ACTION[s_m, a_i] =$ reduce $A \rightarrow B$ - Parser executes a reduce move entering the configuration $(s_0 s_1 \ldots s_{m-r} s, a_i a_{i+1} \ldots a_n)$, where
   - $r =$ length of B
   - $s = GOTO[s_{m-r}, A]$

LR algorithm- Parser Behaviour

Let parser configuration be $(s_0 s_1 \ldots s_m, a_i a_{i+1} \ldots a_n)$

1. $ACTION[s_m, a_i] =$ shift s - Parser enters the configuration $(s_0 s_1 \ldots s_m s, a_{i+1} \ldots a_n)$
2. $ACTION[s_m, a_i] =$ reduce $A \rightarrow B$ - Parser executes a reduce move entering the configuration $(s_0 s_1 \ldots s_{m-r} s, a_i a_{i+1} \ldots a_n)$, where
   - $r =$ length of B
   - $s = GOTO[s_{m-r}, A]$
3. $ACTION[s_m, a_i] =$ accept - Parsing is complete

LR algorithm- Parser Behaviour

Let parser configuration be $(s_0 s_1 \ldots s_m, a_i a_{i+1} \ldots a_n)$

1. $ACTION[s_m, a_i] =$ shift s - Parser enters the configuration $(s_0 s_1 \ldots s_m s, a_{i+1} \ldots a_n)$
2. $ACTION[s_m, a_i] =$ reduce $A \rightarrow B$ - Parser executes a reduce move entering the configuration $(s_0 s_1 \ldots s_{m-r} s, a_i a_{i+1} \ldots a_n)$, where
   - $r =$ length of B
   - $s = GOTO[s_{m-r}, A]$
3. $ACTION[s_m, a_i] =$ accept - Parsing is complete
4. $ACTION[s_m, a_i] =$ error - Report

Construction of $SLR(1)$ parsing table

Input - Augmented grammar G'
Output - SLR parsing table with functions $ACTION$ and $GOTO$

Construction of $SLR(1)$ parsing table

Input - Augmented grammar G'
Output - SLR parsing table with functions $ACTION$ and $GOTO$

1. Construct $C = \{I_0, I_1, \ldots, I_n\}$, the collection of $LR(0)$ items. State i is constructed from I_i

Construction of $SLR(1)$ parsing table

Input - Augmented grammar G'
Output - SLR parsing table with functions $ACTION$ and $GOTO$

1. Construct $C = \{I_0, I_1, \ldots, I_n\}$, the collection of $LR(0)$ items. State i is constructed from I_i
2. $ACTION$ for state i
   - if $[A \rightarrow \alpha.a\beta]$ is in I_i, and $GOTO(I_i, a) = I_j$, set $ACTION[i, a] = sj$, for terminal a.

Construction of $SLR(1)$ parsing table

Input - Augmented grammar G'
Output - SLR parsing table with functions $ACTION$ and $GOTO$

1. Construct $C = \{I_0, I_1, \ldots, I_n\}$, the collection of $LR(0)$ items. State i is constructed from I_i
2. $ACTION$ for state i
   - if $[A \rightarrow \alpha.a\beta]$ is in I_i, and $GOTO(I_i, a) = I_j$, set $ACTION[i, a] = sj$, for terminal a.
   - if $[A \rightarrow \alpha.]$ is in I_i, set $ACTION[i, a] = r`A \rightarrow \alpha'$, for each terminal $a \in FOLLOW(A)$.

Construction of $SLR(1)$ parsing table

Input - Augmented grammar G'
Output - SLR parsing table with functions $ACTION$ and $GOTO$

1. Construct $C = \{I_0, I_1, \ldots, I_n\}$, the collection of $LR(0)$ items. State i is constructed from I_i
2. $ACTION$ for state i
   - if $[A \rightarrow \alpha.a\beta]$ is in I_i, and $GOTO(I_i, a) = I_j$, set $ACTION[i, a] = sj$, for terminal a.
   - if $[A \rightarrow \alpha.]$ is in I_i, set $ACTION[i, a] = r`A \rightarrow \alpha'$, for each terminal $a \in FOLLOW(A)$.
   - if $[S' \rightarrow S.]$ is in I_i, set $ACTION[i, \$]$ =accept.

     if any conflciting actions result from these rules, the grammar is not $SLR(1)$.

Construction of $SLR(1)$ parsing table

Input - Augmented grammar G'
Output - SLR parsing table with functions $ACTION$ and $GOTO$

1. Construct $C = \{I_0, I_1, \ldots, I_n\}$, the collection of $LR(0)$ items. State i is constructed from I_i
2. $ACTION$ for state i
   - if $[A \to \alpha.a\beta]$ is in I_i, and $GOTO(I_i, a) = I_j$, set $ACTION[i, a] = sj$, for terminal a.
   - if $[A \to \alpha.]$ is in I_i, set $ACTION[i, a] = r`A \to \alpha'$, for each terminal $a \in FOLLOW(A)$.
   - if $[S' \to S.]$ is in I_i, set $ACTION[i, \$] =$accept.
     if any conflciting actions result from these rules, the grammar is not $SLR(1)$.
3. For variables A, if $GOTO(I_i, A) = I_j$, set $GOTO[i, A] = j$.

Construction of $SLR(1)$ parsing table

Input - Augmented grammar G'
Output - SLR parsing table with functions $ACTION$ and $GOTO$

1. Construct $C = \{I_0, I_1, \ldots, I_n\}$, the collection of $LR(0)$ items. State i is constructed from I_i
2. $ACTION$ for state i
   - if $[A \rightarrow \alpha.a\beta]$ is in I_i, and $GOTO(I_i, a) = I_j$, set $ACTION[i, a] = sj$, for terminal a.
   - if $[A \rightarrow \alpha.]$ is in I_i, set $ACTION[i, a] = r`A \rightarrow \alpha'$, for each terminal $a \in FOLLOW(A)$.
   - if $[S' \rightarrow S.]$ is in I_i, set $ACTION[i, \$] =$accept.

     if any conflciting actions result from these rules, the grammar is not $SLR(1)$.
3. For variables A, if $GOTO(I_i, A) = I_j$, set $GOTO[i, A] = j$.
4. Entries not defined by above rules are errors.

Construction of $SLR(1)$ parsing table

Input - Augmented grammar G'
Output - SLR parsing table with functions $ACTION$ and $GOTO$

1. Construct $C = \{I_0, I_1, \ldots, I_n\}$, the collection of $LR(0)$ items. State i is constructed from I_i
2. $ACTION$ for state i
   - if $[A \to \alpha.a\beta]$ is in I_i, and $GOTO(I_i, a) = I_j$, set $ACTION[i, a] = sj$, for terminal a.
   - if $[A \to \alpha.]$ is in I_i, set $ACTION[i, a] = r`A \to \alpha'$, for each terminal $a \in FOLLOW(A)$.
   - if $[S' \to S.]$ is in I_i, set $ACTION[i, \$] =$accept.

   if any conflciting actions result from these rules, the grammar is not $SLR(1)$.
3. For variables A, if $GOTO(I_i, A) = I_j$, set $GOTO[i, A] = j$.
4. Entries not defined by above rules are errors.
5. The initial state of the parser is constructed from the set of items containing $[S' \to .S]$.

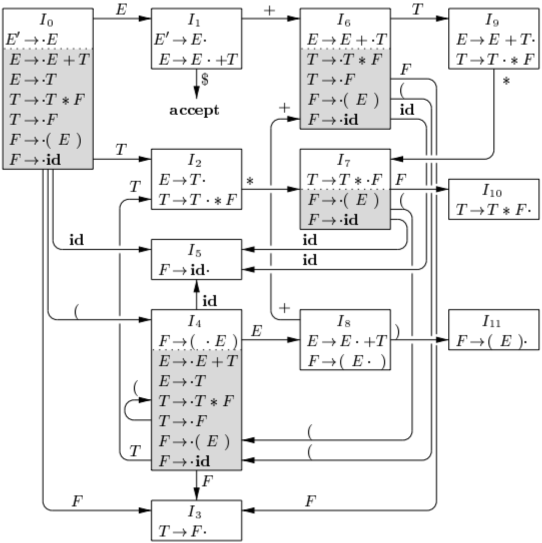
Example

(0) $E' \to E$ (4) $T \to F$
(1) $E \to E + T$ (5)$F \to (E)$
(2) $E \to T$ (6) $F \to id$

(3) $T \to T * F$

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

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



STATE	ACTIONS						GOTO		
	id	$+$	*	(	)	$	E	T	F

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

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

STATE	ACTIONS						GOTO		
	id	+	*	(	)	$	E	T	F
0	s5			s4			1	2	3
1		s6				acc			
2		r2	s7		r2	r2			

STATE	ACTIONS						GOTO		
	id	+	*	(	)	$	E	T	F
0	s5			s4			1	2	3
1		s6				acc			
2		r2	s7		r2	r2			
3		r4	r4		r4	r4			

STATE	ACTIONS						GOTO		
	id	+	*	(	)	\$	E	T	F
0	s5			s4			1	2	3
1		s6				acc			
2		r2	s7		r2	r2			
3		r4	r4		r4	r4			
4	s5			s4			8	2	3

STATE	ACTIONS						GOTO		
	id	$+$	$*$	(	)	\$	E	T	F
0	s5			s4			1	2	3
1		s6				acc			
2		r2	s7		r2	r2			
3		r4	r4		r4	r4			
4	s5			s4			8	2	3
5		r6	r6		r6	r6			

STATE	ACTIONS						GOTO		
	id	+	*	(	)	$	E	T	F
0	s5			s4			1	2	3
1		s6				acc			
2		r2	s7		r2	r2			
3		r4	r4		r4	r4			
4	s5			s4			8	2	3
5		r6	r6		r6	r6			
6	s5			s4				9	3

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

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

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

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

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

Moves of the SLR parser on $id * id + id$

	STACK	SYMBOLS	INPUT	ACTION
(1)	0		$\mathbf{id} * \mathbf{id} + \mathbf{id}\,\$$	shift (s5)
(2)	0 5	$\mathbf{id}$	$* \mathbf{id} + \mathbf{id}\,\$$	reduce by $F \to \mathbf{id}$ (r6)
(3)	0 3	F	$* \mathbf{id} + \mathbf{id}\,\$$	reduce by $T \to F$ (r4)
(4)	0 2	T	$* \mathbf{id} + \mathbf{id}\,\$$	shift (s7)
(5)	0 2 7	$T *$	$\mathbf{id} + \mathbf{id}\,\$$	shift (s5)
(6)	0 2 7 5	$T * \mathbf{id}$	$+ \mathbf{id}\,\$$	reduce by $F \to \mathbf{id}$ (r6)
(7)	0 2 7 10	$T * F$	$+ \mathbf{id}\,\$$	reduce by $T \to T * F$ (r3)
(8)	0 2	T	$+ \mathbf{id}\,\$$	reduce by $E \to T$ (r2)
(9)	0 1	E	$+ \mathbf{id}\,\$$	shift (s6)
(10)	0 1 6	$E +$	$\mathbf{id}\,\$$	shift (s5)
(11)	0 1 6 5	$E + \mathbf{id}$	$\$$	reduce by $F \to \mathbf{id}$ (r6)
(12)	0 1 6 3	$E + F$	$\$$	reduce by $T \to F$ (r4)
(13)	0 1 6 9	$E + T$	$\$$	reduce by $E \to E + T$ (r1)
(14)	0 1	E	$\$$	accept

Every SLR(1) grammar is unambiguous; there are unambiguous grammar that are not SLR(1)

Every SLR(1) grammar is unambiguous; there are unambiguous grammar that are not SLR(1)

(1) $S \rightarrow L = R$ (2)$S \rightarrow R$
(3) $L \rightarrow *R$ (4) $L \rightarrow id$
(5) $R \rightarrow L$

Every SLR(1) grammar is unambiguous; there are unambiguous grammar that are not SLR(1)

(1) $S \rightarrow L = R$ (2)$S \rightarrow R$
(3) $L \rightarrow *R$ (4) $L \rightarrow id$
(5) $R \rightarrow L$

$FIRST(L) = \{*, id\}$,
$FIRST(S) = FIRST(R) = FIRST(L)$
$FOLLOW(S') = \{\$\}$, $FOLLOW(S) = \{\$\}$,

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

Every SLR(1) grammar is unambiguous; there are unambiguous grammar that are not SLR(1)

(1) $S \to L = R$ (2) $S \to R$
(3) $L \to *R$ (4) $L \to id$
(5) $R \to L$

$FIRST(L) = \{*, id\}$,
$FIRST(S) = FIRST(R) = FIRST(L)$
$FOLLOW(S') = \{\$\}$, $FOLLOW(S) = \{\$\}$,

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

$$
\begin{array}{lll}
I_0: & S' \to \cdot S \\
 & S \to \cdot L = R \\
 & S \to \cdot R \\
 & L \to \cdot * R \\
 & L \to \cdot \mathbf{id} \\
 & R \to \cdot L \\
 \\
I_1: & S' \to S\cdot \\
 \\
I_2: & S \to L\cdot = R \\
 & R \to L\cdot \\
 \\
I_3: & S \to R\cdot \\
 \\
I_4: & L \to *\cdot R \\
 & R \to \cdot L \\
 & L \to \cdot * R \\
 & L \to \cdot \mathbf{id}
\end{array}
\qquad
\begin{array}{ll}
I_5: & L \to \mathbf{id}\cdot \\
 \\
I_6: & S \to L = \cdot R \\
 & R \to \cdot L \\
 & L \to \cdot * R \\
 & L \to \cdot \mathbf{id} \\
 \\
I_7: & L \to *R\cdot \\
 \\
I_8: & R \to L\cdot \\
 \\
I_9: & S \to L = R\cdot
\end{array}
$$

$I_2 \xrightarrow{=} I_6$

Show that the following grammar is $LL(1)$, but not $SLR(1)$

$S \to AaAb | BbBa$, $A \to \epsilon$, $B \to \epsilon$

Show that the following grammar is $SLR(1)$, but not $LL(1)$

$S \to SA | A$, $A \to a$

Show that the following grammar is $LL(1)$, but not $SLR(1)$

$S \rightarrow AaAb|BbBa$, $A \rightarrow \epsilon$, $B \rightarrow \epsilon$

Show that the following grammar is $SLR(1)$, but not $LL(1)$

$S \rightarrow SA|A$, $A \rightarrow a$

Neither $LL(1)$ is proper subset of $SLR(1)$ or viceversa.

LR(0) parsing table construction

Input - Augmented grammar G'
Output - LR(0) parsing table with functions *ACTION* and *GOTO*

LR(0) parsing table construction

Input - Augmented grammar G'
Output - LR(0) parsing table with functions $ACTION$ and $GOTO$

1. Construct $C = \{I_0, I_1, \ldots, I_n\}$, the collection of $LR(0)$ items. State i is constructed from I_i

LR(0) parsing table construction

Input - Augmented grammar G'
Output - LR(0) parsing table with functions $ACTION$ and $GOTO$

1. Construct $C = \{I_0, I_1, \ldots, I_n\}$, the collection of $LR(0)$ items. State i is constructed from I_i
2. $ACTION$ for state i
   - if $[A \rightarrow \alpha.a\beta]$ is in I_i, and $GOTO(I_i, a) = I_j$, set $ACTION[i, a] = sj$, for terminal a.

LR(0) parsing table construction

Input - Augmented grammar G'
Output - LR(0) parsing table with functions $ACTION$ and $GOTO$

1. Construct $C = \{I_0, I_1, \ldots, I_n\}$, the collection of $LR(0)$ items. State i is constructed from I_i
2. $ACTION$ for state i
   - if $[A \rightarrow \alpha.a\beta]$ is in I_i, and $GOTO(I_i, a) = I_j$, set $ACTION[i, a] = sj$, for terminal a.
   - if $[A \rightarrow \alpha.]$ is in I_i, set $ACTION[i, a] = r$ "$A \rightarrow \alpha$", for each terminal a.

LR(0) parsing table construction

Input - Augmented grammar G'

Output - LR(0) parsing table with functions $ACTION$ and $GOTO$

1. Construct $C = \{I_0, I_1, \ldots, I_n\}$, the collection of $LR(0)$ items. State i is constructed from I_i
2. $ACTION$ for state i
   - if $[A \to \alpha.a\beta]$ is in I_i, and $GOTO(I_i, a) = I_j$, set $ACTION[i, a] = sj$, for terminal a.
   - if $[A \to \alpha.]$ is in I_i, set $ACTION[i, a] = r\text{“}A \to \alpha\text{”}$, for each terminal a.
   - if $[S' \to S]$ is in I_i, set $ACTION[i, \$]$ =accept.

     if any conflciting actions result from these rules, the grammar is not $LR(0)$.

LR(0) parsing table construction

Input - Augmented grammar G'
Output - LR(0) parsing table with functions $ACTION$ and $GOTO$

1. Construct $C = \{I_0, I_1, \ldots, I_n\}$, the collection of $LR(0)$ items. State i is constructed from I_i
2. $ACTION$ for state i
   - if $[A \rightarrow \alpha.a\beta]$ is in I_i, and $GOTO(I_i, a) = I_j$, set $ACTION[i, a] = sj$, for terminal a.
   - if $[A \rightarrow \alpha.]$ is in I_i, set $ACTION[i, a] = r$ "$A \rightarrow \alpha$", for each terminal a.
   - if $[S' \rightarrow S]$ is in I_i, set $ACTION[i, \$] =$accept.

     if any conflciting actions result from these rules, the grammar is not $LR(0)$.
3. For variables A, if $GOTO(I_i, A) = I_j$, set $GOTO[i, A] = j$.

LR(0) parsing table construction

Input - Augmented grammar G'
Output - LR(0) parsing table with functions $ACTION$ and $GOTO$

1. Construct $C = \{I_0, I_1, \ldots, I_n\}$, the collection of $LR(0)$ items. State i is constructed from I_i
2. $ACTION$ for state i
   - if $[A \to \alpha.a\beta]$ is in I_i, and $GOTO(I_i, a) = I_j$, set $ACTION[i, a] = sj$, for terminal a.
   - if $[A \to \alpha.]$ is in I_i, set $ACTION[i, a] = r$ "$A \to \alpha$", for each terminal a.
   - if $[S' \to S]$ is in I_i, set $ACTION[i, \$] =$accept.

   if any conflciting actions result from these rules, the grammar is not $LR(0)$.
3. For variables A, if $GOTO(I_i, A) = I_j$, set $GOTO[i, A] = j$.
4. Entries not defined by above rules are errors.

LR(0) parsing table construction

Input - Augmented grammar G'
Output - LR(0) parsing table with functions $ACTION$ and $GOTO$

1. Construct $C = \{I_0, I_1, \ldots, I_n\}$, the collection of $LR(0)$ items. State i is constructed from I_i
2. $ACTION$ for state i
   - if $[A \rightarrow \alpha.a\beta]$ is in I_i, and $GOTO(I_i, a) = I_j$, set $ACTION[i, a] = sj$, for terminal a.
   - if $[A \rightarrow \alpha.]$ is in I_i, set $ACTION[i, a] = r$"$A \rightarrow \alpha$", for each terminal a.
   - if $[S' \rightarrow S]$ is in I_i, set $ACTION[i, \$]$ =accept.

     if any conflciting actions result from these rules, the grammar is not $LR(0)$.
3. For variables A, if $GOTO(I_i, A) = I_j$, set $GOTO[i, A] = j$.
4. Entries not defined by above rules are errors.
5. The initial state of the parser is constructed from the set of items containing $[S' \rightarrow S]$.

Viable prefixes

$$E \overset{*}{\underset{rm}{\Longrightarrow}} F * id \Longrightarrow (E) * id$$

Viable prefixes

$$E \underset{rm}{\overset{*}{\Longrightarrow}} F * id \Longrightarrow (E) * id$$

- Stack will contain (, $(E$, (E), but never $(E)*$ - (E) is a handle.

Viable prefixes

$$E \xRightarrow[rm]{*} F * id \Longrightarrow (E) * id$$

- Stack will contain (, $(E$, (E), but never $(E)*$ - (E) is a handle.
- The prefixes of the right sentential form that may appear at the top of the stack are Viable prefixes.

Viable prefixes

$$E \overset{*}{\underset{rm}{\Longrightarrow}} F * id \Longrightarrow (E) * id$$

- Stack will contain (, $(E$, (E), but never $(E)*$ - (E) is a handle.
- The prefixes of the right sentential form that may appear at the top of the stack are Viable prefixes.
- An item $A \rightarrow \beta_1.\beta_2$ is valid for a viable prefix $\alpha\beta_1$ if there exists a derivation $S \overset{*}{\underset{rm}{\Longrightarrow}} \alpha A w \rightarrow \alpha\beta_1\beta_2 w$.

Viable prefixes

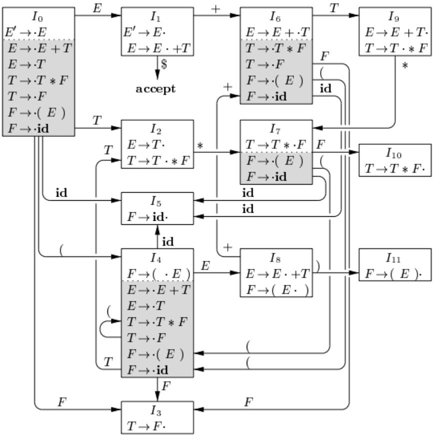
$$E \underset{rm}{\overset{*}{\Longrightarrow}} F * id \Longrightarrow (E) * id$$

- Stack will contain (, $(E$, (E), but never $(E)*$ - (E) is a handle.
- The prefixes of the right sentential form that may appear at the top of the stack are Viable prefixes.
- An item $A \rightarrow \beta_1.\beta_2$ is valid for a viable prefix $\alpha\beta_1$ if there exists a derivation
  $S \underset{rm}{\overset{*}{\Longrightarrow}} \alpha A w \rightarrow \alpha\beta_1\beta_2 w$.
  1. if $\beta_2 \neq \epsilon$, it tells the parser that the handle is not at the top: so the next move is a shift.

Viable prefixes

$$E \xRightarrow[rm]{*} F * id \Longrightarrow (E) * id$$

- Stack will contain (, $(E$, (E), but never $(E)*$ - (E) is a handle.
- The prefixes of the right sentential form that may appear at the top of the stack are Viable prefixes.
- An item $A \to \beta_1.\beta_2$ is valid for a viable prefix $\alpha\beta_1$ if there exists a derivation
  $S \xRightarrow[rm]{*} \alpha A w \to \alpha\beta_1\beta_2 w$.
  1. if $\beta_2 \neq \epsilon$, it tells the parser that the handle is not at the top: so the next move is a shift.
  2. if $\beta_2 = \epsilon$, then we have the handle at the top and its time to reduce.



In the grammar above, $E + T*$ is a viable prefix and the parser is at state 7 after reading $E + T*$. 7 contains items $T \to T * F$, $F \to (E)$, $F \to id$, which are valid items for $E + T*$.

$$
\begin{aligned}
E' &\underset{rm}{\rightarrow} E \\
&\underset{rm}{\rightarrow} E + T \\
&\underset{rm}{\rightarrow} E + T * F
\end{aligned}
$$

$$
\begin{aligned}
E' &\underset{rm}{\rightarrow} E \\
&\underset{rm}{\rightarrow} E + T \\
&\underset{rm}{\rightarrow} E + T * F \\
&\underset{rm}{\rightarrow} E + T * (E)
\end{aligned}
$$

$$
\begin{aligned}
E' &\underset{rm}{\rightarrow} E \\
&\underset{rm}{\rightarrow} E + T \\
&\underset{rm}{\rightarrow} E + T * F \\
&\underset{rm}{\rightarrow} E + T * id
\end{aligned}
$$

Problems with $SLR(1)$ parser

$$
\begin{array}{lll@{\qquad}lll}
I_0: & S' \to \cdot S & & I_5: & L \to \mathbf{id}\cdot \\
 & S \to \cdot L = R & & & \\
 & S \to \cdot R & & I_6: & S \to L = \cdot R \\
 & L \to \cdot * R & & & R \to \cdot L \\
 & L \to \cdot\mathbf{id} & & & L \to \cdot * R \\
 & R \to \cdot L & & & L \to \cdot\mathbf{id} \\
 & & & & \\
I_1: & S' \to S\cdot & & I_7: & L \to *R\cdot \\
 & & & & \\
I_2: & S \to L\cdot = R & & I_8: & R \to L\cdot \\
 & R \to L\cdot & & & \\
 & & & I_9: & S \to L = R\cdot \\
I_3: & S \to R\cdot & & & \\
 & & & & \\
I_4: & L \to *\cdot R & & & \\
 & R \to \cdot L & & & \\
 & L \to \cdot * R & & & \\
 & L \to \cdot\mathbf{id} & & &
\end{array}
$$

SLR parser at state 2 performs reduce "$R \to L$" on input $=$, as well as shift I_6 to $S \to L = .R$.

Problems with $SLR(1)$ parser

$$
\begin{array}{ll}
I_0: & S' \to \cdot S \\
 & S \to \cdot L = R \\
 & S \to \cdot R \\
 & L \to \cdot * R \\
 & L \to \cdot \mathbf{id} \\
 & R \to \cdot L \\
 & \\
I_1: & S' \to S\cdot \\
 & \\
I_2: & S \to L \cdot = R \\
 & R \to L\cdot \\
 & \\
I_3: & S \to R\cdot \\
 & \\
I_4: & L \to * \cdot R \\
 & R \to \cdot L \\
 & L \to \cdot * R \\
 & L \to \cdot \mathbf{id}
\end{array}
\qquad
\begin{array}{ll}
I_5: & L \to \mathbf{id}\cdot \\
 & \\
I_6: & S \to L = \cdot R \\
 & R \to \cdot L \\
 & L \to \cdot * R \\
 & L \to \cdot \mathbf{id} \\
 & \\
I_7: & L \to * R\cdot \\
 & \\
I_8: & R \to L\cdot \\
 & \\
I_9: & S \to L = R\cdot
\end{array}
$$

SLR parser at state 2 performs reduce "$R \to L$" on input $=$, as well as shift I_6 to $S \to L = .R$.
However, no sentential form of the grammar begins with $R = \cdots$.

Problems with $SLR(1)$ parser

I_0: $S' \to \cdot S$
$S \to \cdot L = R$
$S \to \cdot R$
$L \to \cdot * R$
$L \to \cdot \mathbf{id}$
$R \to \cdot L$

I_1: $S' \to S \cdot$

I_2: $S \to L \cdot = R$
$R \to L \cdot$

I_3: $S \to R \cdot$

I_4: $L \to * \cdot R$
$R \to \cdot L$
$L \to \cdot * R$
$L \to \cdot \mathbf{id}$

I_5: $L \to \mathbf{id} \cdot$

I_6: $S \to L = \cdot R$
$R \to \cdot L$
$L \to \cdot * R$
$L \to \cdot \mathbf{id}$

I_7: $L \to * R \cdot$

I_8: $R \to L \cdot$

I_9: $S \to L = R \cdot$

=

SLR parser at state 2 performs reduce "$R \to L$" on input $=$, as well as shift I_6 to $S \to L = .R$.
However, no sentential form of the grammar begins with $R = \cdots$.
Thus, state 2 corresponding to viable prefix L should not call for reduce "$R \to L$" on input $=$.

LR(1) items

Items are of the form $[A \rightarrow \alpha.\beta, a]$, where $A \rightarrow \alpha\beta$ is a production and a is a terminal or the end marker \$.

LR(1) items

Items are of the form $[A \to \alpha.\beta, a]$, where $A \to \alpha\beta$ is a production and a is a terminal or the end marker \$.
a is the lookahead.

LR(1) items

Items are of the form $[A \to \alpha.\beta, a]$, where $A \to \alpha\beta$ is a production and a is a terminal or the end marker \$.
a is the lookahead.
The look ahead has no effect on $[A \to \alpha.\beta, a]$ if $\beta \neq \epsilon$.

LR(1) items

Items are of the form $[A \rightarrow \alpha.\beta, a]$, where $A \rightarrow \alpha\beta$ is a production and a is a terminal or the end marker \$.
a is the lookahead.
The look ahead has no effect on $[A \rightarrow \alpha.\beta, a]$ if $\beta \neq \epsilon$.
An item $[A \rightarrow \alpha., a]$ calls for reduction only if the next input symbol is a.
The set of such a's are a subset of $FOLLOW(A)$.

An item $[A \rightarrow \alpha.\beta, a]$ is *valid* for a viable prefix γ if there is a derivation $S \overset{*}{\underset{rm}{\rightarrow}} \delta A w \underset{rm}{\rightarrow} \delta\alpha\beta w$, where

An item $[A \to \alpha.\beta, a]$ is *valid* for a viable prefix γ if there is a derivation $S \underset{rm}{\overset{*}{\to}} \delta A w \underset{rm}{\to} \delta\alpha\beta w$, where

1. $\gamma = \delta\alpha$;

An item $[A \to \alpha.\beta, a]$ is *valid* for a viable prefix γ if there is a derivation $S \underset{rm}{\overset{*}{\to}} \delta A w \underset{rm}{\to} \delta\alpha\beta w$, where

1. $\gamma = \delta\alpha$;
2. either a is the first symbol of w or, w is ϵ and $a = \$$.

Example 1.18

$$S \to BB$$
$$B \to aB|b.$$

$S \underset{rm}{\overset{*}{\to}} aaBab \underset{rm}{\to} aaaBab$

An item $[A \to \alpha.\beta, a]$ is *valid* for a viable prefix γ if there is a derivation $S \underset{rm}{\overset{*}{\to}} \delta A w \underset{rm}{\to} \delta\alpha\beta w$, where

1. $\gamma = \delta\alpha$;
2. either a is the first symbol of w or, w is ϵ and $a = \$$.

Example 1.18

$$S \to BB$$
$$B \to aB|b.$$

$S \underset{rm}{\overset{*}{\to}} aaBab \underset{rm}{\to} aaaBab$
$[B \to a.B, a]$ is valid for viable prefix $\gamma = aaa$ with

An item $[A \to \alpha.\beta, a]$ is *valid* for a viable prefix γ if there is a derivation $S \overset{*}{\underset{rm}{\to}} \delta A w \underset{rm}{\to} \delta\alpha\beta w$, where

1. $\gamma = \delta\alpha$;
2. either a is the first symbol of w or, w is ϵ and $a = \$$.

Example 1.18

$$S \to BB$$
$$B \to aB|b.$$

$S \overset{*}{\underset{rm}{\to}} aaBab \underset{rm}{\to} aaaBab$
$[B \to a.B, a]$ is valid for viable prefix $\gamma = aaa$ with

- $\delta = aa$

An item $[A \to \alpha.\beta, a]$ is *valid* for a viable prefix γ if there is a derivation $S \underset{rm}{\overset{*}{\to}} \delta A w \underset{rm}{\to} \delta\alpha\beta w$, where

1. $\gamma = \delta\alpha$;
2. either a is the first symbol of w or, w is ϵ and $a = \$$.

Example 1.18

$$S \to BB$$
$$B \to aB|b.$$

$S \underset{rm}{\overset{*}{\to}} aaBab \underset{rm}{\to} aaaBab$
$[B \to a.B, a]$ is valid for viable prefix $\gamma = aaa$ with

- $\delta = aa$
- $\alpha = a$

An item $[A \to \alpha.\beta, a]$ is *valid* for a viable prefix γ if there is a derivation $S \overset{*}{\underset{rm}{\to}} \delta A w \underset{rm}{\to} \delta\alpha\beta w$, where

1. $\gamma = \delta\alpha$;
2. either a is the first symbol of w or, w is ϵ and $a = \$$.

Example 1.18

$$S \to BB$$
$$B \to aB|b.$$

$S \overset{*}{\underset{rm}{\to}} aaBab \underset{rm}{\to} aaaBab$

$[B \to a.B, a]$ is valid for viable prefix $\gamma = aaa$ with

- $\delta = aa$
- $\alpha = a$
- $\beta = B$

An item $[A \to \alpha.\beta, a]$ is *valid* for a viable prefix γ if there is a derivation $S \underset{rm}{\overset{*}{\to}} \delta A w \underset{rm}{\to} \delta\alpha\beta w$, where

1. $\gamma = \delta\alpha$;
2. either a is the first symbol of w or, w is ϵ and $a = \$$.

Example 1.18

$$S \to BB$$
$$B \to aB|b.$$

$S \underset{rm}{\overset{*}{\to}} aaBab \underset{rm}{\to} aaaBab$
$[B \to a.B, a]$ is valid for viable prefix $\gamma = aaa$ with

- $\delta = aa$
- $\alpha = a$
- $\beta = B$
- $A = aB$

An item $[A \to \alpha.\beta, a]$ is *valid* for a viable prefix γ if there is a derivation $S \xrightarrow[rm]{*} \delta A w \xrightarrow[rm]{} \delta\alpha\beta w$, where

1. $\gamma = \delta\alpha$;
2. either a is the first symbol of w or, w is ϵ and $a = \$$.

Example 1.18

$$S \to BB$$
$$B \to aB|b.$$

$S \xrightarrow[rm]{*} aaBab \xrightarrow[rm]{} aaaBab$
$[B \to a.B, a]$ is valid for viable prefix $\gamma = aaa$ with

- $\delta = aa$
- $\alpha = a$
- $\beta = B$
- $A = aB$
- $w = ab$.

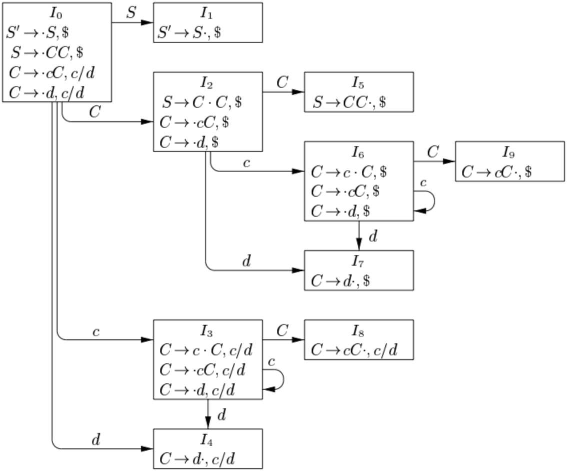
Constructing $LR(1)$ set of items

```
SetOfItems CLOSURE(I) {
    repeat
        for ( each item [A → α·Bβ, a] in I )
            for ( each production B → γ in G' )
                for ( each terminal b in FIRST(βa) )
                    add [B → ·γ, b] to set I;
    until no more items are added to I;
    return I;
}

SetOfItems GOTO(I, X) {
    initialize J to be the empty set;
    for ( each item [A → α·Xβ, a] in I )
        add item [A → αX·β, a] to set J;
    return CLOSURE(J);
}

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

$$(0)\ S' \rightarrow S$$
$$(1)\ S \rightarrow CC$$
$$(2)\ S \rightarrow cC$$
$$(3)\ C \rightarrow d$$



Construction of Cannonical $LR(1)$ parsing table

Input - Augmented grammar G'
Output - CLR(1) parsing table with functions $ACTION$ and $GOTO$

Construction of Cannonical $LR(1)$ parsing table

Input - Augmented grammar G'
Output - CLR(1) parsing table with functions $ACTION$ and $GOTO$

1. Construct $C = \{I_0, I_1, \ldots, I_n\}$, the collection of $LR(1)$ items. State i is constructed from I_i

Construction of Cannonical $LR(1)$ parsing table

Input - Augmented grammar G'
Output - CLR(1) parsing table with functions $ACTION$ and $GOTO$

1. Construct $C = \{I_0, I_1, \ldots, I_n\}$, the collection of $LR(1)$ items. State i is constructed from I_i
2. $ACTION$ for state i
   - if $[A \rightarrow \alpha.a\beta, b]$ is in I_i, and $GOTO(I_i, a) = I_j$, set $ACTION[i, a] = sj$, for terminal a.

Construction of Cannonical $LR(1)$ parsing table

Input - Augmented grammar G'
Output - CLR(1) parsing table with functions $ACTION$ and $GOTO$

1. Construct $C = \{I_0, I_1, \ldots, I_n\}$, the collection of $LR(1)$ items. State i is constructed from I_i
2. $ACTION$ for state i
   - if $[A \rightarrow \alpha.a\beta, b]$ is in I_i, and $GOTO(I_i, a) = I_j$, set $ACTION[i, a] = sj$, for terminal a.
   - if $[A \rightarrow \alpha., a]$ is in I_i, set $ACTION[i, a] = r`A \rightarrow \alpha'$.

Construction of Cannonical $LR(1)$ parsing table

Input - Augmented grammar G'
Output - CLR(1) parsing table with functions $ACTION$ and $GOTO$

1. Construct $C = \{I_0, I_1, \ldots, I_n\}$, the collection of $LR(1)$ items. State i is constructed from I_i
2. $ACTION$ for state i
   - if $[A \rightarrow \alpha.a\beta, b]$ is in I_i, and $GOTO(I_i, a) = I_j$, set $ACTION[i, a] = sj$, for terminal a.
   - if $[A \rightarrow \alpha., a]$ is in I_i, set $ACTION[i, a] = r`A \rightarrow \alpha'$.
   - if $[S' \rightarrow S., \$]$ is in I_i, set $ACTION[i, \$] =$accept.
     if any conflciting actions result from these rules, the grammar is not $SLR(1)$.

Construction of Cannonical $LR(1)$ parsing table

Input - Augmented grammar G'
Output - CLR(1) parsing table with functions $ACTION$ and $GOTO$

1. Construct $C = \{I_0, I_1, \ldots, I_n\}$, the collection of $LR(1)$ items. State i is constructed from I_i
2. $ACTION$ for state i
   - if $[A \rightarrow \alpha.a\beta, b]$ is in I_i, and $GOTO(I_i, a) = I_j$, set $ACTION[i, a] = sj$, for terminal a.
   - if $[A \rightarrow \alpha., a]$ is in I_i, set $ACTION[i, a] = r`A \rightarrow \alpha'$.
   - if $[S' \rightarrow S., \$]$ is in I_i, set $ACTION[i, \$] =$accept.
     if any conflciting actions result from these rules, the grammar is not $SLR(1)$.
3. For variables A, if $GOTO(I_i, A) = I_j$, set $GOTO[i, A] = j$.

Construction of Cannonical $LR(1)$ parsing table

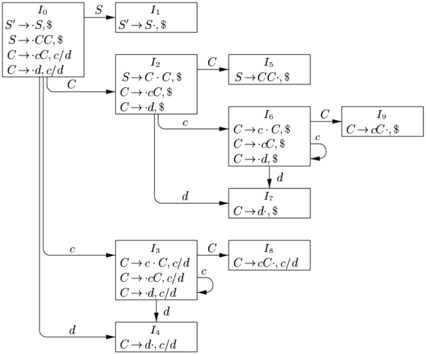
Input - Augmented grammar G'
Output - CLR(1) parsing table with functions $ACTION$ and $GOTO$

1. Construct $C = \{I_0, I_1, \ldots, I_n\}$, the collection of $LR(1)$ items. State i is constructed from I_i
2. $ACTION$ for state i
   - if $[A \rightarrow \alpha.a\beta, b]$ is in I_i, and $GOTO(I_i, a) = I_j$, set $ACTION[i, a] = sj$, for terminal a.
   - if $[A \rightarrow \alpha., a]$ is in I_i, set $ACTION[i, a] = r`A \rightarrow \alpha'$.
   - if $[S' \rightarrow S., \$]$ is in I_i, set $ACTION[i, \$] =$accept.
     if any conflciting actions result from these rules, the grammar is not $SLR(1)$.
3. For variables A, if $GOTO(I_i, A) = I_j$, set $GOTO[i, A] = j$.
4. Entries not defined by above rules are errors.

Construction of Cannonical $LR(1)$ parsing table

Input - Augmented grammar G'
Output - CLR(1) parsing table with functions $ACTION$ and $GOTO$

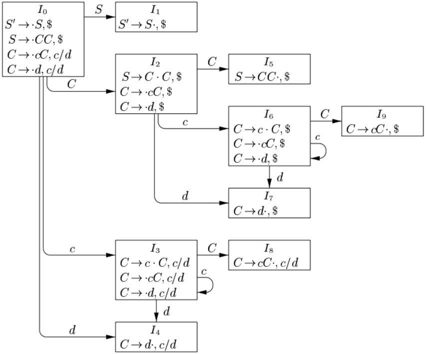
1. Construct $C = \{I_0, I_1, \ldots, I_n\}$, the collection of $LR(1)$ items. State i is constructed from I_i
2. $ACTION$ for state i
   - if $[A \rightarrow \alpha.a\beta, b]$ is in I_i, and $GOTO(I_i, a) = I_j$, set $ACTION[i, a] = sj$, for terminal a.
   - if $[A \rightarrow \alpha., a]$ is in I_i, set $ACTION[i, a] = r`A \rightarrow \alpha'$.
   - if $[S' \rightarrow S., \$]$ is in I_i, set $ACTION[i, \$] =$accept.
     if any conflciting actions result from these rules, the grammar is not $SLR(1)$.
3. For variables A, if $GOTO(I_i, A) = I_j$, set $GOTO[i, A] = j$.
4. Entries not defined by above rules are errors.
5. The initial state of the parser is constructed from the set of items containing $[S' \rightarrow S]$.



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

LALR parser

The problem with $CLR(1)$ parsers is that the number of states is huge.
$LALR(1)$ parsers have the same number of states as $SLR(1)$

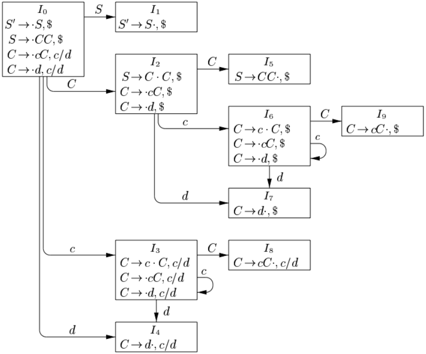


Difference between I_4 and I_7: Consider the string $c * dc * d$.

LALR parser

The problem with $CLR(1)$ parsers is that the number of states is huge.

$LALR(1)$ parsers have the same number of states as $SLR(1)$

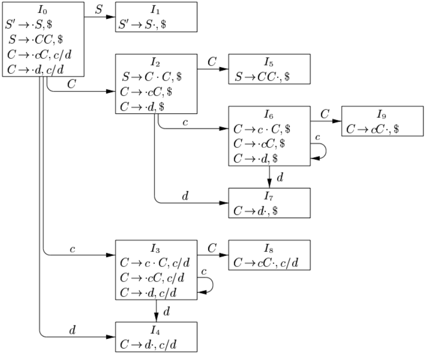


Difference between I_4 and I_7: Consider the string $c * dc * d$.

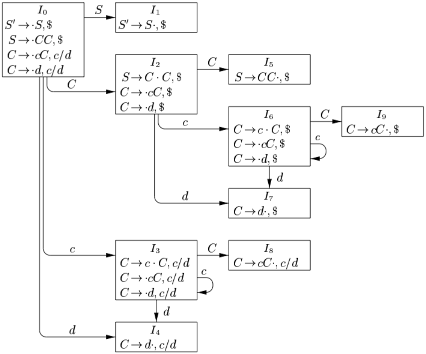
After seeing $c * d$, the parser enters the configuration $033\ldots34$.

LALR parser

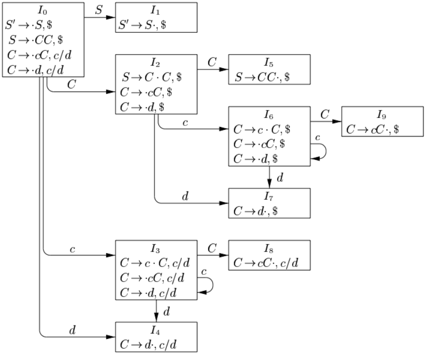
The problem with $CLR(1)$ parsers is that the number of states is huge.
$LALR(1)$ parsers have the same number of states as $SLR(1)$



Difference between I_4 and I_7: Consider the string $c * dc * d$.
After seeing $c * d$, the parser enters the configuration $033 \ldots 34$.
On seeing a \$ next, the parser declares error

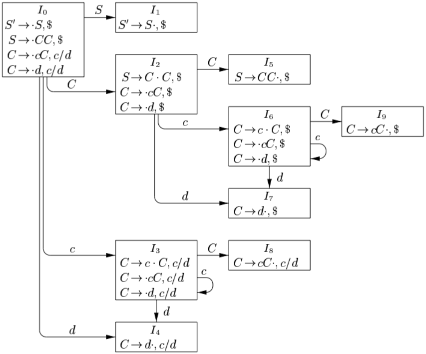


On seeing the next c, the parser reduces $[C \to d., c]$, pops 4 and pushes 8 to enter $033\ldots38$



On seeing the next c, the parser reduces $[C \to d., c]$, pops 4 and pushes 8 to enter $033\ldots38$

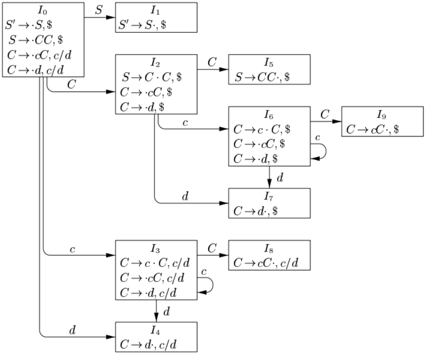
At 8, it reduces $[C \to cC., c]$ until all 3's are popped out: then it enters 02.



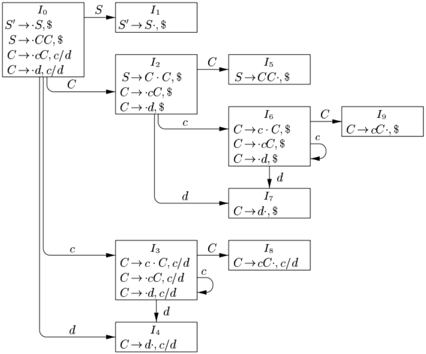
On seeing the next c, the parser reduces $[C \to d., c]$, pops 4 and pushes 8 to enter $033\ldots38$

At 8, it reduces $[C \to cC., c]$ until all 3's are popped out: then it enters 02.

It enters $0266\ldots67$ on seeing the next $c * d$.



On seeing the next c, the parser reduces $[C \to d., c]$, pops 4 and pushes 8 to enter $033\ldots38$
At 8, it reduces $[C \to cC., c]$ until all 3's are popped out: then it enters 02.
It enters $0266\ldots67$ on seeing the next $c * d$.
On seeing \$, it reduces $[C \to d., \$]$, enters $0266\ldots69$.



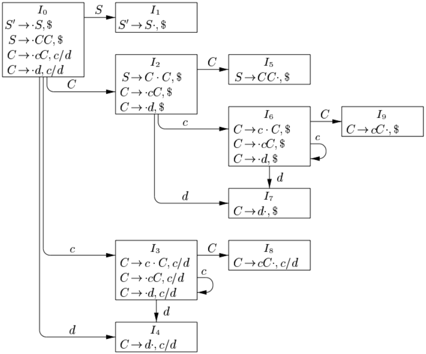
On seeing the next c, the parser reduces $[C \to d., c]$, pops 4 and pushes 8 to enter $033\ldots38$

At 8, it reduces $[C \to cC., c]$ until all 3's are popped out: then it enters 02.

It enters $0266\ldots67$ on seeing the next $c * d$.

On seeing \$, it reduces $[C \to d., \$]$, enters $0266\ldots69$.

At 9, it reduces $[C \to cC., \$]$ until all 6's are popped out: then it enters 025.



On seeing the next c, the parser reduces $[C \to d., c]$, pops 4 and pushes 8 to enter $033\ldots38$

At 8, it reduces $[C \to cC., c]$ until all 3's are popped out: then it enters 02.

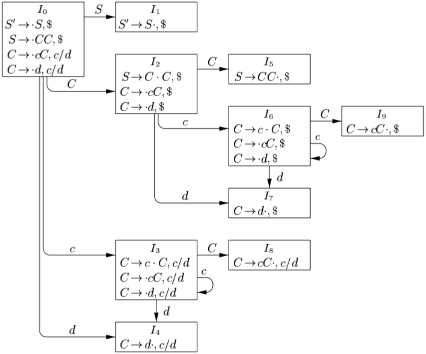
It enters $0266\ldots67$ on seeing the next $c * d$.

On seeing \$, it reduces $[C \to d., \$]$, enters $0266\ldots69$.

At 9, it reduces $[C \to cC., \$]$ until all 6's are popped out: then it enters 025.

It then reduces $[S \to CC., \$]$, pops 2 and 5, enters 01 where it accepts on \$

LALR set of items



Replace I_4 and I_7 by I_{47} - $[C \to d., c|d|\$]$
GOTO's to I_4 and I_7 non enters I_{47}.
ACTION of I_{47} is to reduce on $c, d, \$$.
On $ccd\$$ as input, the parser raises error if I_4 and I_7 are different; the parser reduces at I_{47}.

LALR set of items

Merging of states cannot introduce new shift-reduce conflict

Let $\{[A \rightarrow \alpha., a], [B \rightarrow \gamma.a\eta, b]\} \in I_{ij}$ for the new shift-reduce conflict.
Then, I_i must contain $[A \rightarrow \alpha., a]$ and $[B \rightarrow \gamma.a\eta, c]$: the shift-reduce conflict exists in I_i as well.

Core - the set of items in a state, not considering the lookaheads.

LALR parsing table construction

Input - Augmented grammar G'
Output - CLR(1) parsing table with functions $ACTION$ and $GOTO$

LALR parsing table construction

Input - Augmented grammar G'
Output - CLR(1) parsing table with functions *ACTION* and *GOTO*

1. Construct $C = \{I_0, I_1, \ldots, I_n\}$, the collection of $LR(1)$ items.

LALR parsing table construction

Input - Augmented grammar G'
Output - CLR(1) parsing table with functions *ACTION* and *GOTO*

1. Construct $C = \{I_0, I_1, \ldots, I_n\}$, the collection of $LR(1)$ items.
2. For each core, find the items with same core and merge. Let $C' = \{J_0, J_1, \ldots, J_n\}$ be the resulting set of items.

LALR parsing table construction

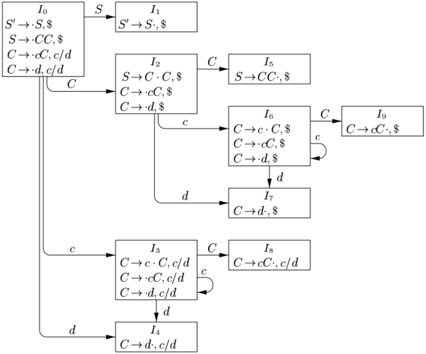
Input - Augmented grammar G'
Output - CLR(1) parsing table with functions $ACTION$ and $GOTO$

1. Construct $C = \{I_0, I_1, \ldots, I_n\}$, the collection of $LR(1)$ items.
2. For each core, find the items with same core and merge. Let $C' = \{J_0, J_1, \ldots, J_n\}$ be the resulting set of items.
3. $ACTION$ for state i: same as $CLR(1)$ parser.

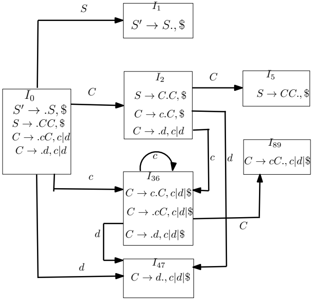
LALR parsing table construction

Input - Augmented grammar G'
Output - CLR(1) parsing table with functions $ACTION$ and $GOTO$

1. Construct $C = \{I_0, I_1, \ldots, I_n\}$, the collection of $LR(1)$ items.
2. For each core, find the items with same core and merge. Let $C' = \{J_0, J_1, \ldots, J_n\}$ be the resulting set of items.
3. $ACTION$ for state i: same as $CLR(1)$ parser.
4. If $J = \{I_1, \ldots, I_l\}$. Then $GOTO(J, X) = K$, where K has the same core as $GOTO(I_1, X)$.



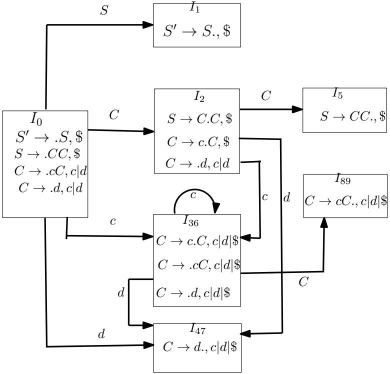
I_0
$S' \to \cdot S, \$$
$S \to \cdot CC, \$$
$C \to \cdot cC, c/d$
$C \to \cdot d, c/d$
S
I_1
$S' \to S\cdot, \$$
C
I_2
$S \to C \cdot C, \$$
$C \to \cdot cC, \$$
$C \to \cdot d, \$$
C
I_5
$S \to CC\cdot, \$$
c
I_6
$C \to c \cdot C, \$$
$C \to \cdot cC, \$$
$C \to \cdot d, \$$
C
I_9
$C \to cC\cdot, \$$
c
d
d
I_7
$C \to d\cdot, \$$
c
I_3
$C \to c \cdot C, c/d$
$C \to \cdot cC, c/d$
$C \to \cdot d, c/d$
C
I_8
$C \to cC\cdot, c/d$
c
d
d
I_4
$C \to d\cdot, c/d$



S
I_1
$S' \to S., \$$
I_0
$S' \to .S, \$$
$S \to .CC, \$$
$C \to .cC, c|d$
$C \to .d, c|d$
C
I_2
$S \to C.C, \$$
$C \to c.C, \$$
$C \to .d, c|d$
C
I_5
$S \to CC., \$$
c
d
I_{89}
$C \to cC., c|d|\$$
c
c
I_{36}
$C \to c.C, c|d|\$$
$C \to .cC, c|d|\$$
$C \to .d, c|d|\$$
C
d
d
I_{47}
$C \to d., c|d|\$$

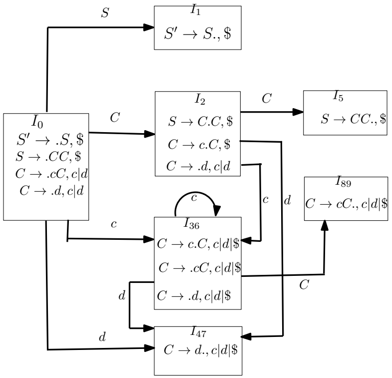
LALR parsing table

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



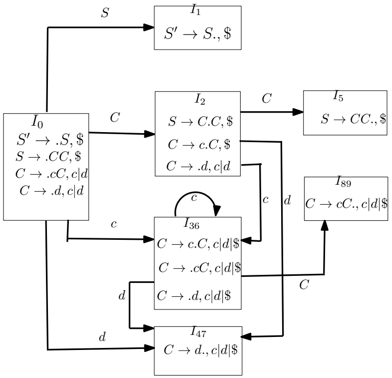
Sample input: *ccdcd*\$

0



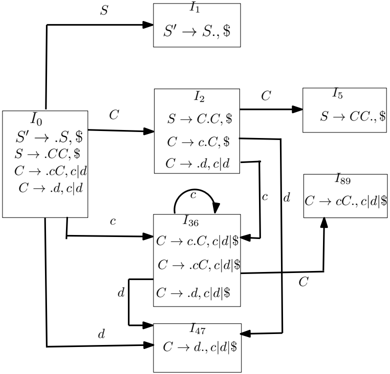
Sample input: $ccdcd\$$

$0 \xrightarrow{c} 036$



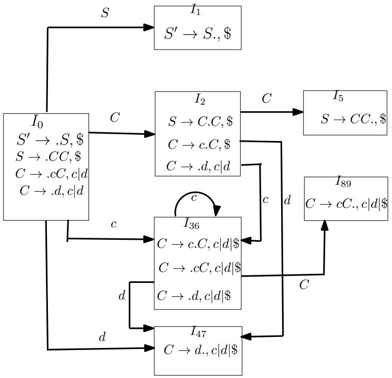
Sample input: *ccdcd*\$

$0 \xrightarrow{c} 036 \xrightarrow{c} 03636$



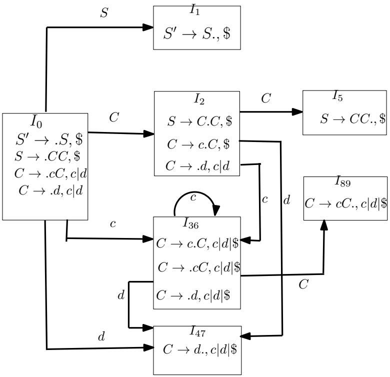
Sample input: $ccdcd\$$

$0 \xrightarrow{c} 036 \xrightarrow{c} 03636 \xrightarrow{d} 0363647$



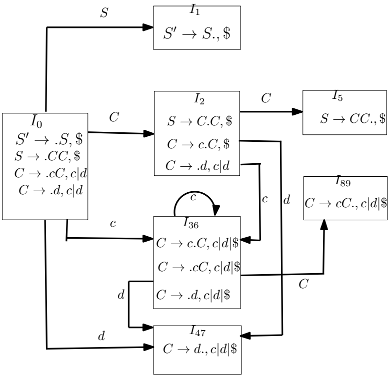
Sample input: *ccdcd*\$

$0 \xrightarrow{c} 036 \xrightarrow{c} 03636 \xrightarrow{d} 0363647 \xrightarrow{reduce C \to d} 0363689$



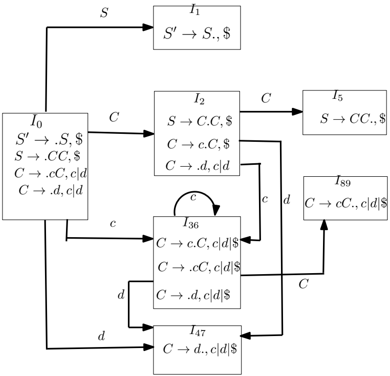
Sample input: *ccdcd*\$

$0 \xrightarrow{c} 036 \xrightarrow{c} 03636 \xrightarrow{d} 0363647 \xrightarrow{reduce C\to d} 0363689 \xrightarrow{reduce C\to cC}$
03689



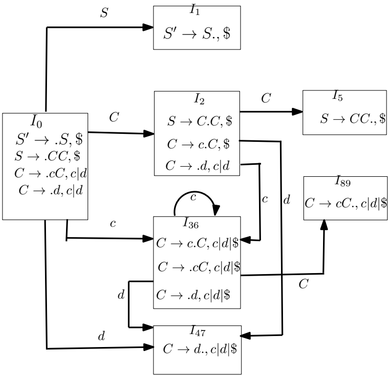
Sample input: *ccdcd*\$

$0 \xrightarrow{c} 036 \xrightarrow{c} 03636 \xrightarrow{d} 0363647 \xrightarrow{reduce C \to d} 0363689 \xrightarrow{reduce C \to cC}$
$03689 \xrightarrow{reduce C \to cC} 02$



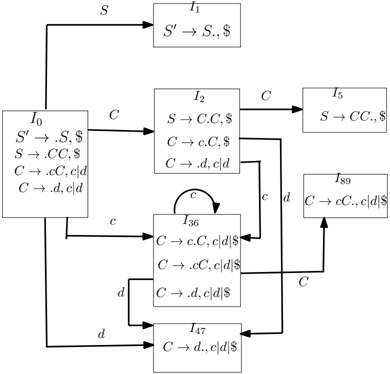
Sample input: *ccdcd*\$

$0 \xrightarrow{c} 036 \xrightarrow{c} 03636 \xrightarrow{d} 0363647 \xrightarrow{reduce C \to d} 0363689 \xrightarrow{reduce C \to cC}$
$03689 \xrightarrow{reduce C \to cC} 02 \xrightarrow{c} 0236$



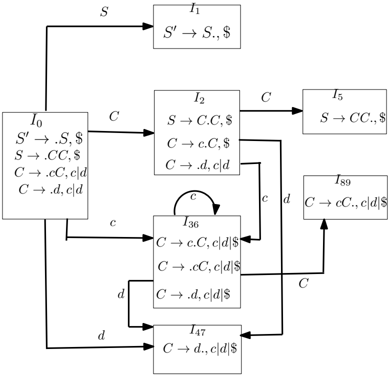
Sample input: *ccdcd*\$

$0 \xrightarrow{c} 036 \xrightarrow{c} 03636 \xrightarrow{d} 0363647 \xrightarrow{reduce C\to d} 0363689 \xrightarrow{reduce C\to cC}$
$03689 \xrightarrow{reduce C\to cC} 02 \xrightarrow{c} 0236 \xrightarrow{d} 023647$



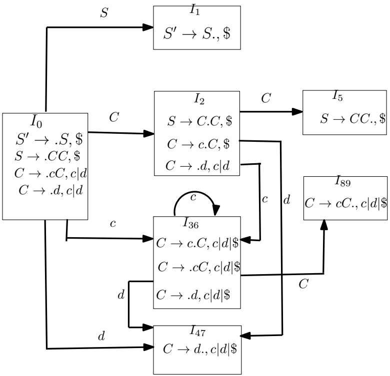
Sample input: *ccdcd*\$

$0 \xrightarrow{c} 036 \xrightarrow{c} 03636 \xrightarrow{d} 0363647 \xrightarrow{reduce C \to d} 0363689 \xrightarrow{reduce C \to cC}$
$03689 \xrightarrow{reduce C \to cC} 02 \xrightarrow{c} 0236 \xrightarrow{d} 023647 \xrightarrow{reduce C \to d}$
023689



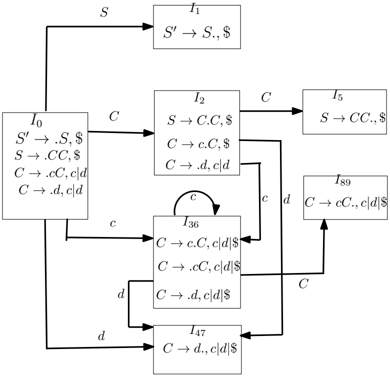
Sample input: *ccdcd*\$

$0 \xrightarrow{c} 036 \xrightarrow{c} 03636 \xrightarrow{d} 0363647 \xrightarrow{reduce C \to d} 0363689 \xrightarrow{reduce C \to cC}$
$03689 \xrightarrow{reduce C \to cC} 02 \xrightarrow{c} 0236 \xrightarrow{d} 023647 \xrightarrow{reduce C \to d}$
$023689 \xrightarrow{reduce C \to cC} 025$



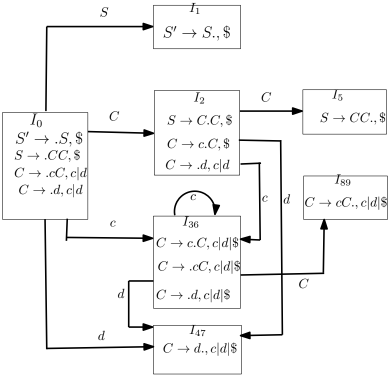
Sample input: *ccdcd*\$

$0 \xrightarrow{c} 036 \xrightarrow{c} 03636 \xrightarrow{d} 0363647 \xrightarrow{reduce C \to d} 0363689 \xrightarrow{reduce C \to cC}$
$03689 \xrightarrow{reduce C \to cC} 02 \xrightarrow{c} 0236 \xrightarrow{d} 023647 \xrightarrow{reduce C \to d}$
$023689 \xrightarrow{reduce C \to cC} 025 \xrightarrow{\$} accept.$



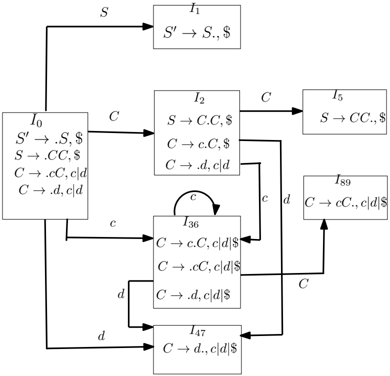
Sample input: *ccd*\$

0



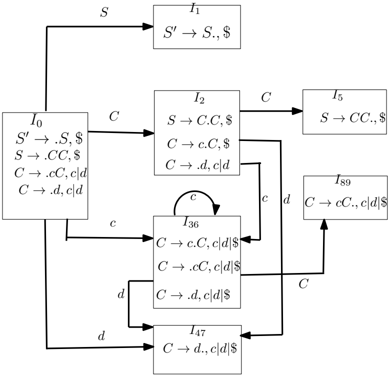
Sample input: *ccd*\$

$0 \xrightarrow{c} 036$



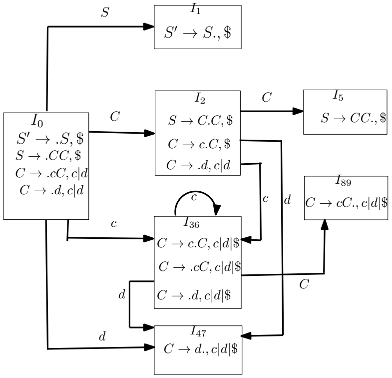
Sample input: *ccd*\$

$0 \xrightarrow{c} 036 \xrightarrow{c} 03636$



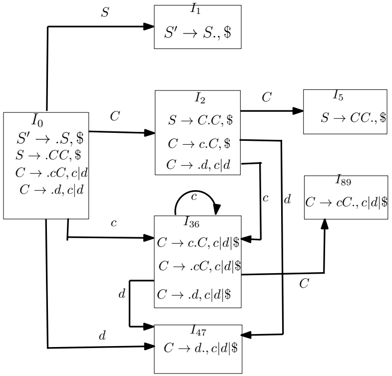
Sample input: *ccd*\$

$0 \xrightarrow{c} 036 \xrightarrow{c} 03636 \xrightarrow{d} 0363647$



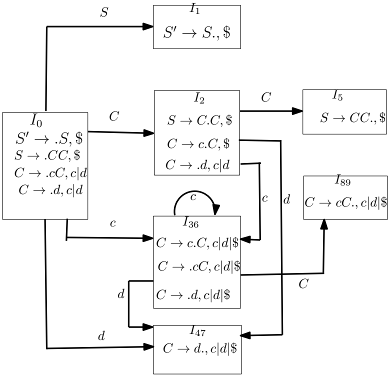
Sample input: *ccd*\$

$0 \xrightarrow{c} 036 \xrightarrow{c} 03636 \xrightarrow{d} 0363647 \xrightarrow{reduce C \to d} 0363689$



Sample input: *ccd*\$

$0 \xrightarrow{c} 036 \xrightarrow{c} 03636 \xrightarrow{d} 0363647 \xrightarrow{reduce C \to d} 0363689 \xrightarrow{reduce C \to cC}$ 03689



Sample input: *ccd*\$

$0 \xrightarrow{c} 036 \xrightarrow{c} 03636 \xrightarrow{d} 0363647 \xrightarrow{reduce C \to d} 0363689 \xrightarrow{reduce C \to cC}$
$03689 \xrightarrow{reduce C \to cC} 02 \xrightarrow{\$} error.$

Parser generator: Bison

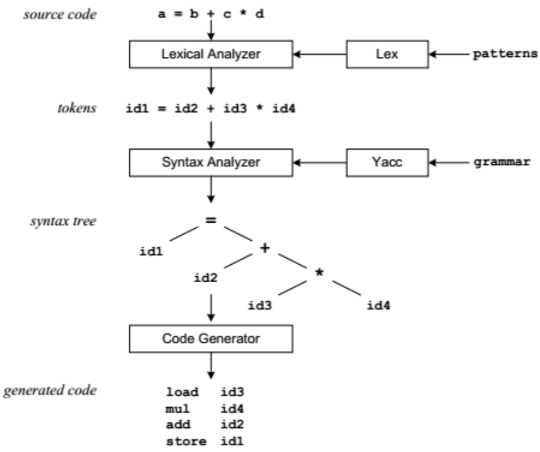


Figure 1: Compilation Sequence

Parser generator: Bison

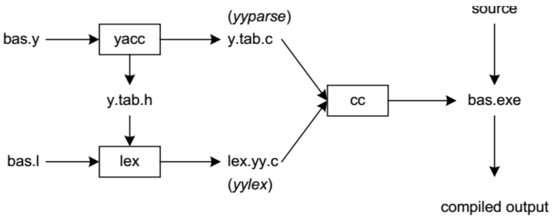


Figure 2: Building a Compiler with Lex/Yacc

Figure 2 illustrates the file naming conventions used by lex and yacc. We'll assume our goal is to write a BASIC compiler. First, we need to specify all pattern matching rules for lex (**bas.l**) and grammar rules for yacc (**bas.y**). Commands to create our compiler, **bas.exe**, are listed below:

bison -d bas.y #creates bas.tab.c and bas.tab.h
flex bas.l #creates lex.yy.c
gcc bas.tab.c lex.yy.c -o bas.exe #creates the executable file bas.exe

Example

(1) $E \to E + E$
(2) $E \to E * E$

(3) $E \to id$

(r1) $E \to E + E$
(r2) $\to E + E * E$
(r3) $\to E + E * id$
(r3) $\to E + id * id$

(r3) $\to id + id * id$

1	. x + y * z	shift	
2	x . + y * z	reduce(r3)	
3	E . + y * z	shift	
4	E + . y * z	shift	
5	E + y . * z	reduce(r3)	
6	E + E . * z	shift	
7	E + E * . z	shift	
8	E + E * z .	reduce(r3)	
9	E + E * E .	reduce(r2)	emit multiply
10	E + E .	reduce(r1)	emit add
11	E .	accept	

The grammar is ambiguous and has shift-reduce conflict.

Shift-reduce conflict: Yacc chooses shift over reduce.
Reduce-reduce conflict: Yacc chooses the first production in the listing.

```
... definitions ...
%%
... rules ...
%%
... subroutines ...
```

The definitions section consists of token declarations and C code bracketed by "%{" and "%}".
The BNF grammar is placed in the rules section and user subroutines are added in the subroutines section.

Bison Stacks

Bison Stacks

- Internally yacc maintains two stacks in memory; a parse stack and a value stack.

Bison Stacks

- Internally yacc maintains two stacks in memory; a parse stack and a value stack.
- The parse stack contains terminals and nonterminals that represent the current parsing state.

Bison Stacks

- Internally yacc maintains two stacks in memory; a parse stack and a value stack.
- The parse stack contains terminals and nonterminals that represent the current parsing state.
- The value stack is an array of YYSTYPE elements and associates a value with each element in the parse stack.

Bison Stacks

- Internally yacc maintains two stacks in memory; a parse stack and a value stack.
- The parse stack contains terminals and nonterminals that represent the current parsing state.
- The value stack is an array of YYSTYPE elements and associates a value with each element in the parse stack.
- For example, when lex returns an INTEGER token, yacc shifts this token to the parse stack.

Bison Stacks

- Internally yacc maintains two stacks in memory; a parse stack and a value stack.
- The parse stack contains terminals and nonterminals that represent the current parsing state.
- The value stack is an array of YYSTYPE elements and associates a value with each element in the parse stack.
- For example, when lex returns an INTEGER token, yacc shifts this token to the parse stack.
- At the same time the corresponding yylval is shifted to the value stack.

Bison Stacks

- Internally yacc maintains two stacks in memory; a parse stack and a value stack.
- The parse stack contains terminals and nonterminals that represent the current parsing state.
- The value stack is an array of YYSTYPE elements and associates a value with each element in the parse stack.
- For example, when lex returns an INTEGER token, yacc shifts this token to the parse stack.
- At the same time the corresponding yylval is shifted to the value stack.
- The parse and value stacks are always synchronized so finding a value related to a token on the stack is easily accomplished.

Bison program for calculator

```
%{
    #include <stdio.h>
    int yylex(void);
    void yyerror(char *);
%}

%token INTEGER

%%

program:
        program expr '\n'         { printf("%d\n", $2); }
        |
        ;

expr:
        INTEGER                   { $$ = $1; }
        | expr '+' expr           { $$ = $1 + $3; }
        | expr '-' expr           { $$ = $1 - $3; }
        ;

%%

void yyerror(char *s) {
    fprintf(stderr, "%s\n", s);
}

int main(void) {
    yyparse();
    return 0;
```

With left-recursion, we have specified that a program consists of zero or more expressions. Each expression terminates with a newline. When a newline is detected we print the value of the expression.

Bison program for calculator

```
%{
    #include <stdio.h>
    int yylex(void);
    void yyerror(char *);
%}

%token INTEGER

%%

program:
        program expr '\n'         { printf("%d\n", $2); }
        |
        ;

expr:
        INTEGER                   { $$ = $1; }
        | expr '+' expr           { $$ = $1 + $3; }
        | expr '-' expr           { $$ = $1 - $3; }
        ;

%%

void yyerror(char *s) {
    fprintf(stderr, "%s\n", s);
}

int main(void) {
    yyparse();
    return 0;
```

When we apply the rule expr: expr '+' expr $\{$ \$\$ $=$ \$1 $+$ \$3; $\}$ we replace the right-hand side of the production in the parse stack with the left-hand side of the same production. In this case we pop expr '+' expr and push expr. We have reduced the stack by popping three terms off the stack and pushing back one term.

Bison program for calculator

```
%{
    #include <stdio.h>
    int yylex(void);
    void yyerror(char *);
%}

%token INTEGER

%%

program:
        program expr '\n'         { printf("%d\n", $2); }
        |
        ;

expr:
        INTEGER                   { $$ = $1; }
        | expr '+' expr           { $$ = $1 + $3; }
        | expr '-' expr           { $$ = $1 - $3; }
        ;

%%

void yyerror(char *s) {
    fprintf(stderr, "%s\n", s);
}

int main(void) {
    yyparse();
    return 0;
```

“\$1” denotes the first term on the right-hand side of the production, “\$2” for the second, and so on. “\$\$” designates the top of the stack after reduction has taken place. The above action adds the value associated with two expressions, pops three terms off the value stack, and pushes back a single sum.

Bison program for calculator

```
%{
    #include <stdio.h>
    int yylex(void);
    void yyerror(char *);
%}

%token INTEGER

%%

program:
        program expr '\n'         { printf("%d\n", $2); }
        |
        ;

expr:
        INTEGER                   { $$ = $1; }
        | expr '+' expr           { $$ = $1 + $3; }
        | expr '-' expr           { $$ = $1 - $3; }
        ;

%%

void yyerror(char *s) {
    fprintf(stderr, "%s\n", s);
}

int main(void) {
    yyparse();
    return 0;
```

%token INTEGER
This definition declares an INTEGER token.
Yacc generates a parser in file y.tab.c and an include file, y.tab.h:
#ifndef YYSTYPE
#define YYSTYPE int
#endif
#define INTEGER 258
extern YYSTYPE yylval;

Lex program for calculator

```
%{
#include <stdlib.h>
void yyerror(char *);
#include "y.tab.h"
%}

%%

[0-9]+      {
                yylval = atoi(yytext);
                return INTEGER;
            }

[-+\n]      return *yytext;

[ \t]       ; /* skip whitespace */

.           yyerror("invalid character");

%%

int yywrap(void) {
    return 1;
}
```

Lex includes this file and utilizes the definitions for token values.
To obtain tokens yacc calls yylex.
Function yylex has a return type of int that returns a token.

Values associated with the token are returned

by lex in variable yylval.