RUTGERS
THE STATE UNIVERSITY OF NEW JERSEY

CS415 Compilers

Syntax Analysis
Part 2

Announcements

- Midterm has been grades. Exams will be handed out in recitation on Wednesday. Please see canvas for grades.

- Third homework has been posted. NEW deadline:
  Due Thursday, March 10

- First project (local instruction scheduler) deadlines:
  code: March 9 @ 11:59pm – single tar file
  report: March 11 @ 11:59pm – single pdf file
  Late policy:
  → Grace period: 1 hour
  → 20% penalty for every started 24 hour period after the deadline.
  → Saturday/Sunday count as a single 24 hour period.

- Warning grades will be submitted by Friday, March 11

RUTGERS

Parsing
(Syntax Analysis)

Top-Down Parsing

EAC Chapters 3.3

Parsing Techniques: Top-down parsers

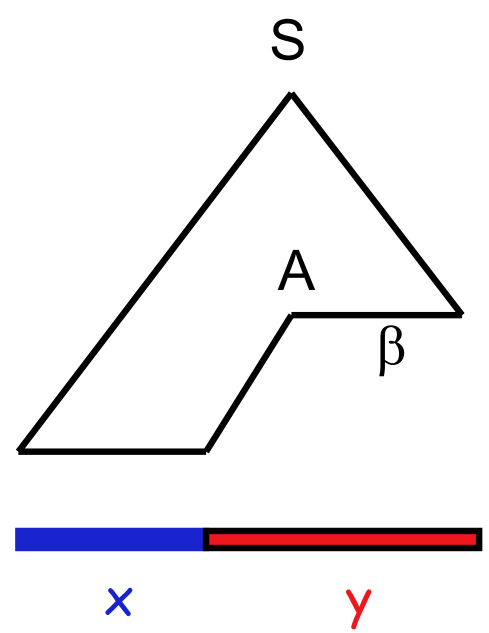
LL(1), recursive descent

1 input symbol lookahead

construct leftmost deriviation (forwards)

input: read left-to-right

$$S \Rightarrow^*_{lm} x\, A\, \beta \Rightarrow_{lm} x\, \delta\, \beta \Rightarrow^*_{lm} x\, y$$



Parsing Techniques: Top-down parsers

LL(1), recursive descent

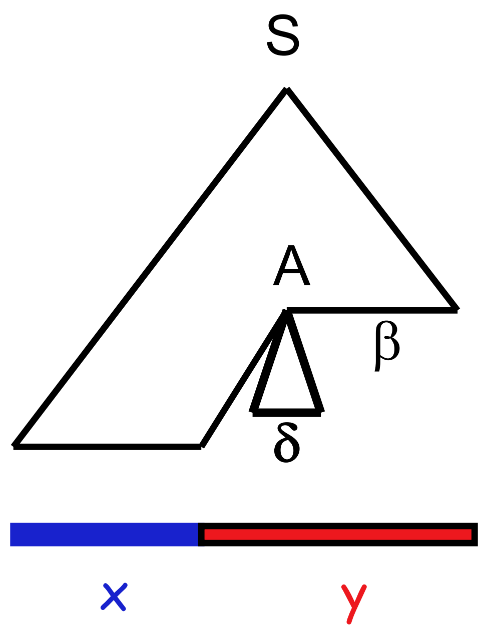
1 input symbol lookahead

construct leftmost derivation (forwards)

input: read left-to-right

rule $A \rightarrow \delta$

$$S \Rightarrow^*_{lm} x A \beta \Rightarrow_{lm} x \delta \beta \Rightarrow^*_{lm} x\ y$$



Top-down vs. Bottom-up decision

Scientist → *Richard_Feynman* | *Albert_Einstein*

Top-down This is what you see on the input before you make your rule decision:

How much lookahead do you need?

Are we looking at either Richard Feynman or Albert Einstein?

RUTGERS

Top-down vs. Bottom-up decision

? ?

Scientist → *Richard_Feynman | Albert_Einstein*

Top-down This is what you see on the input before you make your rule decision:

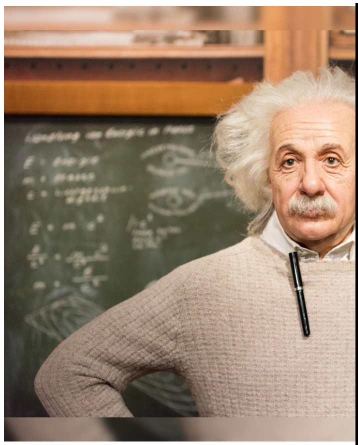
How much lookahead do you need?

Are we looking at either Richard Feynman or Albert Einstein?

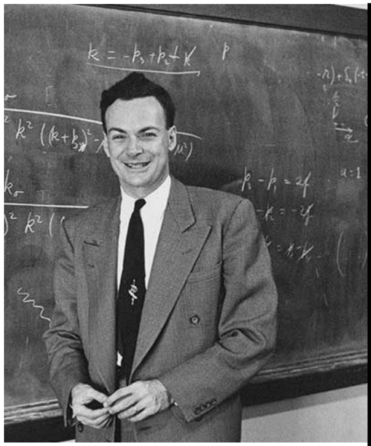
Top-down vs. Bottom-up decision

Scientist → *Richard_Feynman* | *Albert_Einstein*

Top-down This is what you see on the input before you make your rule decision:



How much lookahead do you need?



Are we looking at either Richard Feynman or Albert Einstein?

Remember the expression grammar?

Version with precedence

1	*Goal*	$\rightarrow$	*Expr*
2	*Expr*	$\rightarrow$	*Expr* + *Term*
3		|	*Expr* – *Term*
4		|	*Term*
5	*Term*	$\rightarrow$	*Term* * *Factor*
6		|	*Term* / *Factor*
7		|	*Factor*
8	*Factor*	$\rightarrow$	<u>number</u>
9		|	<u>id</u>

And the input <u>x</u> - <u>2</u> * <u>y</u>

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 a bad property in any part of a compiler

Eliminating Left Recursion

To remove left recursion, we can transform the grammar

Consider a grammar fragment of the form

$$Fee \rightarrow Fee \ \alpha$$
$$\quad\quad | \ \beta$$

where neither α nor β start with *Fee*

We can rewrite this as

$$Fee \rightarrow \beta \ Fie$$
$$Fie \rightarrow \alpha \ Fie$$
$$\quad\quad | \ \varepsilon$$

where *Fie* is a new non-terminal

This accepts the same language, but uses only right recursion

Eliminating Left Recursion

The expression grammar contains two cases of left recursion

Expr	→	*Expr* + *Term*	*Term*	→	*Term* * *Factor*
	|	*Expr* – *Term*		|	*Term* / Factor
	|	*Term*		|	*Factor*

Applying the transformation yields

Expr	→	*Term Expr'*	*Term*	→	*Factor Term'*
Expr'	|	+ *Term Expr'*	*Term'*	|	* *Factor Term'*
	|	- *Term Expr'*		|	/ *Factor Term'*
	|	ε		|	ε

These fragments use only right recursion

Eliminating Left Recursion

Substituting them back into the grammar yields

1	*Goal*	$\rightarrow$	*Expr*
2	*Expr*	$\rightarrow$	*Term Expr'*
3	*Expr'*	$\rightarrow$	+ *Term Expr'*
4		|	- *Term Expr'*
5		|	ε
6	*Term*	$\rightarrow$	*Factor Term'*
7	*Term'*	$\rightarrow$	* *Factor Term'*
8		|	/ *Factor Term'*
9		|	ε
10	*Factor*	$\rightarrow$	number
11		|	id
12		|	(*Expr*)

- This grammar is correct, if somewhat non-intuitive.
- A top-down parser will terminate using it.
- A top-down parser may need to backtrack with it.
- General left recursion removal algorithm in EAC

Roadmap (Where are we?)

We set out to study parsing

- Specifying syntax
  - → Context-free grammars
  - → Ambiguity
- Top-down parsers
  - → Algorithm & its problem with left recursion
  - → Left-recursion removal
  - → Left factoring (will discuss later)
- Predictive top-down parsing
  - → The LL(1) condition
  - → Table-driven LL(1) parsers
  - → Recursive descent parsers
    - Syntax directed translation (example)

Picking the "Right" Production

If it picks the wrong production, a top-down parser may backtrack

Alternative is to look ahead in input & use context to pick correctly

How much lookahead is needed?

- In general, an arbitrarily large amount
- Use the Cocke-Younger, Kasami algorithm or Earley's algorithm

Fortunately,

- Large subclasses of CFGs can be parsed with limited lookahead
- Most programming language constructs fall in those subclasses

Among the interesting subclasses are *LL(1)* and *LR(1)* grammars

Predictive Parsing

Basic idea

Given $A \rightarrow \alpha \mid \beta$, the parser should be able to choose between α & β

FIRST sets

For some *rhs* $\alpha \in G$, define FIRST(α) as the set of tokens that appear as the first symbol in some string that derives from α

That is, $a \in$ FIRST(α) *iff* $\alpha \Rightarrow^* a\,\gamma$, for some γ

The LL(1) Property

If $A \rightarrow \alpha$ and $A \rightarrow \beta$ both appear in the grammar, we would like

$$\text{FIRST}(\alpha) \cap \text{FIRST}(\beta) = \varnothing$$

This would allow the parser to make a correct choice with a lookahead of exactly one symbol !

This is almost correct, but not quite

The FIRST Set – 1 symbol lookahead

$$a \in \text{FIRST}_1(\alpha) \textit{ iff } \alpha \Rightarrow^* a\,\gamma, \text{ for some } \gamma$$

To build FIRST(X) for all grammar symbols X:

1. if X is a terminal (token), FIRST(X) := { X }
2. if $X \to \varepsilon$, then $\varepsilon \in$ FIRST(X)

3. <u>iterate until</u> no more terminals or ε can be added to any FIRST(X):

   if $X \to Y_1 Y_2 \ldots Y_k$ then

   $a \in$ FIRST(X) if $a \in$ FIRST(Y_i) and $\varepsilon \in$ FIRST(Y_j) for all $1 \le j < i$

   $\varepsilon \in$ FIRST(X) if $\varepsilon \in$ FIRST(Y_i) for all $1 \le i \le k$

   <u>end iterate</u>

Note: if $\varepsilon \notin$ FIRST(Y_1), then FIRST(Y_i) is irrelevant, for $1 < i$

The FIRST Set

$a \in \text{FIRST}(\alpha)$ *iff* $\alpha \Rightarrow^* \underline{a}\,\gamma$, for some γ

To build FIRST(α) for $\alpha = X_1 X_2 \ldots X_n$:

1. $a \in \text{FIRST}(\alpha)$ if $a \in \text{FIRST}(X_i)$ and
   $\varepsilon \in \text{FIRST}(X_j)$ for all $1 \le j < i$

2. $\varepsilon \in \text{FIRST}(\alpha)$ if $\varepsilon \in \text{FIRST}(X_i)$ for all $1 \le i \le n$

The FOLLOW Set - 1 symbol

For a non-terminal A, define FOLLOW(A) as

FOLLOW(A) := the set of terminals that can appear immediately to the right of A in some sentential form.

Thus, a non-terminal's FOLLOW set specifies the tokens that can legally appear after it; a terminal has no FOLLOW set

$$\text{FOLLOW}(A) = \{\, a \in (T \cup \{\text{eof}\}) \mid S\ \text{eof} \Rightarrow^{*} \alpha\ A\ a\ \gamma \,\}$$

The FOLLOW Set

To build FOLLOW(X) for all non-terminal X:

1. Place eof in FOLLOW(<*goal*>)

   iterate until no more terminals or eof can be added to any FOLLOW(X):

2. If $A \rightarrow \alpha B \beta$ then put $\{\text{FIRST}(\beta) - \varepsilon\}$ in FOLLOW(B)
3. If $A \rightarrow \alpha B$ then put FOLLOW(A) in FOLLOW(B)
4. If $A \rightarrow \alpha B \beta$ and $\varepsilon \in \text{FIRST}(\beta)$ then put FOLLOW(A) in FOLLOW(B)

Predictive Parsing

If $A \rightarrow \alpha$ and $A \rightarrow \beta$ and $\varepsilon \in$ FIRST(α), then we need to ensure that FIRST(β) is disjoint from FOLLOW(A), too

Define FIRST$^+$(δ) for rule $A \rightarrow \delta$ as

- (FIRST(δ) - { ε }) $\cup$ FOLLOW(A), if $\varepsilon \in$ FIRST(δ)
- FIRST(δ), otherwise

Predictive Parsing

The LL(1) Property

A grammar is *LL(1)* iff $A \to \alpha$ and $A \to \beta$ implies $\text{FIRST}^+(\alpha) \cap \text{FIRST}^+(\beta) = \varnothing$

This would allow the parser to make a correct choice with a lookahead of exactly one symbol !

<u>Question</u>: Can there be two rules $A \to \alpha$ and $A \to \beta$ in a *LL(1)* grammar such that $\varepsilon \in \text{FIRST}(\alpha)$ and $\varepsilon \in \text{FIRST}(\beta)$?

Predictive Parsing

Given a grammar that has the *LL(1)* property

- Problem: NT A needs to be replaced in next derivation step
- Assume $A \rightarrow \beta_1 \mid \beta_2 \mid \beta_3$, with

  $\text{FIRST}^+(\beta_1) \cap \text{FIRST}^+(\beta_2) = \varnothing$, $\text{FIRST}^+(\beta_1) \cap \text{FIRST}^+(\beta_3) = \varnothing$, and $\text{FIRST}^+(\beta_2) \cap \text{FIRST}^+(\beta_3) = \varnothing$ (pair-wise disjoint sets)

```
/* find rule for A */
if (current token ∈ FIRST+ (β1))
    select A → β1
else if (current token ∈ FIRST+(β2))
    select A → β2
else if (current token ∈ FIRST+(β3))
    select A → β3
else
    report an error and return false
```

Grammars with the *LL(1)* property are called *predictive grammars* because the parser can "predict" the correct expansion at each point in the parse.

Parsers that capitalize on the *LL(1)* property are called *predictive parsers*.

One kind of predictive parser is the *recursive descent* parser. *The other is a table-driven parser* *table-driven parser.*

Next class

More Syntax Analysis

Top-down: Read EaC: Chapter 3.3

Bottom-up: Read EaC: Chapter 3.4