Parser: Syntactic Analysis

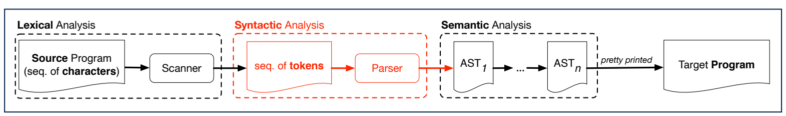
Readings: EAC2 Chapter 3

EECS4302 M:
Compilers and Interpreters
Winter 2020

CHEN-WEI WANG

Parser in Context

- Recall:



- Treats the input programas as a ***a sequence of classified tokens/words***
- Applies rules ***parsing*** token sequences as ***abstract syntax trees (ASTs)*** [***syntactic*** analysis]
- Upon termination:
  - Reports token sequences not derivable as ASTs
  - Produces an ***AST***
- No longer considers ***every character*** in input program.
- *Derivable* token sequences constitute a *context-free language (CFL)*.

Context-Free Languages: Introduction

- We have seen *regular languages*:
  - Can be described using *finite automata* or *regular expressions*.
  - Satisfy the *pumping lemma*.
- Languages with a *recursive* structure are provably *non-regular*.
  e.g., $\{0^n 1^n \mid n \geq 0\}$
- *Context-free grammars (CFG's)* are used to describe strings that can be generated in a *recursive* fashion.
- *Context-free languages (CFL's)* are:
  - Languages that can be described using CFG's.
  - A proper superset of the set of regular languages.

CFG: Example (1.1)

- The language that we previously proved as *non-regular*

$$\{0^n \# 1^n \mid n \geq 0\}$$

can be described using the following *grammar*:

$$\begin{array}{lcl} A & \rightarrow & 0A1 \\ A & \rightarrow & B \\ B & \rightarrow & \# \end{array}$$

- A grammar contains a collection of *substitution* or *production* rules, where:
  - A *terminal* is a word $w \in \Sigma^*$ (e.g., 0, 1, *etc.*).
  - A *variable* or *non-terminal* is a word $w \notin \Sigma^*$ (e.g., A, B, *etc.*).
  - A *start variable* occurs on the LHS of the topmost rule (e.g., A).

CFG: Example (1.2)

- Given a grammar, generate a string by:
  1. Write down the *start variable*.
  2. Choose a production rule where the *start variable* appears on the LHS of the arrow, and *substitute* it by the RHS.
  3. There are two cases of the re-written string:
     - **3.1** It contains *no* variables, then you are done.
     - **3.2** It contains *some* variables, then *substitute* each variable using the relevant *production rules*.
  4. Repeat Step 3.
- e.g., We can generate an *infinite* number of strings from

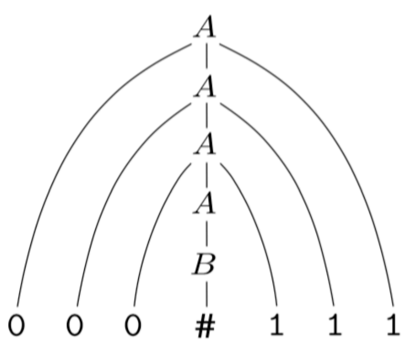
$$\begin{array}{lcl} A & \rightarrow & 0A1 \\ A & \rightarrow & B \\ B & \rightarrow & \# \end{array}$$

  - $A \Rightarrow B \Rightarrow \#$
  - $A \Rightarrow 0A1 \Rightarrow 0B1 \Rightarrow 0\#1$
  - $A \Rightarrow 0A1 \Rightarrow 00A11 \Rightarrow 00B11 \Rightarrow 00\#11$
  - ...

CFG: Example (1.2)

Given a CFG, the *derivation* of a string can be shown as a *parse tree*.

e.g., The derivation of $000\#111$ has the parse tree



CFG: Example (2)

Design a CFG for the following language:

$$\{w \mid w \in \{0,1\}^* \wedge w \texttt{ is a palidrome}\}$$

e.g., 00, 11, 0110, 1001, *etc.*

$$\begin{array}{lcl} P & \rightarrow & \epsilon \\ P & \rightarrow & 0 \\ P & \rightarrow & 1 \\ P & \rightarrow & 0P0 \\ P & \rightarrow & 1P1 \end{array}$$

CFG: Example (3)

Design a CFG for the following language:

$$\{ww^R \mid w \in \{0,1\}^*\}$$

e.g., 00, 11, 0110, *etc.*

$$\begin{array}{lcl} P & \rightarrow & \epsilon \\ P & \rightarrow & 0P0 \\ P & \rightarrow & 1P1 \end{array}$$

CFG: Example (4)

Design a CFG for the set of binary strings, where each block of 0's followed by at least as many 1's.

e.g., 000111, 0001111, *etc.*

- We use S to represent one such string, and A to represent each such block in S.

$$
\begin{array}{lcll}
S & \rightarrow & \epsilon & \{\texttt{BC of } S\} \\
S & \rightarrow & AS & \{\texttt{RC of } S\} \\
A & \rightarrow & \epsilon & \{\texttt{BC of } A\} \\
A & \rightarrow & 01 & \{\texttt{BC of } A\} \\
A & \rightarrow & 0A1 & \{\texttt{RC of } A\texttt{: equal 0's and 1's}\} \\
A & \rightarrow & A1 & \{\texttt{RC of } A\texttt{: more 1's}\}
\end{array}
$$

CFG: Example (5.1) Version 1

Design the grammar for the following small expression language, which supports:

- Arithmetic operations: +, −, ∗, /
- Relational operations: >, <, >=, <=, ==, /=
- Logical operations: `true`, `false`, `!`, `&&`, `||`, `=>`

  Start with the variable *Expression*.
- There are two possible versions:
  1. All operations are mixed together. [e.g., (1 + *true*)/*false*]
  2. Relevant operations are grouped together.

     Try both!

CFG: Example (5.2) Version 1

$$
\begin{array}{lcl}
\textit{Expression} & \rightarrow & \textit{IntegerConstant} \\
 & | & -\textit{IntegerConstant} \\
 & | & \textit{BooleanConstant} \\
 & | & \textit{BinaryOp} \\
 & | & \textit{UnaryOp} \\
 & | & (\ \textit{Expression}\ ) \\
\\
\textit{IntegerConstant} & \rightarrow & \textit{Digit} \\
 & | & \textit{Digit IntegerConstant} \\
\\
\textit{Digit} & \rightarrow & 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9 \\
\\
\textit{BooleanConstant} & \rightarrow & \texttt{TRUE} \\
 & | & \texttt{FALSE}
\end{array}
$$

CFG: Example (5.3) Version 1

$$
\begin{array}{lcl}
\textit{BinaryOp} & \rightarrow & \textit{Expression} + \textit{Expression} \\
 & | & \textit{Expression} - \textit{Expression} \\
 & | & \textit{Expression} * \textit{Expression} \\
 & | & \textit{Expression} \ / \ \textit{Expression} \\
 & | & \textit{Expression}\ \&\&\ \textit{Expression} \\
 & | & \textit{Expression}\ ||\ \textit{Expression} \\
 & | & \textit{Expression} => \textit{Expression} \\
 & | & \textit{Expression} == \textit{Expression} \\
 & | & \textit{Expression}\ /=\ \textit{Expression} \\
 & | & \textit{Expression} > \textit{Expression} \\
 & | & \textit{Expression} < \textit{Expression} \\
\\
\textit{UnaryOp} & \rightarrow & !\ \textit{Expression}
\end{array}
$$

CFG: Example (5.4) Version 1

However, Version 1 of CFG:

- ***Parses*** string that requires further *semantic analysis* (e.g., type checking):
  e.g., `3 => 6`
- Is *ambiguous*, meaning that a string may have more than one ways to interpret it.
  e.g., Draw the *parse tree(s)* for `3 * 5 + 4`

CFG: Example (5.5) Version 2

Expression	→	*ArithmeticOp*
	|	*RelationalOp*
	|	*LogicalOp*
	|	(*Expression*)
IntegerConstant	→	*Digit*
	|	*Digit IntegerConstant*
Digit	→	0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
BooleanConstant	→	`TRUE`
	|	`FALSE`

CFG: Example (5.6) Version 2

ArithmeticOp	→	*ArithmeticOp* + *ArithmeticOp*
	|	*ArithmeticOp* − *ArithmeticOp*
	|	*ArithmeticOp* * *ArithmeticOp*
	|	*ArithmeticOp* / *ArithmeticOp*
	|	(*ArithmeticOp*)
	|	*IntegerConstant*
	|	−*IntegerConstant*
RelationalOp	→	*ArithmeticOp* == *ArithmeticOp*
	|	*ArithmeticOp* /= *ArithmeticOp*
	|	*ArithmeticOp* > *ArithmeticOp*
	|	*ArithmeticOp* < *ArithmeticOp*
LogicalOp	→	*LogicalOp* && *LogicalOp*
	|	*LogicalOp* || *LogicalOp*
	|	*LogicalOp* => *LogicalOp*
	|	! *LogicalOp*
	|	(*LogicalOp*)
	|	*RelationalOp*
	|	*BooleanConstant*

CFG: Example (5.7) Version 2

However, Version 2 of CFG:

- Eliminates some cases for further semantic analysis:
  e.g., `(1 + 2) => (5 / 4)` [no parse tree]
- Still ***Parses*** string that might require further *semantic analysis*:
  e.g., `(1 + 2) / (5 - (2 + 3))`
- Is *ambiguous*, meaning that a string may have more than one ways to interpret it.
  e.g., Draw the *parse tree(s)* for `3 * 5 + 4`

CFG: Formal Definition (1)

- A *context-free grammar (CFG)* is a 4-tuple (V, Σ, R, S):
  - V is a finite set of *variables*.
  - Σ is a finite set of *terminals*. $[V \cap \Sigma = \varnothing]$
  - R is a finite set of *rules* s.t.

$$R \subseteq \{ v \rightarrow s \mid v \in V \land s \in (V \cup \Sigma)^* \}$$

  - $S \in V$ is is the *start variable*.
- Given strings $u, v, w \in (V \cup \Sigma)^*$, variable $A \in V$, and a rule $A \rightarrow w$:
  - $uAv \Rightarrow uwv$ menas that uAv *yields* uwv.
  - $u \overset{*}{\Rightarrow} v$ means that u *derives* v, if:
    - $u = v$; or
    - $u \Rightarrow u_1 \Rightarrow u_2 \Rightarrow \cdots \Rightarrow u_k \Rightarrow v$ [a yield sequence]
- Given a CFG $G = (V, \Sigma, R, S)$, the language of G

$$L(G) = \{ w \in \Sigma^* \mid S \overset{*}{\Rightarrow} w \}$$

CFG: Formal Definition (2): Example

- Design the CFG for strings of properly-nested parentheses.
  e.g., `()`, `()()`, `((()()))()`, *etc.*
  Present your answer in a formal manner.
- $G = (\{S\}, \{ (,) \}, R, S)$, where R is

$$S \rightarrow (\ S\ ) \mid SS \mid \epsilon$$

- Draw *parse trees* for the above three strings that G generates.

CFG: Formal Definition (3): Example

- Consider the grammar $G = (V, \Sigma, R, S)$:
  - R is

$$\begin{array}{lcl} Expr & \rightarrow & Expr + Term \\ & \mid & Term \\ Term & \rightarrow & Term * Factor \\ & \mid & Factor \\ Factor & \rightarrow & (Expr) \\ & \mid & \mathtt{a} \end{array}$$

  - $V = \{Expr, Term, Factor\}$
  - $\Sigma = \{\mathtt{a}, +, *, (,)\}$
  - $S = Expr$
- *Precedence* of operators $+$ and $*$ is embedded in the grammar.
  - "Plus" is specified at a **higher** level (*Expr*) than is "times" (*Term*).
  - Both operands of a multiplication (*Factor*) may be **parenthesized**.

Regular Expressions to CFG's

- Recall the semantics of regular expressions (assuming that we do not consider $\varnothing$):

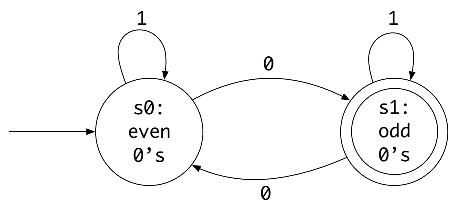
$$\begin{array}{lcl} L(\ \epsilon\ ) & = & \{\epsilon\} \\ L(\ a\ ) & = & \{a\} \\ L(\ E + F\ ) & = & L(E) \cup L(F) \\ L(\ EF\ ) & = & L(E)L(F) \\ L(\ E^*\ ) & = & (L(E))^* \\ L(\ (E)\ ) & = & L(E) \end{array}$$

- e.g., Grammar for $(00 + 1)^* + (11 + 0)^*$

$$\begin{array}{lcl} S & \rightarrow & A \mid B \\ A & \rightarrow & \epsilon \mid AC \\ C & \rightarrow & 00 \mid 1 \\ B & \rightarrow & \epsilon \mid BD \\ D & \rightarrow & 11 \mid 0 \end{array}$$

DFA to CFG's

- Given a DFA $M = (Q,\ \Sigma,\ \delta,\ q_0,\ F)$:
  - Make a variable R_i for each state $q_i \in Q$.
  - Make R_0 the start variable, where q_0 is the start state of M.
  - Add a rule $R_i \rightarrow aR_j$ to the grammar if $\delta(q_i, a) = q_j$.
  - Add a rule $R_i \rightarrow \epsilon$ if $q_i \in F$.
- e.g., Grammar for



$$\begin{array}{lcl} R_0 & \rightarrow & 1R_0 \mid 0R_1 \\ R_1 & \rightarrow & 0R_0 \mid 1R_1 \mid \epsilon \end{array}$$

CFG: Leftmost Derivations (1)

$$\begin{array}{lcl} Expr & \rightarrow & Expr\ +\ Term \mid Term \\ Term & \rightarrow & Term\ \star\ Factor \mid Factor \\ Factor & \rightarrow & (\ Expr\ ) \mid a \end{array}$$

- *Unique leftmost derivation* for the string `a + a * a`:

$$\begin{array}{lcl} Expr & \Rightarrow & Expr\ +\ Term \\ & \Rightarrow & Term\ +\ Term \\ & \Rightarrow & Factor\ +\ Term \\ & \Rightarrow & a\ +\ Term \\ & \Rightarrow & a\ +\ Term\ \star\ Factor \\ & \Rightarrow & a\ +\ Factor\ \star\ Factor \\ & \Rightarrow & a\ +\ a\ \star\ Factor \\ & \Rightarrow & a\ +\ a\ \star\ a \end{array}$$

- This *leftmost derivation* suggests that `a * a` is the right operand of `+`.

CFG: Rightmost Derivations (1)

$$\begin{array}{lcl} Expr & \rightarrow & Expr\ +\ Term \mid Term \\ Term & \rightarrow & Term\ \star\ Factor \mid Factor \\ Factor & \rightarrow & (\ Expr\ ) \mid a \end{array}$$

- *Unique rightmost derivation* for the string `a + a * a`:

$$\begin{array}{lcl} Expr & \Rightarrow & Expr\ +\ Term \\ & \Rightarrow & Expr\ +\ Term\ \star\ Factor \\ & \Rightarrow & Expr\ +\ Term\ \star\ a \\ & \Rightarrow & Expr\ +\ Factor\ \star\ a \\ & \Rightarrow & Expr\ +\ a\ \star\ a \\ & \Rightarrow & Term\ +\ a\ \star\ a \\ & \Rightarrow & Factor\ +\ a\ \star\ a \\ & \Rightarrow & a\ +\ a\ \star\ a \end{array}$$

- This *rightmost derivation* suggests that `a * a` is the right operand of `+`.

CFG: Leftmost Derivations (2)

$$\begin{array}{lcl} Expr & \rightarrow & Expr\ +\ Term \mid Term \\ Term & \rightarrow & Term\ \star\ Factor \mid Factor \\ Factor & \rightarrow & (\ Expr\ ) \mid a \end{array}$$

- *Unique leftmost derivation* for the string `(a + a) * a`:

$$\begin{array}{lcl} Expr & \Rightarrow & Term \\ & \Rightarrow & Term\ \star\ Factor \\ & \Rightarrow & Factor\ \star\ Factor \\ & \Rightarrow & (\ Expr\ )\ \star\ Factor \\ & \Rightarrow & (\ Expr\ +\ Term\ )\ \star\ Factor \\ & \Rightarrow & (\ Term\ +\ Term\ )\ \star\ Factor \\ & \Rightarrow & (\ Factor\ +\ Term\ )\ \star\ Factor \\ & \Rightarrow & (\ a\ +\ Term\ )\ \star\ Factor \\ & \Rightarrow & (\ a\ +\ Factor\ )\ \star\ Factor \\ & \Rightarrow & (\ a\ +\ a\ )\ \star\ Factor \\ & \Rightarrow & (\ a\ +\ a\ )\ \star\ a \end{array}$$

This *leftmost derivation* suggests that `(a + a)` is the left operand of `*`.

CFG: Rightmost Derivations (2)

$$
\begin{array}{lll}
Expr & \rightarrow & Expr + Term \mid Term \\
Term & \rightarrow & Term * Factor \mid Factor \\
Factor & \rightarrow & (Expr) \mid a
\end{array}
$$

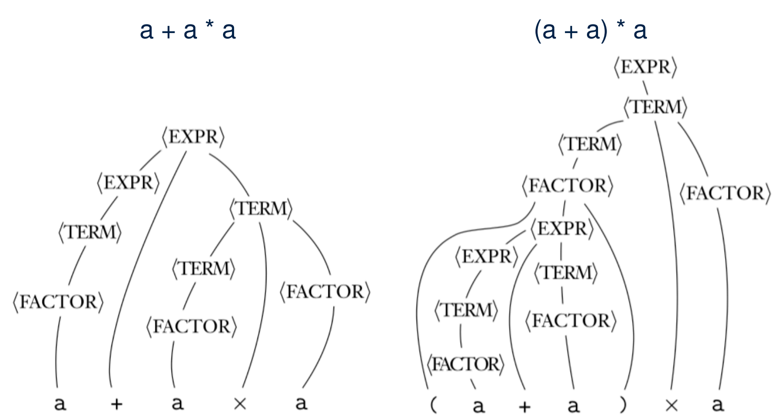
- *Unique rightmost derivation* for the string `(a + a) * a`:

$$
\begin{array}{lll}
Expr & \Rightarrow & Term \\
& \Rightarrow & Term * Factor \\
& \Rightarrow & Term * a \\
& \Rightarrow & Factor * a \\
& \Rightarrow & (Expr) * a \\
& \Rightarrow & (Expr + Term) * a \\
& \Rightarrow & (Expr + Factor) * a \\
& \Rightarrow & (Expr + a) * a \\
& \Rightarrow & (Term + a) * a \\
& \Rightarrow & (Factor + a) * a \\
& \Rightarrow & (a + a) * a
\end{array}
$$

This *rightmost derivation* suggests that `(a + a)` is the left operand of $*$.

CFG: Parse Trees vs. Derivations (1)

- *Parse trees* for (leftmost & rightmost) *derivations* of expressions:



- Orders in which derivations are performed are ***not*** reflected on parse trees.

CFG: Parse Trees vs. Derivations (2)

- A string $w \in \Sigma^*$ may have more than one *derivations*.

  Q: distinct *derivations* for $w \in \Sigma^* \Rightarrow$ distinct *parse trees* for w?

  A: Not in general $\because$ Derivations with ***distinct orders*** of variable substitutions may still result in the ***same parse tree***.
- For example:

$$
\begin{array}{lll}
Expr & \rightarrow & Expr + Term \mid Term \\
Term & \rightarrow & Term * Factor \mid Factor \\
Factor & \rightarrow & (Expr) \mid a
\end{array}
$$

For string `a + a * a`, the *leftmost* and *rightmost* derivations have ***distinct orders*** of variable substitutions, but their corresponding ***parse trees are the same***.

CFG: Ambiguity: Definition

Given a grammar $G = (V, \Sigma, R, S)$:

- A string $w \in \Sigma^*$ is derived *ambiguously* in G if there exist two or more ***distinct*** *parse trees* or, equally, two or more ***distinct*** *leftmost* derivations or, equally, two or more ***distinct*** *rightmost* derivations.

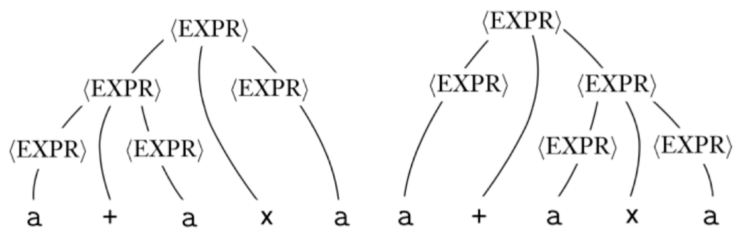
  Here we require that all such derivations have been completed by following a particular order (leftmost or rightmost) to avoid *false alarm*.
- G is *ambiguous* if it generates some string ambiguously.

CFG: Ambiguity: Exercise (1)

- Is the following grammar *ambiguous*?

$$Expr \rightarrow Expr + Expr \mid Expr * Expr \mid (Expr) \mid a$$

- Yes ∵ it generates the string `a + a * a` *ambiguously*:



- ***Distinct ASTs*** (for the ***same input***) mean ***distinct semantic interpretations***: e.g., when a post-order traversal is used to implement evaluation
- **Exercise:** Show *leftmost* derivations for the two parse trees.

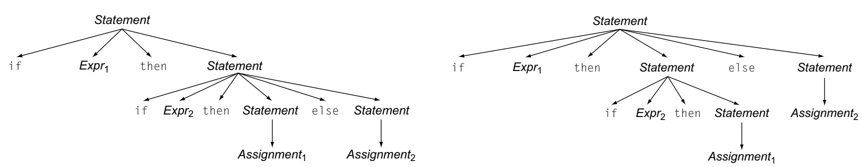
CFG: Ambiguity: Exercise (2.1)

- Is the following grammar *ambiguous*?

Statement	→	`if` *Expr* `then` *Statement*
	|	`if` *Expr* `then` *Statement* `else` *Statement*
	|	*Assignment*
...		

- Yes ∵ it generates the following string *ambiguously*:

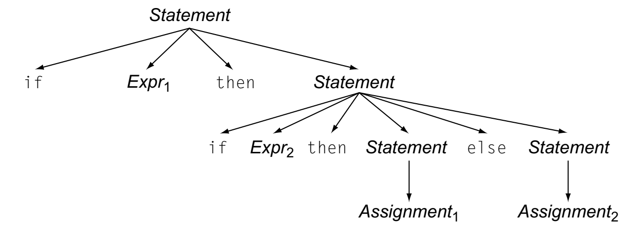
`if` $Expr_1$ `then if` $Expr_2$ `then` $Assignment_1$ `else` $Assignment_2$



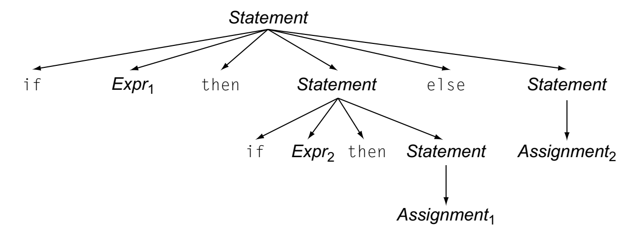
- This is called the *dangling else* problem.
- **Exercise:** Show *leftmost* derivations for the two parse trees.

CFG: Ambiguity: Exercise (2.2)

(***Meaning 1***) $Assignment_2$ may be associated with the inner `if`:



(***Meaning 2***) $Assignment_2$ may be associated with the outer `if`:



CFG: Ambiguity: Exercise (2.3)

- We may remove the ***ambiguity*** by specifying that the *dangling `else`* is associated with the *nearest `if`*:

Statement	→	`if` *Expr* `then` *Statement*
	|	`if` *Expr* `then` *WithElse* `else` *Statement*
	|	*Assignment*
WithElse	→	`if` *Expr* `then` *WithElse* `else` *WithElse*
	|	*Assignment*

- When applying `if ... then` *WithElse* `else` *Statement*:
  - The ***true*** branch will be produced via *WithElse*.
  - The ***false*** branch will be produced via *Statement*.

  There is ***no circularity*** between the two non-terminals.

Discovering Derivations

- Given a CFG $G = (V, \Sigma, R, S)$ and an input program $p \in \Sigma^*$:
  - So far we ***manually*** come up a valid derivation $S \stackrel{*}{\Rightarrow} p$.
  - A parser is supposed to *automate* this derivation process.
    Given an input sequence of (t, c) pairs, where *token* t (e.g., `r241`) belongs to some *syntactic category* c (e.g., `register`):
    Either output a ***valid derivation*** (as an ***AST***), or signal an ***error***.
- In the process of building an *AST* for the input program:
  - **Root** of AST: **start symbol** S of G
  - **Internal nodes**: A subset of variables V of G
  - **Leaves** of AST: **token sequence** input by the scanner

  ⇒ Discovering the *grammatical connections* (according to R) between the root, internal nodes, and leaves is the hard part!
- Approaches to Parsing: $[\, w \in (V \cup \Sigma)^*, A \in V, \boxed{A \rightarrow w} \in R \,]$
  - **Top-down** parsing
    For a node representing ***A***, extend it with a subtree representing ***w***.
  - **Bottom-up** parsing
    For a substring matching ***w***, build a node representing ***A*** accordingly.

TDP: Discovering Leftmost Derivation

```
ALGORITHM: TDParse
  INPUT: CFG G = (V, Σ, R, S)
  OUTPUT: Root of a Parse Tree or Syntax Error
PROCEDURE:
  root := a new node for the start symbol S
  focus := root
  initialize an empty stack trace
  trace.push(null)
  word := NextWord()
  while (true):
    if focus ∈ V then
      if ∃ unvisited rule focus → β1 β2 ... βn ∈ R then
        create β1, β2 ... βn as children of focus
        trace.push(βn βn−1 ... β2)
        focus := β1
      else
        if focus = S then report syntax error
        else backtrack
    elseif word matches focus then
      word := NextWord()
      focus := trace.pop()
    elseif word = EOF ∧ focus = null then  return root
    else backtrack
```

backtrack ≜ pop *focus*.siblings; *focus* := *focus*.parent; *focus*.resetChildren

TDP: Exercise (1)

- Given the following CFG **G**:

$$
\begin{array}{lcl}
Expr & \rightarrow & Expr + Term \\
 & \mid & Term \\
Term & \rightarrow & Term \star Factor \\
 & \mid & Factor \\
Factor & \rightarrow & (\, Expr \,) \\
 & \mid & \mathtt{a}
\end{array}
$$

  Trace *TDParse* on how to build an AST for input `a + a * a`.
- Running *TDParse* with **G** results an ***infinite loop*** !!!
  - *TDParse* focuses on the ***leftmost*** non-terminal.
  - The grammar **G** contains ***left-recursions***.
- We must first convert left-recursions in **G** to ***right-recursions***.

TDP: Exercise (2)

- Given the following CFG **G**:

$$
\begin{array}{lcl}
Expr & \rightarrow & Term \; Expr' \\
Expr' & \rightarrow & + \; Term \; Expr' \\
 & \mid & \epsilon \\
Term & \rightarrow & Factor \; Term' \\
Term' & \rightarrow & \star \; Factor \; Term' \\
 & \mid & \epsilon \\
Factor & \rightarrow & (\, Expr \,) \\
 & \mid & \mathtt{a}
\end{array}
$$

  Exercise. Trace *TDParse* on building AST for `a + a * a`.
  Exercise. Trace *TDParse* on building AST for `(a + a) * a`.
  Q: How to handle ϵ-productions (e.g., $Expr \rightarrow \epsilon$)?
  A: Execute *focus* := *trace*.`pop()` to advance to next node.
- Running *TDParse* will ***terminate*** ∵ **G** is ***right-recursive***.
- We will learn about a systematic approach to converting left-recursions in a given grammar to ***right-recursions***.

Left-Recursions (LR): Direct vs. Indirect

Given CFG $G = (V, \Sigma, R, S)$, $\alpha, \beta, \gamma \in (V \cup \Sigma)^*$, G contains:

- A *cycle* if $\exists A \in V \bullet A \overset{*}{\Rightarrow} A$
- A ***direct*** LR if $A \rightarrow A\alpha \in R$ for non-terminal $A \in V$

e.g.,

$$\begin{array}{lcl} Expr & \rightarrow & Expr + Term \\ & | & Term \\ Term & \rightarrow & Term * Factor \\ & | & Factor \\ Factor & \rightarrow & (Expr) \\ & | & a \end{array}$$

e.g.,

$$\begin{array}{lcl} Expr & \rightarrow & Expr + Term \\ & | & Expr - Term \\ & | & Term \\ Term & \rightarrow & Term * Factor \\ & | & Term \;/\; Factor \\ & | & Factor \end{array}$$

- An ***indirect*** LR if $A \rightarrow B\beta \in R$ for non-terminals $A, B \in V$, $B \overset{*}{\Rightarrow} A\gamma$

$$\begin{array}{lcl} A & \rightarrow & Br \\ B & \rightarrow & Cd \\ C & \rightarrow & At \end{array}$$

$A \rightarrow Br$, $B \overset{*}{\Rightarrow} Atd$

$$\begin{array}{lclcl} A & \rightarrow & Ba & | & b \\ B & \rightarrow & Cd & | & e \\ C & \rightarrow & Df & | & g \\ D & \rightarrow & f & | & Aa \;|\; Cg \end{array}$$

$A \rightarrow Ba$, $B \overset{*}{\Rightarrow} Aafd$

TDP: (Preventively) Eliminating LRs

```
1   ALGORITHM: RemoveLR
2     INPUT: CFG G = (V, Σ, R, S)
3     ASSUME: G acyclic ∧ with no ϵ-productions
4     OUTPUT: G' s.t. G' ≡ G, G' has no
5             indirect & direct left-recursions
6   PROCEDURE:
7     impose an order on V: ⟨⟨A_1, A_2, ..., A_n⟩⟩
8     for i: 1 .. n:
9       for j: 1 .. i − 1:
10        if ∃ A_i → A_j γ ∈ R ∧ A_j → δ_1 | δ_2 | ... | δ_m ∈ R then
11          replace A_i → A_j γ with A_i → δ_1 γ | δ_2 γ | ... | δ_m γ
12        end
13      for A_i → A_i α | β ∈ R:
14        replace it with: A_i → β A', A' → α A' | ϵ
```

L9 to **L11**: Remove ***indirect*** left-recursions from A_1 to A_{i-1}.
L12 to **L13**: Remove ***direct*** left-recursions from A_1 to A_{i-1}.
Loop Invariant (outer for-loop)? At the start of i^{th} iteration:

- No direct or indirect left-recursions for $A_1, A_2, \ldots, A_{i-1}$.
- More precisely: $\forall k : k < i \bullet \neg(\exists l \bullet l \le k \wedge A_k \rightarrow A_l \cdots \in R)$

CFG: Eliminating ϵ-Productions (1)

- Motivations:
  - *TDParse* requires CFG with no ϵ-productions.
  - *RemoveLR* produces CFG which may contain ϵ-productions.
- $\epsilon \notin L \Rightarrow \exists$ CFG $G = (V, \Sigma, R, S)$ s.t. G has no ϵ-productions.

  An *ϵ-production* has the form $A \rightarrow \epsilon$.
- A variable A is *nullable* if $A \overset{*}{\Rightarrow} \epsilon$.
  - Each terminal symbol is *not nullable*.
  - Variable A is *nullable* if either:
    - $A \rightarrow \epsilon \in R$; or
    - $A \rightarrow B_1 B_2 \ldots B_k \in R$, where each variable B_i ($1 \le i \le k$) is a *nullable*.
- Given a production $B \rightarrow CAD$, if only variable A is nullable, then there are 2 versions of B: $B \rightarrow CAD \mid CD$
- In general, given a production $A \rightarrow X_1 X_2 \ldots X_k$ with k symbols, if m of the k symbols are *nullable*:
  - $m < k$: There are 2^m versions of A.
  - $m = k$: There are $2^m - 1$ versions of A. [excluding $A \rightarrow \epsilon$]

CFG: Eliminating ϵ-Productions (2)

- Eliminate ϵ-productions from the following grammar:

$$\begin{array}{lcl} S & \rightarrow & AB \\ A & \rightarrow & aAA \mid \epsilon \\ B & \rightarrow & bBB \mid \epsilon \end{array}$$

- Which are the *nullable* variables? [S, A, B]

$$\begin{array}{lcll} S & \rightarrow & A \mid B \mid AB & \{S \rightarrow \epsilon \texttt{ not included}\} \\ A & \rightarrow & aAA \mid aA \mid a & \{A \rightarrow aA \texttt{ duplicated}\} \\ B & \rightarrow & bBB \mid bB \mid b & \{B \rightarrow bB \texttt{ duplicated}\} \end{array}$$

Backtrack-Free Parsing (1)

- TDParse automates the ***top-down***, ***leftmost*** derivation process by consistently choosing production rules (e.g., in order of their appearance in CFG).
  - This ***inflexibility*** may lead to ***inefficient*** runtime performance due to the need to *backtrack*.
  - e.g., It may take the ***construction of a giant subtree*** to find out a ***mismatch*** with the input tokens, which end up requiring it to *backtrack* all the way back to the ***root*** (start symbol).
- We may avoid backtracking with a modification to the parser:
  - When deciding which production rule to choose, consider:
    (1) the *current* input symbol
    (2) the **consequential** ***first*** symbol if a rule was applied for ***focus***
    [*lookahead* symbol]
  - Using a ***one symbol lookhead***, w.r.t. a ***right-recursive*** CFG, each alternative for the ***leftmost nonterminal*** leads to a ***unique terminal***, allowing the parser to decide on a choice that prevents *backtracking*.
  - Such CFG is ***backtrack free*** with a *lookhead* of one symbol.
  - We also call such backtrack-free CFG a ***predictive grammar***.

The FIRST Set: Definition

- Say we write $T \subset \mathbb{P}(\Sigma^*)$ to denote the set of valid tokens recognizable by the scanner.
- **FIRST** $(\alpha) \triangleq$ set of symbols that can appear as the ***first word*** in some string derived from α.
- More precisely:

$$\text{FIRST}(\alpha) = \begin{cases} \{\alpha\} & \text{if } \alpha \in T \\ \{w \mid w \in \Sigma^* \wedge \alpha \overset{*}{\Rightarrow} w\beta \wedge \beta \in (V \cup \Sigma)^*\} & \text{if } \alpha \in V \end{cases}$$

The FIRST Set: Examples

- Consider this ***right***-recursive CFG:

0	*Goal*	→	*Expr*	6	*Term′*	→	× *Factor Term′*
1	*Expr*	→	*Term Expr′*	7		|	÷ *Factor Term′*
2	*Expr′*	→	+ *Term Expr′*	8		|	ϵ
3		|	- *Term Expr′*	9	*Factor*	→	(*Expr*)
4		|	ϵ	10		|	num
5	*Term*	→	*Factor Term′*	11		|	name

- Compute **FIRST** for each terminal (e.g., num, +, ():

	num	name	+	-	×	÷	(	)	eof	ϵ
FIRST	num	name	+	-	×	÷	(	)	eof	ϵ

- Compute **FIRST** for each non-terminal (e.g., *Expr*, *Term′*):

	Expr	*Expr′*	*Term*	*Term′*	*Factor*
FIRST	(, name, num	+, -, ϵ	(, name, num	×, ÷, ϵ	(, name, num

Computing the FIRST Set

$$\text{FIRST}(\alpha) = \begin{cases} \{\alpha\} & \text{if } \alpha \in T \\ \{w \mid w \in \Sigma^* \wedge \alpha \overset{*}{\Rightarrow} w\beta \wedge \beta \in (V \cup \Sigma)^*\} & \text{if } \alpha \in V \end{cases}$$

```
ALGORITHM: GetFirst
  INPUT: CFG G = (V, Σ, R, S)
  T ⊂ Σ* denotes valid terminals
  OUTPUT: FIRST : V ∪ T ∪ {ϵ, eof} ⟶ ℙ(T ∪ {ϵ, eof})
PROCEDURE:
  for α ∈ (T ∪ {eof, ϵ}): FIRST(α) := {α}
  for A ∈ V: FIRST(A) := ∅
  lastFirst := ∅
  while (lastFirst ≠ FIRST):
    lastFirst := FIRST
    for A → β1β2...βk ∈ R s.t. ∀βj : βj ∈ (T ∪ V):
      rhs := FIRST(β1) − {ϵ}
      for (i := 1; ϵ ∈ FIRST(βi) ∧ i < k; i++):
        rhs := rhs ∪ (FIRST(βi+1) − {ϵ})
      if i = k ∧ ϵ ∈ FIRST(βk) then
        rhs := rhs ∪ {ϵ}
      end
      FIRST(A) := FIRST(A) ∪ rhs
```

Computing the FIRST Set: Extension

- Recall: **FIRST** takes as input a token or a variable.

$$\textbf{FIRST} : V \cup T \cup \{\epsilon, eof\} \longrightarrow \mathbb{P}(T \cup \{\epsilon, eof\})$$

- The computation of variable ***rhs*** in algoritm `GetFirst` actually suggests an extended, overloaded version:

$$\textbf{FIRST} : (V \cup T \cup \{\epsilon, eof\})^* \longrightarrow \mathbb{P}(T \cup \{\epsilon, eof\})$$

FIRST may also take as input a string $\beta_1 \beta_2 \dots \beta_n$ (RHS of rules).

- More precisely:

$$\textbf{FIRST}(\beta_1 \beta_2 \dots \beta_n) = \left\{ \textbf{FIRST}(\beta_1) \cup \textbf{FIRST}(\beta_2) \cup \dots \beta_k \;\middle|\; \begin{array}{l} \forall i : 1 \le i < k \bullet \epsilon \in \textbf{FIRST}(\beta_i) \\ \wedge \\ \epsilon \notin \textbf{FIRST}(\beta_k) \end{array} \right\}$$

Note. β_k is the first symbol whose **FIRST** set does not contain ϵ.

Extended FIRST Set: Examples

Consider this ***right***-recursive CFG:

0	*Goal*	→	*Expr*	6	*Term′*	→	× *Factor Term′*
1	*Expr*	→	*Term Expr′*	7		|	÷ *Factor Term′*
2	*Expr′*	→	+ *Term Expr′*	8		|	ε
3		|	- *Term Expr′*	9	*Factor*	→	(*Expr*)
4		|	ε	10		|	num
5	*Term*	→	*Factor Term′*	11		|	name

e.g., $\textbf{FIRST}(Term\ Expr') = \textbf{FIRST}(Term) = \{ (, \texttt{name}, \texttt{num} \}$

e.g., $\textbf{FIRST}(+\ \ Term\ \ Expr') = \textbf{FIRST}(+) = \{+\}$

e.g., $\textbf{FIRST}(-\ \ Term\ \ Expr') = \textbf{FIRST}(-) = \{-\}$

e.g., $\textbf{FIRST}(\epsilon) = \{\epsilon\}$

Is the FIRST Set Sufficient

- Consider the following three productions:

Expr′	→	+	*Term*	*Term′*	(1)
	|	−	*Term*	*Term′*	(2)
	|	ε			(3)

In TDP, when the parser attempts to expand an *Expr′* node, it ***looks ahead with one symbol*** to decide on the choice of rule: $\textbf{FIRST}(+) = \{+\}$, $\textbf{FIRST}(-) = \{-\}$, and $\textbf{FIRST}(\epsilon) = \{\epsilon\}$.

Q. When to choose rule `(3)` (causing ***focus := trace.pop()***)?

A?. Choose rule `(3)` when $focus \neq \textbf{FIRST}(+) \wedge focus \neq \textbf{FIRST}(-)$?

- ***Correct*** but ***inefficient*** in case of illegal input string: syntax error is only reported after possibly a long series of ***backtrack***.
- Useful if parser knows which words can appear, after an application of the ϵ-production (rule `(3)`), as leadling symbols.

- **FOLLOW** $(v : V) \triangleq$ set of symbols that can appear to the immediate right of a string derived from α.

$$\textbf{FOLLOW}(v) = \{ w \mid w, x, y \in \Sigma^* \wedge v \overset{*}{\Rightarrow} x \wedge S \overset{*}{\Rightarrow} xwy \}$$

The FOLLOW Set: Examples

- Consider this ***right***-recursive CFG:

0	*Goal*	→	*Expr*	6	*Term′*	→	× *Factor Term′*
1	*Expr*	→	*Term Expr′*	7		|	÷ *Factor Term′*
2	*Expr′*	→	+ *Term Expr′*	8		|	ε
3		|	- *Term Expr′*	9	*Factor*	→	(*Expr*)
4		|	ε	10		|	num
5	*Term*	→	*Factor Term′*	11		|	name

- Compute **FOLLOW** for each non-terminal (e.g., *Expr*, *Term′*):

	Expr	*Expr′*	*Term*	*Term′*	*Factor*
FOLLOW	eof,)	eof,)	eof, +, -,)	eof, +, -,)	eof, +, -, x, ÷,)

Computing the FOLLOW Set

$\text{FOLLOW}(v) = \{w \mid w, x, y \in \Sigma^* \wedge v \overset{*}{\Rightarrow} x \wedge S \overset{*}{\Rightarrow} xwy\}$

```
ALGORITHM: GetFollow
  INPUT: CFG G = (V, Σ, R, S)
  OUTPUT: FOLLOW : V ⟶ ℙ(T ∪ {eof})
PROCEDURE:
  for A ∈ V: FOLLOW(A) := ∅
  FOLLOW(S) := {eof}
  lastFollow := ∅
  while(lastFollow ≠ FOLLOW):
    lastFollow := FOLLOW
    for A → β1β2...βk ∈ R:
      trailer := FOLLOW(A)
      for i: k .. 1:
        if βi ∈ V then
          FOLLOW(βi) := FOLLOW(βi) ∪ trailer
          if ϵ ∈ FIRST(βi)
            then trailer := trailer ∪ (FIRST(βi) − ϵ)
            else trailer := FIRST(βi)
        else
          trailer := FIRST(βi)
```


Backtrack-Free Grammar

- A ***backtrack-free grammar*** (for a ***top-down parser***), when expanding the ***focus internal node***, is always able to choose a ***unique*** rule with the ***one-symbol lookahead*** (or report a ***syntax error*** when no rule applies).
- To formulate this, we first define:

$$\text{FIRST}^+(A \to \beta) = \begin{cases} \text{FIRST}(\beta) & \text{if } \epsilon \notin \text{FIRST}(\beta) \\ \text{FIRST}(\beta) \cup \text{FOLLOW}(A) & \text{otherwise} \end{cases}$$

  $\text{FIRST}(\beta)$ is the extended version where β may be $\beta_1\beta_2 \dots \beta_n$
- Now, a ***backtrack-free grammar*** has each of its productions $A \to \gamma_1 \mid \gamma_2 \mid \dots \mid \gamma_n$ satisfying:

$$\forall i, j : 1 \le i, j \le n \wedge i \ne j \bullet \text{FIRST}^+(\gamma_i) \cap \text{FIRST}^+(\gamma_j) = \varnothing$$

TDP: Lookahead with One Symbol

```
ALGORITHM: TDParse
  INPUT: CFG G = (V, Σ, R, S)
  OUTPUT: Root of a Parse Tree or Syntax Error
PROCEDURE:
  root := a new node for the start symbol S
  focus := root
  initialize an empty stack trace
  trace.push(null)
  word := NextWord()
  while (true):
    if focus ∈ V then % use FOLLOW set as well?
      if ∃ unvisited rule focus → β1β2...βn ∈ R ∧ word ∈ FIRST+(β) then
        create β1,β2...βn as children of focus
        trace.push(βnβn−1...β2)
        focus := β1
      else
        if focus = S then report syntax error
        else backtrack
    elseif word matches focus then
      word := NextWord()
      focus := trace.pop()
    elseif word = EOF ∧ focus = null then return root
    else backtrack
```

backtrack ≜ pop *focus*.siblings; *focus* := *focus*.parent; *focus*.resetChildren

Backtrack-Free Grammar: Exercise

Is the following CFG ***backtrack free***?

11	*Factor*	→	name
12		|	name [*ArgList*]
13		|	name (*ArgList*)
15	*ArgList*	→	*Expr MoreArgs*
16	*MoreArgs*	→	, *Expr MoreArgs*
17		|	ϵ

- $\epsilon \notin \text{FIRST}(Factor) \Rightarrow \text{FIRST}^+(Factor) = \text{FIRST}(Factor)$
- $\text{FIRST}(Factor \to \texttt{name}) = \{\texttt{name}\}$
- $\text{FIRST}(Factor \to \texttt{name}\ [ArgList]) = \{\texttt{name}\}$
- $\text{FIRST}(Factor \to \texttt{name}\ (ArgList)) = \{\texttt{name}\}$

∴ The above grammar is ***not*** backtrack free.
⇒ To expand an AST node of *Factor*, with a ***lookahead*** of `name`, the parser has no basis to choose among rules 11, 12, and 13.

Backtrack-Free Grammar: Left-Factoring

- A CFG is not backtrack free if there exists a ***common prefix*** (`name`) among the RHS of ***multiple*** production rules.
- To make such a CFG ***backtrack-free***, we may transform it using *left factoring*: a process of extracting and isolating ***common prefixes*** in a set of production rules.
  - Identify a common prefix α:

$$A \rightarrow \alpha\beta_1 \mid \alpha\beta_2 \mid \ldots \mid \alpha\beta_n \mid \gamma_1 \mid \gamma_2 \mid \ldots \mid \gamma_j$$

[each of $\gamma_1, \gamma_2, \ldots, \gamma_j$ does not begin with α]

  - Rewrite that production rule as:

$$\begin{array}{lcl} A & \rightarrow & \alpha B \mid \gamma_1 \mid \gamma_2 \mid \ldots \mid \gamma_j \\ B & \rightarrow & \beta_1 \mid \beta_2 \mid \ldots \mid \beta_n \end{array}$$

  - New rule $B \rightarrow \beta_1 \mid \beta_2 \mid \ldots \mid \beta_n$ may also contain ***common prefixes***.
  - Rewriting continues until no common prefixes are identified.

Left-Factoring: Exercise

- Use *left-factoring* to remove all ***common prefixes*** from the following grammar.

11	*Factor*	$\rightarrow$	`name`
12		|	`name` [*ArgList*]
13		|	`name` (*ArgList*)
15	*ArgList*	$\rightarrow$	*Expr MoreArgs*
16	*MoreArgs*	$\rightarrow$	, *Expr MoreArgs*
17		|	ϵ

- Identify common prefix `name` and rewrite rules 11, 12, and 13:

$$\begin{array}{lcl} Factor & \rightarrow & \texttt{name} \quad Arguments \\ Arguments & \rightarrow & [\quad ArgList \quad] \\ & \mid & (\quad ArgList \quad) \\ & \mid & \epsilon \end{array}$$

Any more ***common prefixes***? [No]

TDP: Terminating and Backtrack-Free

- Given an arbitrary CFG as input to a ***top-down parser***:
  - **Q.** How do we avoid a ***non-terminating*** parsing process?
    A. Convert left-recursions to right-recursion.
  - **Q.** How do we minimize the need of ***backtracking***?
    A. left-factoring & one-symbol lookahead using FIRST$^+$
- ***Not*** every context-free ***language*** has a corresponding ***backtrack***-free context-free ***grammar***.

  Given a CFL l, the following is *undecidable*:

$$\exists cfg \mid L(cfg) = l \wedge isBacktrackFree(cfg)$$

- Given a CFG $g = (V, \Sigma, R, S)$, whether or not g is ***backtrack-free*** is *decidable*:

  For each $A \rightarrow \gamma_1 \mid \gamma_2 \mid \ldots \mid \gamma_n \in R$:

$$\forall i, j : 1 \leq i, j \leq n \wedge i \neq j \bullet \text{FIRST}^+(\gamma_i) \cap \text{FIRST}^+(\gamma_j) = \varnothing$$

Backtrack-Free Parsing (2.1)

- A *recursive-descent* parser is:
  - A top-down parser
  - Structured as a set of *mutually recursive* procedures

    Each procedure corresponds to a ***non-terminal*** in the grammar.

    See an example.
- Given a ***backtrack-free*** grammar, a tool (a.k.a. *parser generator*) can automatically generate:
  - FIRST, FOLLOW, and FIRST$^+$ sets
  - An efficient ***recursive-descent*** parser

    This generated parser is called an *LL(1) parser*, which:
    - Processes input from Left to right
    - Constructs a Leftmost derivation
    - Uses a lookahead of 1 symbol
- *LL(1) grammars* are those working in an *LL(1)* scheme.

  LL(1) grammars are ***backtrack-free*** by definition.

Backtrack-Free Parsing (2.2)

- Consider this CFG with FIRST$^+$ sets of the RHSs:

	Production				FIRST$^+$
2	*Expr′*	$\rightarrow$	+	*Term Expr′*	{ + }
3		|	-	*Term Expr′*	{ - }
4		|	ϵ		{ ϵ, eof,) }

- The corresponding ***recursive-descent*** parser is structured as:

```
ExprPrim()
  if word = + ∨ word = - then /* Rules 2, 3 */
    word := NextWord()
    if(Term())
      then return ExprPrim()
      else return false
  elseif word = ) ∨ word = eof then /* Rule 4 */
    return true
  else
    report a syntax error
    return false
  end

Term()
  ...
```

See: parser generator

LL(1) Parser: Exercise

Consider the following grammar:

$$
\begin{array}{lcl}
L & \rightarrow & R\ \texttt{a} \\
  & | & Q\ \texttt{ba}
\end{array}
\qquad
\begin{array}{lcl}
R & \rightarrow & \texttt{aba} \\
  & | & \texttt{caba} \\
  & | & R\ \texttt{bc}
\end{array}
\qquad
\begin{array}{lcl}
Q & \rightarrow & \texttt{bbc} \\
  & | & \texttt{bc}
\end{array}
$$

Q. Is it suitable for a ***top-down predictive*** parser?

- If so, show that it satisfies the *LL(1)* condition.
- If not, identify the problem(s) and correct it (them). Also show that the revised grammar satisfies the *LL(1)* condition.

BUP: Discovering Rightmost Derivation

- In TDP, we build the start variable as the ***root node***, and then work towards the ***leaves***. [**leftmost** derivation]
- In Bottom-Up Parsing (BUP):
  - Words (terminals) are still returned from **left** to **right** by the scanner.
  - As terminals, or a mix of terminals and variables, are identified as *reducible* to some variable *A* (i.e., matching the RHS of some production rule for *A*), then a layer is added.
  - Eventually:
    - ***accept***:
      The ***start variable*** is reduced and **all** words have been consumed.
    - ***reject***:
      The next word is not eof, but no further *reduction* can be identified.

Q. Why can BUP find the ***rightmost*** derivation (RMD), if any?

A. BUP discovers steps in a ***RMD*** in its *reverse* order.

BUP: Discovering Rightmost Derivation (1)

- ***table***-driven *LR(1)* parser: an implementation for BUP, which
  - Processes input from **L**eft to right
  - Constructs a **R**ightmost derivation
  - Uses a lookahead of **1** symbol
- A language has the *LR(1)* property if it:
  - Can be parsed in a single **L**eft to right scan,
  - To build a *reversed* **R**ightmost derivation,
  - Using a lookahead of **1** symbol to determine parsing actions.
- Critical step in a bottom-up parser is to find the ***next*** *handle*.

BUP: Discovering Rightmost Derivation (2)

```
ALGORITHM: BUParse
  INPUT: CFG G = (V, Σ, R, S), Action & Goto Tables
  OUTPUT: Report Parse Success or Syntax Error
PROCEDURE:
  initialize an empty stack trace
  trace.push(S) /* start state */
  word := NextWord()
  while(true)
    state := trace.top()
    act := Action[state, word]
    if act = ``accept'' then
      succeed()
    elseif act = ``reduce A → β'' then
      trace.pop() 2 × |β| times /* word + state */
      state := trace.top()
      trace.push(A)
      next := Goto[state, A]
      trace.push(next)
    elseif act = ``shift s_i'' then
      trace.push(word)
      trace.push(s_i)
      word := NextWord()
    else
      fail()
```


BUP: Example Tracing (1)

- Consider the following grammar for parentheses:

1	*Goal*	→	*List*
2	*List*	→	*List Pair*
3		|	*Pair*
4	*Pair*	→	(*Pair*)
5		|	( )

- Assume: tables ***Action*** and ***Goto*** constructed accordingly:

State	`Action` Table: eof	(	)	`Goto` Table: *List*	*Pair*
0		s 3		1	2
1	acc	s 3			4
2	r 3	r 3			
3		s 6	s 7		5
4	r 2	r 2			
5			s 8		
6		s 6	s 10		9
7	r 5	r 5			
8	r 4	r 4			
9			s 11		
10			r 5		
11			r 4		

In **Action** table:

- s_i: shift to state *i*
- r_j: reduce to the LHS of production #*j*

BUP: Example Tracing (2.1)

Consider the steps of performing BUP on input `( )`:

Iteration	State	word	Stack	Handle	Action
initial	—	(	$ 0	— *none* —	—
1	0	(	$ 0	— *none* —	*shift 3*
2	3	)	$ 0 (3	— *none* —	*shift 7*
3	7	eof	$ 0 (3) 7	( )	*reduce 5*
4	2	eof	$ 0 *Pair* 2	*Pair*	*reduce 3*
5	1	eof	$ 0 *List* 1	*List*	*accept*

BUP: Example Tracing (2.2)

Consider the steps of performing BUP on input `( ( ) ) ( )`:

Iteration	State	word	Stack	Handle	Action
initial	—	(	$ 0	— *none* —	—
1	0	(	$ 0	— *none* —	*shift 3*
2	3	(	$ 0 (3	— *none* —	*shift 6*
3	6	)	$ 0 (3 (6	— *none* —	*shift 10*
4	10	)	$ 0 (3 (6) 10	( )	*reduce 5*
5	5	)	$ 0 (3 *Pair* 5	— *none* —	*shift 8*
6	8	(	$ 0 (3 *Pair* 5) 8	(*Pair*)	*reduce 4*
7	2	(	$ 0 *Pair* 2	*Pair*	*reduce 3*
8	1	(	$ 0 *List* 1	— *none* —	*shift 3*
9	3	)	$ 0 *List* 1 (3	— *none* —	*shift 7*
10	7	eof	$ 0 *List* 1 (3) 7	( )	*reduce 5*
11	4	eof	$ 0 *List* 1 *Pair* 4	*List Pair*	*reduce 2*
12	1	eof	$ 0 *List* 1	*List*	*accept*

BUP: Example Tracing (2.3)

Consider the steps of performing BUP on input `( ) )`:

Iteration	State	word	Stack	Handle	Action
initial	—	(	$ 0	— *none* —	—
1	0	(	$ 0	— *none* —	*shift 3*
2	3	)	$ 0 (3	— *none* —	*shift 7*
3	7	)	$ 0 (3) 7	— *none* —	*error*

LR(1) Items: Definition

- In ***LR(1)*** parsing, **Action** and **Goto** tabeles encode legitimate ways (w.r.t. a grammar) for finding ***handles*** (for ***reductions***).
- In a ***table***-driven ***LR(1)*** parser, the table-construction algorithm represents each potential ***handle*** (for a ***reduction***) with an ***LR(1) item*** e.g.,

$$[A \rightarrow \beta \bullet \gamma,\ \mathtt{a}]$$

where:

- A production rule $A \rightarrow \beta\gamma$ is currently being applied.
- A placeholder, $\bullet$, indicates the position of the parser's ***stack top***.
  - ✓ The parser's stack contains β ("left context").
  - ✓ γ is yet to be matched.

  Remark. Upon matching $\beta\gamma$, if $\mathtt{a}$ matches the current word, then we "replace" $\beta\gamma$ (and their corresponding states) with A (and its corresponding state).
- A terminal symbol $\mathtt{a}$ servers as a ***lookahead symbol***.

LR(1) Items: Scenarios

An ***LR(1) item*** can be:

1. **POSSIBILITY** $[A \rightarrow \bullet\beta\gamma,\ \mathtt{a}]$
   - In the current parsing context, an A would be valid.
   - $\bullet$ represents the position of the parser's ***stack top***
   - Recognizing a β next would be one step towards discovering an A.
2. **PARTIALLY COMPLETION** $[A \rightarrow \beta \bullet \gamma,\ \mathtt{a}]$
   - The parser has progressed from $[A \rightarrow \bullet\beta\gamma,\ \mathtt{a}]$ by recognizing β.
   - Recognizing a γ next would be one step towards discovering an A.
3. **COMPLETION** $[A \rightarrow \beta\gamma\bullet,\ \mathtt{a}]$
   - Parser has progressed from $[A \rightarrow \bullet\beta\gamma,\ \mathtt{a}]$ by recognizing $\beta\gamma$.
   - $\beta\gamma$ found in a context where an A followed by $\mathtt{a}$ would be valid.
   - If the current input word matches $\mathtt{a}$, then:
     - Current ***complet item*** is a ***handle***.
     - Parser can ***reduce*** $\beta\gamma$ to A (and replace $\beta\gamma$ with A in its stack).

LR(1) Items: Example (1.1)

Consider the following grammar for parentheses:

1	*Goal*	$\rightarrow$	*List*
2	*List*	$\rightarrow$	*List Pair*
3		|	*Pair*
4	*Pair*	$\rightarrow$	(*Pair*)
5		|	( )

Initial State: $[Goal \rightarrow \bullet List,\ \mathtt{eof}]$

Desired Final State: $[Goal \rightarrow List\bullet,\ \mathtt{eof}]$

Intermediate States: Subset Construction

Q. Derive all ***LR(1) items*** for the above grammar.

- FOLLOW$(List) = \{\mathtt{eof},\ (\}$ FOLLOW$(Pair) = \{\mathtt{eof},\ (,\ )\}$
- For each production $A \rightarrow \beta$, given FOLLOW(A), ***LR(1) items*** are:

$$\{\ [A \rightarrow \bullet\beta\gamma,\ \mathtt{a}] \mid a \in \text{FOLLOW}(A)\ \}$$
$$\cup$$
$$\{\ [A \rightarrow \beta \bullet \gamma,\ \mathtt{a}] \mid a \in \text{FOLLOW}(A)\ \}$$
$$\cup$$
$$\{\ [A \rightarrow \beta\gamma\bullet,\ \mathtt{a}] \mid a \in \text{FOLLOW}(A)\ \}$$

LR(1) Items: Example (1.2)

Q. Given production $A \to \beta$ (e.g., *Pair* → `(` *Pair* `)`), how many *LR(1) items* can be generated?

- The current parsing progress (on matching the RHS) can be:
  1. •`(` *Pair* `)`
  2. `(` •*Pair* `)`
  3. `(` *Pair*• `)`
  4. `(` *Pair* `)`•
- Lookahead symbol following *Pair*? **Follow**(*Pair*) = {`eof`, `(`, `)`}
- All possible *LR(1) items* related to *Pair* → `(` *Pair* `)`?

✓ [•`(` *Pair* `)`, `eof`]	[•`(` *Pair* `)`, `(`]	[•`(` *Pair* `)`, `)`]
✓ [`(` •*Pair* `)`, `eof`]	[`(` •*Pair* `)`, `(`]	[`(` •*Pair* `)`, `)`]
✓ [`(` *Pair*• `)`, `eof`]	[`(` *Pair*• `)`, `(`]	[`(` *Pair*• `)`, `)`]
✓ [`(` *Pair* `)`•, `eof`]	[`(` *Pair* `)`•, `(`]	[`(` *Pair* `)`•, `)`]

A. How many in general (in terms of A and β)?

$$\underbrace{|\beta| + 1}_{\texttt{possible positions of } \bullet} \times \underbrace{|\textsc{Follow}(A)|}_{\texttt{possible lookahead symbols}}$$

LR(1) Items: Example (1.3)

A. There are 33 *LR(1) items* in the parentheses grammar.

[*Goal* → • *List*,`eof`]		
[*Goal* → *List* •,`eof`]		
[*List* → • *List Pair*,`eof`]	[*List* → • *List Pair*,`(`]	
[*List* → *List* • *Pair*,`eof`]	[*List* → *List* • *Pair*,`(`]	
[*List* → *List Pair* •,`eof`]	[*List* → *List Pair* •,`(`]	
[*List* → • *Pair*,`eof`]	[*List* → • *Pair*,`(`]	
[*List* → *Pair* •,`eof`]	[*List* → *Pair* •,`(`]	
[*Pair* → • `(` *Pair* `)`,`eof`]	[*Pair* → • `(` *Pair* `)`,`)`]	[*Pair* → • `(` *Pair* `)`,`(`]
[*Pair* → `(` • *Pair* `)`,`eof`]	[*Pair* → `(` • *Pair* `)`,`)`]	[*Pair* → `(` • *Pair* `)`,`(`]
[*Pair* → `(` *Pair* • `)`,`eof`]	[*Pair* → `(` *Pair* • `)`,`)`]	[*Pair* → `(` *Pair* • `)`,`(`]
[*Pair* → `(` *Pair* `)` •,`eof`]	[*Pair* → `(` *Pair* `)` •,`)`]	[*Pair* → `(` *Pair* `)` •,`(`]
[*Pair* → • `(` `)`,`eof`]	[*Pair* → • `(` `)`,`(`]	[*Pair* → • `(` `)`,`)`]
[*Pair* → `(` • `)`,`eof`]	[*Pair* → `(` • `)`,`(`]	[*Pair* → `(` • `)`,`)`]
[*Pair* → `(` `)` •,`eof`]	[*Pair* → `(` `)` •,`(`]	[*Pair* → `(` `)` •,`)`]

LR(1) Items: Example (2)

Consider the following grammar for expressions:

0	*Goal*	→	*Expr*	6	*Term′*	→	`x` *Factor Term′*
1	*Expr*	→	*Term Expr′*	7		|	`÷` *Factor Term′*
2	*Expr′*	→	`+` *Term Expr′*	8		|	ϵ
3		|	`-` *Term Expr′*	9	*Factor*	→	`(` *Expr* `)`
4		|	ϵ	10		|	`num`
5	*Term*	→	*Factor Term′*	11		|	`name`

Q. Derive all *LR(1) items* for the above grammar.
Hints. First compute **Follow** for each non-terminal:

	Expr	*Expr′*	*Term*	*Term′*	*Factor*
FOLLOW	`eof`, `)`	`eof`, `)`	`eof`, `+`, `-`, `)`	`eof`, `+`, `-`, `)`	`eof`, `+`, `-`, `x`, `÷`, `)`

Tips. Ignore ϵ ***production*** such as *Expr′* → ϵ since the **Follow** sets already take them into consideration.

Canonical Collection ($\mathcal{CC}$) vs. LR(1) items

1	*Goal* → *List*
2	*List* → *List Pair*
3	| *Pair*
4	*Pair* → `(` *Pair* `)`
5	| `(` `)`

Recall:

LR(1) Items: 33 items

Initial State: [*Goal* → •*List*, `eof`]

Desired Final State: [*Goal* → *List*•, `eof`]

- The *canonical collection*
  $$\mathcal{CC} = \{cc_0, cc_1, cc_2, \ldots, cc_n\}$$
  denotes the set of ***valid states*** of a ***LR(1)*** parser.
  - Each $cc_i \in \mathcal{CC}$ ($0 \le i \le n$) is a set of **LR(1) items**.
  - $\mathcal{CC} \subseteq \mathbb{P}$(**LR(1) items**) $|\mathcal{CC}|$? [$|\mathcal{CC}| \le 2^{|\textbf{LR(1) items}|}$]
- To model a ***LR(1)*** parser, we use techniques similar to how we construct a DFA from an NFA (subset construction and ϵ-closure).
- **Analogies.**
  ✓ **LR(1) items** ≈ states of source *NFA*
  ✓ $\mathcal{CC}$ ≈ states of target *DFA*

Constructing $\mathcal{CC}$: The *closure* Procedure (1)

```
1   ALGORITHM: closure
2     INPUT: CFG G = (V, Σ, R, S), a set s of LR(1) items
3     OUTPUT: a set of LR(1) items
4   PROCEDURE:
5     lastS := ∅
6     while(lastS ≠ s):
7       lastS := s
8       for [A → ··· • C δ, a] ∈ s:
9         for C → γ ∈ R:
10          for b ∈ FIRST(δa):
11            s := s ∪ { [C → •γ, b] }
12    return s
```

- **Line 8**: $[A \rightarrow \cdots \bullet C\ \delta,\ a] \in s$ indicates that the parser's next task is to match $C\ \delta$ with a lookahead symbol a.
- **Line 9**: Given: matching γ can reduce to C
- **Line 10**: Given: $b \in$ **FIRST**(δa) is a valid lookahead symbol after reducing γ to C
- **Line 11**: Add a new item $[C \rightarrow \bullet\gamma,\ b]$ into *s*.
- **Line 6**: Termination is guaranteed.

  ∵ Each iteration adds ≥ 1 item to *s* (otherwise *lastS* ≠ *s* is *false*).

Constructing $\mathcal{CC}$: The *closure* Procedure (2.1)

1	$Goal \rightarrow List$
2	$List \rightarrow List\ Pair$
3	$\mid\ Pair$
4	$Pair \rightarrow \underline{(}\ Pair\ \underline{)}$
5	$\mid\ \underline{(}\ \underline{)}$

Initial State: $[Goal \rightarrow \bullet List,\ \texttt{eof}]$

Calculate $cc_0 = closure([Goal \rightarrow \bullet List,\ \texttt{eof}])$.

Constructing $\mathcal{CC}$: The *goto* Procedure (1)

```
1   ALGORITHM: goto
2     INPUT: a set s of LR(1) items, a symbol x
3     OUTPUT: a set of LR(1) items
4   PROCEDURE:
5     moved := ∅
6     for item ∈ s:
7       if item = [α → β • xδ, a] then
8         moved := moved ∪ { [α → βx • δ, a] }
9       end
10    return closure(moved)
```

Line 7: Given: item $[\alpha \rightarrow \beta \bullet x\delta,\ a]$ (where *x* is the next to match)
Line 8: Add $[\alpha \rightarrow \beta x \bullet \delta,\ a]$ (indicating x is matched) to *moved*
Line 10: Calculate and return *closure*(*moved*) as the "next state" from *s* with a "transition" x.

Constructing $\mathcal{CC}$: The *goto* Procedure (2)

1	$Goal \rightarrow List$
2	$List \rightarrow List\ Pair$
3	$\mid\ Pair$
4	$Pair \rightarrow \underline{(}\ Pair\ \underline{)}$
5	$\mid\ \underline{(}\ \underline{)}$

$$cc_0 = \left\{ \begin{array}{ccc} [Goal \rightarrow \bullet List,\ \texttt{eof}] & [List \rightarrow \bullet List\ Pair,\ \texttt{eof}] & [List \rightarrow \bullet List\ Pair,\ \underline{(}] \\ [List \rightarrow \bullet Pair,\ \texttt{eof}] & [List \rightarrow \bullet Pair,\ \underline{(}] & [Pair \rightarrow \bullet \underline{(}\ Pair\ \underline{)},\ \texttt{eof}] \\ [Pair \rightarrow \bullet \underline{(}\ Pair\ \underline{)}, \underline{(}] & [Pair \rightarrow \bullet \underline{(}\ \underline{)},\ \texttt{eof}] & [Pair \rightarrow \bullet \underline{(}\ \underline{)}, \underline{(}] \end{array} \right\}$$

Calculate $goto(cc_0,\ \underline{(})$. ["next state" from cc_0 taking $\underline{(}$]

Constructing $\mathcal{CC}$: The Algorithm (1)

```
ALGORITHM: BuildCC
  INPUT: a grammar G = (V, Σ, R, S), goal production S → S'
  OUTPUT:
    (1) a set CC = {cc0, cc1, ..., ccn} where cci ⊆ G's LR(1) items
    (2) a transition function
PROCEDURE:
  cc0 := closure({[S' → •S, eof]})
  CC := {cc0}
  processed := {cc0}
  lastCC := ∅
  while(lastCC ≠ CC):
    lastCC := CC
    for cci s.t. cci ∈ CC ∧ cci ∉ processed:
      processed := processed ∪ {cci}
      for x s.t. [··· → ···•x...] ∈ cci
        temp := goto(cci, x)
        if temp ∉ CC then
          CC := CC ∪ {temp}
        end
        δ := δ ∪ (cci, x, temp)
```


Constructing $\mathcal{CC}$: The Algorithm (2.1)

1	*Goal* → *List*
2	*List* → *List Pair*
3	| *Pair*
4	*Pair* → (*Pair*)
5	| ( )

- Calculate $\mathcal{CC} = \{cc_0, cc_1, \ldots, cc_{11}\}$
- Calculate the transition function $\delta : \mathcal{CC} \times \Sigma \rightarrow \mathcal{CC}$

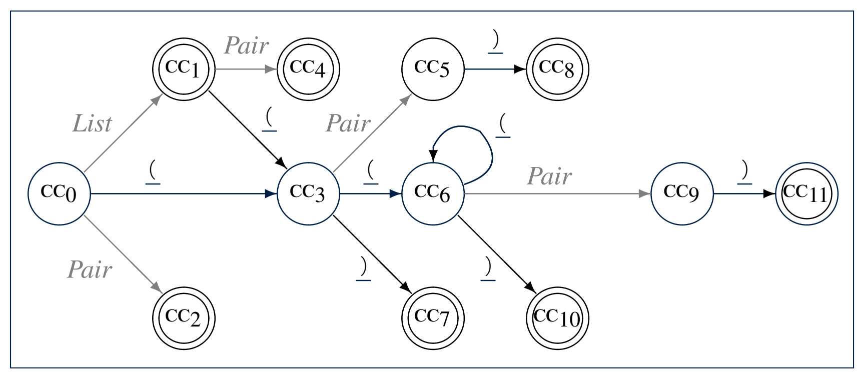
Constructing $\mathcal{CC}$: The Algorithm (2.2)

Resulting transition table:

Iteration	Item	*Goal*	*List*	*Pair*	(	)	eof
0	cc_0	∅	cc_1	cc_2	cc_3	∅	∅
1	cc_1	∅	∅	cc_4	cc_3	∅	∅
	cc_2	∅	∅	∅	∅	∅	∅
	cc_3	∅	∅	cc_5	cc_6	cc_7	∅
2	cc_4	∅	∅	∅	∅	∅	∅
	cc_5	∅	∅	∅	∅	cc_8	∅
	cc_6	∅	∅	cc_9	cc_6	cc_{10}	∅
	cc_7	∅	∅	∅	∅	∅	∅
3	cc_8	∅	∅	∅	∅	∅	∅
	cc_9	∅	∅	∅	∅	cc_{11}	∅
	cc_{10}	∅	∅	∅	∅	∅	∅
4	cc_{11}	∅	∅	∅	∅	∅	∅

Constructing $\mathcal{CC}$: The Algorithm (2.3)

Resulting DFA for the parser:



Constructing $\mathcal{CC}$: The Algorithm (2.4.1)

Resulting canonical collection $\mathcal{CC}$:

$$cc_0 = \left\{\begin{matrix} [Goal \rightarrow \bullet\, List, \texttt{eof}] & [List \rightarrow \bullet\, List\ Pair, \texttt{eof}] & [List \rightarrow \bullet\, List\ Pair, \underline{(}] \\ [List \rightarrow \bullet\, Pair, \texttt{eof}] & [List \rightarrow \bullet\, Pair, \underline{(}] & [Pair \rightarrow \bullet\, \underline{(}\ Pair\ \underline{)}, \texttt{eof}] \\ [Pair \rightarrow \bullet\, \underline{(}\ Pair\ \underline{)}, \underline{(}] & [Pair \rightarrow \bullet\, \underline{(}\ \underline{)}, \texttt{eof}] & [Pair \rightarrow \bullet\, \underline{(}\ \underline{)}, \underline{(}] \end{matrix}\right\}$$

$$cc_1 = \left\{\begin{matrix} [Goal \rightarrow List\, \bullet, \texttt{eof}] & [List \rightarrow List\, \bullet\, Pair, \texttt{eof}] & [List \rightarrow List\, \bullet\, Pair, \underline{(}] \\ [Pair \rightarrow \bullet\, \underline{(}\ Pair\ \underline{)}, \texttt{eof}] & [Pair \rightarrow \bullet\, \underline{(}\ Pair\ \underline{)}, \underline{(}] & [Pair \rightarrow \bullet\, \underline{(}\ \underline{)}, \texttt{eof}] \\ & [Pair \rightarrow \bullet\, \underline{(}\ \underline{)}, \underline{(}] & \end{matrix}\right\}$$

$$cc_2 = \left\{[List \rightarrow Pair\, \bullet, \texttt{eof}] \quad [List \rightarrow Pair\, \bullet, \underline{(}]\right\}$$

$$cc_3 = \left\{\begin{matrix} [Pair \rightarrow \bullet\, \underline{(}\ Pair\ \underline{)}, \underline{)}] & [Pair \rightarrow \underline{(}\, \bullet\, Pair\ \underline{)}, \texttt{eof}] & [Pair \rightarrow \underline{(}\, \bullet\, Pair\ \underline{)}, \underline{(}] \\ [Pair \rightarrow \bullet\, \underline{(}\ \underline{)}, \underline{)}] & [Pair \rightarrow \underline{(}\, \bullet\, \underline{)}, \texttt{eof}] & [Pair \rightarrow \underline{(}\, \bullet\, \underline{)}, \underline{(}] \end{matrix}\right\}$$

$$cc_4 = \left\{[List \rightarrow List\ Pair\, \bullet, \texttt{eof}] \quad [List \rightarrow List\ Pair\, \bullet, \underline{(}]\right\}$$

$$cc_5 = \left\{[Pair \rightarrow \underline{(}\ Pair\, \bullet\, \underline{)}, \texttt{eof}] \quad [Pair \rightarrow \underline{(}\ Pair\, \bullet\, \underline{)}, \underline{(}]\right\}$$

$$cc_6 = \left\{\begin{matrix} [Pair \rightarrow \bullet\, \underline{(}\ Pair\ \underline{)}, \underline{)}] & [Pair \rightarrow \underline{(}\, \bullet\, Pair\ \underline{)}, \underline{)}] \\ [Pair \rightarrow \bullet\, \underline{(}\ \underline{)}, \underline{)}] & [Pair \rightarrow \underline{(}\, \bullet\, \underline{)}, \underline{)}] \end{matrix}\right\}$$

$$cc_7 = \left\{[Pair \rightarrow \underline{(}\ \underline{)}\, \bullet, \texttt{eof}] \quad [Pair \rightarrow \underline{(}\ \underline{)}\, \bullet, \underline{(}]\right\}$$

$$cc_8 = \left\{[Pair \rightarrow \underline{(}\ Pair\ \underline{)}\, \bullet, \texttt{eof}] \quad [Pair \rightarrow \underline{(}\ Pair\ \underline{)}\, \bullet, \underline{(}]\right\}$$

$$cc_9 = \left\{[Pair \rightarrow \underline{(}\ Pair\, \bullet\, \underline{)}, \underline{)}]\right\}$$

$$cc_{10} = \left\{[Pair \rightarrow \underline{(}\ \underline{)}\, \bullet, \underline{)}]\right\}$$

$$cc_{11} = \left\{[Pair \rightarrow \underline{(}\ Pair\ \underline{)}\, \bullet, \underline{)}]\right\}$$

Constructing *Action* and *Goto* Tables (1)

```
1  ALGORITHM: BuildActionGotoTables
2    INPUT:
3      (1) a grammar G = (V, Σ, R, S)
4      (2) goal production S → S′
5      (3) a canonical collection CC = {cc0, cc1, ..., ccn}
6      (4) a transition function δ : CC × Σ → CC
7    OUTPUT: Action Table & Goto Table
8  PROCEDURE:
9    for cc_i ∈ CC:
10     for item ∈ cc_i:
11       if item = [A → β • xγ, a]\pause∧δ(cc_i, x) = cc_j then
12         Action[i, x] := shift j
13       elseif item = [A → β•, a] then
14         Action[i, a] := reduce A → β
15       elseif item = [S → S′•, eof] then
16         Action[i, eof] := accept
17       end
18     for v ∈ V:
19       if δ(cc_i, v) = cc_j then
20         Goto[i, v] = j
21       end
```

- **L12, 13**: Next valid step in discovering A is to match terminal symbol x.
- **L14, 15**: Having recognized β, if current word matches lookahead a, reduce β to A.
- **L16, 17**: Accept if input exhausted and what's recognized reducible to start var. S.
- **L20, 21**: Record consequence of a reduction to non-terminal v from state i

Constructing *Action* and *Goto* Tables (2)

Resulting **Action** and **Goto** tables:

	Action Table			Goto Table	
State	eof	(	)	***List***	***Pair***
0		s 3		1	2
1	acc	s 3			4
2	r 3	r 3			
3		s 6	s 7		5
4	r 2	r 2			
5			s 8		
6		s 6	s 10		9
7	r 5	r 5			
8	r 4	r 4			
9			s 11		
10			r 5		
11			r 4		

BUP: Discovering Ambiguity (1)

1	*Goal*	→	*Stmt*
2	*Stmt*	→	if expr then *Stmt*
3		|	if expr then *Stmt* else *Stmt*
4		|	assign

- Calculate $\mathcal{CC} = \{cc_0, cc_1, \ldots, \}$
- Calculate the transition function $\delta : \mathcal{CC} \times \Sigma \rightarrow \mathcal{CC}$

BUP: Discovering Ambiguity (2.1)

Resulting transition table:

	Item	*Goal*	*Stmt*	`if`	`expr`	`then`	`else`	`assign`	`eof`
0	cc_0	$\emptyset$	cc_1	cc_2	$\emptyset$	$\emptyset$	$\emptyset$	cc_3	$\emptyset$
1	cc_1	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
	cc_2	$\emptyset$	$\emptyset$	$\emptyset$	cc_4	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
	cc_3	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
2	cc_4	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	cc_5	$\emptyset$	$\emptyset$	$\emptyset$
3	cc_5	$\emptyset$	cc_6	cc_7	$\emptyset$	$\emptyset$	$\emptyset$	cc_8	$\emptyset$
4	cc_6	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	cc_9	$\emptyset$	$\emptyset$
	cc_7	$\emptyset$	$\emptyset$	$\emptyset$	cc_{10}	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
	cc_8	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
5	cc_9	$\emptyset$	cc_{11}	cc_2	$\emptyset$	$\emptyset$	$\emptyset$	cc_3	$\emptyset$
	cc_{10}	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	cc_{12}	$\emptyset$	$\emptyset$	$\emptyset$
6	cc_{11}	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
	cc_{12}	$\emptyset$	cc_{13}	cc_7	$\emptyset$	$\emptyset$	$\emptyset$	cc_8	$\emptyset$
7	cc_{13}	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	cc_{14}	$\emptyset$	$\emptyset$
8	cc_{14}	$\emptyset$	cc_{15}	cc_7	$\emptyset$	$\emptyset$	$\emptyset$	cc_8	$\emptyset$
9	cc_{15}	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$

BUP: Discovering Ambiguity (2.2.1)

Resulting canonical collection $\mathcal{CC}$:

$$cc_0 = \left\{ \begin{array}{ll} [Goal \rightarrow \bullet\, Stmt, \texttt{eof}] & [Stmt \rightarrow \bullet\, \texttt{if expr then}\ Stmt, \texttt{eof}] \\ [Stmt \rightarrow \bullet\, \texttt{assign}, \texttt{eof}] & [Stmt \rightarrow \bullet\, \texttt{if expr then}\ Stmt\ \texttt{else}\ Stmt, \texttt{eof}] \end{array} \right\}$$

$$cc_1 = \left\{ [Goal \rightarrow Stmt\, \bullet, \texttt{eof}] \right\}$$

$$cc_2 = \left\{ \begin{array}{l} [Stmt \rightarrow \texttt{if}\ \bullet\ \texttt{expr then}\ Stmt, \texttt{eof}], \\ [Stmt \rightarrow \texttt{if}\ \bullet\ \texttt{expr then}\ Stmt\ \texttt{else}\ Stmt, \texttt{eof}] \end{array} \right\}$$

$$cc_3 = \left\{ [Stmt \rightarrow \texttt{assign}\, \bullet, \texttt{eof}] \right\}$$

$$cc_4 = \left\{ \begin{array}{l} [Stmt \rightarrow \texttt{if expr}\ \bullet\ \texttt{then}\ Stmt, \texttt{eof}], \\ [Stmt \rightarrow \texttt{if expr}\ \bullet\ \texttt{then}\ Stmt\ \texttt{else}\ Stmt, \texttt{eof}] \end{array} \right\}$$

$$cc_5 = \left\{ \begin{array}{l} [Stmt \rightarrow \texttt{if expr then}\ \bullet\ Stmt, \texttt{eof}], \\ [Stmt \rightarrow \texttt{if expr then}\ \bullet\ Stmt\ \texttt{else}\ Stmt, \texttt{eof}], \\ [Stmt \rightarrow \bullet\ \texttt{if expr then}\ Stmt, \{\texttt{eof}, \texttt{else}\}], \\ [Stmt \rightarrow \bullet\ \texttt{assign}, \{\texttt{eof}, \texttt{else}\}], \\ [Stmt \rightarrow \bullet\ \texttt{if expr then}\ Stmt\ \texttt{else}\ Stmt, \{\texttt{eof}, \texttt{else}\}] \end{array} \right\}$$

$$cc_6 = \left\{ \begin{array}{l} [Stmt \rightarrow \texttt{if expr then}\ Stmt\ \bullet, \texttt{eof}], \\ [Stmt \rightarrow \texttt{if expr then}\ Stmt\ \bullet\ \texttt{else}\ Stmt, \texttt{eof}] \end{array} \right\}$$

$$cc_7 = \left\{ \begin{array}{l} [Stmt \rightarrow \texttt{if}\ \bullet\ \texttt{expr then}\ Stmt, \{\texttt{eof}, \texttt{else}\}], \\ [Stmt \rightarrow \texttt{if}\ \bullet\ \texttt{expr then}\ Stmt\ \texttt{else}\ Stmt, \{\texttt{eof}, \texttt{else}\}] \end{array} \right\}$$

BUP: Discovering Ambiguity (2.2.2)

Resulting canonical collection $\mathcal{CC}$:

$$cc_8 = \{[Stmt \rightarrow \texttt{assign}\, \bullet, \{\texttt{eof}, \texttt{else}\}]\}$$

$$cc_9 = \left\{ \begin{array}{l} [Stmt \rightarrow \texttt{if expr then}\ Stmt\ \texttt{else}\ \bullet\ Stmt, \texttt{eof}], \\ [Stmt \rightarrow \bullet\ \texttt{if expr then}\ Stmt, \texttt{eof}], \\ [Stmt \rightarrow \bullet\ \texttt{if expr then}\ Stmt\ \texttt{else}\ Stmt, \texttt{eof}], \\ [Stmt \rightarrow \bullet\ \texttt{assign}, \texttt{eof}] \end{array} \right\}$$

$$cc_{10} = \left\{ \begin{array}{l} [Stmt \rightarrow \texttt{if expr}\ \bullet\ \texttt{then}\ Stmt, \{\texttt{eof}, \texttt{else}\}], \\ [Stmt \rightarrow \texttt{if expr}\ \bullet\ \texttt{then}\ Stmt\ \texttt{else}\ Stmt, \{\texttt{eof}, \texttt{else}\}] \end{array} \right\}$$

$$cc_{11} = \{[Stmt \rightarrow \texttt{if expr then}\ Stmt\ \texttt{else}\ Stmt\ \bullet, \texttt{eof}]\}$$

$$cc_{12} = \left\{ \begin{array}{l} [Stmt \rightarrow \texttt{if expr then}\ \bullet\ Stmt, \{\texttt{eof}, \texttt{else}\}], \\ [Stmt \rightarrow \texttt{if expr then}\ \bullet\ Stmt\ \texttt{else}\ Stmt, \{\texttt{eof}, \texttt{else}\}], \\ [Stmt \rightarrow \bullet\ \texttt{if expr then}\ Stmt, \{\texttt{eof}, \texttt{else}\}], \\ [Stmt \rightarrow \bullet\ \texttt{if expr then}\ Stmt\ \texttt{else}\ Stmt, \{\texttt{eof}, \texttt{else}\}], \\ [Stmt \rightarrow \bullet\ \texttt{assign}, \{\texttt{eof}, \texttt{else}\}] \end{array} \right\}$$

$$cc_{13} = \left\{ \begin{array}{l} [Stmt \rightarrow \texttt{if expr then}\ Stmt\ \bullet\ , \{\texttt{eof}, \texttt{else}\}], \\ [Stmt \rightarrow \texttt{if expr then}\ Stmt\ \bullet\ \texttt{else}\ Stmt, \{\texttt{eof}, \texttt{else}\}] \end{array} \right\}$$

$$cc_{14} = \left\{ \begin{array}{l} [Stmt \rightarrow \texttt{if expr then}\ Stmt\ \texttt{else}\ \bullet\ Stmt, \{\texttt{eof}, \texttt{else}\}], \\ [Stmt \rightarrow \bullet\ \texttt{if expr then}\ Stmt, \{\texttt{eof}, \texttt{else}\}], \\ [Stmt \rightarrow \bullet\ \texttt{if expr then}\ Stmt\ \texttt{else}\ Stmt, \{\texttt{eof}, \texttt{else}\}], \\ [Stmt \rightarrow \bullet\ \texttt{assign}, \{\texttt{eof}, \texttt{else}\}] \end{array} \right\}$$

BUP: Discovering Ambiguity (3)

- Consider cc_{13}

$$cc_{13} = \left\{ \begin{array}{l} [Stmt \rightarrow \texttt{if expr then}\ Stmt\ \bullet\ , \{\texttt{eof}, \texttt{else}\}], \\ [Stmt \rightarrow \texttt{if expr then}\ Stmt\ \bullet\ \texttt{else}\ Stmt, \{\texttt{eof}, \texttt{else}\}] \end{array} \right\}$$

  Q. What does it mean if the current word to consume is `else`?
  A. We can either ***shift*** (then expecting to match another *Stmt*) or reduce to a *Stmt*.
  A single *Action* table entry cannot hold these two alternatives.
  This is known as the *shift-reduce conflict*.
- Consider another scenario, say:
  $[A \rightarrow \gamma\delta\bullet,\ \texttt{a}]$
  $[B \rightarrow \gamma\delta\bullet,\ \texttt{a}]$
  Q. What does it mean if the current word to consume is `a`?
  A. We can either ***reduce*** to *A* or ***reduce*** to *B*.
  A single *Action* table entry cannot hold these two alternatives.
  This is known as the *reduce-reduce conflict*.

Index (1)

Parser in Context

Context-Free Languages: Introduction

CFG: Example (1.1)

CFG: Example (1.2)

CFG: Example (1.2)

CFG: Example (2)

CFG: Example (3)

CFG: Example (4)

CFG: Example (5.1) Version 1

CFG: Example (5.2) Version 1

CFG: Example (5.3) Version 1

Index (2)

CFG: Example (5.4) Version 1

CFG: Example (5.5) Version 2

CFG: Example (5.6) Version 2

CFG: Example (5.7) Version 2

CFG: Formal Definition (1)

CFG: Formal Definition (2): Example

CFG: Formal Definition (3): Example

Regular Expressions to CFG's

DFA to CFG's

CFG: Leftmost Derivations (1)

CFG: Rightmost Derivations (1)

Index (3)

CFG: Leftmost Derivations (2)

CFG: Rightmost Derivations (2)

CFG: Parse Trees vs. Derivations (1)

CFG: Parse Trees vs. Derivations (2)

CFG: Ambiguity: Definition

CFG: Ambiguity: Exercise (1)

CFG: Ambiguity: Exercise (2.1)

CFG: Ambiguity: Exercise (2.2)

CFG: Ambiguity: Exercise (2.3)

Discovering Derivations

TDP: Discovering Leftmost Derivation

Index (4)

TDP: Exercise (1)

TDP: Exercise (2)

Left-Recursions (LF): Direct vs. Indirect

TDP: (Preventively) Eliminating LRs

CFG: Eliminating ϵ-Productions (1)

CFG: Eliminating ϵ-Productions (2)

Backtrack-Free Parsing (1)

The first Set: Definition

The first Set: Examples

Computing the first Set

Computing the first Set: Extension

Index (5)

Extended first Set: Examples

Is the first Set Sufficient?

The follow Set: Examples

Computing the follow Set

Backtrack-Free Grammar

TDP: Lookahead with One Symbol

Backtrack-Free Grammar: Exercise

Backtrack-Free Grammar: Left-Factoring

Left-Factoring: Exercise

TDP: Terminating and Backtrack-Free

Backtrack-Free Parsing (2.1)

Index (6)

Backtrack-Free Parsing (2.2)

LL(1) Parser: Exercise

BUP: Discovering Rightmost Derivation

BUP: Discovering Rightmost Derivation (1)

BUP: Discovering Rightmost Derivation (2)

BUP: Example Tracing (1)

BUP: Example Tracing (2.1)

BUP: Example Tracing (2.2)

BUP: Example Tracing (2.3)

LR(1) Items: Definition

LR(1) Items: Scenarios

Index (7)

LR(1) Items: Example (1.1)

LR(1) Items: Example (1.2)

LR(1) Items: Example (1.3)

LR(1) Items: Example (2)

Canonical Collection ($\mathcal{CC}$) vs. LR(1) items

Constructing $\mathcal{CC}$: The *closure* Procedure (1)

Constructing $\mathcal{CC}$: The *closure* Procedure (2.1)

Constructing $\mathcal{CC}$: The *goto* Procedure (1)

Constructing $\mathcal{CC}$: The *goto* Procedure (2)

Constructing $\mathcal{CC}$: The Algorithm (1)

Constructing $\mathcal{CC}$: The Algorithm (2.1)

Index (8)

Constructing $\mathcal{CC}$: The Algorithm (2.2)

Constructing $\mathcal{CC}$: The Algorithm (2.3)

Constructing $\mathcal{CC}$: The Algorithm (2.4)

Constructing *Action* and *Goto* Tables (1)

Constructing *Action* and *Goto* Tables (2)

BUP: Discovering Ambiguity (1)

BUP: Discovering Ambiguity (2.1)

BUP: Discovering Ambiguity (2.2.1)

BUP: Discovering Ambiguity (2.2.2)

BUP: Discovering Ambiguity (3)