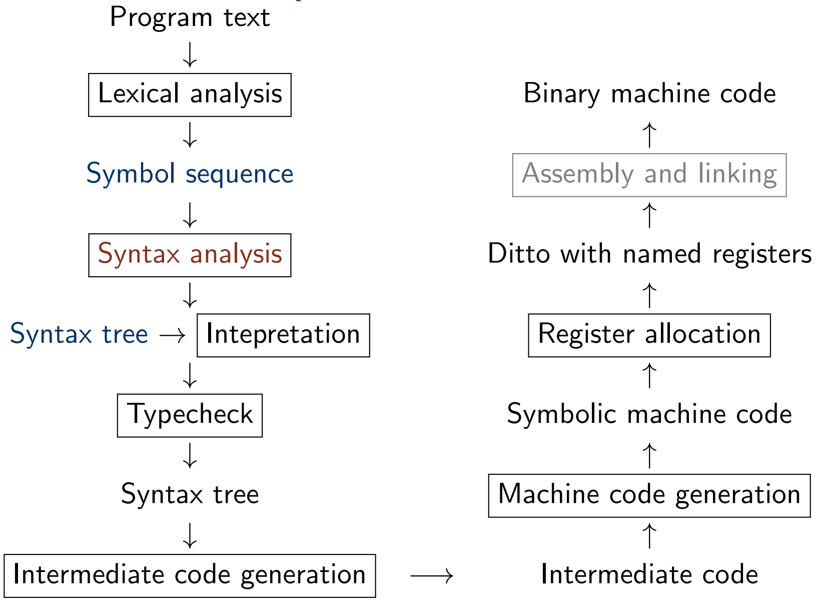
UNIVERSITÀ di VERONA
Dipartimento di INFORMATICA

# Syntax Analysis

Alessandra Di Pierro
alessandra.dipierro@univr.it
Using Cosmin E. Oancea and Jost Berthold's material (DIKU - University of Copenhagen)

Compiler Lecture Notes

## Structure of a Compiler



## Syntax Analysis (Parsing)

Relates to the correct construction of sentences, i.e., grammar.

1. Checks that grammar is respected, otherwise syntax error, and

2. Arranges tokens into a syntax tree reflecting the text structure: leaves are tokens, which if read from left to right results in the original text!

```
mother
cokes
dinner
My.
```

*syntax error*

```
My dinner
cokes
mother.
```

*semantic error*

Essential tool and theory used are *Context-Free Grammars*:
a notation suitable for human understanding that can be transformed into an efficient implementation.

## Context-Free Grammar (CFG) Definition

1. a set of *terminals* $\sum$ – the language alphabet, e.g., the set of tokens produced by lexer. (Convention: use small letters.)
2. a set of *non-terminals* $N$, denoting sets of recursively defined strings.
3. *a start symbol* $S \in N$, denoting the lang defined by the grammar.
4. a set $P$ of productions of form $Y \rightarrow X_1 \dots X_n$, where $Y \in N$ is a (single) non-terminal, and $X_i \in (\sum \cup N)^*, \forall i$ can be a terminal or non-terminal. Each production describes some of the strings of the corresponding non-terminal $Y$.

G: $S \rightarrow aS$
$S \rightarrow \epsilon$
regular-expression language $a^*$

G: $S \rightarrow aSb$
$S \rightarrow \epsilon$
describes language $\{a^n b^n, \forall n \geq 0\}$

G: $S \rightarrow aSa \mid bSb \mid \dots$
$S \rightarrow a \mid b \mid \dots \mid \epsilon$
describes palyndromes, e.g., *abba*, *babab*.

The latter two languages cannot be described with regular expressions.

## Transforming Regular Expressions (*RE*) To CFGs

***RE*s can systematically be rewritten as CFGs, hence CFGs are strictly more powerful than *RE*s:** A regular expression $s_i$ is translated to a nonterminal $N_i$, which is defined by one or two productions, by pattern-matching the shape of $s_i$, as defined below:

| Form of $s_i$ | Productions for $N_i$ |
|---|---|
| $\epsilon$ | $N_i \rightarrow \epsilon$ |
| a | $N_i \rightarrow$ a |
| $s_j \ s_k$ | $N_i \rightarrow N_j N_k$ |
| $s_j \mid s_k$ | $N_i \rightarrow N_j$ |
| | $N_i \rightarrow N_k$ |
| $s_j^*$ | $N_i \rightarrow N_j N_i$ |
| | $N_i \rightarrow \epsilon$ |
| $s_j^+$ | $N_i \rightarrow N_j N_i$ |
| | $N_i \rightarrow N_j$ |
| $s_j?$ | $N_i \rightarrow N_j$ |
| | $N_i \rightarrow \epsilon$ |

## Example: Deriving Words

Nonterminals can recursively refer to each other
(cannot do that with regular expressions):

$$\begin{array}{ll} \text{G: } S \to aSB & (1) \\ S \to \epsilon & (2) \\ B \to Bb & (3) \\ B \to b & (4) \end{array} \qquad \begin{array}{l} \text{G: } S \to aSB \mid \epsilon \\ B \to Bb \mid \epsilon \\ B \to \epsilon \end{array} \qquad \begin{array}{l} S = \{a \cdot x \cdot y \mid x \in S,\ y \in B\} \cup \{\epsilon\} \\ B = \{x \cdot b \mid x \in B\} \cup \{b\} \end{array}$$

- Words of the language can be constructed by
- starting with the start symbol $S$, and
- successively replacing nonterminals with right-hand sides.

Deriving `aaabbbb`:

$$S \Rightarrow^1 \underline{aSB} \Rightarrow^1 a\underline{aSB}B \Rightarrow^4 aaS\underline{b}B \Rightarrow^1 aa\underline{aSB}bB$$
$$\Rightarrow^1 aaa\underline{\ }BbB \Rightarrow^3 aaa\underline{Bb}bB \Rightarrow^4 aaaBbb\underline{b} \Rightarrow^4 aaa\underline{b}bbb.$$

## Definition: Derivation Relation

Let $G = (\sum, N, S, P)$ be a grammar.
The derivation relation $\Rightarrow$ on $(\sum \cup N)^*$ is defined as:

- for a nonterminal $X \in N$ and a production $(X \rightarrow \beta) \in P$, $\alpha_1 X \alpha_2 \;\Rightarrow\; \alpha_1 \beta \alpha_2$, for all $\alpha_1, \alpha_2 \in (\sum \cup N)^*$
- describes one derivation step using one of the productions.
- Production can be numbered with the grammar-rule number.

G: $S \rightarrow aSB$ (1)
$S \rightarrow \epsilon$ (2)
$B \rightarrow Bb$ (3)
$B \rightarrow b$ (4)

$$S \Rightarrow^1 \underline{aSB} \Rightarrow^1 a\underline{aSB}B \Rightarrow^2 aa\_BB$$
$$\Rightarrow^3 aa\underline{Bb}B \Rightarrow^4 aa\underline{b}bB \Rightarrow^4 aabb\underline{b}.$$

- Here we have used leftmost derivation, i.e., always expanded the leftmost terminal first. Could also use right-most derivation.
- $aaabbbb$ and $aabbb \in L(G)$.

## Transitive Derivation Relation Definition

Let $G = (\sum, N, S, P)$ be a grammar and $\Rightarrow$ its derivation relation. The transitive derivation relation is defined as:

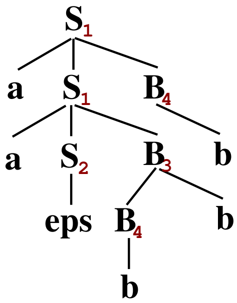
- $\alpha \Rightarrow^* \alpha$, for $\alpha \in (\sum \cup N)^*$, derived in 0 steps,
- for $\alpha, \beta \in (\sum \cup N)^*$, $\alpha \Rightarrow^* \beta$ *iff* there exists $\gamma \in (\sum \cup N)^*$ such that $\alpha \Rightarrow \gamma$, and $\gamma \Rightarrow^* \beta$, i.e., derived in at least one step.

The Language of a Grammar consists of all the words that can be obtained via the transitive derivation relation:
$L(G) = \{w \in \sum^* \mid S \Rightarrow^* w\}$.

For example *aaabbbb* and *aabbb* $\in L(G)$,
because $S \Rightarrow^* aaabbbb$ and $S \Rightarrow^* aabbb$.

## Syntax Trees

G: $S \rightarrow aSB$ (1)
$S \rightarrow \epsilon$ (2)
$B \rightarrow Bb$ (3)
$B \rightarrow b$ (4)



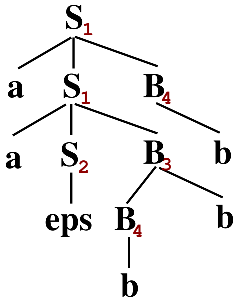
**Syntax trees** describe the "structure" of the derivation (independent of the order in which nonterminals have been chosen to be derived).

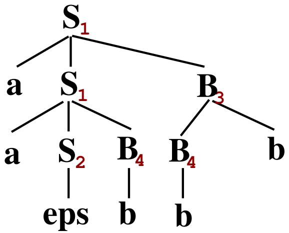
Leftmost derivation always derives the leftmost nonterminal first, and corresponds to a *depth-first, left-to-right tree traversal*:

$S \Rightarrow^1 \underline{aSB} \Rightarrow^1 a\underline{aSB}B \Rightarrow^2 aa\_BB \Rightarrow^3 aa\underline{Bb}B \Rightarrow^4 aa\underline{b}bB \Rightarrow^4 aabb\underline{b}.$

## Syntax Trees & Ambiguous Grammars

G: $S \to aSB$ (1)
$S \to \epsilon$ (2)
$B \to Bb$ (3)
$B \to b$ (4)





Syntax trees describe the "structure" of the derivation; the order of derivation (left- or rightmost) is irrelevant.

**The grammar is said to be ambiguous** if there exists a word that can be derived in two ways corresponding to different syntax trees.

$S \Rightarrow^1 \underline{aSB} \Rightarrow^1 a\underline{aSB}B \Rightarrow^2 aa\_BB \Rightarrow^3 aa\underline{Bb}B \Rightarrow^4 aa\underline{b}bB \Rightarrow^4 aabb\underline{b}$.

$S \Rightarrow^1 \underline{aSB} \Rightarrow^1 a\underline{aSB}B \Rightarrow^2 aa\_BB \Rightarrow^4 aa\underline{b}B \Rightarrow^3 aab\underline{bB} \Rightarrow^4 aabb\underline{b}$.

## Why is Grammar Ambiguity Important?

When grammar is used to describe sets of strings, ambiguity is OK.
**Most often grammars are used to impose structure on strings**

Ambiguity is in general undecidable, i.e., there is no method that for all grammars can answer the question "Is this grammar ambiguous?"

However, in many cases, it is not difficult to detect & prove ambiguity.

For example, $N \rightarrow N\alpha N$

is ambiguous, where $\alpha$ is any sequence of grammar symbols. WHY?

Remove ambiguity by transforming the grammar into an equivalent unambiguous one, i.e., that generates the same language.
(No known method to decide equivalence between grammars either.)

**The methods for constructing parsers from grammars, which we will learn a bit later, only work for unambiguous grammars!**

## Handling/Removing Grammar Ambiguity

$$\begin{aligned} E &\rightarrow E + E \mid E - E \\ E &\rightarrow E * E \mid E \, / \, E \\ E &\rightarrow a \mid (E) \end{aligned}$$

- Can this be represented as a *RE*?
- *Precedence and Associativity* guide decision:
- ambiguity resolved by parsing directives,
- or by rewriting the grammar.

What are the problems:

- Ambiguous derivation of $a - a - a$ can be resolved by fixing *a left-associative* derivation: $(a - a) - a$.
- Ambiguous derivation of $a + a * a$ can be resolved by setting *the precedence* of $*$ higher than $+$: $a + (a * a)$.

## Defining/Resolving Operator Associativity

A binary operator is called:

- *left associative* if expression $x \oplus y \oplus z$ should be evaluated from left to right: $(x \oplus y) \oplus z$
- *right associative* if expression $x \oplus y \oplus z$ should be evaluated from right to left: $x \oplus (y \oplus z)$
- *non-associative* if expressions such as $x \oplus y \oplus z$ are disallowed,
- associative if both left-to-right and right-to-left evaluations lead to the same result.

Examples:

- left associative operators: $-$ and $/$,
- right associative operators: exponentiation, assignment.

## Establishing Intended Associativity

$$E \rightarrow E \oplus E$$

$$E \rightarrow \mathsf{num}$$

Why is the grammar above ambiguous?

Disambiguated via a new nonterminal that fixes associativity:

- If $\oplus$ is left-associative $\Rightarrow$ by making the grammar **left recursive**:

$$E \rightarrow E \oplus E' \mid E'$$
$$E' \rightarrow \mathsf{num}$$

- If $\oplus$ right-associative $\Rightarrow$ by making the grammar **right recursive**:

$$E \rightarrow E' \oplus E \mid E'$$
$$E' \rightarrow \mathsf{num}$$

- If $\oplus$ is non-associative $\Rightarrow$ by making the grammar **non recursive**:

$$E \rightarrow E' \oplus E' \mid E'$$
$$E' \rightarrow \mathsf{num}$$

This changes the language because `num+num+num` is now illegal!

## Establishing Intended Associativity

What if we have two operators at **the same precedence level** (+,-)?

$$E \to E + E \mid E \oplus E$$

$$E \to \text{num}$$

Disambiguated in the same way, **but if and only if $+$ and $\oplus$ have the same associativity!** For example, with left associative $+$ and $\oplus$:

$$E \to E + E' \mid E \oplus E' \mid E'$$

$$E' \to \text{num}$$

Why does it NOT work if $+$ and $\oplus$ have different associativity?

$$E \to E + E' \mid E' \oplus E \mid E'$$

$$E' \to \text{num}$$

It does not accept the same language: $\texttt{num}+\texttt{num}\oplus\texttt{num}$ not derivable!

**Same-precedence operators need to have identical associativity!**

## Defining/Resolving Operator Precedence

- Introduce precedence levels to set operator priorities. **Idea:** If an exp uses an operator of certain precedence, then its subexps cannot use lower-precedence operators (without parentheses).
- for example precedence of $*$ and $/$ over (higher than) $+$ and $-$,
- and more precedence levels can be added, e.g., exponentiation.

At grammar level: this can be accomplished by introducing one nonterminal for each level of precedence. More complex example:

$$\begin{array}{lll} E & \to & E + E \mid E - E \\ E & \to & E * E \mid E \,/\, E \\ E & \to & a \mid (E) \end{array} \qquad \begin{array}{lll} E & \to & E + E \mid E - E \mid T \\ T & \to & T * T \mid T \,/\, T \\ T & \to & a \mid (E) \end{array}$$

Finally, one can solve associativity as discussed before:

## More Complex Example: Establishing Associativity

- Can be declared in the parser file via directives
- when operators are associative use same associativity as comparable operators,
- **cannot mix left- and right-associative operators at the same precedence level.**

At grammar level: this can be accomplished by introducing new nonterminals that establish explicitly operator's associativity :

$$\begin{array}{lll} E & \rightarrow & E + E \mid E - E \mid T \\ T & \rightarrow & T * T \mid T / T \\ T & \rightarrow & a \mid (E) \end{array} \qquad \begin{array}{lll} E & \rightarrow & E + T \mid E - T \mid T \\ T & \rightarrow & T * F \mid T / F \mid F \\ F & \rightarrow & a \mid (E) \end{array}$$

## Parsing

- Leftmost or rightmost derivation produces a syntax tree from a sequence of tokens.

Two approaches:

- *Top-Down Parsing:* builds syntax tree from the root going down.
  - also called *predictive parsing*: guesses the to-be-used production.
  - uses leftmost-recurrence derivation of look-ahead 1, i.e., “LL(1)”
- *Bottom-Up Parsing:* builds syntax tree starting from the leaves and going up.
  - search for parts of input string that match right-hand side of productions and rewrite these to left-hand side nonterminals,
  - syntax tree is completed when the string has been rewritten by inverse derivation to the start symbol.

  - Also called *shift-reduce parsing*: shifts input to stack,
  - then reduces right-hand side to left-hand side nonterminal,
  - until the start symbol is reached.
- Both use a stack to keep track of the derivation sequence.

## Top-Down Parsing Look-Ahead 1, LL(1)

- First, make the grammar unambiguous:

G: $S \to aSB \mid B \mid \epsilon$
$B \to Bb \mid b$

Ambiguous.

G': $S \to aSb \mid B$ (1,2)
$B \to bB \mid \epsilon$ (3,4)

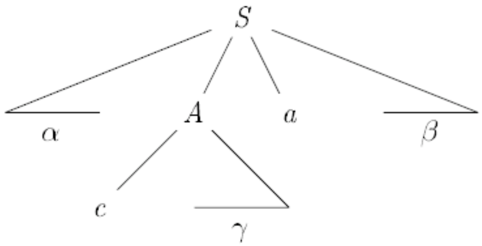
Unambiguous (and same language).

- Compute info to choose the right production.
  For each right-hand side: *What input token may come first?*
- Special attention to empty right-hand sides. What may follow?
- Then, we can decide from a look-ahead of one token, which production to use. Choose a production $N \to \alpha$ if:
  1. look-ahead is $c$ and $\alpha \Rightarrow^*$ something that starts with $c$,
  2. look-ahead is $c$ and $\alpha \Rightarrow^* \epsilon$ and $c$ can follow $N$.

On the blackboard, assume input *aabbb*:
$S \Rightarrow^1 \underline{a}Sb \Rightarrow^1 a\underline{a}Sbb \Rightarrow^2 aa\underline{B}bb \Rightarrow^3 aa\underline{b}Bbb \Rightarrow^4 aabbb.$

## Use of First and Follow Sets

Consider a derivation $S \Rightarrow^* \alpha A a \beta$, and a derivation from $A$ $A \Rightarrow^* c\gamma$.



We say that $c \in \textsc{First}(A)$ and $a \in \textsc{Follow}(A)$.

## First Sets and Nullable Property

**Definition (FIRST Set and NULLABLE)**

*Let $G = (\sum, N, S, P)$ a grammar and $\Rightarrow$ its derivation relation. For all sequences of grammar symbols $\alpha \in (\sum \cup N)^*$, define*

- $\textsc{First}(\alpha) \; = \; \{c \in \sum \mid \exists_{\beta \in (\sum \cup N)^*} \texttt{ s.t. } \alpha \Rightarrow^* c\beta\}$
  *(all input tokens at the start of what can be derived from $\alpha$)*
- $\textsc{Nullable}(\alpha) = \begin{cases} \textit{true}, & \textit{if } \alpha \Rightarrow^* \epsilon \\ \textit{false}, & \textit{otherwise} \end{cases}$

Computing NULLABLE and FIRST for right-hand sides:

- Set equations recursively use results for nonterminals
- Smallest solution found by computing a smallest fixed point
- Solved simultaneously for all right-hand sides of the productions.

## Recursive Rules for Nullable

$$\begin{array}{ll}
\textsc{Nullable}(\epsilon) & = \textit{true} \\
\textsc{Nullable}(\mathsf{a}) & = \textit{false} \\
\textsc{Nullable}(\alpha\beta) & = \textsc{Nullable}(\alpha) \wedge \textsc{Nullable}(\beta),\ \alpha, \beta \in (\sum \cup N)^* \\
\textsc{Nullable}(N) & = \textsc{Nullable}(\alpha_1)\ \vee \cdots \vee\ \textsc{Nullable}(\alpha_n), \\
 & \text{for all productions of } N \text{ of form } N \rightarrow \alpha_i,\ \ i \in \{1 \ldots n\}
\end{array}$$

- With our grammar, Equations for nonterminals:

G': $S \rightarrow aSb \mid B$ $\quad \textsc{Nullable}(S) = \textsc{Nullable}(aSb) \vee \textsc{Nullable}(B)$

$B \rightarrow bB \mid \epsilon$ $\quad \textsc{Nullable}(B) = \textsc{Nullable}(bB) \vee \textsc{Nullable}(\epsilon)$

- With our grammar, Equations for the right-hand side:

$$\begin{array}{ll}
\textsc{Nullable}(aSb) & = \textsc{Nullable}(a) \wedge \textsc{Nullable}(S) \wedge \textsc{Nullable}(b) \\
\textsc{Nullable}(B) & = \textsc{Nullable}(B) \\
\textsc{Nullable}(bB) & = \textsc{Nullable}(b) \wedge \textsc{Nullable}(B) \\
\textsc{Nullable}(\epsilon) & = \textit{true}
\end{array}$$

Compute fix-point solution of the system by starting with *false* for all
$\textsc{Nullable}(aSb) = \textit{false}$ and $\textsc{Nullable}(bB) = \textit{false}$,
$\textsc{Nullable}(B) = \textit{true}$ and $\textsc{Nullable}(S) = \textit{true}$,

## Recursive Rules for First, i.e., Set Equations

$$\begin{array}{ll}
\textsc{First}(\epsilon) & = \emptyset \\
\textsc{First}(\mathsf{a}) & = \mathsf{a},\ \forall\ \mathsf{a} \in \sum \\
\textsc{First}(\alpha\beta) & = \begin{cases} \textsc{First}(\alpha)\ \cup\ \textsc{First}(\beta), & \text{if } \textsc{Nullable}(\alpha) \\ \textsc{First}(\alpha), & \text{otherwise} \end{cases} \\
\textsc{First}(N) & = \textsc{First}(\alpha_1)\ \cup \cdots \cup\ \textsc{First}(\alpha_n), \\
 & \text{for all productions of } N \text{ of form } N \to \alpha_i,\ i \in \{1 \ldots n\}
\end{array}$$

- With our grammar, Equations for nonterminals:

G': $S \to aSb \mid B$ $\quad \textsc{First}(S) = \textsc{First}(aSb) \cup \textsc{First}(B)$

$B \to bB \mid \epsilon$ $\quad \textsc{First}(B) = \textsc{First}(bB) \cup \textsc{First}(\epsilon)$

- With our grammar, Equations for the right-hand side:

$$\begin{array}{ll}
\textsc{First}(aSb) & = \textsc{First}(a) \\
\textsc{First}(B) & = \textsc{First}(B) \\
\textsc{First}(bB) & = \textsc{First}(b) \\
\textsc{First}(\epsilon) & = \emptyset
\end{array}$$

Compute fix-point solution of the system by starting with $\emptyset$ for all:

## Recursive Rules for First, i.e., Set Equations

$$
\begin{array}{ll}
\textsc{First}(\epsilon) & = \emptyset \\
\textsc{First}(\mathsf{a}) & = \mathsf{a},\ \forall\ \mathsf{a} \in \sum \\
\textsc{First}(\alpha\beta) & = \begin{cases} \textsc{First}(\alpha)\ \cup\ \textsc{First}(\beta), & \text{if } \textsc{Nullable}(\alpha) \\ \textsc{First}(\alpha), & \text{otherwise} \end{cases} \\
\textsc{First}(N) & = \textsc{First}(\alpha_1)\ \cup \cdots \cup\ \textsc{First}(\alpha_n), \\
 & \text{for all productions of } N \text{ of form } N \to \alpha_i,\ i \in \{1 \dots n\}
\end{array}
$$

- With our grammar, Equations for nonterminals:

G': $S \to aSb \mid B$ $\qquad \textsc{First}(S) = \textsc{First}(aSb) \cup \textsc{First}(B) = \{\mathtt{a},\ \mathtt{b}\}$

$B \to bB \mid \epsilon$ $\qquad \textsc{First}(B) = \textsc{First}(bB) \cup \textsc{First}(\epsilon) = \{\mathtt{b}\}$

- With our grammar, Equations for the right-hand side:

$$
\begin{array}{lll}
\textsc{First}(aSb) & = \textsc{First}(a) & = \{\mathtt{a}\} \\
\textsc{First}(B) & = \textsc{First}(B) & = \{\mathtt{b}\} \\
\textsc{First}(bB) & = \textsc{First}(b) & = \{\mathtt{b}\} \\
\textsc{First}(\epsilon) & = \emptyset &
\end{array}
$$

Compute fix-point solution of the system by starting with $\emptyset$ for all:

## Follow Sets for Nonterminals

- $\textsc{First}$ sets might not be enough for parsing.
- Assume a production $X \to \alpha$ and $\textsc{Nullable}(\alpha) = \mathit{true}$.
- If the look-ahead can follow $X$ then this production may be chosen (but the $\textsc{First}$ set cannot provide this information!)

**Definition ($\textsc{Follow}$ Set of a Nonterminal)**

*Let $G = (\sum, N, S, P)$ a grammar and $\Rightarrow$ its derivation relation. For each nonterminal $X \in N$ define*

- $\textsc{Follow}(X) \ = \ \{c \in \sum \mid \exists_{\alpha,\beta \in (\sum \cup N)^*} \texttt{ s.t. } S \Rightarrow^* \alpha \underline{Xc} \beta\}$

  *(all input tokens that follow $X$ in sequences derivable from $S$)*

*To recognize the end of the input, we also extend the grammar with a new start symbol $S'$, a new character \$ and a production $S' \to S\$$. It follows that we always have $\$ \in \textsc{Follow}(S)$.*

## Recursive Rules for Follow, i.e., Set Equations

- requires solving a collection of set constraints,
- which are derived from *right-hand side productions*.

**Definition (Follow Set Constraints)**

*For $X \in N$, consider all productions $Y \to \alpha X \beta$, where $X$ appears on the right-hand side:*

- $\textsc{First}(\beta) \subseteq \textsc{Follow}(X)$
- *If* $\textsc{Nullable}(\beta)$ *or* $\beta = \epsilon$ *then* $\textsc{Follow}(Y) \subseteq \textsc{Follow}(X)$

*If $X$ appears multiple times in the right-hand side of a production, then each occurrence contributes with a separate constraint equation.*

## Example: Computation of Follow Sets

| | | | |
|---|---|---|---|
| G': $S' \to S\$$ | … | $\textsc{First}(\$) = \{\$\}$ | $\subseteq \textsc{Follow}(S)$ |
| $S \to aSb$ | … | $\textsc{First}(\text{b}) = \{\text{b}\}$ | $\subseteq \textsc{Follow}(S)$ |
| $S \to B$ | … | $\textsc{Follow}(S)$ | $\subseteq \textsc{Follow}(B)$ |
| $B \to bB$ | … | $\textsc{Follow}(B)$ | $\subseteq \textsc{Follow}(B)$ |
| $B \to \epsilon$ | … | no equation | |

All nonterminals initialized with $\emptyset$: $\textsc{Follow}(S) = \textsc{Follow}(B) = \emptyset$

Iteration 1 $\textsc{Follow}(S) = \{\$, \text{b}\}$, $\textsc{Follow}(B) = \{\$, \text{b}\}$

Iteration 2 $\textsc{Follow}(S) = \{\$, \text{b}\}$, $\textsc{Follow}(B) = \{\$, \text{b}\}$

Fix Point: $\textsc{Follow}(S) = \{\$, \text{b}\}$, $\textsc{Follow}(B) = \{\$, \text{b}\}$

## Look-Ahead Sets and LL(1) Grammars

An LL(1) parser requires computation of the:

- $\textsc{Nullable}$ and $\textsc{First}$ sets for all right-hand sides of prods, and
- $\textsc{Follow}$ sets for all nonterminals.

**Definition (Look-Ahead Sets for a Grammar)**

*The Look-Ahead Set of a grammar production $X \to \alpha$ is:*

$$la(X \to \alpha) = \begin{cases} \textsc{First}(\alpha) \cup \textsc{Follow}(X), & \textit{if } \textsc{Nullable}(\alpha) \\ \textsc{First}(\alpha), & \textit{otherwise} \end{cases}$$

**Definition (LL(1) Grammar)**

*If for each nonterminal $X \in N$, the look-ahead sets of all productions of $X$ are disjoint, then a syntax tree can be build by looking ahead at exactly one token.*

*The grammar is called LL(1): left-to-right, left-most, look-ahead* 1*. The parser can predict the next production from the look-ahead token.*

## Example: Look-Ahead Sets

| | | | |
|---|---|---|---|
| G': $S' \to S\$$ | ... | $la(S' \to S\$)$ | $= \textsc{First}(S\$) = \{a, b, \$\}$ |
| $S \to aSb$ | ... | $la(S \to aSb)$ | $= \textsc{First}(aSb) = \{a\}$ |
| $S \to B$ | ... | $la(S \to B)$ | $= \textsc{First}(B) \cup \textsc{Follow}(B) = \{b\}$ |
| $B \to bB$ | ... | $la(B \to bB)$ | $= \textsc{First}(bB) = \{b\}$ |
| $B \to \epsilon$ | ... | $la(B \to \epsilon)$ | $= \textsc{First}(\epsilon) \cup \textsc{Follow}(B) = \{\$, b\}$ |

**Our grammar is not LL(1), albeit it is unambiguous!**

To make it LL(1) one can modify the grammar to separate the generation of the additional bs at the end:

| | | |
|---|---|---|
| G': $S' \to S\$$ | $la(S' \to S\$)$ | $= \textsc{First}(S\$) = \{a, b, \$\}$ |
| 1: $S \to AB$ | $la(S \to AB)$ | $= \textsc{First}(AB) \cup \textsc{Follow}(S) = \{a, b, \$\}$ |
| 2: $A \to aAb$ | $la(A \to aAb)$ | $= \textsc{First}(aAb) = \{a\}$ |
| 3: $A \to \epsilon$ | $la(A \to \epsilon)$ | $= \textsc{First}(\epsilon) \cup \textsc{Follow}(A) = \{b, \$\}$ |
| 4: $B \to bB$ | $la(B \to bB)$ | $= \textsc{First}(bB) = \{b\}$ |
| 5: $B \to \epsilon$ | $la(B \to \epsilon)$ | $= \textsc{First}(\epsilon) \cup \textsc{Follow}(B) = \{\$\}$ |

## Recursive Descent Parsing

- Implem: a set of recursive procedures, one for each nonterminal.
- The syntax tree is implicitly (partially) built on the call stack.
- Each procedure matches tokens (error if unmatchable), and
- calls the corresp. procedure for each right-hand side nonterminal
- When a production is used, its number is added to the output.

```
fun parseS'() = (* check lookahead la(S' -> S $) = {a,b,$} *)
    if   next = 'a' or next = 'b' or next = '$'
    then print("0"); parseS(); match('$'); (* follow production *)
    else error("unacceptable input in rule S' -> S $: " ^ next)

and parseS()  = (* check lookahead la(S -> AB) = {a,b,$} *)
    if   next = 'a' or next = 'b' or next = '$'
    then print("1"); parseA(); parseB(); (* follow production *)
    else error("unacceptable input in rule S -> AB $: " ^ next)

and parseA()  = (* branch on lookahead la(A -> aAb) = {a} *)
    if      next = 'a' then print("2"); match('a'); parseA(); match('b');
                (* branch on lookahead la(A -> ϵ) = {b, $} *)
    else if next = 'b' or next = '$' then print("3");
    else    error("unacceptable input in rule S -> aAb | ϵ: " ^ next)
```

## Recursive Descent Parsing Continuation

```
fun parseS'() = (* check lookahead la(S' -> S $) = {a,b,$} *)
    if   next = 'a' or next = 'b' or next = '$'
    then print("0"); parseS(); match('$'); (* follow production *)
    else error("unacceptable input in rule S' -> S $: " ^ next)

and parseS()  = (* check lookahead la(S -> AB) = {a,b,$} *)
    if   next = 'a' or next = 'b' or next = '$'
    then print("1"); parseA(); parseB(); (* follow production *)
    else error("unacceptable input in rule S -> AB $: " ^ next)

and parseA()  = (* branch on lookahead la(A -> aAb) = {a} *)
    if      next = 'a' then print("2"); match('a'); parseA(); match('b');
                (* branch on lookahead la(A -> ϵ) = {b, $} *)
    else if next = 'b' or next = '$' then print("3");
    else    error("unacceptable input in rule A -> aAb | ϵ: " ^ next)

and parseB()  = (* branch on lookahead la(B -> bB) = {b} *)
    if      next = 'b' then print("4"); match('b'); parseB();
                (* branch on lookahead la(B -> ϵ) = {$} *)
    else if next = '$' then print("5");
    else    error("unacceptable input in rule B -> bB | ϵ: " ^ next)
```

## Table-Driven LL(1) Parsing

- *Stack:* contains unprocessed part of production, initially $S\$$,
- *Parse Table:* gives the action to take, depending on stack & next token,
- *Pop action:* removes/consumes terminal from stack on matching input,
- *Derive action:* pops nonterminal from stack, pushes its right side into parse table,
- *Accept* input when stack is empty and end of input.

Look-Ahead/Input

| Stack | a | b | $ |
|---|---|---|---|
| S' | S, 0 | S, 0 | S, 0 |
| S | AB, 1 | AB,1 | AB,1 |
| A | aAb,2 | $\epsilon$,3 | $\epsilon$, 3 |
| B | error | bB, 4 | $\epsilon$, 5 |
| a | *pop* | error | error |
| b | error | *pop* | error |
| $ | error | error | accept |

Example On Input aabbb:

| Input | Stack | Action | Output |
|---|---|---|---|
| aabbb$ | S$ | derive | 0 |
| aabbb$ | AB$ | derive | 01 |
| aabbb$ | aAbB$ | pop | 012 |
| abbb$ | AbB$ | derive | 012 |
| abbb$ | aAbbB$ | pop | 0122 |
| bbb$ | AbbB$ | derive | 0122 |
| bbb$ | bbB$ | pop | 01223 |
| bb$ | bB$ | pop | 01223 |
| b$ | B$ | derive | 01223 |
| b$ | bB$ | pop | 012234 |
| $ | B$ | derive | 012234 |
| $ | $ | accept | 0122345 |

## Eliminating Identical Prefixes by Left Factorization

Two main problems with constructing LL(1) parsers:

- **Identical Prefixes**
- **Left Recursion**

If several productions of the same nonterminal have identical prefixes:

$$X \to \alpha\beta \mid \alpha\gamma \mid \theta$$

then, in order to disambiguate them, the required look-ahead must be longer than the common prefix $\alpha$!

**The solution is Left-Factorization**: rewrite the common-factor productions by introducing a new nonterminal to derive their suffixes:

$$X \to \alpha Y \mid \theta$$

$$Y \to \beta \mid \gamma$$

## Eliminating Left Recursion

**The second problem is Left Recursion**: when a nonterminal reproduces itself on the left-hand side of its production,

$$\text{directly: } X \to X\alpha \mid \beta \mid \ldots$$

$$\text{or indirectly: } X \Rightarrow^* X\alpha$$

The problem is that $\textsc{First}(\beta) \subseteq \textsc{First}(X) \subseteq \textsc{First}(X\alpha)$, hence both productions of $X$, namely $X\alpha$ and $\beta$, have $\beta$ as a prefix.

**Solution: avoid left recursion by rewriting the grammar**:
Hint: the grammar "resembles" regular expression $\beta(\alpha)^*$, which can be rewritten in an equivalent right-recursive form as

$$X \to \beta X'$$

$$X' \to \alpha X' \mid \epsilon$$

## More General: Eliminating Left Recursion

Eliminating left-recursion can be easily extended to work in the case of multiple left-recursive productions:

$$X \to X\alpha_1 \mid \ldots \mid X\alpha_m \mid \beta_1 \mid \ldots \mid \beta_n$$

"resembles" regular expression $(\beta_1 \mid \ldots \mid \beta_n)\ (\alpha_1 \mid \ldots \mid \alpha_m)^*$, which can be rewritten in right-recursive form:

$$X \to \beta_1 X' \mid \ldots \mid \beta_n X'$$

$$X' \to \alpha_1 X' \mid \ldots \mid \alpha_m X' \mid \epsilon$$

**Indirect Left Recursion**, for example

- $X_1 \to X_2\alpha_1,\ \ldots,\ X_k \to X_1\alpha_k$ OR/AND
- $Y \to \alpha Y \beta$ where $\alpha$ is *Nullable*.

Solved by systematically rewriting grammar to make left-recursion direct, then apply the above. **Beyond the scope of this lecture.**

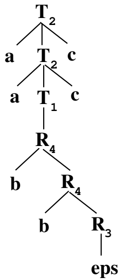
## Summary: Constructing LL(1) Parser

1. Eliminate grammar ambiguity.
2. Eliminate left recursion.
3. Perform left factorization where required.
4. Add an extra start production $S' \rightarrow S\$$ to grammar.
5. Compute the $\textsc{First}$ set for every production
6. Compute the $\textsc{Follow}$ set for every nonterminal
   (Note that 5 & 6 also require the computation of $\textsc{Nullable}$.)
7. For nonterminal $N$ and input symbol $c$, choose $N \rightarrow \alpha$ if
   - $c \in \textsc{First}(\alpha)$, or
   - $\textsc{Nullable}(\alpha)$ and $c \in \textsc{Follow}(N)$

   (The choice is made by either looking in a parse table or is encoded directly in a recursive-descent program.)

## Bottom-Up Parsing Intuition

| | | | |
|---|---|---|---|
| $T'$ | $\rightarrow$ | $T$ | (0) |
| $T$ | $\rightarrow$ | $R$ | (1) |
| $T$ | $\rightarrow$ | $aTc$ | (2) |
| $R$ | $\rightarrow$ | $\epsilon$ | (3) |
| $R$ | $\rightarrow$ | $bR$ | (4) |

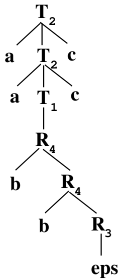


$T \rightarrow^2 aTc \rightarrow^2 aaTcc \rightarrow^1 aaRcc \rightarrow^4 aabRcc \rightarrow^4 aabbRcc \rightarrow^3 aabbcc$

- LL(1) parser works top-down by "guessing" the productions
- bottom-up: build syntax tree from the leaves; intuitively inverse derivation.

## Bottom-Up Parsing Intuition

$$\begin{array}{llll} T' & \rightarrow & T & (0) \\ T & \rightarrow & R & (1) \\ T & \rightarrow & aTc & (2) \\ R & \rightarrow & \epsilon & (3) \\ R & \rightarrow & bR & (4) \end{array}$$

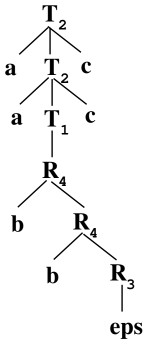


$T \rightarrow^2 aTc \rightarrow^2 aaTcc \rightarrow^1 aaRcc \rightarrow^4 aabRcc \rightarrow^4 aabbRcc \rightarrow^3 aabbcc$

- LL(1) parser works top-down by “guessing” the productions
- bottom-up: build syntax tree from the leaves; intuitively inverse derivation.

## Bottom-Up Parsing or Shift-Reduce Parsers

| Stack | Input | Action |
|---|---:|---|
| $\epsilon$ | aabbcc$ | shift |
| a | abbcc$ | shift |
| aa | bbcc$ | shift |
| aab | bcc$ | shift |
| aabb | cc$ | reduce 3 |
| aab$\underline{\text{bR}}$ | cc$ | reduce 4 |
| aa$\underline{\text{bR}}$ | cc$ | reduce 4 |
| aa$\underline{\text{R}}$ | cc$ | reduce 1 |
| aaT | cc$ | shift |
| a$\underline{\text{aTc}}$ | c$ | reduce 2 |
| aT | c$ | shift |
| $\underline{\text{aTc}}$ | $ | reduce 2 |
| $\underline{\text{T}}$ | $ | accept |



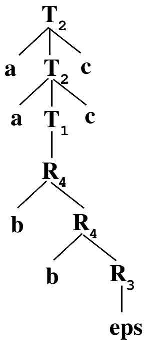
| | | | |
|---|---|---|---|
| $T'$ | $\rightarrow$ | $T$ | (0) |
| $T$ | $\rightarrow$ | $R$ | (1) |
| $T$ | $\rightarrow$ | $aTc$ | (2) |
| $R$ | $\rightarrow$ | $\epsilon$ | (3) |
| $R$ | $\rightarrow$ | $bR$ | (4) |

Questions:

- When to accept: use separate start production with $T'$
- When to shift or to reduce? Especially for $\epsilon$ productions.

## Shift-Reduce Parsers: Machine Idea

| Stack | Input | Action |
|---|---:|---|
| $\epsilon$ | aabbcc$ | shift |
| a | abbcc$ | shift |
| aa | bbcc$ | shift |
| aab | bcc$ | shift |
| aabb | cc$ | reduce 3 |
| aab$\underline{\text{bR}}$ | cc$ | reduce 4 |
| aa$\underline{\text{bR}}$ | cc$ | reduce 4 |
| aa$\underline{\text{R}}$ | cc$ | reduce 1 |
| aaT | cc$ | shift |
| a$\underline{\text{aTc}}$ | c$ | reduce 2 |
| aT | c$ | shift |
| $\underline{\text{aTc}}$ | $ | reduce 2 |
| $\underline{\text{T}}$ | $ | accept |



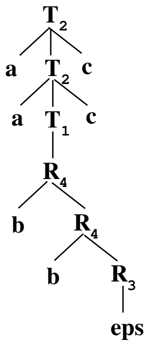
$$\begin{array}{llll} T' & \to & T & (0) \\ T & \to & R & (1) \\ T & \to & aTc & (2) \\ R & \to & \epsilon & (3) \\ R & \to & bR & (4) \end{array}$$

Conceptually:

- use a DFA that reads stack's content starting with bottom:
- If DFA in accepting state when reaching the top $\Rightarrow$ reduce by a production determined by **next input & the accepting state**.
- If not accepting then shift, i.e., transition on input symbol.

## Shift-Reduce Parsers: Machine Idea

| Stack | Input | Action |
|---|---:|---|
| $\epsilon$ | aabbcc\$ | shift |
| a | abbcc\$ | shift |
| aa | bbcc\$ | shift |
| aab | bcc\$ | shift |
| aabb | cc\$ | reduce 3 |
| $\text{aab}\underline{\text{bR}}$ | cc\$ | reduce 4 |
| $\text{aa}\underline{\text{bR}}$ | cc\$ | reduce 4 |
| $\text{aa}\underline{\text{R}}$ | cc\$ | reduce 1 |
| aaT | cc\$ | shift |
| $\text{a}\underline{\text{aTc}}$ | c\$ | reduce 2 |
| aT | c\$ | shift |
| $\underline{\text{aTc}}$ | \$ | reduce 2 |
| $\underline{\text{T}}$ | \$ | accept |



| | | | |
|---|---|---|---|
| $T'$ | $\rightarrow$ | $T$ | (0) |
| $T$ | $\rightarrow$ | $R$ | (1) |
| $T$ | $\rightarrow$ | $aTc$ | (2) |
| $R$ | $\rightarrow$ | $\epsilon$ | (3) |
| $R$ | $\rightarrow$ | $bR$ | (4) |

- DFA $\equiv$ one way street: on reduce need to read again the whole stack to find out where it was previously.
- Store on the stack, with each element the state of the DFA when it reads this element.
- we do not actually need to store the input symbol.

## Shift-Reduce Parsers: Machine Idea

| | a | b | c | $ | T | R |
|---|---|---|---|---|---|---|
| 0 | s3 | s4 | r3 | r3 | g1 | g2 |
| 1 | | | | a | | |
| 2 | | | r1 | r1 | | |
| 3 | s3 | s4 | r3 | r3 | g5 | g2 |
| 4 | | s4 | r3 | r3 | | g6 |
| 5 | | | s7 | | | |
| 6 | | | r4 | r4 | | |
| 7 | | | r2 | r2 | | |

$$\begin{array}{llll} T' & \rightarrow & T & (0) \\ T & \rightarrow & R & (1) \\ T & \rightarrow & aTc & (2) \\ R & \rightarrow & \epsilon & (3) \\ R & \rightarrow & bR & (4) \end{array}$$

| Stack | Input | Action |
|---|---|---|
| 0 | aabbcc$ | s3 |
| 03 | abbcc$ | s3 |
| 033 | bbcc$ | s4 |
| 0334 | bcc$ | s4 |
| 03344 | cc$ | r3; g6 |
| 033446 | cc$ | r4; g6 |
| 03346 | cc$ | r4; g2 |
| 0332 | cc$ | r1; g5 |
| 0335 | cc$ | s7 |
| 03357 | c$ | r2; g5 |
| 035 | c$ | s7 |
| 0357 | $ | r2; g1 |
| 01 | $ | a |

```
stack := empty(); push(0,stack); read(next);
while(true)
    case parseTable[top(stack), next] of
        shift s:    push(s, stack); read(next);

        reduce p:   n := left-hand side of p;
                    r := # symbols on right-hand side of p;
                    pop r elements from stack;
                    push( parseTable[top(stack),n], stack);

        accept:     success;     break;
        error:      reportError; break;
```

## Building The Parse Table: Algorithm

1 Build an NFA from the grammar: build an NFA for each graammar production and connect them with $\epsilon$ transitions.

2 Transform the NFA into an DFA, and record for each DFA state the set of NFA states it represents.

3 For any transition between two NFA states on a terminal $x$, denoted $s_1^x s_2$, **add action shift** $s_2$ to the parse-table entry cross-indexed by the **DFA state** corresponding to $s_1$ and token $x$

4 For any transition between two NFA states on a non-terminal $N$ denoted $s_1^N s_2$, **add action go** $s_2$ to the parse-table entry cross-indexed by the **DFA state** corresponding to $s_1$ and nonterminal $N$

5 Compute the $\textsc{Follow}$ sets for all nonterminals.

6 For every DFA state `k` that contains an accepting NFA state corresponding to *grammar-production number p*, i.e., $N \to \alpha$ $(p)$ in the grammar, add a **reduce** $p$ action to all parse-table entries cross-indexed by `k` and the tokens in $\textsc{Follow}(N)$.

## Build an NFA from each Grammar Production

- Make an NFA for each production, by treating both terminals and nonterminals as alphabet symbols.
- Accepting state is labeled with grammar production number!
- Transitions by terminals: will correspond to shift actions.
- Transitions on nonterminals: will correspond to go actions.

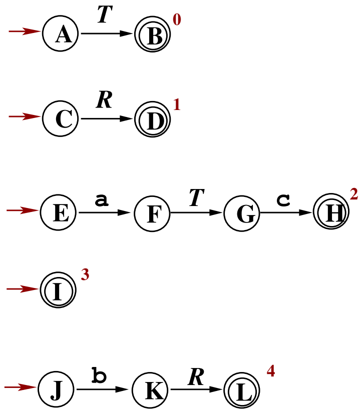
*T' –> T*

*T –> R*

*T –> aTc*

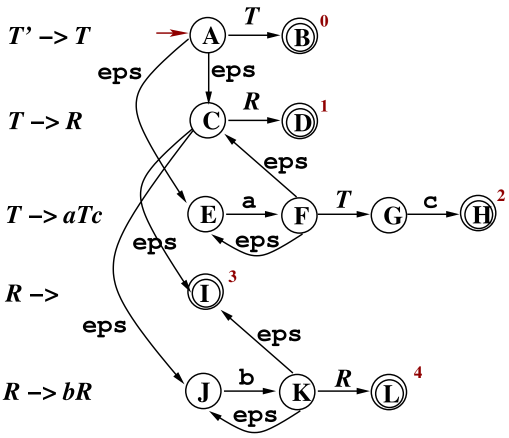
*R –>*

*R –> bR*



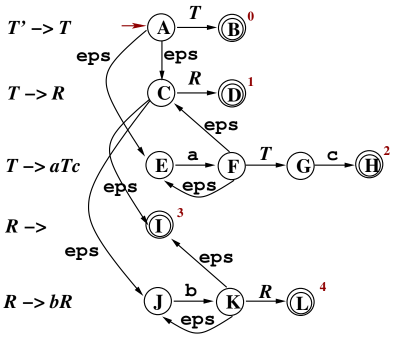
## Unified NFA: Connect NFAs with $\epsilon$ Transitions

- Make an NFA for each production, by treating both terminals and nonterminals as alphabet symbols.
- Accepting state is labeled with grammar production number!
- Transitions by terminals: shift actions
- Transitions on nonterminals: go actions
- Whenever a transition by a nonterminal $N$ is possible also insert an $\epsilon$-transition from the *first state* to the *initial states of the NFAs* corresponding to nonterminal $N$.
- Finally convert the resulted NFA to a DFA!



## Adding SHIFT and GO Actions to Parse Table

| | | a | b | c | $ | T | R |
|---|---|---|---|---|---|---|---|
| 0 | A,C,E,I,J | | | | | | |
| 1 | B | | | | | | |
| 2 | D | | | | | | |
| 3 | F,C,E,I,J | | | | | | |
| 4 | K,I,J | | | | | | |
| 5 | G | | | | | | |
| 6 | L | | | | | | |
| 7 | H | | | | | | |



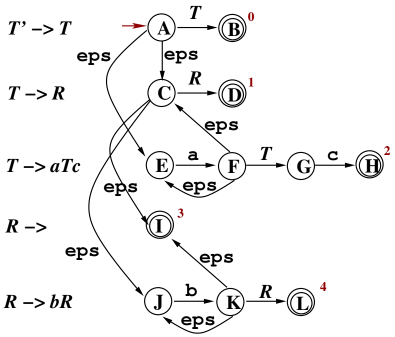
3 For any transition between two NFA states on a terminal $x$, denoted $s_1^x s_2$, **add action shift** $s_2$ to the parse-table entry cross-indexed by the **DFA state** corresponding to $s_1$ and token $x$.

4 For any transition between two NFA states on a non-terminal $N$ denoted $s_1^N s_2$, **add action go** $s_2$ to the parse-table entry cross-indexed by the **DFA state** corresponding to $s_1$ and nonterminal $N$

- for example for $E^a F$ add *shift* 3 to the entry cross-indexed by DFA state 0 and token a, because state $E$ belongs to DFA state 0 and state $F$ belongs to DFA state 3.
- for $A^T B$ add *go* 1 to the entry cross-indexed by DFA state 0 and nonterminal $N$, because state $A$ belongs to DFA state 0 and state $B$ belongs to DFA state 1.

## Adding REDUCE Actions to Parse Table

| | | a | b | c | $ | T | R |
|---|---|---|---|---|---|---|---|
| 0 | A,C,E,I,J | s3 | s4 | | | g1 | g2 |
| 1 | B | | | | a | | |
| 2 | D | | | | | | |
| 3 | F,C,E,I,J | s3 | s4 | | | g5 | g2 |
| 4 | K,I,J | | s4 | | | | g6 |
| 5 | G | | | s7 | | | |
| 6 | L | | | | | | |
| 7 | H | | | | | | |

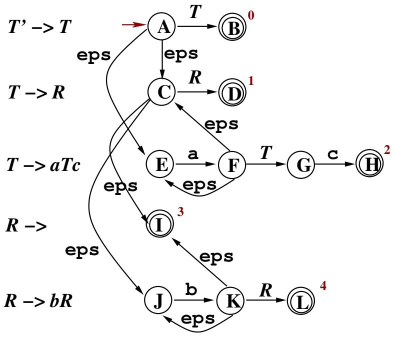


- Compute FOLLOW sets for each nonterminal:
- FOLLOW($T'$) $= \{\$\}$
  FOLLOW($T$) $= \{c, \$\}$
  FOLLOW($R$) $= \{c, \$\}$

- For every DFA state `k` that contains an accepting NFA state corresponding to *grammar-production number p*, i.e., $N \rightarrow \alpha$ ($p$) in the grammar, add a **reduce** $p$ action to all parse-table entries cross-indexed by `k` and the tokens in FOLLOW($N$).
- For example, for NFA final state $H$ corresponding to grammar production $T \rightarrow aTc$ `(2)`, **add actions reduce 2** to entries cross-indexed by DFA state 7 (to which $H$ belongs) and tokens $\{c, \$\}$ $=$ FOLLOW($T$).
- Reduction on state `0` is written accept (necessarily on $)!

## Completed SLR Parse Table

|   |   | a | b | c | $ | T | R |
|---|---|---|---|---|---|---|---|
| 0 | A,C,E,I,J | s3 | s4 | r3 | r3 | g1 | g2 |
| 1 | B |  |  |  | a |  |  |
| 2 | D |  |  | r1 | r1 |  |  |
| 3 | F,C,E,I,J | s3 | s4 | r3 | r3 | g5 | g2 |
| 4 | K,I,J |  | s4 | r3 | r3 |  | g6 |
| 5 | G |  |  | s7 |  |  |  |
| 6 | L |  |  | r4 | r4 |  |  |
| 7 | H |  |  | r2 | r2 |  |  |



Reduction on state 0 is written **accept** (a) because it signals that the whole word has been accepted by the start symbol of the grammar!