Parser 1

Ronghui Gu

Spring 2019

Columbia University

* Course website: https://www.cs.columbia.edu/ rgu/courses/4115/spring2019

** These slides are borrowed from Prof. Edwards.

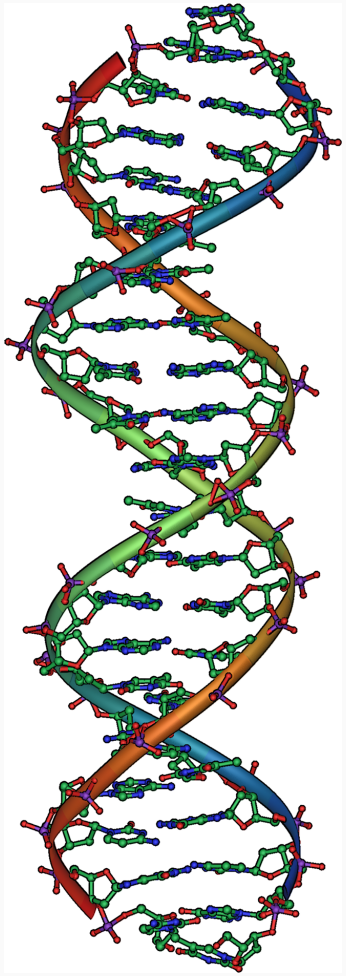
The Big Picture

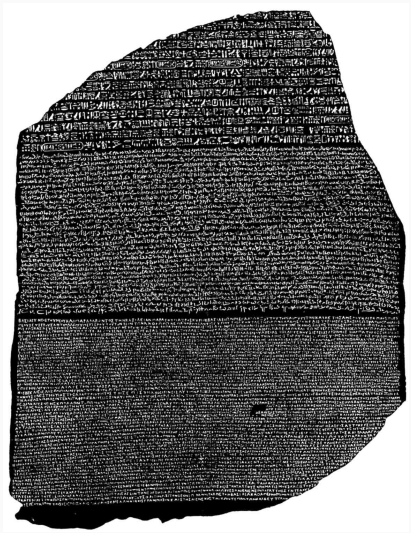
The First Question

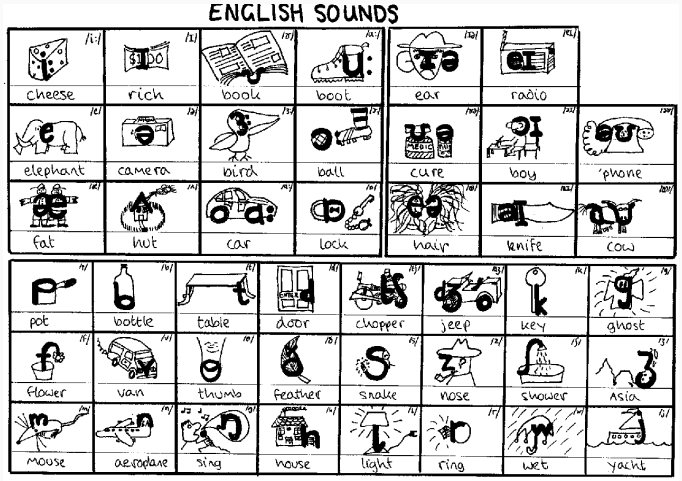
How do we describe/construct a program?

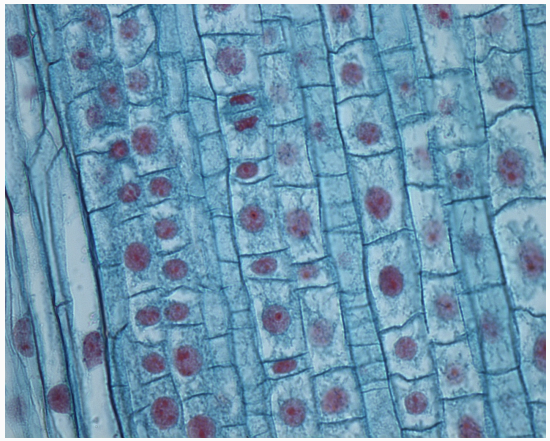
Solution: Use a Discrete Combinatorial System

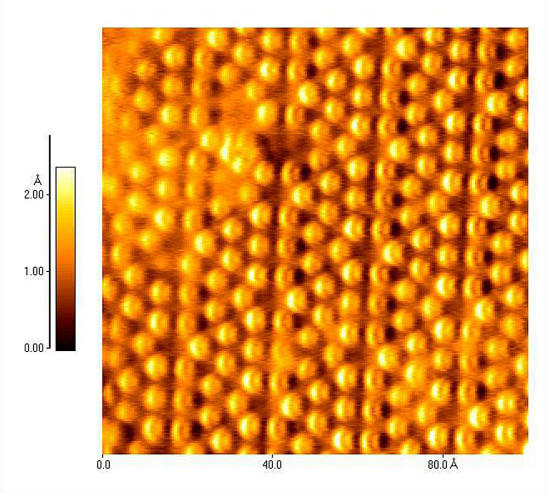
Use *combinations* of a *small number of things* to represent (exponentially) many different things.







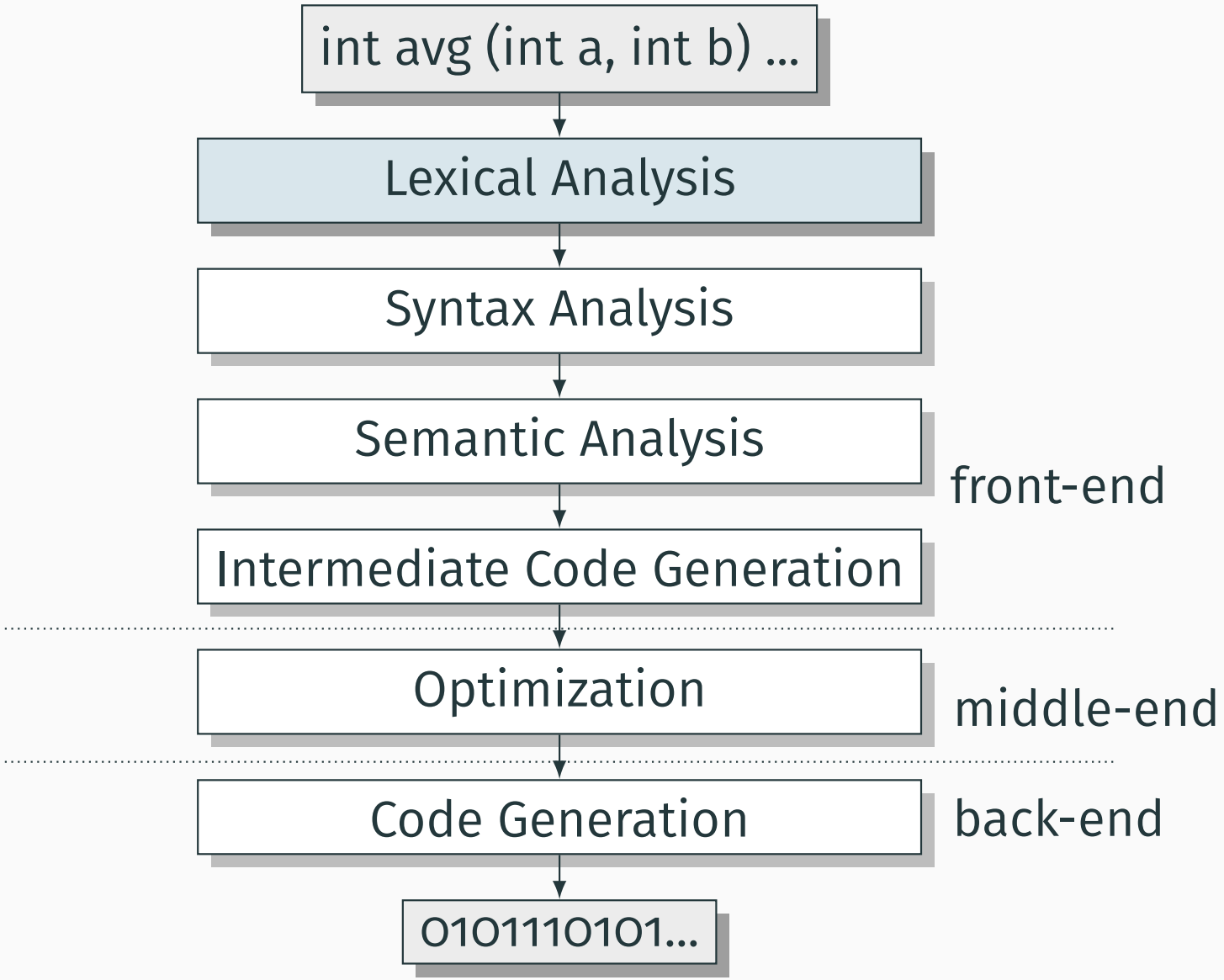




The Second Question

How do we combine characters into words?

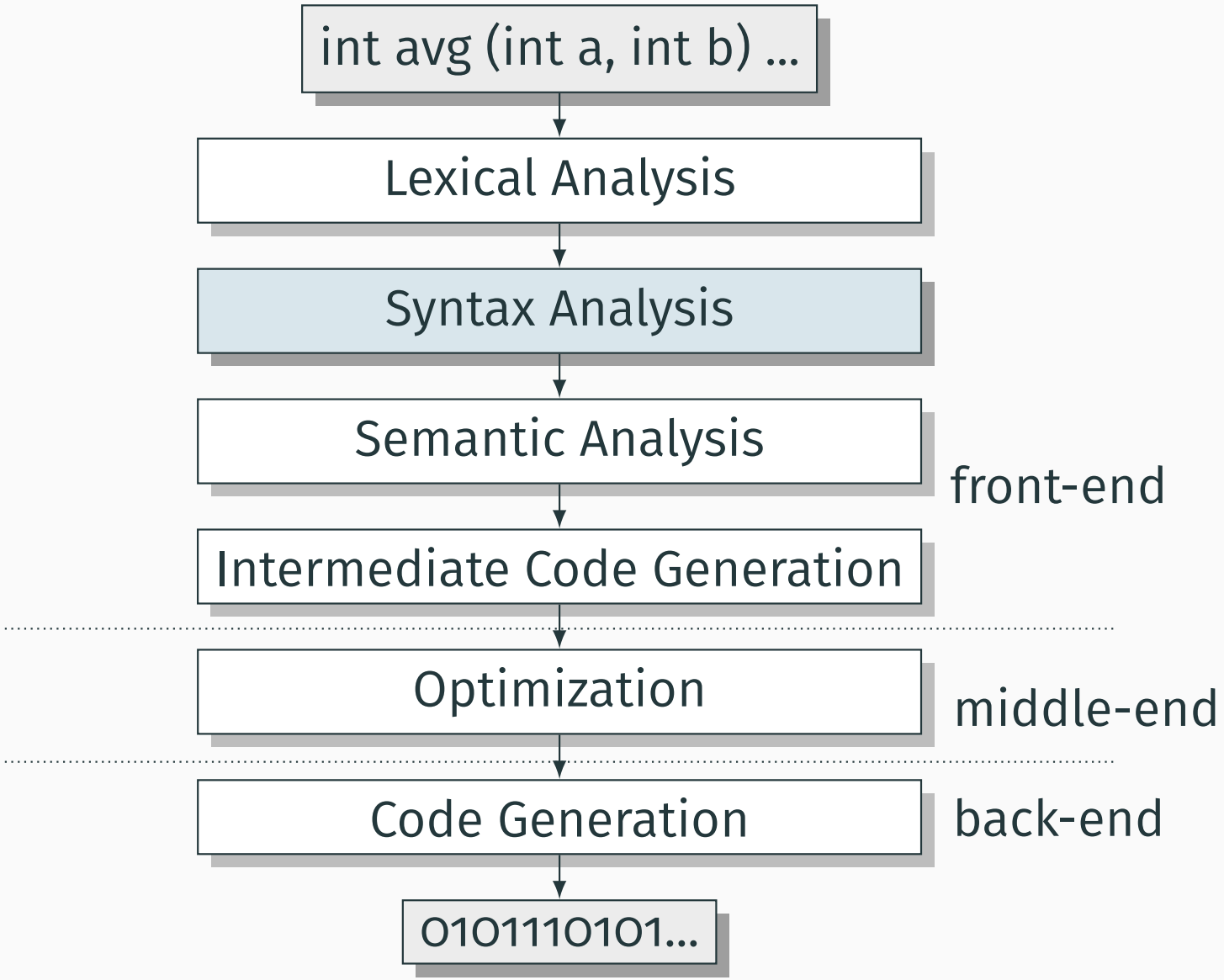
Scanner


int avg (int a, int b) ...
Lexical Analysis
Syntax Analysis
Semantic Analysis
front-end
Intermediate Code Generation
Optimization
middle-end
Code Generation
back-end
01011110101...

The Third Question

How do we combine words into sentences?

Parser


int avg (int a, int b) ...
Lexical Analysis
Syntax Analysis
Semantic Analysis
front-end
Intermediate Code Generation
Optimization
middle-end
Code Generation
back-end
01011110101...

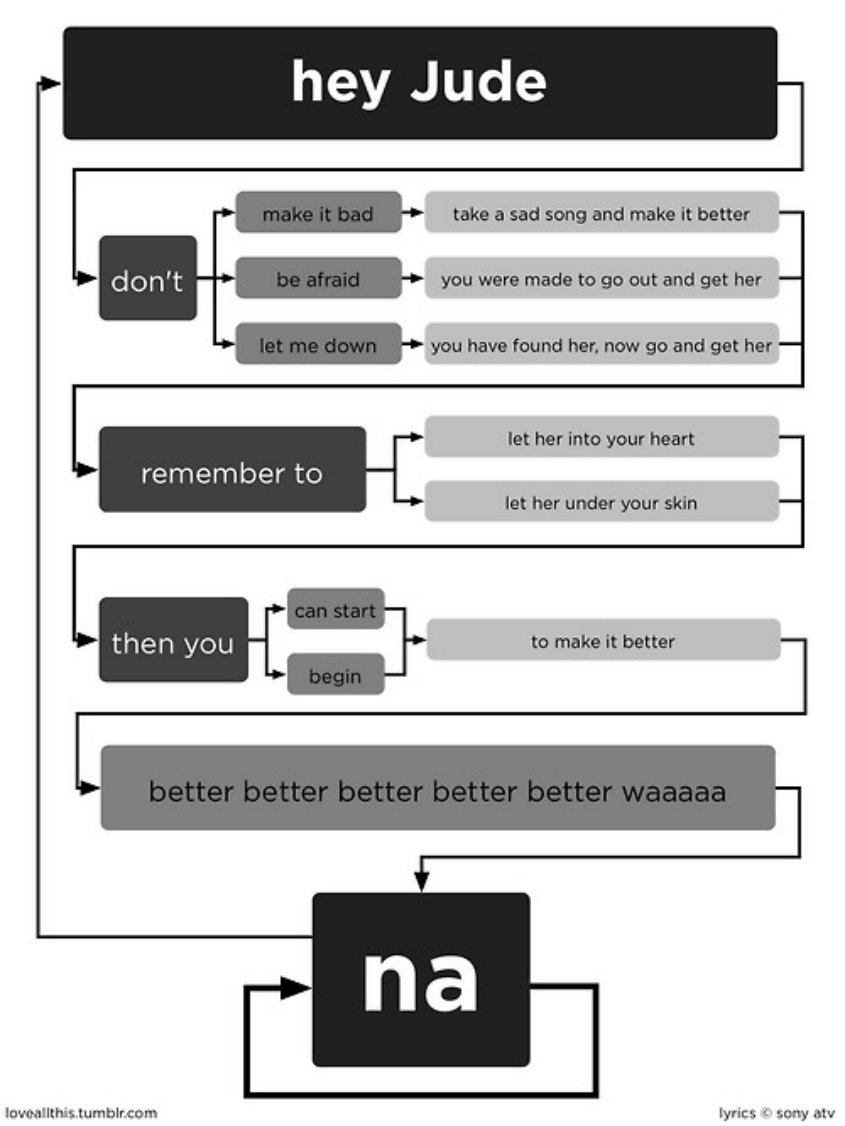
Choices: CS Research Jargon Generator

Pick one from each column

an integrated	mobile	network
a parallel	functional	preprocessor
a virtual	programmable	compiler
an interactive	distributed	system
a responsive	logical	interface
a synchronized	digital	protocol
a balanced	concurrent	architecture
a virtual	knowledge-based	database
a meta-level	multimedia	algorithm

E.g., “a responsive knowledge-based preprocessor.”

http://www.cs.purdue.edu/homes/dec/essay.topic.generator.html



http://loveallthis.tumblr.com/post/506873221

How about more structured collections of things?

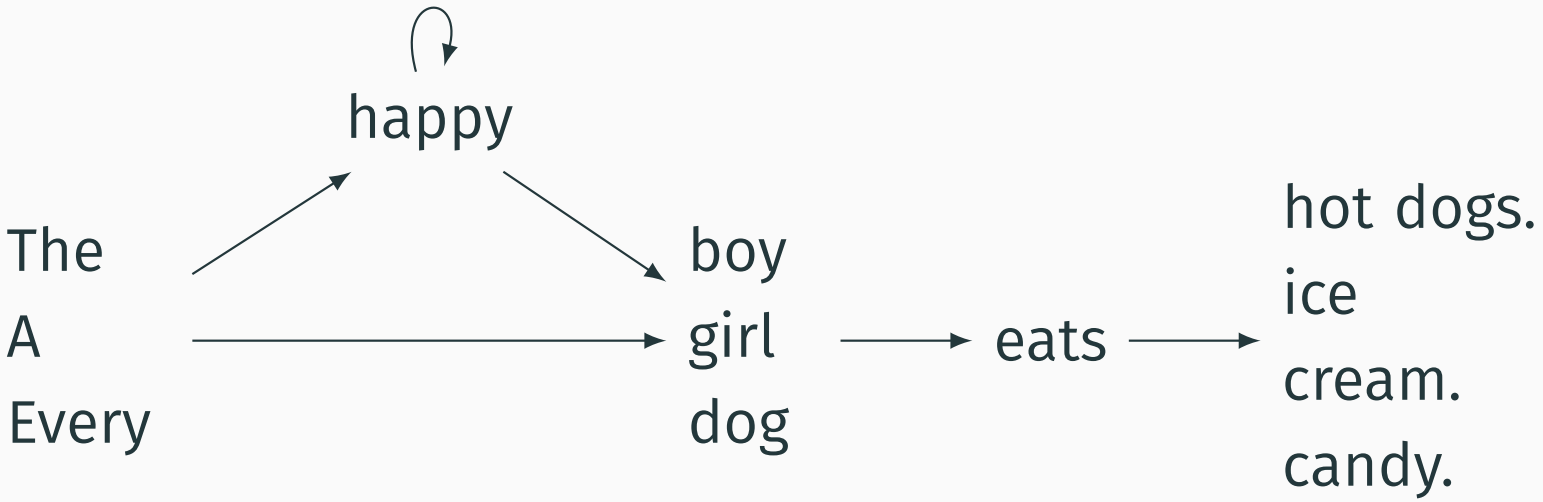
The boy eats hot dogs.

The dog eats ice cream.

Every happy girl eats candy.

A dog eats candy.

The happy happy dog eats hot dogs.

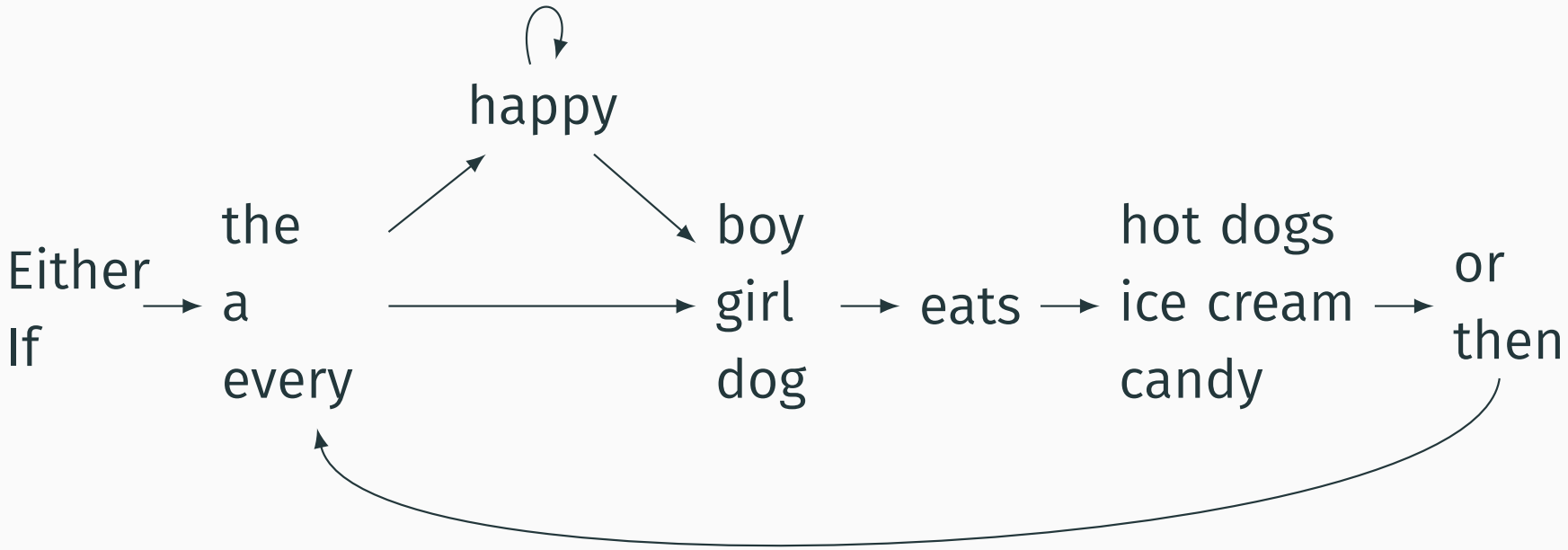


Pinker, *The Language Instinct*

Richer Sentences Are Harder

If the boy eats hot dogs, then the girl eats ice cream.

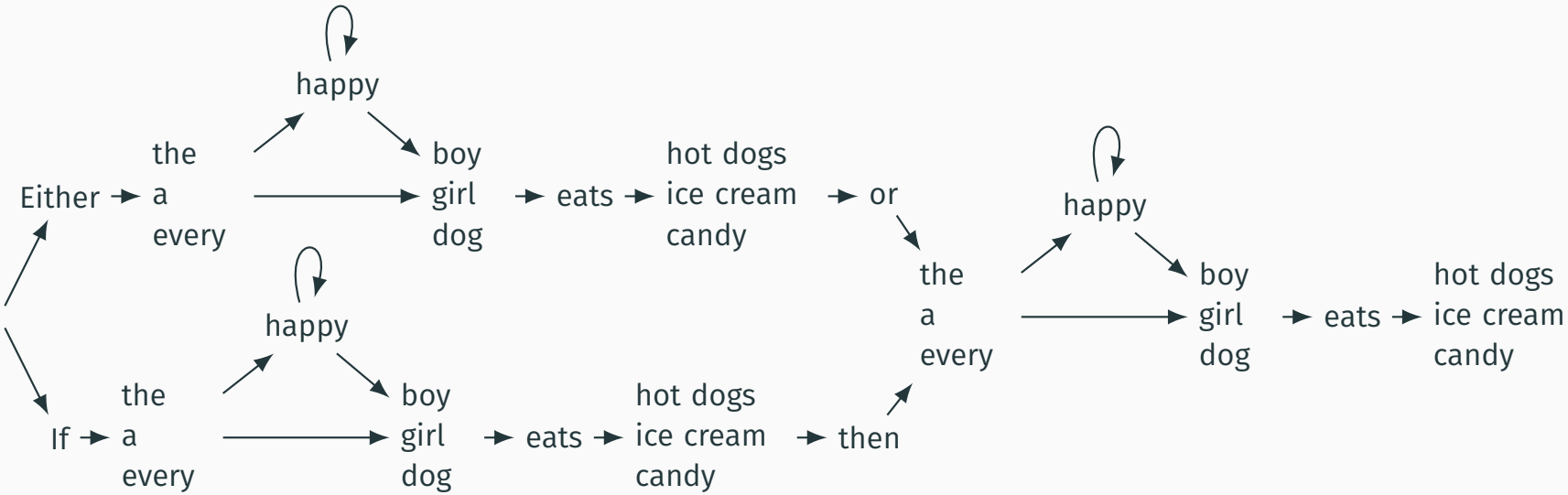
Either the boy eats candy, or every dog eats candy.



Does this work?

Automata Have Poor Memories

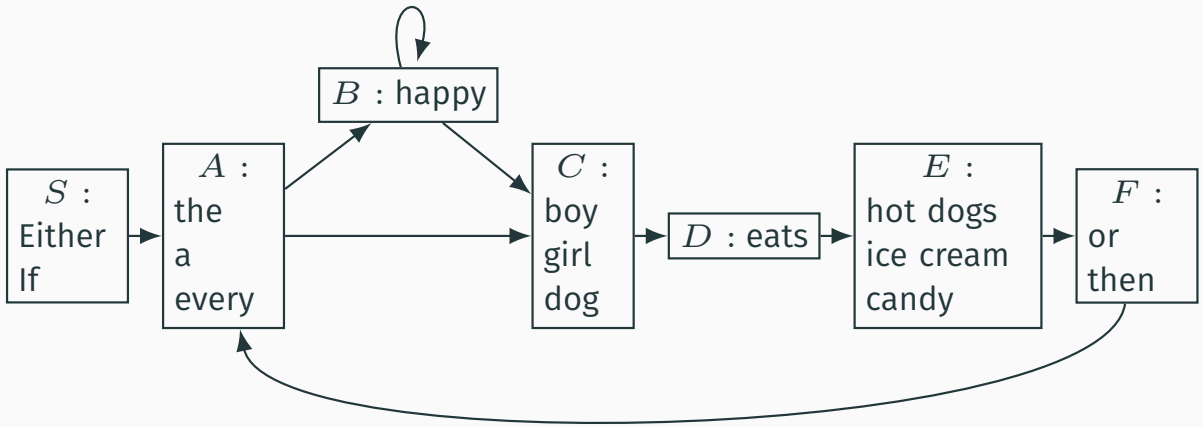
Want to “remember” whether it is an “either-or” or “if-then” sentence. Only solution: duplicate states.



Automata in the form of Production Rules

Problem: automata do not remember where they've been

$S \to \text{Either } A$
$S \to \text{If } A$
$A \to \text{the } B$
$A \to \text{the } C$
$A \to \text{a } B$
$A \to \text{a } C$
$A \to \text{every } B$
$A \to \text{every } C$
$B \to \text{happy } B$
$B \to \text{happy } C$
$C \to \text{boy } D$
$C \to \text{girl } D$
$C \to \text{dog } D$
$D \to \text{eats } E$
$E \to \text{hot dogs } F$
$E \to \text{ice cream } F$
$E \to \text{candy } F$



Solution: Context-Free Grammars

Context-Free Grammars have the ability to "call subroutines:"

$S \to$ Either P, or P.	Exactly two Ps
$S \to$ If P, then P.	
$P \to A\ H\ N$ eats O	One each of A, H, N, and O
$A \to$ the	
$A \to$ a	
$A \to$ every	
$H \to$ happy H	H is "happy" zero or more times
$H \to \epsilon$	
$N \to$ boy	
$N \to$ girl	
$N \to$ dog	
$O \to$ hot dogs	
$O \to$ ice cream	
$O \to$ candy	

An Example

n 0's followed by n 1's, e.g., 000111, 01

$S \to 0\ S\ 1.$

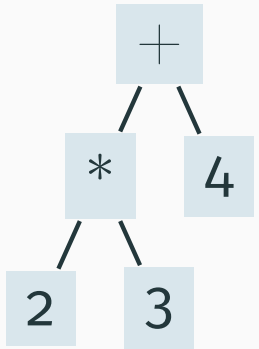
$S \to \epsilon.$

Constructing Grammars and Ocamlyacc

Parsing

Objective: build an abstract syntax tree (AST) for the token sequence from the scanner.

$$2 * 3 + 4 \quad \Rightarrow$$



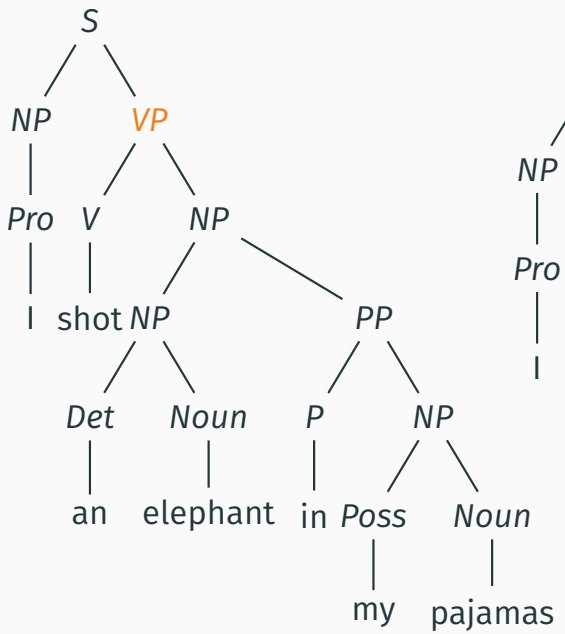
Goal: verify the syntax of the program, discard irrelevant information, and “understand” the structure of the program.

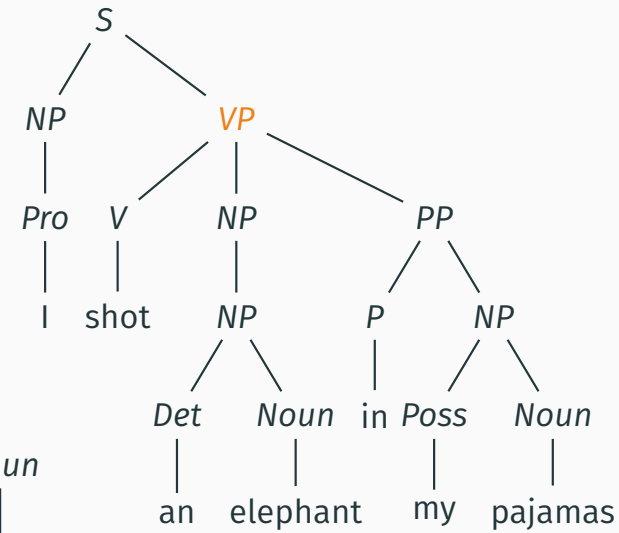
Parentheses and most other forms of punctuation removed.

Ambiguity in English

I shot an elephant in my pajamas

S	→	*NP VP*
VP	→	*V NP*
VP	→	*V NP PP*
NP	→	*NP PP*
NP	→	*Pro*
NP	→	*Det Noun*
NP	→	*Poss Noun*
PP	→	*P NP*
V	→	shot
Noun	→	elephant
Noun	→	pajamas
Pro	→	I
Det	→	an
P	→	in
Poss	→	my





Jurafsky and Martin, *Speech and Language Processing*

The Dangling Else Problem

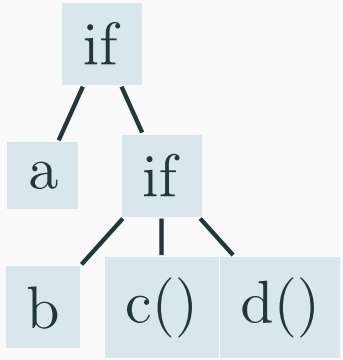
Who owns the *else*?

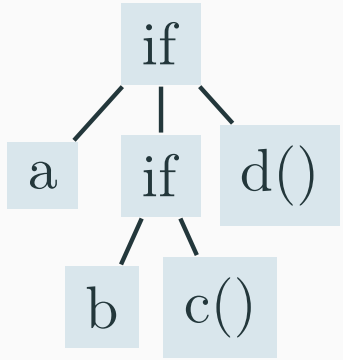
$$\text{if (a) if (b) c(); else d();}$$

```
stmt : IF expr THEN stmt
     | IF expr THEN stmt ELSE stmt
```

Problem comes after matching the first statement. Question is whether an “else” should be part of the current statement or a surrounding one since the second line tells us “stmt ELSE” is possible.

The Dangling Else Problem

Should this be 

or 

?

Grammars are usually ambiguous; manuals give disambiguating rules such as C's:

> *As usual the "else" is resolved by connecting an else with the last encountered elseless if.*

The Dangling Else Problem

Idea: break into two types of statements: those that have a dangling “then” (“dstmt”) and those that do not (“cstmt”). A statement may be either, but the statement just before an “else” must not have a dangling clause because if it did, the “else” would belong to it.

```
stmt : dstmt
     | cstmt

dstmt : IF expr THEN stmt
      | IF expr THEN cstmt ELSE dstmt

cstmt : IF expr THEN cstmt ELSE cstmt
      | other statements ...
```

if (a) if (b) c(); else d();

Another Solution to the Dangling Else Problem

We are effectively carrying an extra bit of information during parsing: whether there is an open “then” clause.
Unfortunately, duplicating rules is the only way to do this in a context-free grammar.

Another Solution to the Dangling Else Problem

Some languages resolve this problem by insisting on nesting everything.

E.g., Algol 68:

```
if a < b then a else b fi;
```

“fi” is “if” spelled backwards. The language also uses do–od and case–esac.

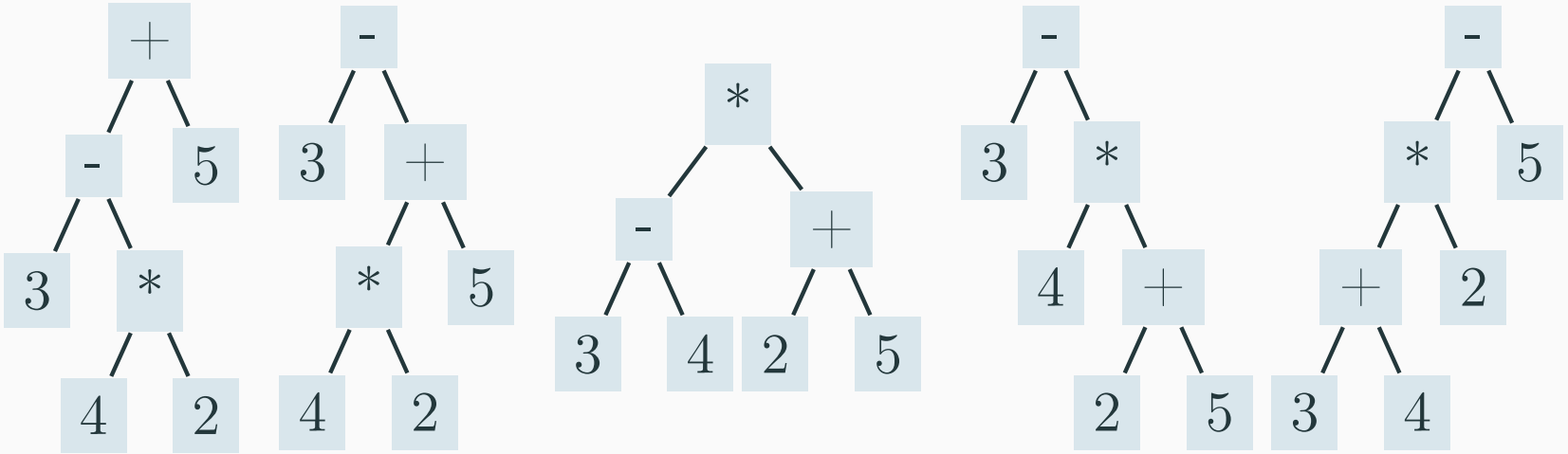
Ambiguous Arithmetic

Ambiguity can be a problem in expressions. Consider parsing

$$3 - 4 * 2 + 5$$

with the grammar

$$e \to e + e \mid e - e \mid e * e \mid e / e \mid N$$



Operator Precedence and Associativity

Usually resolve ambiguity in arithmetic expressions

Like you were taught in elementary school:

“My Dear Aunt Sally”

Mnemonic for multiplication and division before addition and subtraction.

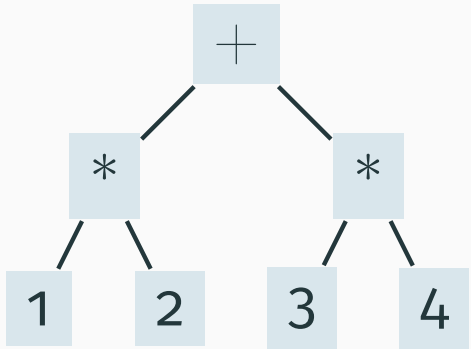
Operator Precedence

Defines how “sticky” an operator is.

$$1 * 2 + 3 * 4$$

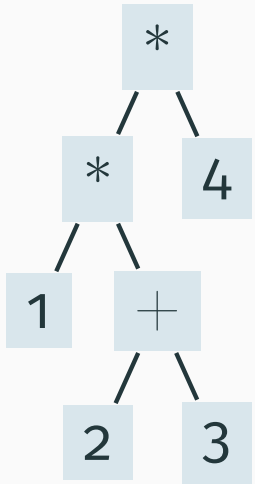
* at higher precedence than $+$:

$(1 * 2) + (3 * 4)$



$+$ at higher precedence than *:

$1 * (2 + 3) * 4$

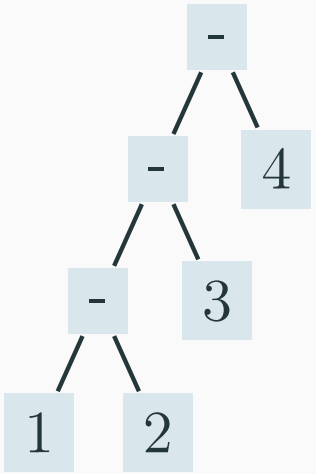


Associativity

Whether to evaluate left-to-right or right-to-left

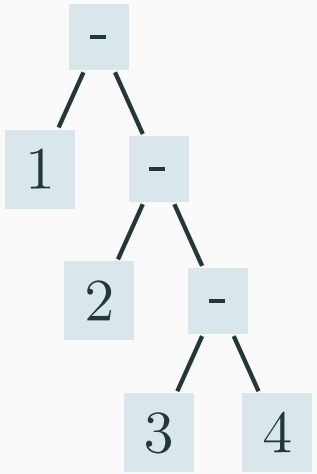
Most operators are left-associative

$$1 - 2 - 3 - 4$$



$$((1 - 2) - 3) - 4$$

left associative



$$1 - (2 - (3 - 4))$$

right associative

Fixing Ambiguous Grammars

A grammar specification:

```
expr :
      expr PLUS expr
    | expr MINUS expr
    | expr TIMES expr
    | expr DIVIDE expr
    | NUMBER
```

Ambiguous: no precedence or associativity.

Ocamlyacc's complaint: "16 shift/reduce conflicts."

$$1 * 2 + 3?$$

Assigning Precedence Levels

Split into multiple rules, one per level

```
expr : expr PLUS expr
     | expr MINUS expr
     | term

term : term TIMES term
     | term DIVIDE term
     | atom

atom : NUMBER
```

Still ambiguous: associativity not defined

Ocamlyacc's complaint: "8 shift/reduce conflicts."

$$1 * 2 + 3$$

$$1 * 2 * 3?$$

Assigning Associativity

Make one side the next level of precedence

```
expr : expr PLUS term
     | expr MINUS term
     | term

term : term TIMES atom
     | term DIVIDE atom
     | atom

atom : NUMBER
```

This is left-associative.

No shift/reduce conflicts.

$$1 * 2 * 3$$

Ocamlyacc Specifications

```
%{
  (* Header: verbatim OCaml; optional *)
%}

  /* Declarations: tokens, precedence, etc. */

%%

  /* Rules: context-free rules */

%%

  (* Trailer: verbatim OCaml; optional *)
```

Declarations

- %token *symbol* ...
  Define symbol names (exported to .mli file)
- %token < *type* > *symbol* ...
  Define symbols with attached attribute (also exported)
- %start *symbol* ...
  Define start symbols (entry points)
- %type < *type* > *symbol* ...
  Define the type for a symbol (mandatory for start)
- %left *symbol* ...
- %right *symbol* ...
- %nonassoc *symbol* ...
  Define predecence and associtivity for the given symbols, listed in order from lowest to highest precedence

Rules

```
nonterminal :
    symbol ... symbol { semantic-action }
  | ...
  | symbol ... symbol { semantic-action }
```

- *nonterminal* is the name of a rule, e.g., “program,” “expr”
- *symbol* is either a terminal (token) or another rule
- *semantic-action* is OCaml code evaluated when the rule is matched
- In a *semantic-action*, $1, $2, ... returns the value of the first, second, ... symbol matched
- A rule may include “%prec *symbol*” to override its default precedence

An Example .mly File

```
%token <int> INT
%token PLUS MINUS TIMES DIV LPAREN RPAREN EOL

%left PLUS MINUS /* lowest precedence */
%left TIMES DIV
%nonassoc UMINUS /* highest precedence */

%start main       /* the entry point */
%type <int> main

main:
    expr EOL                    { $1 }

expr:
    INT                         { $1 }
  | LPAREN expr RPAREN          { $2 }
  | expr PLUS expr              { $1 + $3 }
  | expr MINUS expr             { $1 - $3 }
  | expr TIMES expr             { $1 * $3 }
  | expr DIV expr               { $1 / $3 }
  | MINUS expr %prec UMINUS     { - $2 }
```