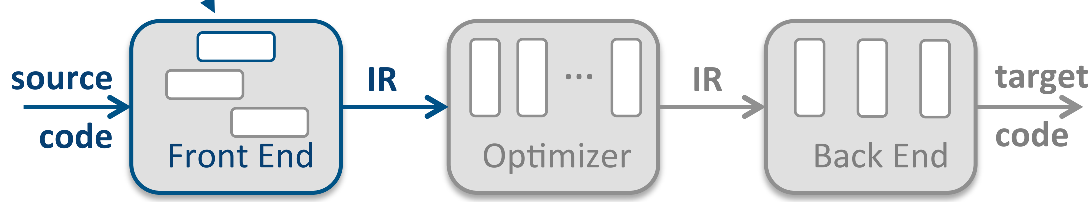
COMP 412
FALL 2017

# Introduction to Parsing

*Context-free grammars*

## Comp 412

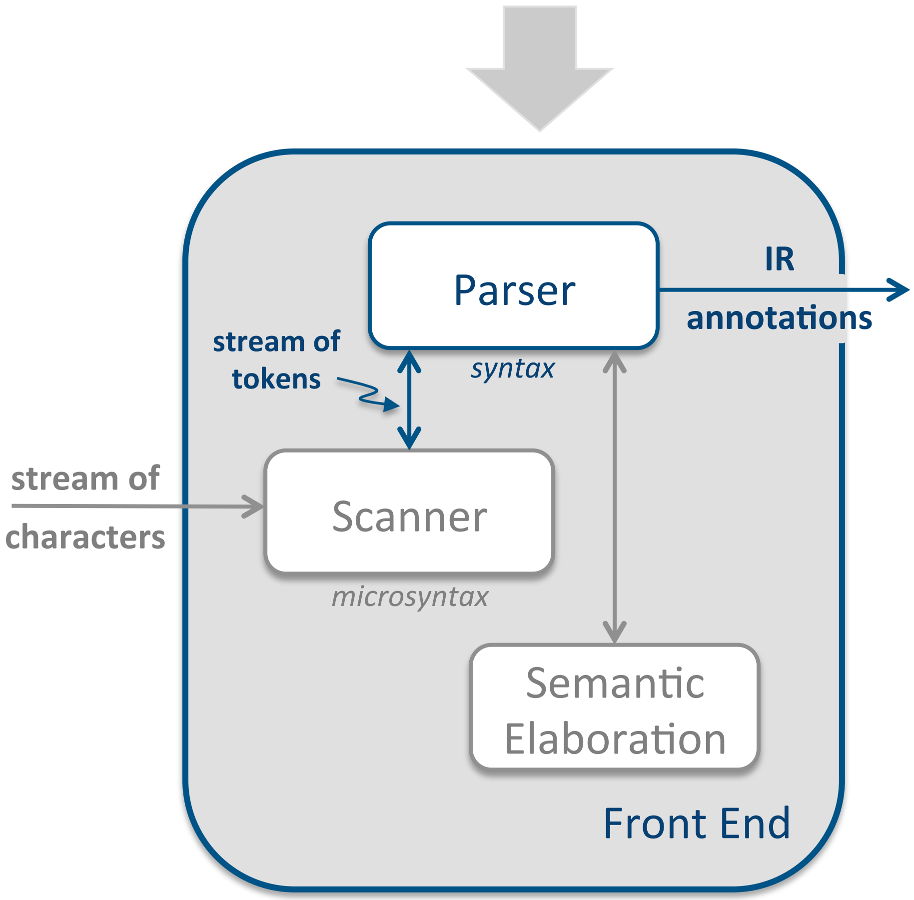
Finally, the box moved …






**Chapter 3 in EaC2e**

# The Front End


**Scanner** looks at every character

- Converts stream of chars to stream of classified words:
  - <word, part of speech>
  - Sometimes call this pair a "token"
- Efficiency & scalability matter

**Parser** looks at every token

- Determines if the stream of tokens forms a sentence in the source language
- Fits tokens to some syntactic model, or grammar, for the source language

# The Study of Parsing

**Parsing is the process of discovering a *derivation* for some sentence**

- Need mathematical model of syntax — a grammar *G*
- Need an algorithm to test membership in $L(G)$
- Need to remember that our goal is to build parsers, not to study the interesting if arcane mathematics of arbitrary languages

**Roadmap for our study of parsing**

1. Context-free grammars & derivations
2. Top-down parsing
   - Top-down parsers explore the possibilities of syntax in a systematic way
   - A file of code has a limited number of words that can occur at its start
3. Bottom-up parsing
   - Bottom-up parsers build on the detailed structure of the input stream
   - Each classified word can affect the interpretation of past & future words

# Specifying Syntax: Context-Free Grammars

**Context-free syntax is specified with a *context-free grammar* (CFG)**

This **CFG** defines the set of noises that sheep normally make

| | | | |
|---|---|---|---|
| 0 | ***SheepNoise*** | → | ***SheepNoise*** **baa** |
| 1 | | \| | **baa** |

See the digression about **BNF** on p. 87 of EaC2e

It is written in a variant of Backus-Naur form (**BNF**)

Formally, a **CFG** is a four tuple, $G = (S, N, T, P)$

- *S* is the *start symbol* of the grammar
  - *L(G) is the set of sentences that can be derived from* S
- *N* is a set of *nonterminal symbols* or syntactic variables — *SheepNoise*
- *T* is a set of *terminal symbols* or words — baa
- *P* is a set of *productions* or *rewrite rules* — $P: N \rightarrow (N \cup T)^+$

**We will defer the definition of "context free" for a few slides.**

# Deriving Sentences with a CFG

**We can use the *SheepNoise* grammar to derive sentences**

– *use the productions as rewrite rules*

| *Rule* | *Sentential Form* |
|---|---|
| — | *SheepNoise* |
| 1 | <u>baa</u> |

| *Rule* | *Sentential Form* |
|---|---|
| — | *SheepNoise* |
| 0 | *SheepNoise* <u>baa</u> |
| 1 | <u>baa</u> <u>baa</u> |

| *Rule* | *Sentential Form* |
|---|---|
| — | *SheepNoise* |
| 0 | *SheepNoise* <u>baa</u> |
| 0 | *SheepNoise* <u>baa</u> <u>baa</u> |
| 1 | <u>baa</u> <u>baa</u> <u>baa</u> |

***And, so on …***

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

A ***sentential form*** is a string of terminal & nonterminal symbols that is a valid step in some derivation.

# Context-Free Grammars

## What makes a grammar "context free" ?

Productions in the *SheepNoise* grammar have a specific form:

| | | | |
|---|---|---|---|
| 0 | ***SheepNoise*** | → | ***SheepNoise*** **baa** |
| 1 | | \| | **baa** |

Each production has *a single nonterminal symbol on its left hand side*, which makes it impossible to encode either left or right context.

⇒ The grammar is *context free*

A context-sensitive grammar can have ≥ 1 symbol on its lhs.

- **CSG**'s have not found widespread application in compilers

lhs ≅ *left-hand side*
rhs ≅ *right hand side*

Notice that $L$(*SheepNoise*) is actually a regular language: baa baa*

**RLs ⊂ CFLs**

# Digression: The Chomsky Hierarchy

## Noam Chomsky proposed a hierarchy of languages

PL microsyntax

Type 3 grammars are regular grammars (equivalent to **RE**s)

- Single ***NT*** on *lhs*; *rhs* has one ***T*** & (optionally) one ***NT***
- Corresponds to a **DFA**

PL syntax

Type 2 grammars are context-free grammars

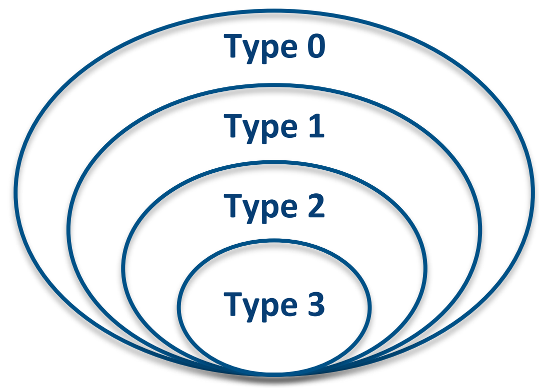
- Single ***NT*** on *lhs*; *rhs* has string of grammar symbols
- Corresponds to a **push-down automaton**

Type 1 grammars are *context-sensitive grammars*

- Productions of form $\alpha A \beta \rightarrow \gamma$, where $\alpha$, $\beta$, and $\gamma$ are strings in $(\boldsymbol{T} \cup \boldsymbol{NT})$
- Corresponds to a **linear bounded automaton**

Type 0 grammars are unrestricted grammars

- Includes all formal grammars
- Corresponds to a **Turing machine**




**The Chomsky Hierarchy of Grammars**

**Relating COMP 412 to your friends who major in linguistics.**

# Limits of Regular Languages

**RL**'s $\subset$ **CFL**'s

## Does it matter that RL's $\subset$ CFL's ?

You cannot construct **DFA**'s to recognize these languages

- $L = \{ p^k q^k \}$ *(parentheses, brackets)*
- $L = \{ wcw^r \mid w \in \Sigma^* \}$

Neither of these is a regular language *(nor an RE)*

Constructs like these are important to programming languages

But, this is a little subtle. You <u>can</u> construct **DFA**'s for

- Strings with alternating 0's and 1's

  $( \varepsilon \mid 1 ) ( 01 )^* ( \varepsilon \mid 0 )$

- Strings with and even number of 0's and 1's

**RE**'s can count bounded sets and bounded differences

# Terminology for Derivations

## The point of parsing is to discover a derivation

A derivation consists of a series of rewrite steps

$$S \Rightarrow \gamma_0 \Rightarrow \gamma_1 \Rightarrow \gamma_2 \Rightarrow \ldots \Rightarrow \gamma_{n-1} \Rightarrow \gamma_n \Rightarrow \textit{sentence}$$

- Each $\gamma_i$ is a sentential form
  - If $\gamma$ contains only terminal symbols, $\gamma$ is a **sentence** in *L(G)*
  - If $\gamma$ contains 1 or more non-terminals, $\gamma$ is a **sentential form**
- To get $\gamma_i$ from $\gamma_{i-1}$, expand some **NT** $A \in \gamma_{i-1}$ by using $A \rightarrow \beta$
  - Replace the occurrence of $A \in \gamma_{i-1}$ with $\beta$ to get $\gamma_i$
  - Replacing the leftmost **NT** at each step, creates a **leftmost** derivation
  - Replacing the rightmost **NT** at each step, creates a **rightmost** derivation

A **left-sentential form** occurs in a *leftmost* derivation

A **right-sentential form** occurs in a rightmost derivation

**NT** ≅ *nonterminal symbol*

*S* is the start symbol for the grammar *G*

# Terminology for Derivations

## The point of parsing is to discover a derivation

Top down ↓ / Bottom up ↑

| Rule | Sentential Form |
|---|---|
| — | *SheepNoise* |
| 0 | *SheepNoise* baa |
| 0 | *SheepNoise* baa baa |
| 1 | baa baa baa |

***Three-word SheepNoise***

- A top-down parse begins with the grammar's start symbol and works toward the sentence
- A bottom-up parse starts with the words in the sentence and works towards the start symbol

## In the general case[1], discovering a derivation looks expensive

- Many alternatives & combinations, possible backtracking
- Derivation must be guided by the actual words in the sentence
- Fortunately, programming languages tend to have simple syntax
- Understanding parsing will help you see why PLs look as they do!

[1] e.g., Chomsky 0 or 1 grammars

# A Better Example

**Not a regular language**

***SheepNoise* is quite limited. Let's consider a more interesting example.**

| | | | |
|---|---|---|---|
| 0 | *Start* | → | *Brackets* |
| 1 | *Brackets* | → | **(** *Brackets* **)** |
| 2 | | \| | **[** *Brackets* **]** |
| 3 | | \| | **( )** |
| 4 | | \| | **[ ]** |

***Two flavors of nested brackets***

| ***Rule*** | ***Sentential Form*** |
|---|---|
| — | *Start* |
| 0 | *Brackets* |
| 1 | **(** *Brackets* **)** |
| 2 | **( [** *Brackets* **] )** |
| 3 | **( [ ( ) ] )** |

***Derivation of "( [ ( ) ] )"***

- A sequence of rewrites that produces a sentence is a *derivation*
- Process of discovering a derivation is called *parsing*

We denote this derivation: *Start* $\Rightarrow^*$ ( [ ( ) ] )

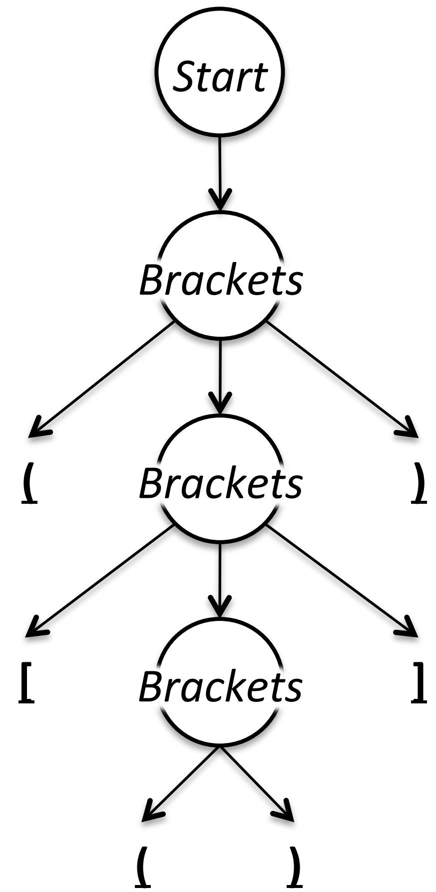
# Brackets

## A derivation corresponds to a *derivation tree* or *parse tree*

| *Rule* | *Sentential Form* |
|---|---|
| — | *Start* |
| 0 | *Brackets* |
| 1 | ( *Brackets* ) |
| 2 | ( [ *Brackets* ] ) |
| 3 | ( [ ( ) ] ) |

***Start*** $\Rightarrow^*$ **( [ ( ) ] )**

The derivation gives us the grammatical structure of the input sentence, which was completely missing in **RE** / **DFA** recognizers.




***Parse tree for this derivation***

# A Simple Expression Grammar

## CFGs are used to define many programming language constructs

| | | | |
|---|---|---|---|
| 0 | *Expr* | → | *Expr Op Expr* |
| 1 | | \| | number |
| 2 | | \| | identifier |
| 3 | *Op* | → | plus |
| 4 | | \| | minus |
| 5 | | \| | times |
| 6 | | \| | divide |

**Expressions over +, -, *, /**
**numbers, & identifiers**

| Rule | Sentential Form |
|---|---|
| — | *Expr* |
| 0 | *Expr Op Expr* |
| 2 | <id,x> *Op Expr* |
| 4 | <id,x> — *Expr* |
| 0 | <id,x> — *Expr Op Expr* |
| 1 | <id,x> — <num,2> *Op Expr* |
| 5 | <id,x> — <num,2> * *Expr* |
| 2 | <id,x> — <num,2> * <id,y> |

***Derivation of x – 2 * y***

When a syntactic category has just one lexeme, as with plus and minus, we will often write it as just the lexeme.

*And, if you skipped class & are reading the slides, you should know that this grammar is a very bad way to define expressions*

# A Simple Expression Grammar

## Constructing a derivation

- At each step, we select an **NT** in the current string to replace
- Different choices can lead to different derivations

Two derivations are of interest

- ***Leftmost derivation*** — replace, at each step, the leftmost **NT**
- ***Rightmost derivation*** — replace, at each step, the rightmost **NT**

These are the two systematic derivations (*We don't care about random orders*)

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

- Of course, there is also a *rightmost* derivation
- In this example, the rightmost derivation is different

# Leftmost and Rightmost Derivations

| *Rule* | *Sentential Form* |
|---|---|
| — | *Expr* |
| 0 | *Expr Op Expr* |
| 2 | <id,x> *Op Expr* |
| 4 | <id,x> — *Expr* |
| 0 | <id,x> — *Expr Op Expr* |
| 1 | <id,x> — <num,2> *Op Expr* |
| 5 | <id,x> — <num,2> * *Expr* |
| 2 | <id,x> — <num,2> * <id,y> |

***Leftmost Derivation of x – 2 * y***

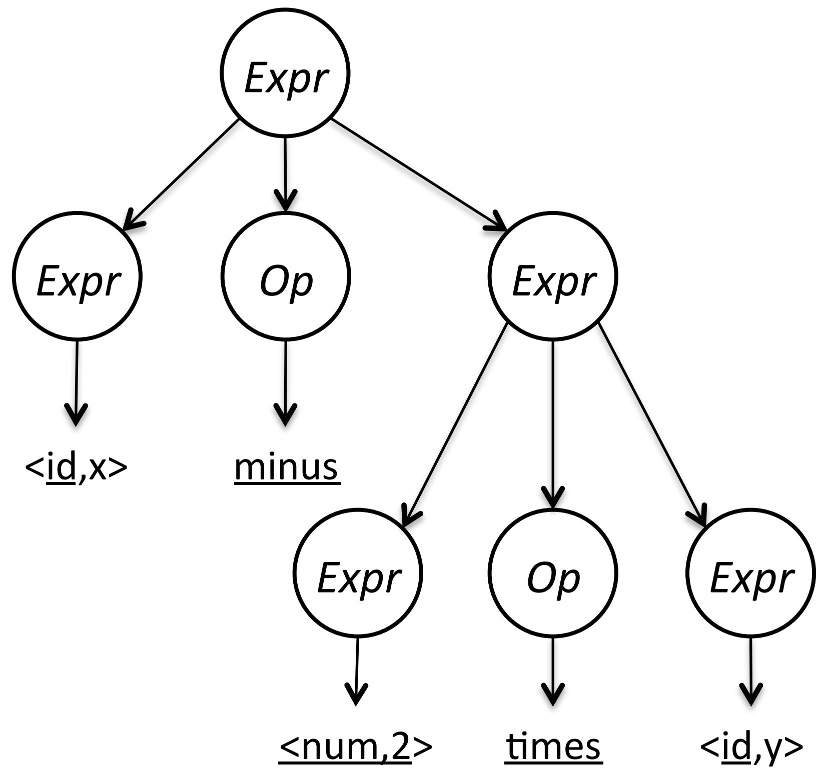
| *Rule* | *Sentential Form* |
|---|---|
| — | *Expr* |
| 0 | *Expr Op Expr* |
| 2 | *Expr Op* <id,y> |
| 5 | *Expr* * <id,y> |
| 0 | *Expr Op Expr* * <id,y> |
| 1 | *Expr Op* <num,2> * <id,y> |
| 4 | *Expr* — <num,2> * <id,y> |
| 2 | <id,x> — <num,2> * <id,y> |

***Rightmost Derivation of x – 2 * y***

- In both cases, *Expr* $\Rightarrow^*$ identifier — number + identifier
- The two derivations produce different parse trees & evaluation orders

# Leftmost Derivation

| ***Rule*** | ***Sentential Form*** |
|---|---|
| — | *Expr* |
| 0 | *Expr Op Expr* |
| 2 | <id,x> *Op Expr* |
| 4 | <id,x> — *Expr* |
| 0 | <id,x> — *Expr Op Expr* |
| 1 | <id,x> — <num,2> *Op Expr* |
| 5 | <id,x> — <num,2> * *Expr* |
| 2 | <id,x> — <num,2> * <id,y> |

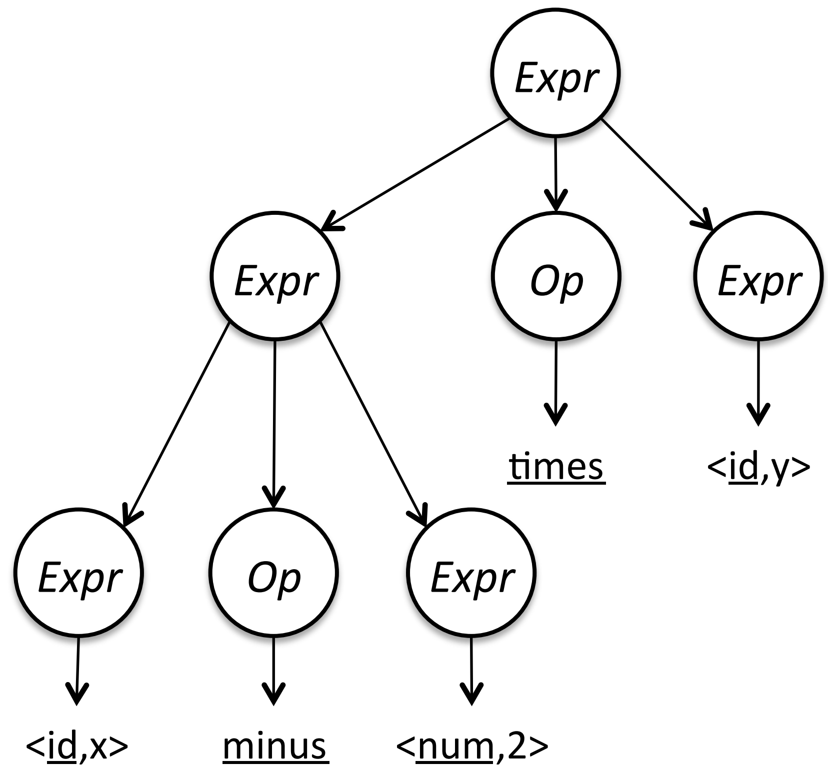



*In a postorder treewalk, this parse tree evaluates as* **x – (2 * y)**

***Parse tree for the leftmost derivation***

# Rightmost Derivation

| *Rule* | *Sentential Form* |
|---|---|
| — | *Expr* |
| 0 | *Expr Op Expr* |
| 2 | *Expr Op* <id,y> |
| 5 | *Expr* * <id,y> |
| 0 | *Expr Op Expr* * <id,y> |
| 1 | *Expr Op* <num,2> * <id,y> |
| 4 | *Expr* — <num,2> * <id,y> |
| 2 | <id,x> — <num,2> * <id,y> |




*In a postorder treewalk, this parse tree evaluates as* **(x – 2) * y**

***Parse tree for the rightmost derivation***

# Evaluation Order: Why Do We Care?

**The leftmost & rightmost derivations for x – 2 * y produced different evaluation orders.**

- These two orders may produce different results, even with integers
  - x – ( 2 * y ) is different than (x – 2) * y, for most values of x & y
  - In floating point, the problem can arise with a string of the same operator
- Standard algebra specifies both an evaluation order (*left to right*) and a precedence (*parentheses; multiply and divide; add and subtract*)

The compiler must pay attention to the intended order of evaluation

**Floating-point Numbers are not Real Numbers**

- Finite magnitude (e.g., $-2^{31}$ to $2^{31}-1$) introduces overflow & underflow
- Floating-point arithmetic causes unexpected losses of precision
  - There exist $x$, $y$, & $z$ such that $x + y > 0$, $(x + y) + z > z$, but $x + (y + z) = z$

# **Reminder**: Why Do We Care About Ambiguity?

## It is easy to get lost in language theory

The point of this course is language translation

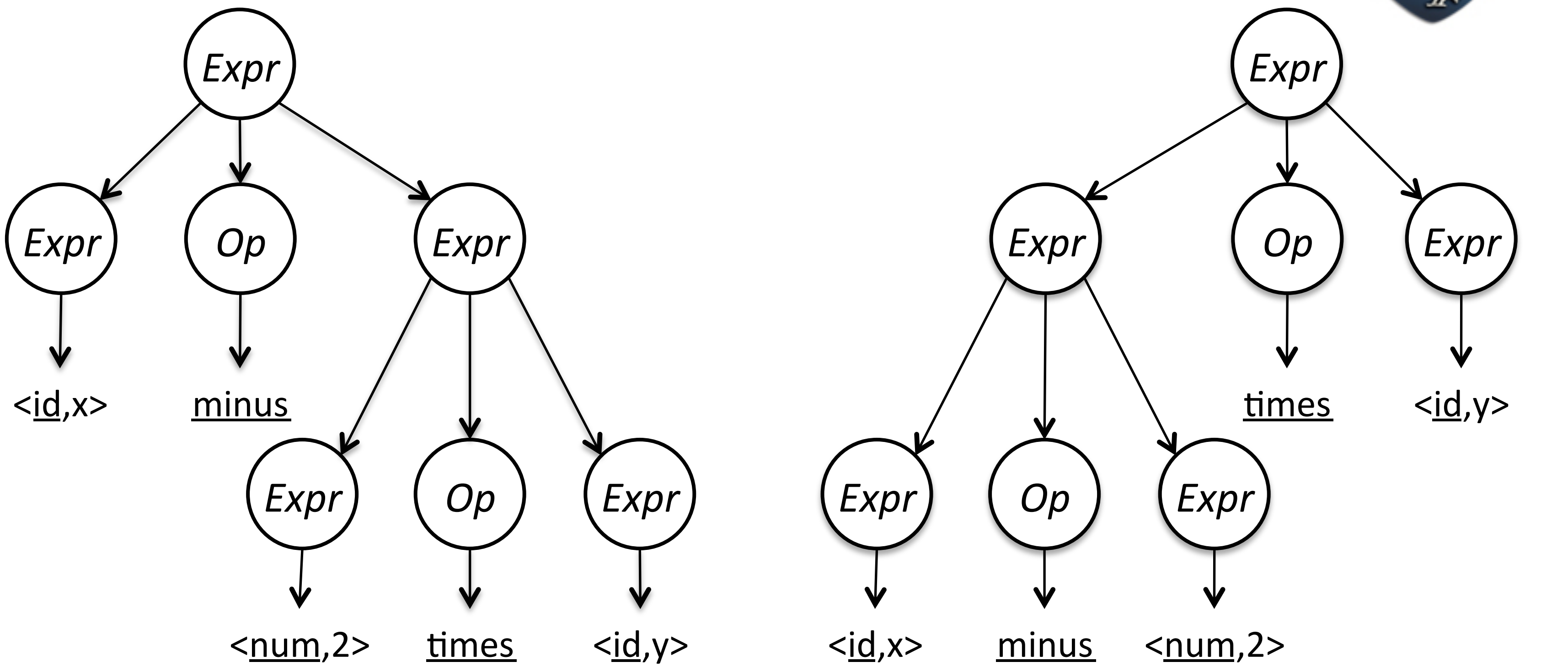
- Build an executable image that implements the source program
- *Implements* implies that the source program has well-defined meaning

Ambiguity is the opposite of "well-defined"

- Ambiguous constructs have multiple meanings
- A program with multiple meanings is not, in general, a good thing

**Programming languages (& their designers) should abhor ambiguity**

# Leftmost & Rightmost Parse Trees




The two parse trees generate different evaluation orders

Different parse trees for leftmost & rightmost derivations means that the underlying grammar is ambiguous

$\longrightarrow$ Not a good thing.

# Ambiguity

**The grammar has multiple leftmost derivations (& rightmost derivations)**

| ***Rule*** | ***Sentential Form*** |
|---|---|
| — | *Expr* |
| 0 | *Expr Op Expr* |
| 2 | <id,x> *Op Expr* |
| 4 | <id,x> — *Expr* |
| 0 | <id,x> — *Expr Op Expr* |
| 1 | <id,x> — <num,2> *Op Expr* |
| 5 | <id,x> — <num,2> * *Expr* |
| 2 | <id,x> — <num,2> * <id,y> |

| ***Rule*** | ***Sentential Form*** |
|---|---|
| — | *Expr* |
| 0 | *Expr Op Expr* |
| 0 | *Expr Op Expr Op Expr* |
| 2 | <id,x> *Op Expr Op Expr* |
| 4 | <id,x> — *Expr Op Expr* |
| 1 | <id,x> — <num,2> *Op Expr* |
| 5 | <id,x> — <num,2> * *Expr* |
| 1 | <id,x> — <num,2> * <id,y> |

Any grammar that has multiple leftmost (or multiple rightmost) derivations for a single sentence is an **ambiguous** grammar.

⇒ *Ambiguity is bad in a programming language*

# Ambiguity

## What should you do with an ambiguous grammar?

You rewrite it to remove the ambiguity.

| | | | |
|---|---|---|---|
| 0 | *Expr* | → | *Expr Op Expr* |
| 1 | | \| | number |
| 2 | | \| | identifier |
| 3 | *Op* | → | plus |
| 4 | | \| | minus |
| 5 | | \| | times |
| 6 | | \| | divide |

***Ambiguous Grammar***

| | | | |
|---|---|---|---|
| 0 | *Expr* | → | *Expr Op Value* |
| 1 | | \| | *Value* |
| 2 | *Value* | → | number |
| 3 | | \| | identifier |
| 4 | *Op* | → | plus |
| 5 | | \| | minus |
| 6 | | \| | times |
| 7 | | \| | divide |

***Rewritten Grammar***

In this case, the ambiguity that we see arises from the fact that rule 0 generates *Expr*, its lhs, at both the right & left ends of its rhs.

⇒ *The ambiguity allows derivations with the wrong evaluation order*

# Ambiguity

## Leftmost derivation of x – 2 * y

| | | | |
|---|---|---|---|
| 0 | *Expr* | → | *Expr Op Value* |
| 1 | | \| | *Value* |
| 2 | *Value* | → | <u>number</u> |
| 3 | | \| | <u>identifier</u> |
| 4 | *Op* | → | <u>plus</u> |
| 5 | | \| | <u>minus</u> |
| 6 | | \| | <u>times</u> |
| 7 | | \| | <u>divide</u> |

| *Rule* | *Sentential Form* |
|---|---|
| — | *Expr* |
| 0 | *Expr Op Value* |
| 0 | *Expr Op Value Op Value* |
| 1 | *Value Op Value Op Value* |
| 3 | <id,x> *Op Value Op Value* |
| 5 | <id,x> — *Value Op Value* |
| 2 | <id,x> — <num,2> *Op Value* |
| 6 | <id,x> — <num,2> * Value |
| 3 | <id,x> — <num,2> * <id,y> |

The unambiguous grammar requires one more step in this derivation: the rewrite through *Value* for **x**.

Seems like a small price to pay. (**TANSTAAFL**)

# Ambiguity

## Definitions

- A context-free grammar $G$ is **ambiguous** if there exists has more than one leftmost derivation for some sentence in $L(G)$
- A context-free grammar $G$ is **ambiguous** if there exists has more than one rightmost derivation for some sentence in $L(G)$
- A context-free grammar $G$ is **ambiguous** if the rightmost and leftmost derivations produce different parse trees
  - *However, the rightmost and leftmost derivations may differ*

## The classic example — the if-then-else problem

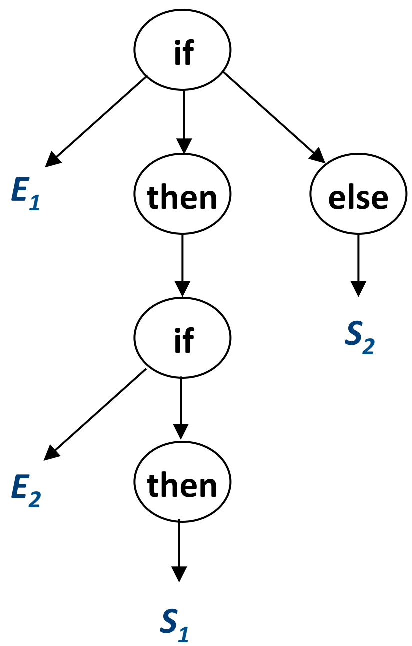
| | | | |
|---|---|---|---|
| 0 | *Stmt* | → | <u>if</u> *Expr* <u>then</u> *Stmt* |
| 1 | | \| | <u>if</u> *Expr* <u>then</u> *Stmt* <u>else</u> *Stmt* |
| 2 | | \| | … *other statements* … |

# Ambiguity

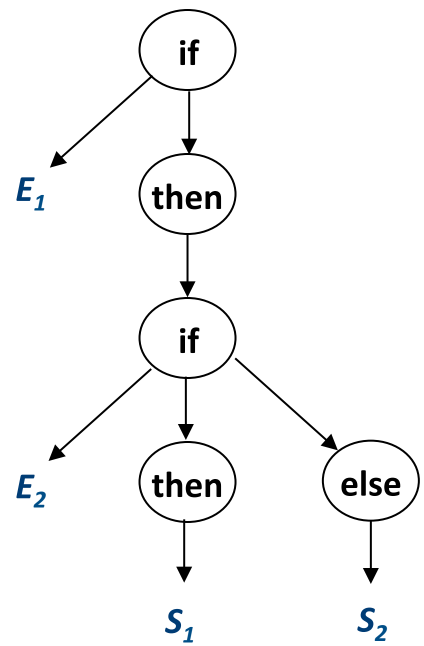
## The straightforward if-then-else grammar is ambiguous

Consider the sentential form:

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




*production 2, then production 1*




*production 1, then production 2*

## Two parse trees, two meanings

# Ambiguity

**The new grammar forces the structure to match the desired meaning.**

## Rewriting the grammar to remove the ambiguity

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

| | | | |
|---|---|---|---|
| 0 | *Stmt* | → | if *Expr* then *Stmt* |
| 1 | | \| | if *Expr* then *WithElse* else *Stmt* |
| 2 | | \| | … *other statements* … |
| 3 | *WithElse* | → | if *Expr* then *WithElse* else *WithElse* |
| 4 | | \| | … *other statements* … |

**The critical point:** the **if-then** case is not in *… other statements …*

With this grammar, the example has only one rightmost derivation

**Intuition:** once inside a *WithElse*, derivation cannot generate an unmatched else

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

# Ambiguity

## Derivation for the example sentence

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

| *Rule* | *Sentential Form* |
|---|---|
| — | *Stmt* |
| 0 | <u>if</u> *Expr* <u>then</u> *Stmt* |
| 1 | <u>if</u> *Expr* <u>then</u> <u>if</u> *Expr* <u>then</u> *WithElse* <u>else</u> *Stmt* |
| 2 | <u>if</u> *Expr* <u>then</u> <u>if</u> *Expr* <u>then</u> *WithElse* <u>else</u> $S_2$ |
| 4 | <u>if</u> *Expr* <u>then</u> <u>if</u> *Expr* <u>then</u> $S_1$ <u>else</u> $S_2$ |
| ? | <u>if</u> *Expr* <u>then</u> <u>if</u> $E_2$ <u>then</u> $S_1$ <u>else</u> $S_2$ |
| ? | <u>if</u> $E_1$ <u>then</u> <u>if</u> $E_2$ <u>then</u> $S_1$ <u>else</u> $S_2$ |

Other productions to derive *Expr*s

The new grammar has only one **rightmost** derivation for the example

# A Final Word on **IF-THEN-ELSE**

## The IF-THEN-ELSE ambiguity is a bit more subtle than it looks

| | | |
|---|---|---|
| *Stmts* | → | *Stmts Stmt* |
| | \| | *Stmt* |
| *Stmt* | → | *Reference* = *Expr* |
| | \| | IF ( *Expr* ) THEN *Stmt* |
| | \| | IF ( *Expr* ) THEN *Stmt* ELSE *Stmt* |

… where *Reference* and *Expr* are **NT**s defined elsewhere

We know how to fix this ambiguity using the "withelse" rewrite

## What happens if we add a *Stmt* that contains *Stmt* ?

| | | |
|---|---|---|
| *Stmt* | → | WHILE ( *Expr* ) *Stmts* |

*Stmt* can derive **IF-THEN-ELSE**, which creates an ambiguity when a **WHILE** is inside an **IF-THEN** or an **IF-THEN-ELSE**

→ *Either disallow **IF-THEN** inside while or require brackets around Stmts list*

# Deeper Ambiguity

## Ambiguity usually refers to confusion in the CFG

Overloading can create deeper confusions about meaning

a = f(17)

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

## Disambiguating this confusion requires context

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

**The alternative:** *change the syntax*
C introduced square brackets for subscripts
BCPL used !, the indirection operator

# 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
- 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