RUTGERS

THE STATE UNIVERSITY OF NEW JERSEY

CS415 Compilers

Syntax Analysis

These slides are based on slides copyrighted by Keith Cooper, Ken Kennedy & Linda Torczon at Rice University

Announcements

- Third homework has been posted. Due Monday, March 7

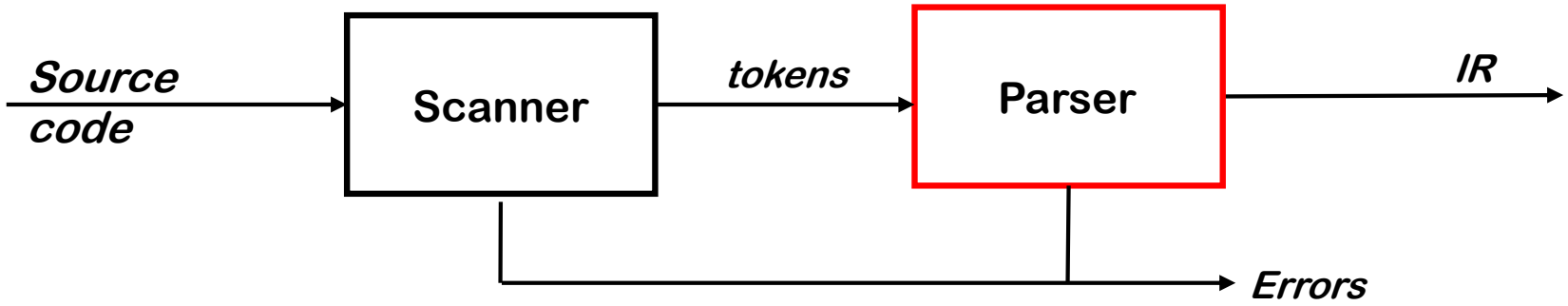
- First project (local instruction scheduler) NEW deadlines:
  code: March 9 @ 11:59pm – single tar file
  report: March 11 @ 11:59pm – single pdf file
  Submission site is now open on canvas

- 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.

Parsing (Syntax Analysis)

EAC Chapters 3.1 - 3.2

Review: The Front End



Parser

- Checks the stream of words and their parts of speech (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

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 and derivations
2. Top-down parsing
   - → LL(1) parsers, hand-coded recursive descent parsers
3. Bottom-up parsing
   - → Automatically generated LR(1) parsers

Specifying Syntax with a Grammar

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

$$\begin{aligned} \textit{SheepNoise} &\rightarrow \textit{SheepNoise}\ \underline{\text{baa}} \\ &\mid\ \underline{\text{baa}} \end{aligned}$$

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 or tokens)*
- P is a set of *productions* or *rewrite rules* $(P : N \rightarrow (N \cup T)^*)$

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

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 …

A Simple Expression Grammar

x – 2 * y ∈ L(G) ?

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

1	*Expr*	→	*Expr Op Expr*
2		|	number
3		|	id
4	*Op*	→	+
5		|	–
6		|	*
7		|	/

Rule	Sentential Form
—	*Expr*
1	*Expr Op Expr*
3	‹id,x› *Op Expr*
5	‹id,x› – *Expr*
1	‹id,x› – *Expr Op Expr*
2	‹id,x› – ‹num,2› *Op Expr*
6	‹id,x› – ‹num,2› * *Expr*
3	‹id,x› – ‹num,2› * ‹id,y›

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

We denote this derivation: *Expr* ⇒* id – num * id

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;
  generates left sentential forms ($\Rightarrow^*_{lm}$)
- *Rightmost derivation* — replace rightmost NT at each step;
  generates right sentential forms ($\Rightarrow^*_{rm}$)

These are the two *systematic* derivations

(We don't care about randomly-ordered derivations!)

The example on the preceding slide was a *leftmost* derivation

- Of course, there is also a *rightmost* derivation
- Interestingly, the resulting parse trees may be different

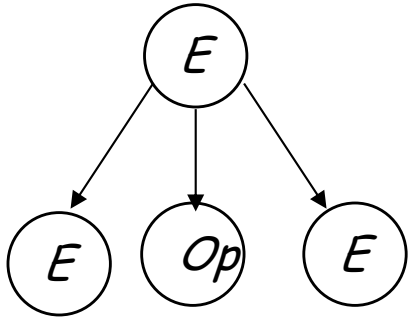
Parse Trees

Rule in our grammar:
$E \rightarrow$ E Op E

A single derivation step

$\ldots E \ldots \Rightarrow \ldots E\ Op\ E \ldots$

can be represented as a tree structure with the left-hand side non-terminal as the root, and all right-hand side symbols as the children (ordered left to right).



The entire derivation of a sentence in the language can be represented as a **parse tree** with the start symbol as its root, and leave nodes that are all terminal symbols.

NOTE: The structure of the parse tree has semantic significance!

Structure Encodes Semantics

<sentence> ::= <subject> <verb> <rest>

Example:

time flies like an arrow
fruit flies like a banana.

The Two Derivations for x - 2 * y

Rule	Sentential Form
—	*Expr*
1	*Expr Op Expr*
3	‹id,x› *Op Expr*
5	‹id,x› – *Expr*
1	‹id,x› – *Expr Op Expr*
2	‹id,x› – ‹num,2› *Op Expr*
6	‹id,x› – ‹num,2› * *Expr*
3	‹id,x› – ‹num,2› * ‹id,y›

Leftmost derivation

Rule	Sentential Form
—	*Expr*
1	*Expr Op Expr*
3	*Expr Op* ‹id,y›
6	*Expr* * ‹id,y›
1	*Expr Op Expr* * ‹id,y›
2	*Expr Op* ‹num,2› * ‹id,y›
5	*Expr* – ‹num,2› * ‹id,y›
3	‹id,x› – ‹num,2› * ‹id,y›

Rightmost derivation

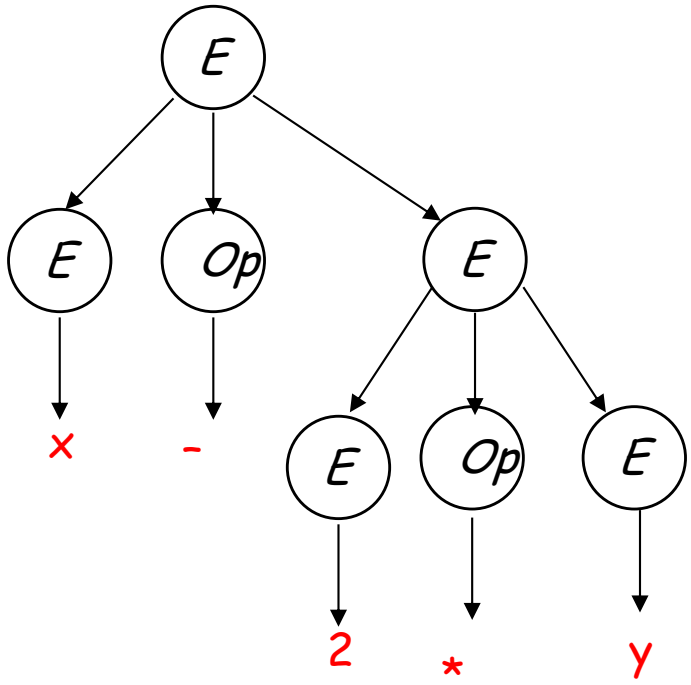
In both cases, *Expr* ⇒* id – num * id

- The two derivations produce different parse trees
- The parse trees imply different evaluation orders!

Derivations and Parse Trees

Leftmost derivation

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

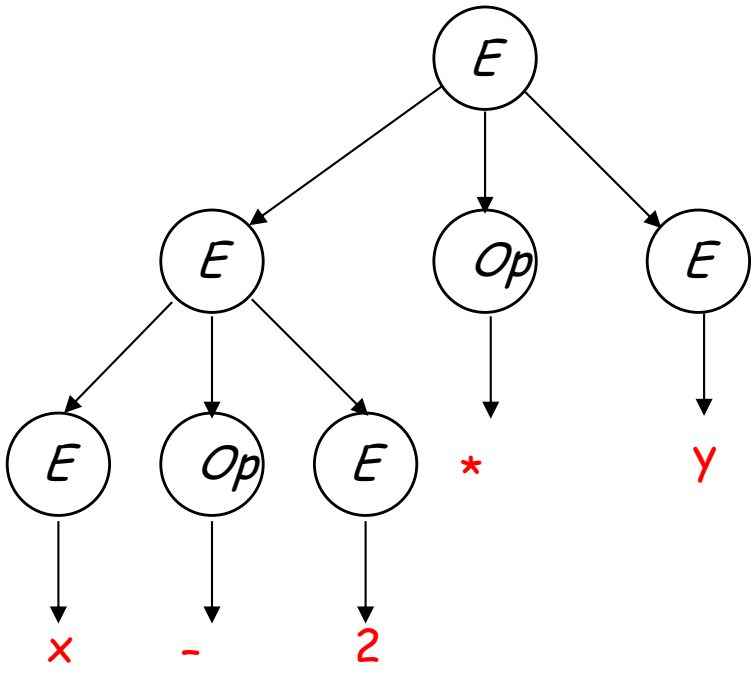


This evaluates as x - (2 * y)

Derivations and Parse Trees

This corresponds to our rightmost derivation.

Can we get this with another leftmost derivation as well?



This evaluates as (x - 2) * y

Two Leftmost Derivations for x - 2 * y

The Difference:

- Different productions chosen on the second step

Rule	*Sentential Form*
—	*Expr*
1	*Expr Op Expr*
(3)	<id,x> *Op Expr*
5	<id,x> - *Expr*
1	<id,x> - *Expr Op Expr*
2	<id,x> - <num,2> *Op Expr*
6	<id,x> - <num,2> * *Expr*
3	<id,x> - <num,2> * <id,y>

Original choice

Rule	*Sentential Form*
—	*Expr*
1	*Expr Op Expr*
(1)	*Expr Op Expr Op Expr*
3	<id,x> *Op Expr Op Expr*
5	<id,x> - *Expr Op Expr*
2	<id,x> - <num,2> *Op Expr*
6	<id,x> - <num,2> * *Expr*
3	<id,x> - <num,2> * <id,y>

New choice

- Both derivations succeed in producing x - 2 * y

Derivations and Precedence

These two derivations point out a problem with the grammar.

*How to resolve **ambiguity?***

Answer: Change expression grammar to enforce operator precedence and associativity

To add precedence

- Create a non-terminal for each *level of precedence*
- Isolate the corresponding part of the grammar
- Force the parser to recognize high precedence subexpressions first

For algebraic expressions

- Multiplication and division, first (*level one*)
- Subtraction and addition, next (*level two*)

Note: we are ignoring the issue of associativity for now

Derivations and Precedence

Adding the standard algebraic precedence produces:

	1	*Goal*	→	*Expr*
level two	2	*Expr*	→	*Expr* + *Term*
	3		|	*Expr* – *Term*
	4		|	*Term*
level one	5	*Term*	→	*Term* * *Factor*
	6		|	*Term* / *Factor*
	7		|	*Factor*
	8	*Factor*	→	number
	9		|	id

This grammar is slightly larger

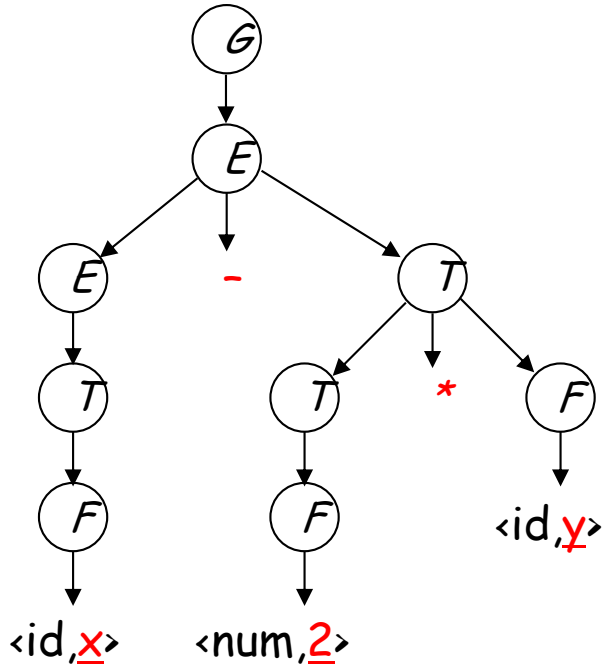
- Takes more rewriting to reach some of the terminal symbols
- Encodes expected precedence
- Produces same parse tree under leftmost & rightmost derivations

*Let's see how it parses x - 2 * y*

Derivations and Precedence

Rule	Sentential Form
—	*Goal*
1	*Expr*
3	*Expr* – *Term*
5	*Expr* – *Term* * *Factor*
9	*Expr* – *Term* * ‹id,y›
7	*Expr* – *Factor* * ‹id,y›
8	*Expr* – ‹num,2›* ‹id,y›
4	*Term* – ‹num,2›* ‹id,y›
7	*Factor* – ‹num,2›* ‹id,y›
9	‹id,x›– ‹num,2›* ‹id,y›

The rightmost derivation



Its parse tree

This produces x – (2 * y), along with an appropriate parse tree.

Both the leftmost and rightmost derivations give the same expression, because the grammar directly encodes the desired precedence.

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*
- The leftmost and rightmost derivations for a sentential form may differ, even in an unambiguous grammar

Classic example — the *if-then-else* problem

Stmt → if *Expr* then *Stmt*
| if *Expr* then *Stmt* else *Stmt*
| *… other stmts …*

This ambiguity is entirely grammatical in nature

Ambiguity

This sentential form has two derivations

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

Ambiguity

Removing the ambiguity

- Must rewrite the grammar to avoid generating the problem
- Match each else to innermost unmatched if *(common sense rule)*

1	*Stmt*	→	*WithElse*
2		|	*NoElse*
3	*WithElse*	→	if *Expr* then *WithElse* else *WithElse*
4		|	*OtherStmt*
5	*NoElse*	→	if *Expr* then *Stmt*
6		|	if *Expr* then *WithElse* else *NoElse*

With this grammar, the example has only one derivation

Ambiguity

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

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

This binds the else controlling S_2 to the inner if

Deeper Ambiguity

Ambiguity usually refers to confusion in the CFG

Overloading can create deeper ambiguity

a = f(17)

In many Algol-like languages, f could be either a function or a subscripted array variable

Disambiguating this one requires context

- Really an issue of *type*, not context-free syntax
- Requires an extra-grammatical solution (not in CFG)
- Must handle these with a different mechanism
  - → Step outside grammar rather than use a more complex grammar

Ambiguity - the Final Word

Ambiguity arises from two distinct sources

- Confusion in the context-free syntax (*if-then-else*)
- Confusion that requires context to resolve (*overloading*)

Resolving ambiguity

- To remove context-free ambiguity, rewrite the grammar
- Change language (e.g.: if … endif)
- To handle context-sensitive ambiguity takes cooperation
  - → Knowledge of declarations, types, …
  - → Accept a superset of *L(G)* & check it by other means†
  - → This is a language design problem

Sometimes, the compiler writer accepts an ambiguous grammar

- → Parsing techniques that "do the right thing"
- → *i.e.,* always select the same derivation

†See Chapter 4

Parsing (Syntax Analysis)

Top-Down Parsing

EAC Chapters 3.3

Parsing Techniques: Top-down parsers

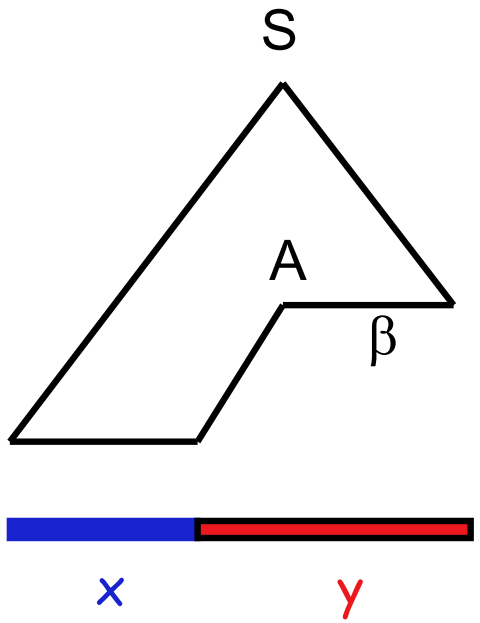
LL(1), recursive descent

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

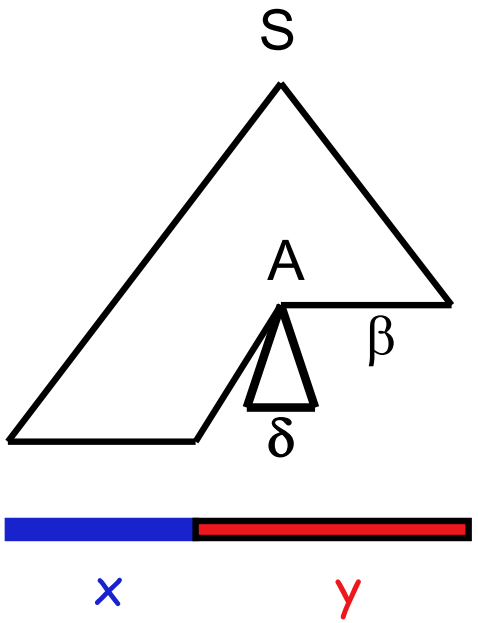
1 input symbol lookahead

construct leftmost deriviation (forwards)

input: read left-to-right

rule $A \rightarrow \delta$

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



Parsing Techniques: Top-down parsers

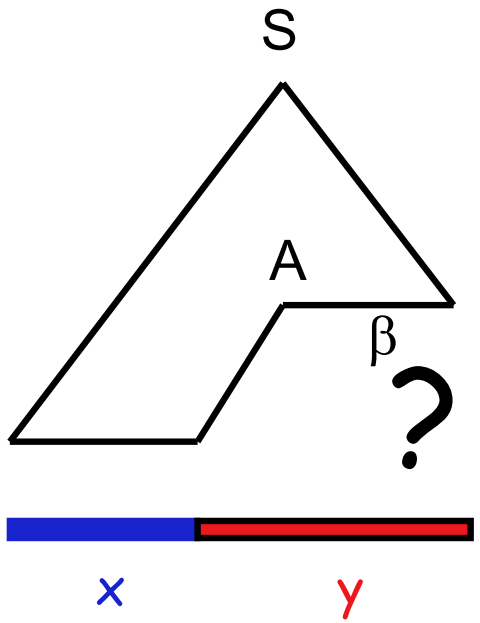
LL(1), recursive descent

1 input symbol lookahead

construct leftmost deriviation (forwards)

input: read left-to-right

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



? Means that we don't know yet this part of the parse tree

Parsing Techniques: Bottom-up parsers

LR(1), operator precedence

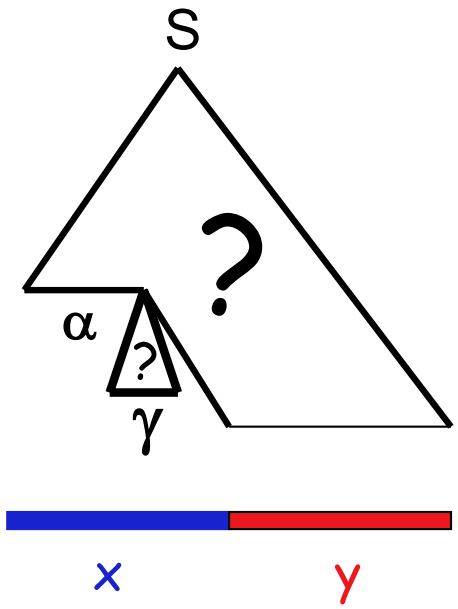
1 input symbol lookahead

construct rightmost deriviation (backwards)

input: read left-to-right

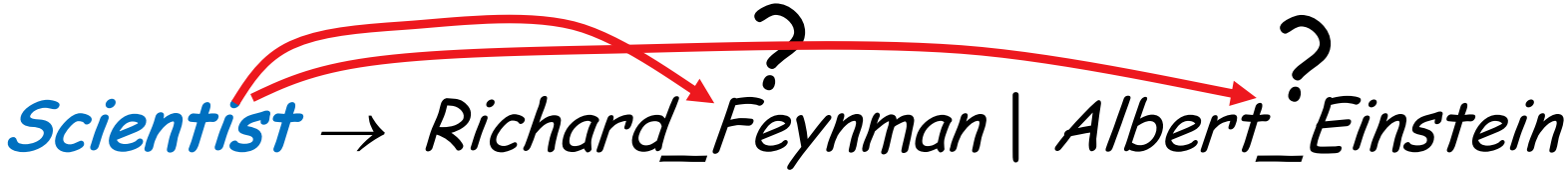
rule B ::= γ

$S \Rightarrow^*_{rm} \alpha B y \Rightarrow_{rm} \alpha \gamma y \Rightarrow^*_{rm} x y$



? Means that we don't know yet this part of the parse tree

Top-down vs. Bottom-up decision



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

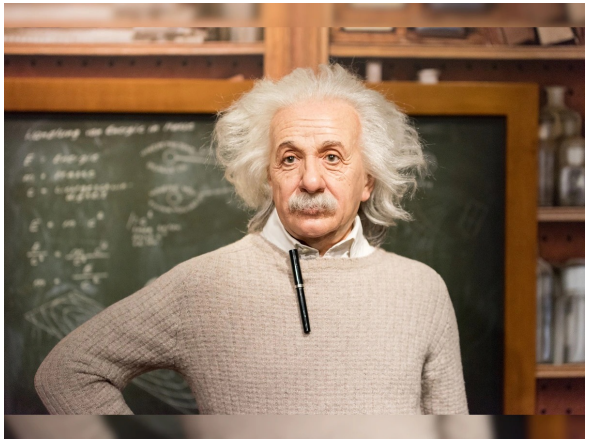
Are we looking at either Richard Feynman or Albert Einstein?

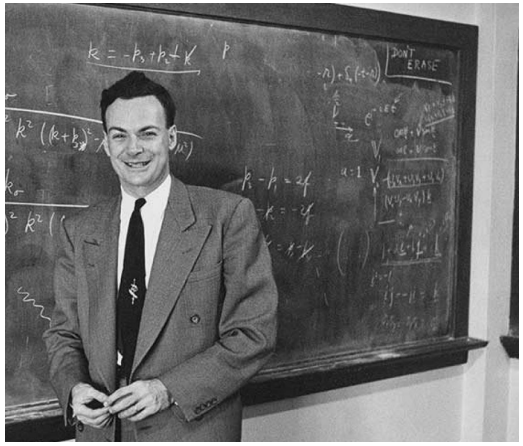
Top-down vs. Bottom-up decision

?

Scientist → *Richard_Feynman* | *Albert_Einstein*

Bottom-up This is what you see on the input before you make your rule decision:





Is this the scientist
Richard Feynman or Albert Einstein?

Next class

Syntax Analysis (top-down)

Read EaC: Chapter 3.3

Syntax Analysis (bottom-up)

Read EaC: Chapter 3.4