CS 4240: Compilers

Lecture 21: Context Free Grammars, Top Down Parsing

Instructor: Vivek Sarkar (vsarkar@gatech.edu)

April 8, 2019

ANNOUNCEMENTS & REMINDERS

- **Project 3 assigned today**
  - **Due by 11:59pm on Tuesday, April 23rd**
  - **Automatic penalty-free extension until 11:59pm on Tuesday, April 30th**
  - 10% of course grade
- **Homework 3 will be assigned latertoday**
  - **Due by 11:59pm on Tuesday, April 23rd**
  - **Automatic penalty-free extension until 11:59pm on Friday, April 26th**
  - 5% of course grade
- FINAL EXAM: Wednesday, May 1, 2:40 pm - 5:30 pm
  - 30% of course grade

Worksheet #20 Solution

(From Lecture #20 given on 4/1/2019)

Write regular expressions for each of the following languages

Language L1,
consisting of strings of 0's and 1's that may have a 0, but **whenever a 0 occurs it must be followed by a 1.**

$\Sigma = \{0, 1\}$

To include **ε** in L1 **: (1 | 01)***

To not include **ε** in L1 **: (1 | 01)+**

Language L1

To include **ε** in L1 **: (1 | 01)***

To not include **ε** in L1 **: (1 | 01)+**

Incorrect RegEx from students (L1)

L(**(01)***) = {**ε** , 01, 0101, 010101, …} : Doesn't contain **11**

L(**(01+)***) : Doesn't contain **101**

L(**1*(01)*1***) : Doesn't contain **01101**

L(**(01*)***) : Contains **00**

L(**(0?1*)***) : Contains **00** **-> Possible Fix : (0?1+)***

L(**(1* | 01*)***) : Contains **00** **-> Possible Fix : (1* | 01+)***

Language L1

To include **ε** in L1 **: (1 | 01)***

To not include **ε** in L1 **: (1 | 01)+**

Other Correct RegEx from students (L1)

1*(01+)*

(1*(01)*)*

(1 | 01 | ε)*

ε is redundant, since the outermost * indicates that the language for the RegEx already includes ε

Language L2,
consisting of strings from the alphabet
{ a-z, 0-9, _ }. A string in L2 **must start with a lowercase letter or an underscore**, and **followed by one or more** of lowercase letters and digits and underscores. Further, **each underscore must be followed by a letter or a digit**.

Strings in L2 must satisfy the following

1. constructed from the alphabet Σ = {a-z, 0-9, _}
2. must start with a lowercase letter or an underscore.
3. length is at least 2.
4. each underscore must be followed by a letter or a digit.

Strings in Language L2…

1. constructed from the alphabet Σ = {a-z, 0-9, _}
2. must start with a lowercase letter or an underscore.
3. length is at least 2.
4. each underscore must be followed by a letter or a digit.

RegEx for strings in L2 which begin with _ :

_[a-z0-9]([a-z0-9] | _[a-z0-9])*

RegEx for strings in L2 which begin with **[a-z]** :

[a-z]([a-z0-9] | _[a-z0-9])+

Correct :

_[a-z0-9]([a-z0-9] | _[a-z0-9])* | **[a-z]([a-z0-9] | _[a-z0-9])+**

Strings in Language L2…

1. constructed from the alphabet $\Sigma = \{a\text{-}z,\ 0\text{-}9,\ \_\}$
2. must start with a lowercase letter or an underscore.
3. length is at least 2.
4. each underscore must be followed by a letter or a digit.

Incorrect RegEx from students (L2) :
Most of the students' answers were slightly incorrect

([a-z] | _[a-z0-9])([a-z0-9] | _[a-z0-9])* : matches **_**

([a-z] | _[a-z0-9])([a-z0-9] | _[a-z0-9])+ : can't match **_a**

([a-z] | _)[a-z0-9]([a-z0-9] | _[a-z0-9])* : can't match **a_a**

Language L3,
consisting of strings of 0’s and 1’s such that each string **contains at least 3 consecutive 0’s**.

$\Sigma = \{0, 1\}$

(0 | 1)* 000 (0 | 1)*

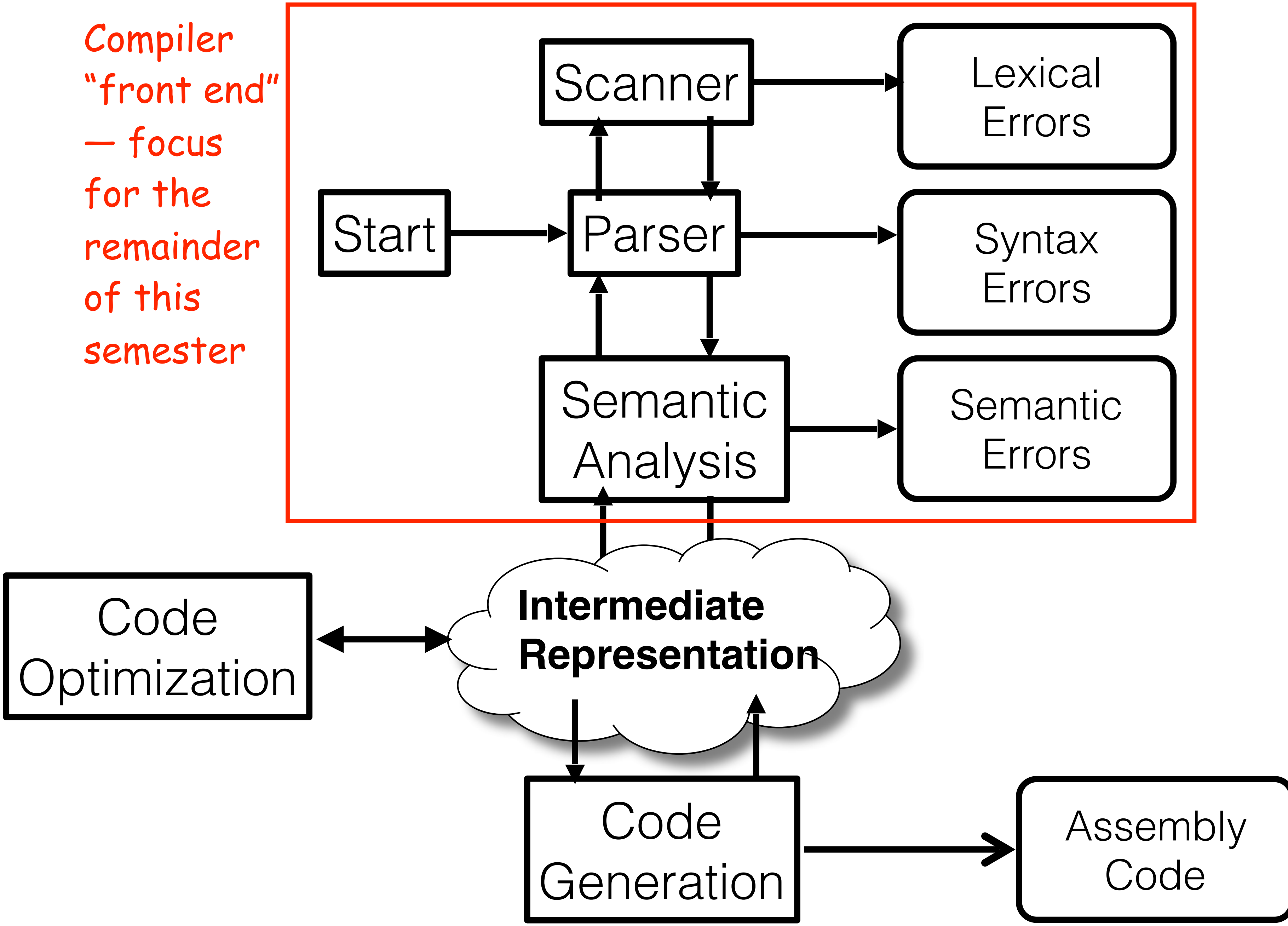
Some students used
( ) and [] as if the two were equivalent,
but they are different in meaning.

( ) : Used to change precedence, or grouping

[ ] : character classes

Structure of a Full Compiler (Recap from Lecture 1)

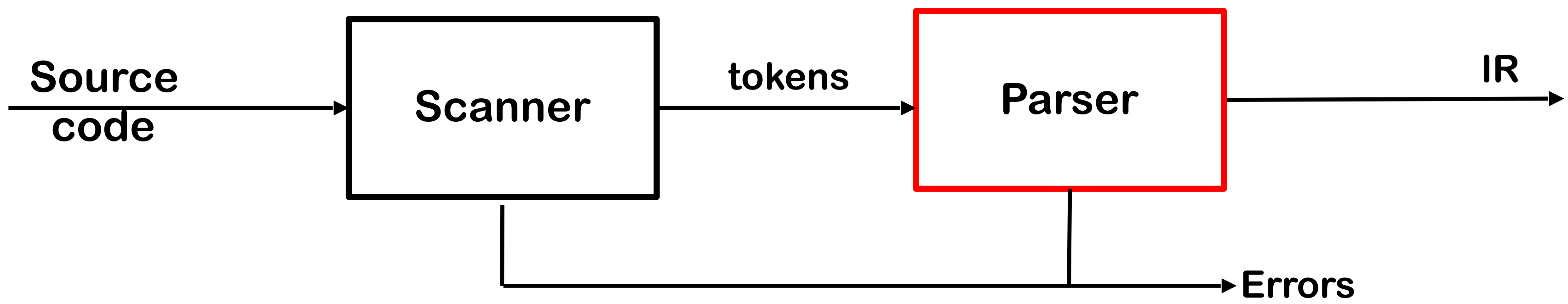
Compiler "front end" — focus for the remainder of this semester



Front-end architecture

- **Scanning**: converting source code into stream of known chunks called **tokens**
  - **Lexical** rules of language dictate how legal **token** is formed as a sequence of **alphabet symbols**
    - **Formalized using regular expressions**
- **Parsing**: building **tree-based** representation of code
  - **Grammar** dictates how legal **tree** is formed as a sequence of **tokens**
    - **Formalized using context-free grammars**

The Front End



Parser

- Checks the stream of <u>words</u> and their <u>parts of speech</u> (produced by the scanner) for grammatical correctness
- Determines if the input is syntactically well formed
- Guides checking at deeper levels than syntax
- Builds an IR representation of the code

Think of this as the mathematics of diagramming sentences

The Study of Parsing

The process of discovering a derivation for some sentence

- Need a mathematical model of syntax — a grammar *G*
- Need an algorithm for testing membership in *L*(*G*)
- Need to keep in mind that our goal is building parsers, not studying the mathematics of arbitrary languages

Roadmap

1. Context-free grammars (CFGs) and derivations
2. Top-down parsing
   - Generated LL(1) parsers & hand-coded recursive descent parsers

Specifying Syntax with a Grammar

Context-free syntax is specified with a context-free grammar

SheepNoise → SheepNoise <u>baa</u>
| <u>baa</u>

This CFG defines the set of noises sheep normally make

It is written in a variant of Backus–Naur form

Formally, a grammar is a four tuple, G = (S,N,T,P)

- S is the start symbol (set of strings in L(G))
- N is a set of non-terminal symbols (syntactic variables)
- T is a set of terminal symbols (words)
- P is a set of productions or rewrite rules $(P : N \rightarrow (N \cup T)^+)$

Example due to Dr. Scott K. Warren

Deriving Syntax

We can use the *SheepNoise* grammar to create sentences

- use the productions as rewriting rules

Rule	Sentential Form
—	*SheepNoise*
2	baa

Rule	Sentential Form
—	*SheepNoise*
1	*SheepNoise* baa
2	baa baa

Rule	Sentential Form
—	*SheepNoise*
1	*SheepNoise* baa
1	*SheepNoise* baa baa
2	baa baa baa

And so on ...

While this example is cute, it quickly runs out of intellectual steam ...

A More Useful Grammar

To explore the uses of CFGs,we need a more complex grammar

1	*Expr*	→	*Expr Op Expr*
2		|	<u>number</u>
3		|	<u>id</u>
4	*Op*	→	+
5		|	-
6		|	*
7		|	/

Rule	*Sentential Form*
—	*Expr*
1	*Expr Op Expr*
3	‹id,<u>x</u>› *Op Expr*
5	‹id,<u>x</u>› - *Expr*
1	‹id,<u>x</u>› - *Expr Op Expr*
2	‹id,<u>x</u>› - ‹num,<u>2</u>› *Op Expr*
6	‹id,<u>x</u>› - ‹num,<u>2</u>› * *Expr*
3	‹id,<u>x</u>› - ‹num,<u>2</u>› * ‹id,<u>y</u>›

- » Such a sequence of rewrites is called a derivation
- » Process of discovering a derivation is called parsing

We denote this derivation: Expr $\Rightarrow^*$ <u>id</u> - <u>num</u> * <u>id</u>

Derivations

- At each step, we choose a non-terminal to replace
- Different choices can lead to different derivations

Two derivations are of interest

- Leftmost derivation — replace leftmost NT at each step
  - Used in top-down LL parsing
- Rightmost derivation — replace rightmost NT at each step
  - Used in bottom-up LR parsing

Ambiguous Grammars

Definitions

- If a grammar has more than one leftmost derivation for a single sentential form, the grammar is ambiguous
- If a grammar has more than one rightmost derivation for a single sentential form, the grammar is ambiguous

Classic example — the if-then-else problem

$$
\begin{aligned}
\text{Stmt} \rightarrow\ & \underline{\text{if}}\ \text{Expr}\ \underline{\text{then}}\ \text{Stmt} \\
|\ & \underline{\text{if}}\ \text{Expr}\ \underline{\text{then}}\ \text{Stmt}\ \underline{\text{else}}\ \text{Stmt} \\
|\ & \ldots \text{ other stmts } \ldots
\end{aligned}
$$

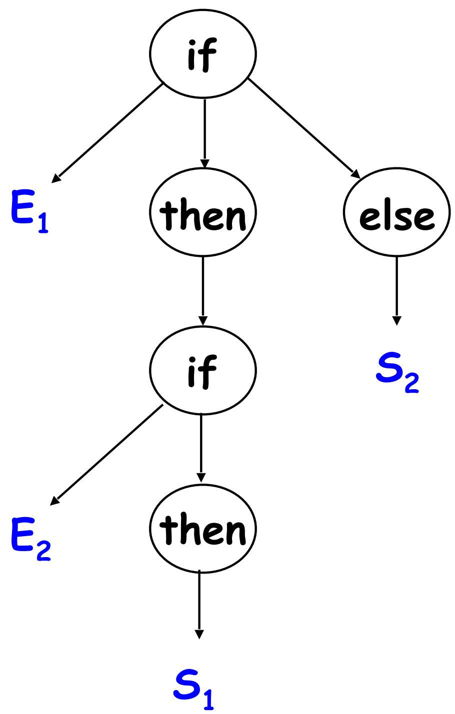
This ambiguity is entirely grammatical in nature

Ambiguity

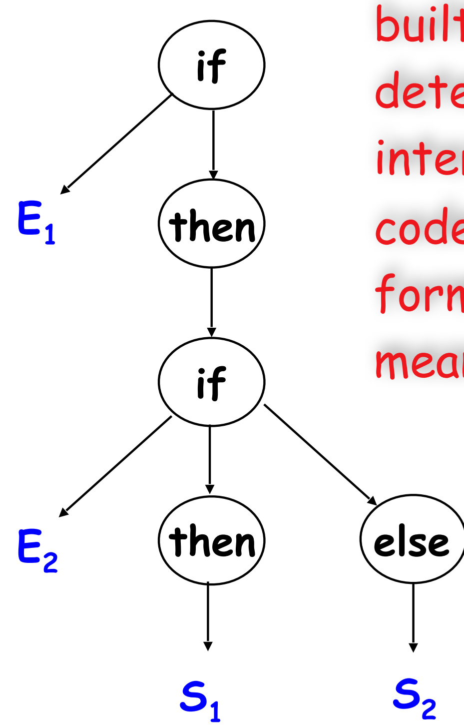
This sentential form has two derivations

<u>if</u> $Expr_1$ <u>then</u> <u>if</u> $Expr_2$ <u>then</u> $Stmt_1$ <u>else</u> $Stmt_2$

Part of the problem is that the structure built by the parser will determine the interpretation of the code, and these two forms have different meanings!



production 2, then production 1



production 1, then production 2

Ambiguity

The grammar forces the structure to match the desired meaning.

Removing the ambiguity

» Must rewrite the grammar to avoid generating the problem

» Match each else to innermost unmatched if (common sense rule)

1	Stmt	→	if Expr then Stmt
2		|	if Expr then WithElse else Stmt
3		|	Other Statements
4	WithElse	→	if Expr then WithElse else WithElse
5		|	Other Statements

With this grammar, example has only one rightmost derivation

Intuition: once into WithElse, we cannot generate an unmatched else

… a final if without an else can only come through rule 2 …

Ambiguity

___ if $Expr_1$ then if $Expr_2$ then $Stmt_1$ else $Stmt_2$

Rule	*Sentential Form*
—	*Stmt*
1	if *Expr* then *Stmt*
2	if *Expr* then if *Expr* then *WithElse* else *Stmt*
3	if *Expr* then if *Expr* then *WithElse* else S_2
5	if *Expr* then if *Expr* then S_1 else S_2
?	if *Expr* then if E_2 then S_1 else S_2
?	if E_1 then if E_2 then S_1 else S_2

some other production

This grammar has only one rightmost derivation for the example

Parsing Techniques

Top-down parsers (LL(1), recursive descent)

- Start at the root of the parse tree and grow toward leaves
- Pick a production & try to match the input
- Bad "pick" $\Rightarrow$ may need to backtrack
- Some grammars are backtrack-free (predictive parsing)

Bottom-up parsers (LR(1), operator precedence)

- Start at the leaves and grow toward root
- As input is consumed, encode possibilities in an internal state
- Start in a state valid for legal first tokens
- Bottom-up parsers handle a large class of grammars

Top-down Parsing

A top-down parser starts with the root of the parse tree

The root node is labeled with the goal symbol of the grammar

Top-down parsing algorithm:

> Construct the root node of the parse tree
>
> Repeat until the lower fringe of the parse tree matches the input string
>
> 1. At a node labeled *A*, select a production with *A* on its lhs and, for each symbol on its rhs, construct the appropriate child
> 2. When a terminal symbol is added to the fringe and it doesn't match the fringe, backtrack
> 3. Find the next node to be expanded (label $\in$ NT)

The key is picking the right production in step 1

- That choice should be guided by the input string

Left Recursion

Top-down parsers cannot handle left-recursive grammars

Formally,

A grammar is left recursive if $\exists\, A \in NT$ such that
$\exists$ a derivation $A \Rightarrow^{+} A\alpha$, for some string $\alpha \in (NT \cup T)^{+}$

Our expression grammar is left recursive

» This can lead to non-termination in a top-down parser
» For a top-down parser, any recursion must be right recursion
» We would like to convert the left recursion to right recursion

Non-termination is <u>always</u> a bad property in a compiler

Eliminating Left Recursion

To remove left recursion, we can transform the grammar

Consider a grammar fragment of the form

$$\begin{aligned} \text{Fee} &\rightarrow \text{Fee}\ \alpha \\ &\mid\ \beta \end{aligned}$$

where neither α nor β start with Fee

We can rewrite this fragment as

$$\begin{aligned} \text{Fee} &\rightarrow \beta\ \text{Fie} \\ \text{Fie} &\rightarrow \alpha\ \text{Fie} \\ &\mid\ \varepsilon \end{aligned}$$

where Fie is a new non-terminal

The new grammar defines the same language as the old grammar, using only right recursion.

Added a reference to the empty string

Eliminating Left Recursion

The expression grammar contains two cases of left recursion

Expr	$\rightarrow$	*Expr* + *Term*	*Term*	$\rightarrow$	*Term* * *Factor*
	|	*Expr* - *Term*		|	*Term* / Factor
	|	*Term*		|	*Factor*

Applying the transformation yields

Expr	$\rightarrow$	*Term* *Expr′*	*Term*	$\rightarrow$	*Factor* *Term′*
Expr′	$\rightarrow$	+ *Term* *Expr′*	*Term′*	$\rightarrow$	* *Factor* *Term′*
	|	- *Term* *Expr′*		|	/ *Factor* *Term′*
	|	ε		|	ε

These fragments use only right recursion

They retain the original left associativity

Eliminating Left Recursion

Substituting them back into the grammar yields

1	*Goal*	→	*Expr*
2	*Expr*	→	*Term Expr′*
3	*Expr′*	→	+ *Term Expr′*
4		|	- *Term Expr′*
5		|	ε
6	*Term*	→	*Factor Term′*
7	*Term′*	→	* *Factor Term′*
8		|	/ *Factor Term′*
9		|	ε
10	*Factor*	→	<u>number</u>
11		|	<u>id</u>
12		|	<u>(</u> *Expr* <u>)</u>

- This grammar is correct, if somewhat non-intuitive.
- It is left associative, as was the original
- A top-down parser will terminate using it.
- A top-down parser may need to backtrack with it.

Eliminating Left Recursion

The transformation eliminates immediate left recursion

What about more general, indirect left recursion ?

The general algorithm:

arrange the NTs into some order $A_1, A_2, \ldots, A_n$

for i $\leftarrow$ 1 to n

for s $\leftarrow$ 1 to i - 1

replace each production $A_i \rightarrow A_s\gamma$ with $A_i \rightarrow \delta_1\gamma \mid \delta_2\gamma \mid \ldots \mid \delta_k\gamma$,

where $A_s \rightarrow \delta_1 \mid \delta_2 \mid \ldots \mid \delta_k$ are all the current productions for A_s

eliminate any immediate left recursion on A_i

using the direct transformation

This assumes that the initial grammar has no cycles ($A_i \Rightarrow^+ A_i$),

and no epsilon productions

Eliminating Left Recursion

How does this algorithm work?

1. Impose arbitrary order on the non-terminals
2. Outer loop cycles through NT in order
3. Inner loop ensures that a production expanding A_i has no non-terminal A_s in its rhs, for $s < i$
4. Last step in outer loop converts any direct recursion on A_i to right recursion using the transformation showed earlier
5. New non-terminals are added at the end of the order & have no left recursion

At the start of the i^{th} outer loop iteration

For all $k < i$, no production that expands A_k contains a non-terminal A_s in its rhs, for $s < k$

Example

» Order of symbols: G, E, T

1. A_i = G	2. A_i = E	3. A_i = T, A_s = E	4. A_i = T
G → E	G → E	G → E	G → E
E → E + T	E → T E'	E → T E'	E → T E'
E → T	E' → + T E'	E' → + T E'	E' → + T E'
T → E ~ T	E' → ε	E' → ε	E' → ε
T → <u>id</u>	T → E ~ T	T → T E' ~ T	T → <u>id</u> T'
	T → <u>id</u>	T → <u>id</u>	T' → E' ~ T T'
			T' → ε