CSC488/2107 Winter 2019 — Compilers & Interpreters

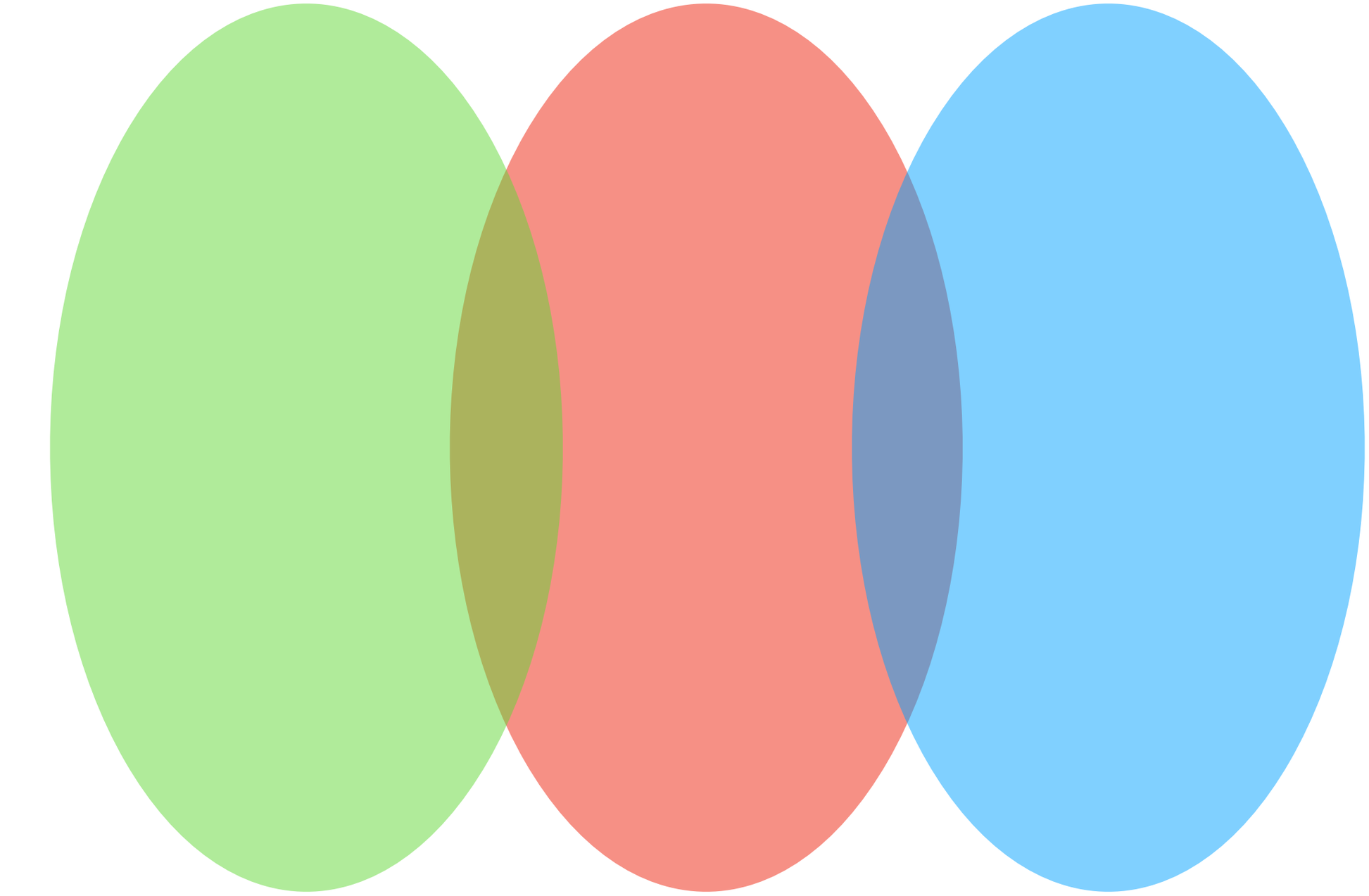
https://www.cs.toronto.edu/~csc488h/

Peter McCormick
pdm@cs.toronto.edu

Agenda

- Recognize, Analyze, Transform
- Lexical analysis
- Building lexical analyzers

Recognize Analyze Transform



Frontend

Backend

Syntax Analysis

- The *syntax* of a language defines the rules by which a sequence of tokens can be recognized as a legal construction in that language.

- Recognize and distinguish legal and illegal *sentences* of the language

Addition expressions

Addition expressions tokens as regular expressions

Digits = ‘0’ … ‘9’

Plus = ‘+’

Literal = Digits^{+}

Token = Plus | Literal

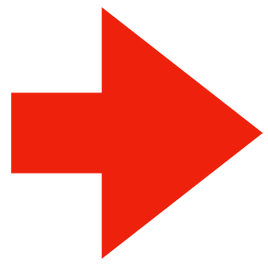
Addition expressions *as* regular expressions

Digits = (‘0’ … ‘9’)+

Expr = (Digits ‘+’)* Digits

Digits = (‘0’ … ‘9’)+

Expr = (Digits ‘+’)* Digits



Expr = ((‘0’ … ‘9’)+ ‘+’)* (‘0’ … ‘9’)+

Expressions with parentheses

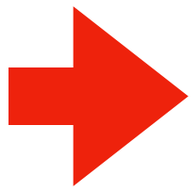
Digits = (‘0’ … ‘9’)+
Add = Expr ‘+’ Expr
Expr = Digits | ‘(’ Add ‘)’

```
488
(400+88)
(400+(44+44))
((400+4)+(42+42))
```

Digits = (‘0’ … ‘9’)+

Add = Expr ‘+’ Expr

Expr = Digits | ‘(‘ Add ‘)’



Expr = Digits | (Expr ‘+’ Expr)

Expr = Digits | (
(Digits | (Expr ‘+’ Expr)) ’+’
(Digits | (Expr ‘+’ Expr)))

Abbreviations in regular expressions are just syntactic sugar…

They add no additional expressive power

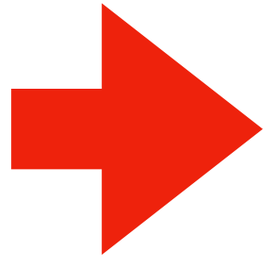
Recursion adds expressive power

Recursive abbreviations?

- Simplifies regular expressions
    - Alternation within expressions is no longer required
    - Alternation pipe | is no longer required
    - Kleene closure is no longer required

Recursive abbreviations

prod = α β (γ | δ) ε

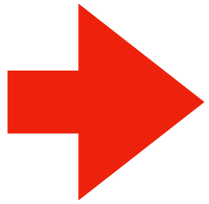


helper = γ | δ

prod = α β helper ε

Recursive abbreviations

helper = γ | δ
prod = α β helper ε



helper = γ
helper = δ
prod = α β helper ε

Recursive abbreviations

prod = (α β γ)*

prod = (α β γ) prod

prod = λ

This simple but powerful notation of recursive abbreviations is referred to as *context-free grammars* (CFG)

Context-free grammars

- CFGs can describe richer languages than regular expressions
    - Thus need something more powerful than finite automata to recognize them (key is having *memory*)
- A *language* is a set of strings over an alphabet Σ
- Alphabet Σ ranges over *tokens*, not characters
    - Example: Σ = { IF, IDENT, PLUS, LBRACE, EQ, … }
- A CFG consists of a set of *productions*

Context-free grammars

A *production* is of the form:

symbol ⟶ symbol symbol … symbol

- Right hand side has 0 or more symbols (0 means λ)
- A *symbol* is either:
  - *Terminal*: a token from alphabet Σ
  - *Non-terminal*: appears on the left hand side of a production
- Left hand side symbol is always a non-terminal
- Distinguished *start* production (typically the first)

S ⟶ S ‘;’ S

S ⟶ id ‘=‘ E

S ⟶ ‘print’ E

```
id = num ;
id = num ;
print id + ( id + num )
```

E ⟶ id

E ⟶ num

E ⟶ E ‘+’ E

E ⟶ ‘(‘ E ‘)’

```
x = 400;
y = 42;
print x + (y + 46)
```

Derivations

- Derive a sentence from the grammar to show that it is in the language

- Begin with the start symbol, and repeatedly replace right hand side non-terminals with symbols from productions

- Many possible derivation orders

  - *Left-most derivation*: always expand the left-most non-terminal first, working towards the right

  - *Right-most derivation*

```
id = num ;
id = num ;
print id + ( id + num )
```

S ⟶ S ‘;’ S

S ⟶ id ‘=‘ E

S ⟶ ‘print’ E

E ⟶ id

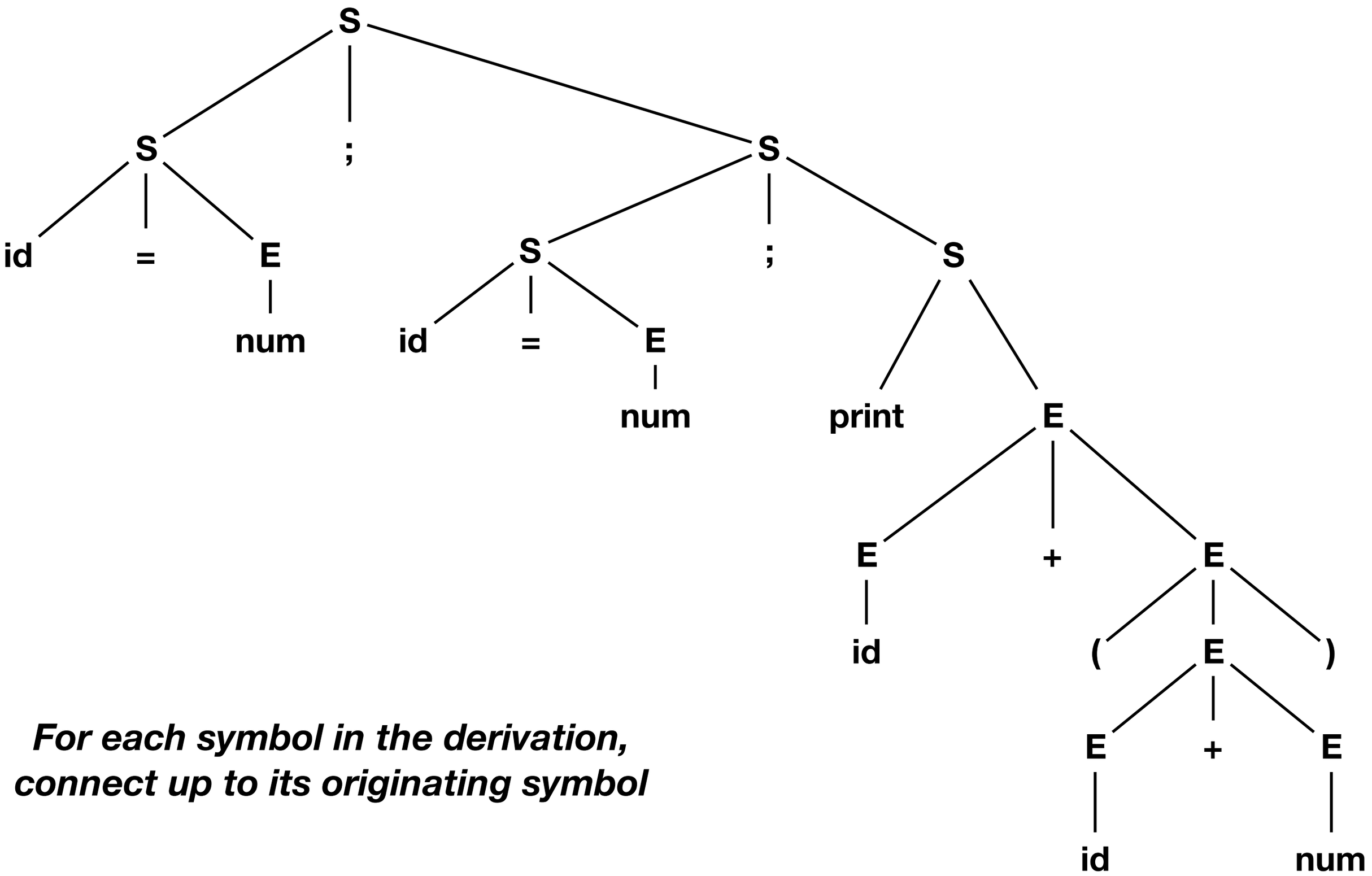
E ⟶ num

E ⟶ E ‘+’ E

E ⟶ ‘(‘ E ‘)’

S
S ; S
S ; S ; S
S ; id = E ; S
S ; id = num ; S
id = E ; id = num ; S
id = num ; id = num ; S
id = num ; id = num ; print E
id = num ; id = num ; print E + E
id = num ; id = num ; print E + (E)
id = num ; id = num ; print E + (E + E)
id = num ; id = num ; print id + (E + E)
id = num ; id = num ; print id + (id + E)
id = num ; id = num ; print id + (id + num)

Parse trees

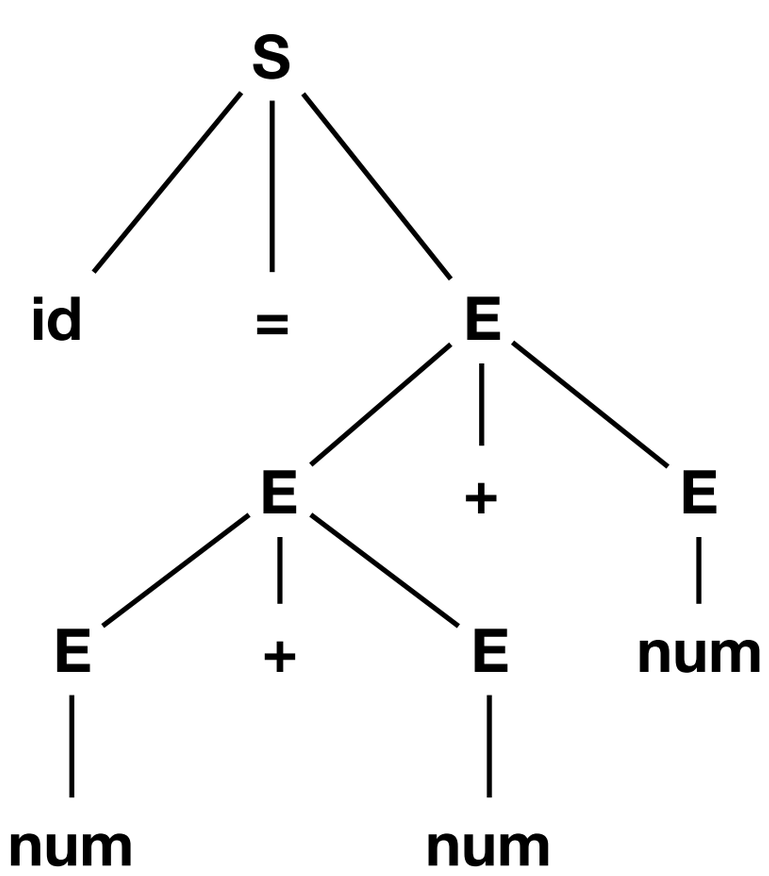


For each symbol in the derivation, connect up to its originating symbol

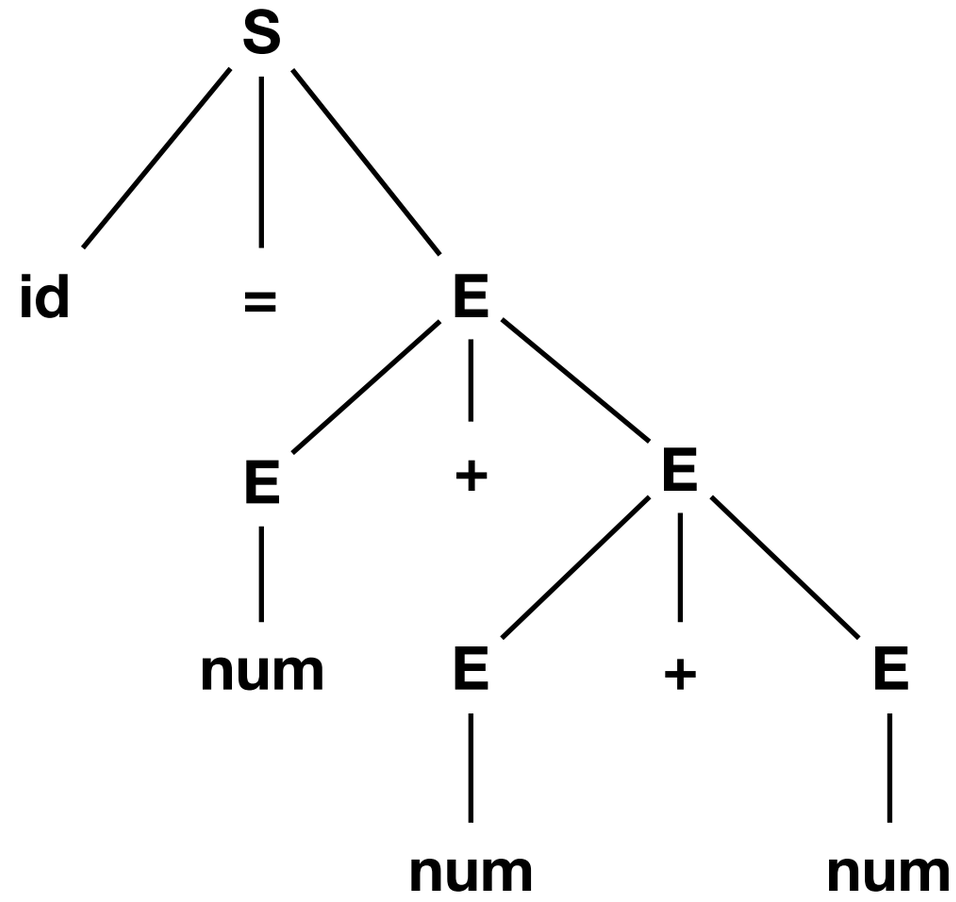
Ambiguous grammars

Two possible parse trees for the same sentence

```
x = 1 + 2 + 3
```



(1 + 2) + 3



1 + (2 + 3)

Removing ambiguity

We want **left-associativity**

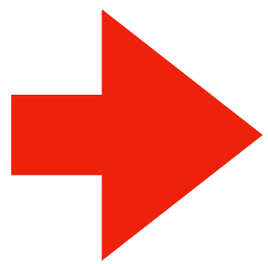
$$S \longrightarrow E$$

$$E \longrightarrow id$$

$$E \longrightarrow num$$

$$E \longrightarrow E + E$$

$$E \longrightarrow (E)$$



$$S \longrightarrow E$$

$$E \longrightarrow T$$

$$E \longrightarrow E + T$$

$$T \longrightarrow id$$

$$T \longrightarrow num$$

$$T \longrightarrow (E)$$

T for **terms** ***because they are added together***

Removing ambiguity

$S \longrightarrow E$

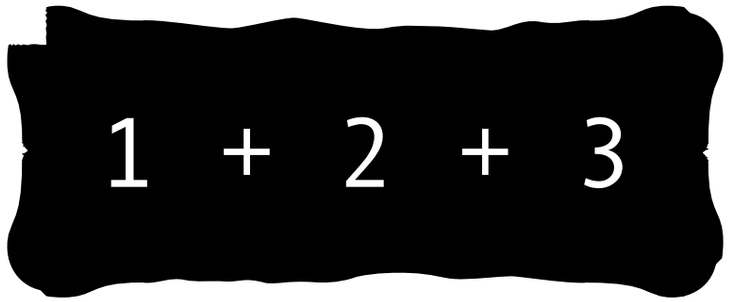
$E \longrightarrow T$

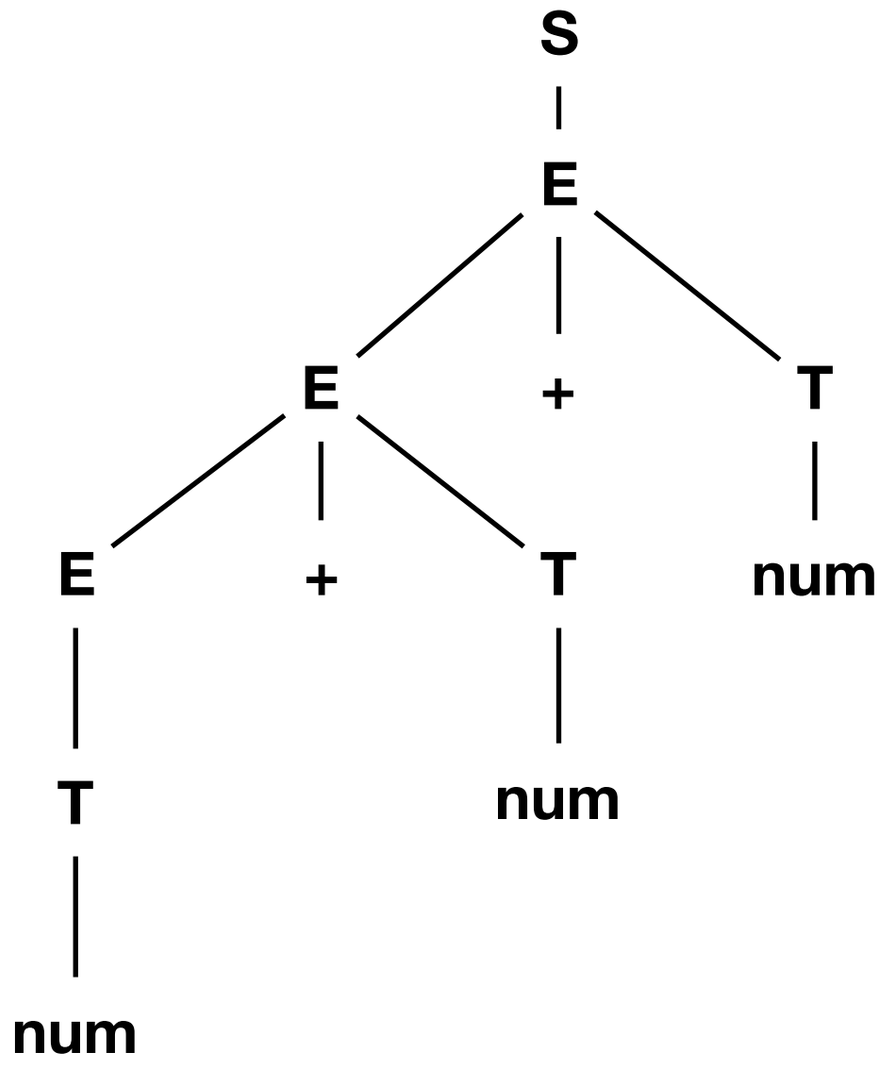
$E \longrightarrow E + T$

$T \longrightarrow id$

$T \longrightarrow num$

$T \longrightarrow (E)$





More ambiguity

***We want multiplication & division to be* higher-precedence*, or to* bind more tightly**

$S \longrightarrow E$

$E \longrightarrow id$

$E \longrightarrow num$

$E \longrightarrow E * E$

$E \longrightarrow E / E$

$E \longrightarrow E + E$

$E \longrightarrow E - E$

$E \longrightarrow (E)$

$S \longrightarrow E$

$E \longrightarrow T$

$E \longrightarrow E + T$

$E \longrightarrow E - T$

$T \longrightarrow T * F$

$T \longrightarrow T / F$

$T \longrightarrow F$

$F \longrightarrow id$

$F \longrightarrow num$

$F \longrightarrow (E)$

F for* factors *because they are multiplied together

Impossible to derive

S ⟶ E

E ⟶ T

E ⟶ E + T

E ⟶ E - T

T ⟶ T * F

T ⟶ T / F

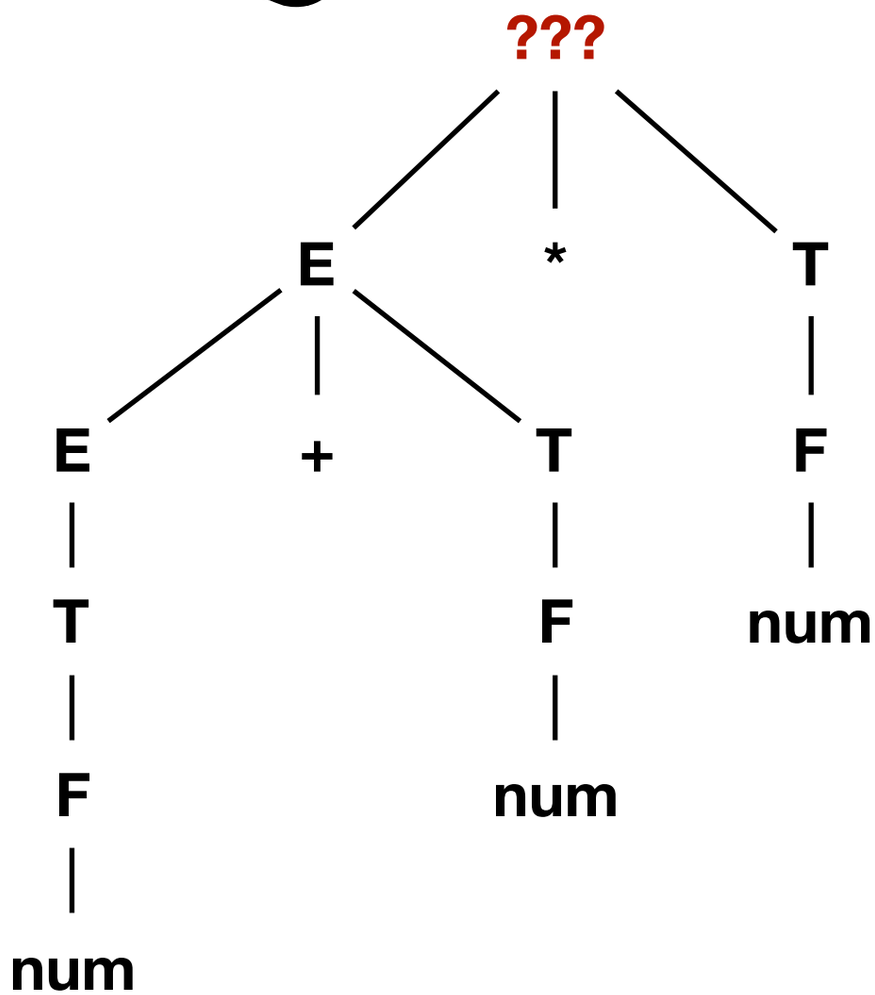
T ⟶ F

F ⟶ id

F ⟶ num

F ⟶ (E)

```
(1 + 2) * 3
```



Backus-Naur Form (BNF)

Backus-Naur Form (BNF) — ALGOL 60 Report

```
<expression> ::= <term>
               | <expression> “+” <term>
               | <expression> “-” <term>

<term>       ::= <factor>
               | <term> “*” <factor>
               | <term> “/” <factor>

<factor>     ::= <identifier>
               | <number>
               | “(” <expression> “)”
```

PLY *(Python Lex-Yacc)*

```
expression : term
           | expression PLUS term
           | expression MINUS term

term       : factor
           | term STAR factor
           | term SLASH factor

factor     : identifier
           | number
           | LPAREN expression RPAREN
```

Parsing Top Down

vs

Bottom Up

Top Down Parsing

- Starting from the start symbol *S*, find the correct sequence of production expansions that will transform *S* into the input token stream (if possible)
  - Called a *derivation*
- Build the parse tree from the root (*S*) to the leaves (terminals)
- Top down techniques: LL(*k*), Recursive Descent

Bottom Up Parsing

- Given a stream of input tokens, find the correct sequence of production contractions that take the input back to the start symbol
  - Called a *reduction* or *reverse derivation*
- Build the parse tree from the leaves (terminals) to the root (S)
- Bottom up techniques: LR(*k*), LALR(*k*), SLR(*k*)

$LL(1)$ Grammars

Peter McCormick

January 24, 2018

Context-free Grammars

- Define a *context-free grammar* $G = (N, \Sigma, S, P)$ where
  - Σ is the set of *terminals* (the *alphabet*), each represented as lower case Roman letters (t, x, y, z)
  - N is the set of *non-terminals*, each represented by capitalized Roman letters (A, B, X, Y)
  - $S \in N$ is the distinguished *start* symbol
  - P is a finite set of *productions*
  - $V = \Sigma \cup N$ is the vocabulary of the grammar. Strings ranging over V, such as $A\,w\,X\,t$, are each represented by Greek letters (α, β, γ)
- $X \Rightarrow^{+} \alpha$ means that there is a valid sequence of derivations starting from the non-terminal X to the string α
- Let \$ be a special *end-of-input* marker

Productions

A production is a *rewriting rule* of the form:

$$X \quad \rightarrow \quad \alpha_1 \alpha_2 \cdots \alpha_m$$

where $X \in N$, $\alpha_i \in V^*$ for each i, and $m \geq 0$. If $m = 0$, then

$$X \quad \rightarrow$$

That is, X can be replaced by λ, the empty string.

- *Rewriting* is the act of replacing a non-terminal X with the right hand of a production for X, so $\alpha_1 \cdots \alpha_m$ replacing X
- $X \Rightarrow^+ \alpha$ means that there is a valid sequence of derivations starting from the non-terminal X to the vocabulary string α

Definitions

A vocabulary string is in *sentential form* if it is in the set of all strings that can be derived from the start symbol:

$$\{w : S \Rightarrow^* w\}$$

A *language* is the set of all terminal strings that can be derived from the start symbol:

$$\{w : S \Rightarrow^* w\} \cap \Sigma^*$$

Definitions

Given a production $X \to \gamma$ and a sentenial form $\alpha X \beta$, then

$$\begin{array}{lll} \alpha X \beta & \Rightarrow & \alpha \gamma \beta \text{ is a } \textit{derivation} \text{ in one step} \\ \alpha X \beta & \Rightarrow^* & \alpha \gamma \beta \text{ is a derivation in zero or more steps} \\ \alpha X \beta & \Rightarrow^+ & \alpha \gamma \beta \text{ is a derivation in one or more steps} \end{array}$$

- A *left-most derivation* expands non-terminals left to right, while a *right-most derivation* expands right to left

$LL(1)$ Grammars

- LL is a set of all languages that can be parser by an LL parser
- An LL parser consumes its input **L**eft-to-right, producing a **L**eftmost-derivation
- $LL(k)$ means LL with k tokens worth of input look ahead
- $LL(1)$ means 1 input token lookahead

Nullable

$\alpha \in V^+$ is *nullable* iff $\alpha \Rightarrow^* \lambda$

Nullable

Given the productions:

$$\begin{aligned} X &\rightarrow Y\,Z\,W \\ Y &\rightarrow \lambda \\ Z &\rightarrow z \mid Y \\ W &\rightarrow w \mid Y\,Z \end{aligned}$$

Then X is nullable since:

$$\begin{aligned} X &\Rightarrow \underline{Y}\,Z\,W \\ &\Rightarrow \underline{Z}\,W \text{ (since } Y \text{ is nullable)} \\ &\Rightarrow \underline{Y}\,W \\ &\Rightarrow \underline{W} \text{ (again since } Y \text{ is nullable)} \\ &\Rightarrow \underline{Y}\,Z \\ &\Rightarrow \underline{Z} \text{ (again)} \\ &\Rightarrow \underline{Y} \Rightarrow \lambda \end{aligned}$$

First Sets

$First(z\,\beta)$	$=$	$\{z\}$	z is a terminal symbol
$First(\lambda)$	$=$	$\{\}$	(the empty set)
$First(B\gamma)$	$=$	$First(B)$	(if B is *not* nullable)
	$=$	$First(B) \cup First(\gamma)$	(if B *is* nullable)

Follow Sets

$$Follow(X) = \left\{ t \in \Sigma : S \Rightarrow^{+} \alpha X \boldsymbol{t} \beta \right\}$$

Follow Set Construction Rules

Starting from initially empty follow sets, iteratively apply these rules until the sets no longer change.

#1.	If S is the goal symbol	Add $\{\$\}$ to $\mathit{Follow}(S)$
#2.	If $S \Rightarrow^+ \alpha X$	Add $\{\$\}$ to $\mathit{Follow}(X)$
#3.	If $S \Rightarrow^+ \alpha X \mathbf{t} \beta$	Add $\{t\}$ to $\mathit{Follow}(X)$
#4.	If $S \Rightarrow^+ \alpha X \mathbf{Y} \beta$	If Y is *not* nullable: · Add $\mathit{First}(Y)$ to $\mathit{Follow}(X)$ Else if Y *is* nullable: · Add $\mathit{First}(Y) \cup \mathit{First}(\beta)$ to $\mathit{Follow}(X)$
#5.	If $X \rightarrow \alpha \mathbf{Y}$	Add $\mathit{Follow}(X)$ to $\mathit{Follow}(Y)$

Predict Sets for $LL(1)$ Grammars

Given a non-terminal X defined with several alternate productions:

$$\begin{array}{lll} X & \rightarrow & \gamma_1 \\ & \rightarrow & \gamma_2 \\ & & \ldots \\ & \rightarrow & \gamma_m \end{array}$$

The predict set for each production $X \rightarrow \gamma_i$ is defined as

$$\begin{array}{lll} \mathit{Predict}(X \rightarrow \gamma_i) & = & \mathit{First}(\gamma_i) \qquad\qquad (\text{if } \gamma_i \text{ is } \textit{not} \text{ nullable}) \\ & = & \mathit{First}(\gamma_i) \cup \mathit{Follow}(X) \quad (\text{if } \gamma_i \textit{ is} \text{ nullable}) \end{array}$$

Predict Sets for $LL(1)$ Grammars

- To be an $LL(1)$ grammar, the Predict sets for all productions for a given non-terminal X must be mutually disjoint

$$\begin{array}{rcl} X & \rightarrow & Y\,\alpha \\ & \rightarrow & Y\,\beta \end{array}$$

is not $LL(1)$ because the Predict sets for the productions of X are not mutually disjoint:

$$\begin{array}{rcl} \mathit{Predict}(X \rightarrow Y\,\alpha) & = & \mathit{First}(Y\,\alpha) \text{ (if } Y \text{ is } \textit{not} \text{ nullable)} \\ & = & \mathit{First}(Y) \\ & = & \mathit{First}(Y\,\beta) \\ & = & \mathit{Predict}(X \rightarrow Y\,\beta) \\ \mathit{Predict}(X \rightarrow Y\,\alpha) & = & \mathit{First}(Y\,\alpha) \cup \mathit{Follow}(X) \text{ (if } Y \text{ is nullable)} \\ \mathit{Predict}(X \rightarrow Y\,\beta) & = & \mathit{First}(Y\,\beta) \cup \mathit{Follow}(X) \text{ (if } Y \text{ is nullable)} \end{array}$$

Factoring Out Common Prefix

Transform:

$$\begin{aligned} X &\rightarrow Y\,\alpha \\ &\rightarrow Y\,\beta \end{aligned}$$

into:

$$\begin{aligned} X &\rightarrow Y\,X_{tail} \\ X_{tail} &\rightarrow \alpha \\ &\rightarrow \beta \end{aligned}$$

Left Recursion

$LL(1)$ grammars cannot handle left recursion:

$$\begin{aligned} X &\rightarrow X\,\alpha \\ &\rightarrow \beta \end{aligned}$$

If β is *not* nullable:

$$\begin{aligned} \mathit{Predict}(X \rightarrow X\,\alpha) &= \mathit{First}(X\,\alpha) \\ &= \mathit{First}(X) \\ &= \beta \\ \mathit{Predict}(X \rightarrow \beta) &= \mathit{First}(\beta) \\ &= \beta \end{aligned}$$

Similarly argument for existence of an intersection if β is nullable.

Factoring Out Left Recursion

Transform:

$$\begin{aligned} X &\rightarrow X\,\alpha \\ &\rightarrow \beta \end{aligned}$$

into:

$$\begin{aligned} X &\rightarrow \beta\,X_{tail} \\ X_{tail} &\rightarrow \alpha\,X_{tail} \\ &\rightarrow \lambda \end{aligned}$$

Next Week

- More on practical parser construction
- Bottom up parsing
- LR(1) grammars