04 - Testing and Manipulating Grammars

Dr. Robert Lowe

Division of Mathematics and Computer Science
Maryville College

Outline

Outline

Sample Grammar G

For this discussion, we will be using the following grammar (found on page 39 of your textbook):

$$
\begin{aligned}
S &\rightarrow E \\
E &\rightarrow T \mid E + T \\
T &\rightarrow F \mid T * F \\
F &\rightarrow U \mid (E) \\
U &\rightarrow 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9
\end{aligned}
$$

General Top Down Parsing

1. Look at the next symbol of input. This is the **target** symbol.

General Top Down Parsing

1. Look at the next symbol of input. This is the **target** symbol.
2. Expand the next non-terminal in the sentence.

General Top Down Parsing

1. Look at the next symbol of input. This is the **target** symbol.
2. Expand the next non-terminal in the sentence.
3. If the target symbol does not match, backtrack and select a different non-terminal.

General Top Down Parsing

1. Look at the next symbol of input. This is the **target** symbol.
2. Expand the next non-terminal in the sentence.
3. If the target symbol does not match, backtrack and select a different non-terminal.
4. Keep repeating the process until there are either no non-terminal candidates or until there are no non-terminals left in the sentence.

The Backtracking Problem

- Even with the best of luck, a backtracking parser would be exponential in runtime!

The Backtracking Problem

- Even with the best of luck, a backtracking parser would be exponential in runtime!
- A left-recursive grammar could lead to an infinite number of candidates.

The Backtracking Problem

- Even with the best of luck, a backtracking parser would be exponential in runtime!
- A left-recursive grammar could lead to an infinite number of candidates.
- Recall that the sample grammar in the textbook has the rule $E \rightarrow E + T$

The Backtracking Problem

- Even with the best of luck, a backtracking parser would be exponential in runtime!
- A left-recursive grammar could lead to an infinite number of candidates.
- Recall that the sample grammar in the textbook has the rule $E \rightarrow E + T$
- Consider the following expansion for the grammar from the textbook:

The Backtracking Problem

- Even with the best of luck, a backtracking parser would be exponential in runtime!
- A left-recursive grammar could lead to an infinite number of candidates.
- Recall that the sample grammar in the textbook has the rule $E \rightarrow E + T$
- Consider the following expansion for the grammar from the textbook:

$$E + T \qquad\qquad 1 + 2 * 3$$

The Backtracking Problem

- Even with the best of luck, a backtracking parser would be exponential in runtime!
- A left-recursive grammar could lead to an infinite number of candidates.
- Recall that the sample grammar in the textbook has the rule $E \rightarrow E + T$
- Consider the following expansion for the grammar from the textbook:

$$\begin{array}{ll} E + T & 1 + 2 * 3 \\ E + T + T & 1 + 2 * 3 \end{array}$$

The Backtracking Problem

- Even with the best of luck, a backtracking parser would be exponential in runtime!
- A left-recursive grammar could lead to an infinite number of candidates.
- Recall that the sample grammar in the textbook has the rule $E \rightarrow E + T$
- Consider the following expansion for the grammar from the textbook:

$$\begin{array}{ll} E + T & 1 + 2 * 3 \\ E + T + T & 1 + 2 * 3 \\ E + T + T + T & 1 + 2 * 3 \end{array}$$

The Backtracking Problem

- Even with the best of luck, a backtracking parser would be exponential in runtime!
- A left-recursive grammar could lead to an infinite number of candidates.
- Recall that the sample grammar in the textbook has the rule $E \rightarrow E + T$
- Consider the following expansion for the grammar from the textbook:

$$\begin{array}{ll} E + T & 1 + 2 * 3 \\ E + T + T & 1 + 2 * 3 \\ E + T + T + T & 1 + 2 * 3 \\ E + T + T + T + T & 1 + 2 * 3 \end{array}$$

The Backtracking Problem

- Even with the best of luck, a backtracking parser would be exponential in runtime!
- A left-recursive grammar could lead to an infinite number of candidates.
- Recall that the sample grammar in the textbook has the rule $E \to E + T$
- Consider the following expansion for the grammar from the textbook:

$$\begin{array}{ll} E + T & 1 + 2 * 3 \\ E + T + T & 1 + 2 * 3 \\ E + T + T + T & 1 + 2 * 3 \\ E + T + T + T + T & 1 + 2 * 3 \\ \ldots & \end{array}$$

Ordering Recursion

- Left recursion causes problems in candidate expansions.

Ordering Recursion

- Left recursion causes problems in candidate expansions.
- Perhaps we could organize a grammar to mitigate the expansion problem.

Ordering Recursion

- Left recursion causes problems in candidate expansions.
- Perhaps we could organize a grammar to mitigate the expansion problem.
- If we move left recursive choices to the end, maybe this would fix it!

Ordering Recursion

- Left recursion causes problems in candidate expansions.
- Perhaps we could organize a grammar to mitigate the expansion problem.
- If we move left recursive choices to the end, maybe this would fix it!
- What if we took the grammar G and imposed the order $\langle T, F, U \rangle$ on expansions?

Ordering Recursion

- Left recursion causes problems in candidate expansions.
- Perhaps we could organize a grammar to mitigate the expansion problem.
- If we move left recursive choices to the end, maybe this would fix it!
- What if we took the grammar G and imposed the order $\langle T, F, U \rangle$ on expansions?

$$T \qquad\qquad 1 + 2 * 3$$

Ordering Recursion

- Left recursion causes problems in candidate expansions.
- Perhaps we could organize a grammar to mitigate the expansion problem.
- If we move left recursive choices to the end, maybe this would fix it!
- What if we took the grammar G and imposed the order $\langle T, F, U \rangle$ on expansions?

$$\begin{array}{ll} T & 1 + 2 * 3 \\ F & 1 + 2 * 3 \end{array}$$

Ordering Recursion

- Left recursion causes problems in candidate expansions.
- Perhaps we could organize a grammar to mitigate the expansion problem.
- If we move left recursive choices to the end, maybe this would fix it!
- What if we took the grammar G and imposed the order $\langle T, F, U \rangle$ on expansions?

T	$1 + 2 * 3$
F	$1 + 2 * 3$
U	$1 + 2 * 3$

Ordering Recursion

- Left recursion causes problems in candidate expansions.
- Perhaps we could organize a grammar to mitigate the expansion problem.
- If we move left recursive choices to the end, maybe this would fix it!
- What if we took the grammar G and imposed the order $\langle T, F, U \rangle$ on expansions?

T	$1 + 2 * 3$
F	$1 + 2 * 3$
U	$1 + 2 * 3$
1	$1 + 2 * 3$

Ordering Recursion

- Left recursion causes problems in candidate expansions.
- Perhaps we could organize a grammar to mitigate the expansion problem.
- If we move left recursive choices to the end, maybe this would fix it!
- What if we took the grammar G and imposed the order $\langle T, F, U\rangle$ on expansions?

$$\begin{array}{ll} T & 1 + 2 * 3 \\ F & 1 + 2 * 3 \\ U & 1 + 2 * 3 \\ 1 & 1 + 2 * 3 \\ \lambda & +2 * 3 \end{array}$$

Ordering Recursion

- Left recursion causes problems in candidate expansions.
- Perhaps we could organize a grammar to mitigate the expansion problem.
- If we move left recursive choices to the end, maybe this would fix it!
- What if we took the grammar G and imposed the order $\langle T, F, U \rangle$ on expansions?

T	$1 + 2 * 3$
F	$1 + 2 * 3$
U	$1 + 2 * 3$
1	$1 + 2 * 3$
λ	$+2 * 3$
Mismatch! Backtrack!	

Ordering Recursion (Continued)

$T * F$ $\qquad\qquad$ $1 + 2 * 3$

Ordering Recursion (Continued)

$T * F$	$1 + 2 * 3$
$F * F$	$1 + 2 * 3$

Ordering Recursion (Continued)

$T * F$	$1 + 2 * 3$
$F * F$	$1 + 2 * 3$
$U * F$	$1 + 2 * 3$

Ordering Recursion (Continued)

$T * F$	$1 + 2 * 3$
$F * F$	$1 + 2 * 3$
$U * F$	$1 + 2 * 3$
$1 * F$	$1 + 2 * 3$

Ordering Recursion (Continued)

$T * F$	$1 + 2 * 3$
$F * F$	$1 + 2 * 3$
$U * F$	$1 + 2 * 3$
$1 * F$	$1 + 2 * 3$

Mismatch! Backtrack!

Ordering Recursion (Continued)

$T * F$	$1 + 2 * 3$
$F * F$	$1 + 2 * 3$
$U * F$	$1 + 2 * 3$
$1 * F$	$1 + 2 * 3$
Mismatch! Backtrack!	
$T * F * F$	$1 + 2 * 3$

Ordering Recursion (Continued)

$T * F$	$1 + 2 * 3$
$F * F$	$1 + 2 * 3$
$U * F$	$1 + 2 * 3$
$1 * F$	$1 + 2 * 3$
Mismatch! Backtrack!	
$T * F * F$	$1 + 2 * 3$
$\ldots$	

Ordering Recursion (Continued)

$T * F$	$1 + 2 * 3$
$F * F$	$1 + 2 * 3$
$U * F$	$1 + 2 * 3$
$1 * F$	$1 + 2 * 3$
Mismatch! Backtrack!	
$T * F * F$	$1 + 2 * 3$
$\ldots$	
And there's the loop again...	

Outline

1. Grammars and Recursion
2. LL(1) Grammars
3. Manipulating Grammars

Deterministic Parsing

- Backtracking is parsing by “brute force”.

Deterministic Parsing

- Backtracking is parsing by “brute force”.
- Backtracking essentially explores every possible production, searching for a match.

Deterministic Parsing

- Backtracking is parsing by “brute force”.
- Backtracking essentially explores every possible production, searching for a match.
- Generally, we want parse times to be proportional to the size of the input, not exponential.

Deterministic Parsing

- Backtracking is parsing by “brute force”.
- Backtracking essentially explores every possible production, searching for a match.
- Generally, we want parse times to be proportional to the size of the input, not exponential.
- Undoing parsing is difficult!

Deterministic Parsing

- Backtracking is parsing by “brute force”.
- Backtracking essentially explores every possible production, searching for a match.
- Generally, we want parse times to be proportional to the size of the input, not exponential.
- Undoing parsing is difficult!
- We need some way to determine what production we must have based on the symbols being examined.

LL(k) Grammars

- Instead of guessing and checking, we maintain a buffer of terminals.

LL(k) Grammars

- Instead of guessing and checking, we maintain a buffer of terminals.
- If a grammar is decidable using k terminals, we call this a k-lookahead grammar.

LL(k) Grammars

- Instead of guessing and checking, we maintain a buffer of terminals.
- If a grammar is decidable using k terminals, we call this a k-lookahead grammar.
- We can further classify the grammar by its scanning order and which production it expands first.

$\mathrm{LL}(k)$ Grammars

- Instead of guessing and checking, we maintain a buffer of terminals.
- If a grammar is decidable using k terminals, we call this a k-lookahead grammar.
- We can further classify the grammar by its scanning order and which production it expands first.
- An $\mathrm{LL}(k)$ grammar is a grammar that is scanned from left to right and expands the left most derivation.

$\mathrm{LL}(k)$ Grammars

- Instead of guessing and checking, we maintain a buffer of terminals.
- If a grammar is decidable using k terminals, we call this a k-lookahead grammar.
- We can further classify the grammar by its scanning order and which production it expands first.
- An $\mathrm{LL}(k)$ grammar is a grammar that is scanned from left to right and expands the left most derivation.
- $\mathrm{RL}(k)$ scans input from right to left, expanding left-most derivations.

$\mathrm{LL}(k)$ Grammars

- Instead of guessing and checking, we maintain a buffer of terminals.
- If a grammar is decidable using k terminals, we call this a k-lookahead grammar.
- We can further classify the grammar by its scanning order and which production it expands first.
- An $\mathrm{LL}(k)$ grammar is a grammar that is scanned from left to right and expands the left most derivation.
- $\mathrm{RL}(k)$ scans input from right to left, expanding left-most derivations.
- $\mathrm{LR}(k)$ scans from left to right, expanding left-most derivations.

$\mathrm{LL}(k)$ Grammars

- Instead of guessing and checking, we maintain a buffer of terminals.
- If a grammar is decidable using k terminals, we call this a k-lookahead grammar.
- We can further classify the grammar by its scanning order and which production it expands first.
- An $\mathrm{LL}(k)$ grammar is a grammar that is scanned from left to right and expands the left most derivation.
- $\mathrm{RL}(k)$ scans input from right to left, expanding left-most derivations.
- $\mathrm{LR}(k)$ scans from left to right, expanding left-most derivations.
- All of the above have a look-ahead buffer of k terminals.

$\mathrm{LL}(k)$ Grammars

- Instead of guessing and checking, we maintain a buffer of terminals.
- If a grammar is decidable using k terminals, we call this a k-lookahead grammar.
- We can further classify the grammar by its scanning order and which production it expands first.
- An $\mathrm{LL}(k)$ grammar is a grammar that is scanned from left to right and expands the left most derivation.
- $\mathrm{RL}(k)$ scans input from right to left, expanding left-most derivations.
- $\mathrm{LR}(k)$ scans from left to right, expanding left-most derivations.
- All of the above have a look-ahead buffer of k terminals.
- We are really interested in $\mathrm{LL}(1)$ grammars.

Defining $\mathrm{LL}(1)$ Grammars

- Suppose we have a target expansion $A \rightarrow \alpha_1|\alpha_2|\dots|\alpha_n$

Defining LL(1) Grammars

- Suppose we have a target expansion $A \rightarrow \alpha_1 | \alpha_2 | \ldots | \alpha_n$
- We must be able to select α_i by looking at the next symbol.

Defining LL(1) Grammars

- Suppose we have a target expansion $A \to \alpha_1|\alpha_2|\dots|\alpha_n$
- We must be able to select α_i by looking at the next symbol.
- For each production, we must have a disjoin **director set** $D(A \to \alpha_i)$.

Defining $\mathrm{LL}(1)$ Grammars

- Suppose we have a target expansion $A \rightarrow \alpha_1|\alpha_2|\ldots|\alpha_n$
- We must be able to select α_i by looking at the next symbol.
- For each production, we must have a disjoin **director set** $D(A \rightarrow \alpha_i)$.
- For lookup buffer s, $A \rightarrow \alpha_i$ iff $s \in D(A \rightarrow \alpha_i)$.

Defining $\mathrm{LL}(1)$ Grammars

- Suppose we have a target expansion $A \rightarrow \alpha_1|\alpha_2|\ldots|\alpha_n$
- We must be able to select α_i by looking at the next symbol.
- For each production, we must have a disjoin **director set** $D(A \rightarrow \alpha_i)$.
- For lookup buffer s, $A \rightarrow \alpha_i$ iff $s \in D(A \rightarrow \alpha_i)$.
- We can also have a set of symbols which immediately identify as an error if they are encountered.

Calculating Director Sets

- If $\alpha_i \overset{*}{\Rightarrow} t\gamma$ for some terminal t

Calculating Director Sets

- If $\alpha_i \stackrel{*}{\Rightarrow} t\gamma$ for some terminal t
- Then $t \in D(A \rightarrow \alpha_i)$

Calculating Director Sets

- If $\alpha_i \stackrel{*}{\Rightarrow} t\gamma$ for some terminal t
- Then $t \in D(A \rightarrow \alpha_i)$
- Because $A \stackrel{+}{\Rightarrow} t\gamma$ is a valid derivation.

Calculating Director Sets

- If $\alpha_i \stackrel{*}{\Rightarrow} t\gamma$ for some terminal t
- Then $t \in D(A \rightarrow \alpha_i)$
- Because $A \stackrel{+}{\Rightarrow} t\gamma$ is a valid derivation.
- Let $<<$ be an operator over $(N \cup T)$ such that $\beta << \alpha \iff \exists \alpha \rightarrow \beta$

Calculating Director Sets

- If $\alpha_i \overset{*}{\Rightarrow} t\gamma$ for some terminal t
- Then $t \in D(A \rightarrow \alpha_i)$
- Because $A \overset{+}{\Rightarrow} t\gamma$ is a valid derivation.
- Let $<<$ be an operator over $(N \cup T)$ such that $\beta << \alpha \iff \exists \alpha \rightarrow \beta$
- The reflexive transitive closure $<<^*$ is therefore the "Can Start" relation

Calculating Director Sets

- If $\alpha_i \overset{*}{\Rightarrow} t\gamma$ for some terminal t
- Then $t \in D(A \rightarrow \alpha_i)$
- Because $A \overset{+}{\Rightarrow} t\gamma$ is a valid derivation.
- Let $<<$ be an operator over $(N \cup T)$ such that $\beta << \alpha \iff \exists \alpha \rightarrow \beta$
- The reflexive transitive closure $<<^*$ is therefore the "Can Start" relation
- The **start set** is $\text{START}(\alpha) = \beta : \beta <<^* \alpha$

Calculating Director Sets

- If $\alpha_i \stackrel{*}{\Rightarrow} t\gamma$ for some terminal t
- Then $t \in D(A \to \alpha_i)$
- Because $A \stackrel{+}{\Rightarrow} t\gamma$ is a valid derivation.
- Let $<<$ be an operator over $(N \cup T)$ such that $\beta << \alpha \iff \exists \alpha \to \beta$
- The reflexive transitive closure $<<^*$ is therefore the "Can Start" relation
- The **start set** is $\mathrm{START}(\alpha) = \beta : \beta <<^* \alpha$
- Considering $\alpha_i = \beta_1 \beta_2 \ldots \beta_r$ then $t \in \mathrm{START}(\beta_1) \implies t \in D(A \to \alpha_i)$

Start Sets of G

$$\text{START}(U) = \{U, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$$

Start Sets of G

$$\text{START}(U) = \{U, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$$
$$\text{START}(F) = \{\{F, (\} \cup \text{START}(U)\}$$

Start Sets of G

$$
\begin{aligned}
\mathrm{START}(U) &= \{U, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9\} \\
\mathrm{START}(F) &= \{\{F, (\} \cup \mathrm{START}(U)\} \\
\mathrm{START}(T) &= \{\{T\} \cup \mathrm{START}(F)\}
\end{aligned}
$$

Start Sets of G

$$
\begin{aligned}
\mathrm{START}(U) &= \{U, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9\} \\
\mathrm{START}(F) &= \{\{F, (\} \cup \mathrm{START}(U)\} \\
\mathrm{START}(T) &= \{\{T\} \cup \mathrm{START}(F)\} \\
\mathrm{START}(E) &= \{\{E\} \cup \mathrm{START}(T)\}
\end{aligned}
$$

Start Sets of G

$$
\begin{aligned}
\mathrm{START}(U) &= \{U, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9\} \\
\mathrm{START}(F) &= \{\{F, (\} \cup \mathrm{START}(U)\} \\
\mathrm{START}(T) &= \{\{T\} \cup \mathrm{START}(F)\} \\
\mathrm{START}(E) &= \{\{E\} \cup \mathrm{START}(T)\} \\
\mathrm{START}(S) &= \{\{S\} \cup \mathrm{START}(E)\}
\end{aligned}
$$

Start Sets of G

$$
\begin{aligned}
\mathrm{START}(U) &= \{U, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9\} \\
\mathrm{START}(F) &= \{\{F, (\} \cup \mathrm{START}(U)\} \\
\mathrm{START}(T) &= \{\{T\} \cup \mathrm{START}(F)\} \\
\mathrm{START}(E) &= \{\{E\} \cup \mathrm{START}(T)\} \\
\mathrm{START}(S) &= \{\{S\} \cup \mathrm{START}(E)\}
\end{aligned}
$$

- Is G an $\mathrm{LL}(1)$ grammar?

Start Sets of G

$$\begin{aligned}
\mathrm{START}(U) &= \{U, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9\} \\
\mathrm{START}(F) &= \{\{F, (\} \cup \mathrm{START}(U)\} \\
\mathrm{START}(T) &= \{\{T\} \cup \mathrm{START}(F)\} \\
\mathrm{START}(E) &= \{\{E\} \cup \mathrm{START}(T)\} \\
\mathrm{START}(S) &= \{\{S\} \cup \mathrm{START}(E)\}
\end{aligned}$$

- Is G an $\mathrm{LL}(1)$ grammar?
- NO! In fact, no grammar containing left-recursive rules is $\mathrm{LL}(1)$!

Start Sets of G

$$
\begin{aligned}
\mathrm{START}(U) &= \{U, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9\} \\
\mathrm{START}(F) &= \{\{F, (\} \cup \mathrm{START}(U)\} \\
\mathrm{START}(T) &= \{\{T\} \cup \mathrm{START}(F)\} \\
\mathrm{START}(E) &= \{\{E\} \cup \mathrm{START}(T)\} \\
\mathrm{START}(S) &= \{\{S\} \cup \mathrm{START}(E)\}
\end{aligned}
$$

- Is G an $\mathrm{LL}(1)$ grammar?
- NO! In fact, no grammar containing left-recursive rules is $\mathrm{LL}(1)$!
- $D(A \to A\gamma) \subseteq \mathrm{START}(A)$

The First Function

- Are the start set symbols the only ones in $D(A \to \alpha_i)$?

The First Function

- Are the start set symbols the only ones in $D(A \to \alpha_i)$?
- Extend the function START to FIRST which operates on whole strings $\beta_1\beta_2\ldots\beta_r$ over $(N \cup T)^*$ and finds terminals which can start the string.

The First Function

- Are the start set symbols the only ones in $D(A \to \alpha_i)$?
- Extend the function START to FIRST which operates on whole strings $\beta_1\beta_2\ldots\beta_r$ over $(N \cup T)^*$ and finds terminals which can start the string.
- This function is defined recursively (where $\gamma \in (N \cup T)$ and $\delta \in (N \cup T)^*$):

The First Function

- Are the start set symbols the only ones in $D(A \to \alpha_i)$?
- Extend the function START to FIRST which operates on whole strings $\beta_1\beta_2\ldots\beta_r$ over $(N \cup T)^*$ and finds terminals which can start the string.
- This function is defined recursively (where $\gamma \in (N \cup T)$ and $\delta \in (N \cup T)^*$):

$$\text{FIRST}(\lambda) = \emptyset$$

The First Function

- Are the start set symbols the only ones in $D(A \to \alpha_i)$?
- Extend the function START to FIRST which operates on whole strings $\beta_1 \beta_2 \dots \beta_r$ over $(N \cup T)^*$ and finds terminals which can start the string.
- This function is defined recursively (where $\gamma \in (N \cup T)$ and $\delta \in (N \cup T)^*$):

$$\text{FIRST}(\lambda) = \emptyset$$
$$\text{FIRST}(\gamma\delta) = \text{terminals of START}(\gamma) \cup \text{FIRST}(\delta) \quad \text{if } \gamma \stackrel{*}{\Rightarrow} \lambda$$

The First Function

- Are the start set symbols the only ones in $D(A \to \alpha_i)$?
- Extend the function START to FIRST which operates on whole strings $\beta_1\beta_2\ldots\beta_r$ over $(N \cup T)^*$ and finds terminals which can start the string.
- This function is defined recursively (where $\gamma \in (N \cup T)$ and $\delta \in (N \cup T)^*$):

$$\begin{aligned}
\text{FIRST}(\lambda) &= \emptyset \\
\text{FIRST}(\gamma\delta) &= \text{terminals of START}(\gamma) \cup \text{FIRST}(\delta) && \text{if } \gamma \stackrel{*}{\Rightarrow} \lambda \\
\text{FIRST}(\gamma\delta) &= \text{terminals of START}(\gamma) && \text{o.w.}
\end{aligned}$$

The EMPTY Property

- We need to find if $\gamma \stackrel{*}{\Rightarrow}$ exists.

The EMPTY Property

- We need to find if $\gamma \overset{*}{\Rightarrow}$ exists.
- If ti does we say $\text{EMPTY}(\gamma)$ is true.

The EMPTY Property

- We need to find if $\gamma \overset{*}{\Rightarrow}$ exists.
- If ti does we say $\mathrm{EMPTY}(\gamma)$ is true.
- the EMPTY property can be defined as follows:

The EMPTY Property

- We need to find if $\gamma \stackrel{*}{\Rightarrow}$ exists.
- If ti does we say $\mathrm{EMPTY}(\gamma)$ is true.
- the EMPTY property can be defined as follows:
    1. If $\gamma \in T$ then $\mathrm{EMPTY}(\gamma) = \mathrm{false}$

The EMPTY Property

- We need to find if $\gamma \stackrel{*}{\Rightarrow}$ exists.
- If ti does we say $\mathrm{EMPTY}(\gamma)$ is true.
- the EMPTY property can be defined as follows:
  1. If $\gamma \in T$ then $\mathrm{EMPTY}(\gamma) = \mathrm{false}$
  2. If $\gamma \in N$ then

The EMPTY Property

- We need to find if $\gamma \stackrel{*}{\Rightarrow}$ exists.
- If ti does we say $\mathrm{EMPTY}(\gamma)$ is true.
- the EMPTY property can be defined as follows:
  1. If $\gamma \in T$ then $\mathrm{EMPTY}(\gamma) = \mathrm{false}$
  2. If $\gamma \in N$ then
     1. If $\exists \gamma \to \lambda$ then $\mathrm{EMPTY}(\gamma) = \mathrm{true}$

The EMPTY Property

- We need to find if $\gamma \stackrel{*}{\Rightarrow}$ exists.
- If ti does we say $\mathrm{EMPTY}(\gamma)$ is true.
- the EMPTY property can be defined as follows:
  1. If $\gamma \in T$ then $\mathrm{EMPTY}(\gamma) = \mathrm{false}$
  2. If $\gamma \in N$ then
     1. If $\exists\gamma \rightarrow \lambda$ then $\mathrm{EMPTY}(\gamma) = \mathrm{true}$
     2. If $\exists\gamma \rightarrow \delta_1 \ldots \delta_k$ where $\forall 1 \leq i \leq k$ $\mathrm{EMPTY}(\delta_i) = \mathrm{true}$ then $\mathrm{EMPTY}(\gamma) = \mathrm{true}$

The EMPTY Property

- We need to find if $\gamma \stackrel{*}{\Rightarrow}$ exists.
- If ti does we say $\mathrm{EMPTY}(\gamma)$ is true.
- the EMPTY property can be defined as follows:
  1. If $\gamma \in T$ then $\mathrm{EMPTY}(\gamma) = \mathrm{false}$
  2. If $\gamma \in N$ then
     1. If $\exists\gamma \rightarrow \lambda$ then $\mathrm{EMPTY}(\gamma) = \mathrm{true}$
     2. If $\exists\gamma \rightarrow \delta_1 \ldots \delta_k$ where $\forall 1 \leq i \leq k$ $\mathrm{EMPTY}(\delta_i) = \mathrm{true}$ then $\mathrm{EMPTY}(\gamma) = \mathrm{true}$
     3. For all other γ, $\mathrm{EMPTY}(\gamma) = \mathrm{false}$

The EMPTY Property

- We need to find if $\gamma \stackrel{*}{\Rightarrow}$ exists.
- If ti does we say $\mathrm{EMPTY}(\gamma)$ is true.
- the EMPTY property can be defined as follows:
  1. If $\gamma \in T$ then $\mathrm{EMPTY}(\gamma) = \mathrm{false}$
  2. If $\gamma \in N$ then
     1. If $\exists \gamma \rightarrow \lambda$ then $\mathrm{EMPTY}(\gamma) = \mathrm{true}$
     2. If $\exists \gamma \rightarrow \delta_1 \ldots \delta_k$ where $\forall 1 \leq i \leq k$ $\mathrm{EMPTY}(\delta_i) = \mathrm{true}$ then $\mathrm{EMPTY}(\gamma) = \mathrm{true}$
     3. For all other γ, $\mathrm{EMPTY}(\gamma) = \mathrm{false}$
- Let's calculate the FIRST for the productions in G.

The FOLLOW Function

- Suppose we have a terminal t in our look-ahead buffer.

The FOLLOW Function

- Suppose we have a terminal *t* in our look-ahead buffer.
- When $\alpha_i \stackrel{*}{\Rightarrow}$, A production *A* is the correct choice for the parser if *t* can follow *A*.

The FOLLOW Function

- Suppose we have a terminal *t* in our look-ahead buffer.
- When $\alpha_i \stackrel{*}{\Rightarrow}$, A production *A* is the correct choice for the parser if *t* can follow *A*.
- We calculate FOLLOW like this:

The FOLLOW Function

- Suppose we have a terminal *t* in our look-ahead buffer.
- When $\alpha_i \stackrel{*}{\Rightarrow}$, A production *A* is the correct choice for the parser if *t* can follow *A*.
- We calculate FOLLOW like this:
    1. First, caclulate FINISH (the set of all terminals that can end the prouction)

The FOLLOW Function

- Suppose we have a terminal *t* in our look-ahead buffer.
- When $\alpha_i \stackrel{*}{\Rightarrow}$, A production *A* is the correct choice for the parser if *t* can follow *A*.
- We calculate FOLLOW like this:
  1. First, caclulate FINISH (the set of all terminals that can end the prouction)
  2. Next, we add the $\mathrm{START}(\beta_{\mathrm{i}})$ for all β_i that can follow *A*

The FOLLOW Function

- Suppose we have a terminal *t* in our look-ahead buffer.
- When $\alpha_i \overset{*}{\Rightarrow}$, A production *A* is the correct choice for the parser if *t* can follow *A*.
- We calculate FOLLOW like this:
  1. First, caclulate FINISH (the set of all terminals that can end the prouction)
  2. Next, we add the $\mathrm{START}(\beta_\mathrm{i})$ for all β_i that can follow *A*
- Let's do this for *G*!

The FOLLOW Function

- Suppose we have a terminal *t* in our look-ahead buffer.
- When $\alpha_i \stackrel{*}{\Rightarrow}$, A production *A* is the correct choice for the parser if *t* can follow *A*.
- We calculate FOLLOW like this:
  1. First, caclulate FINISH (the set of all terminals that can end the prouction)
  2. Next, we add the $\mathrm{START}(\beta_\mathrm{i})$ for all β_i that can follow *A*
- Let's do this for *G*!
- What are the complete director sets for *G*?

Outline