



COMP 412
FALL 2009

Introduction to Parsing

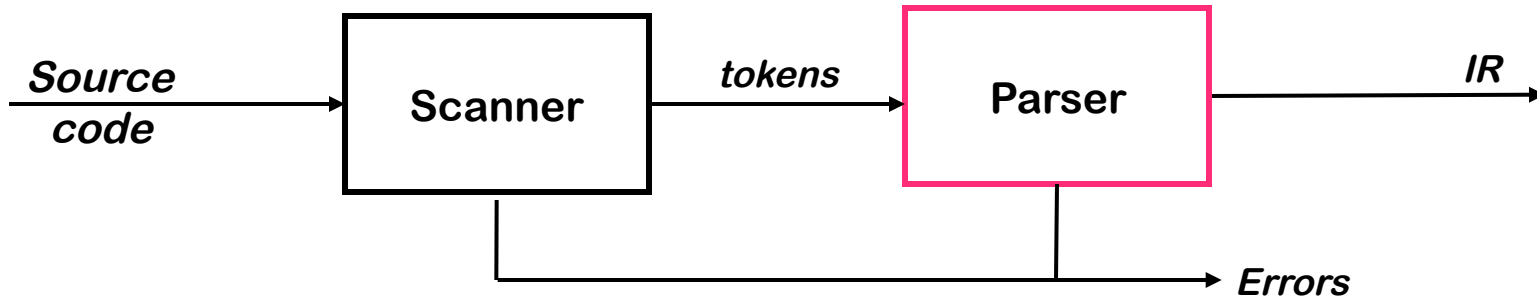
Comp 412

Copyright 2009, Keith D. Cooper & Linda Torczon, all rights reserved.

Students enrolled in Comp 412 at Rice University have explicit permission to make copies of these materials for their personal use.

Faculty from other educational institutions may use these materials for nonprofit educational purposes, provided this copyright notice is preserved.

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

Think of this as the mathematics of diagramming sentences



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 for our study of parsing

- 1 Context-free grammars and derivations Today
 - 2 Top-down parsing
 - Generated LL(1) parsers & hand-coded recursive descent parsers
 - 3 Bottom-up parsing
 - Generated LR(1) parsers
- | Lab 2
-



Specifying Syntax with a Grammar

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

$$\begin{array}{l} \textit{SheepNoise} \rightarrow \textit{SheepNoise} \underline{\textit{baa}} \\ \quad \quad \quad | \quad \underline{\textit{baa}} \end{array}$$

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

Example due to Dr. Scott K. Warren

Deriving Syntax



We can use the *SheepNoise* grammar to create sentences
— use the productions as *rewriting rules*

<i>Rule</i>	<i>Sentential Form</i>
—	<i>SheepNoise</i>
2	<u>baa</u>

<i>Rule</i>	<i>Sentential Form</i>
—	<i>SheepNoise</i>
1	<i>SheepNoise</i> <u>baa</u>
1	<i>SheepNoise</i> <u>baa</u> <u>baa</u>
2	<u>baa</u> <u>baa</u> <u>baa</u>

<i>Rule</i>	<i>Sentential Form</i>
—	<i>SheepNoise</i>
1	<i>SheepNoise</i> <u>baa</u>
2	<u>baa</u> <u>baa</u>

And so on ...

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



Why Not Use Regular Languages & DFAs?

Not all languages are regular

(RL's \subset CFL's \subset CSL's)

You cannot construct DFA's to recognize these languages

- $L = \{ p^k q^k \}$ (parenthesis languages)
- $L = \{ w c w^r \mid w \in \Sigma^* \}$

Neither of these is a regular language

(nor an RE)

To recognize these features requires an arbitrary amount of context (left or right ...)

But, this issue is somewhat subtle. You can construct DFA's for

- Strings with alternating 0's and 1's
($\epsilon \mid 1$)(01)*($\epsilon \mid 0$)
- Strings with an even number of 0's and 1's

RE's can count bounded sets and bounded differences



Limits of Regular Languages

Advantages of Regular Expressions

- Simple & powerful notation for specifying patterns
- Automatic construction of fast recognizers
- Many kinds of syntax can be specified with REs

Example — a regular expression for arithmetic expressions

$Term \rightarrow [a-zA-Z] ([a-zA-Z] | [0-9])^*$

$Op \rightarrow + | - | * | /$

$Expr \rightarrow (Term Op)^* Term$

$[a-zA-Z] ([a-zA-Z] | [0-9])^* (+ | - | * | /)^* [a-zA-Z] ([a-zA-Z] | [0-9])^*$

Of course, this would generate a DFA ...

If REs are so useful ... *Why not use them for everything?*

Cannot add
parentheses!



A More Useful Grammar Than Sheep Noise

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

1	<i>Expr</i>	→	<i>Expr Op Expr</i>
2			<u>number</u>
3			<u>id</u>
4	<i>Op</i>	→	+
5			-
6			*
7			/

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

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

We denote this derivation: $Expr \Rightarrow^* \underline{id} - \underline{num} * \underline{id}$



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
- *Rightmost derivation* — replace rightmost NT at each step

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, it turns out to be different



The Two Derivations for $\underline{x} - \underline{2} * \underline{y}$

Rule	Sentential Form
—	<i>Expr</i>
1	<i>Expr Op Expr</i>
3	$\langle id, \underline{x} \rangle Op Expr$
5	$\langle id, \underline{x} \rangle - Expr$
1	$\langle id, \underline{x} \rangle - Expr Op Expr$
2	$\langle id, \underline{x} \rangle - \langle num, \underline{2} \rangle Op Expr$
6	$\langle id, \underline{x} \rangle - \langle num, \underline{2} \rangle * Expr$
3	$\langle id, \underline{x} \rangle - \langle num, \underline{2} \rangle * \langle id, \underline{y} \rangle$

Leftmost derivation

Rule	Sentential Form
—	<i>Expr</i>
1	<i>Expr Op Expr</i>
3	<i>Expr Op</i> $\langle id, \underline{y} \rangle$
6	<i>Expr</i> * $\langle id, \underline{y} \rangle$
1	<i>Expr Op Expr</i> * $\langle id, \underline{y} \rangle$
2	<i>Expr Op</i> $\langle num, \underline{2} \rangle$ * $\langle id, \underline{y} \rangle$
5	<i>Expr</i> - $\langle num, \underline{2} \rangle$ * $\langle id, \underline{y} \rangle$
3	$\langle id, \underline{x} \rangle - \langle num, \underline{2} \rangle * \langle id, \underline{y} \rangle$

Rightmost derivation

In both cases, $Expr \Rightarrow^* \underline{id} - \underline{num} * \underline{id}$

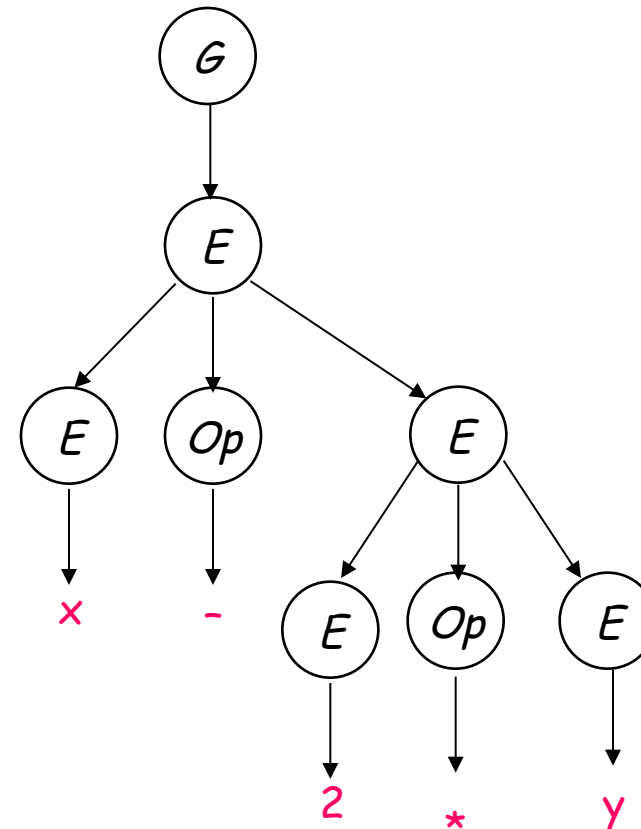
- The two derivations produce different parse trees
- The parse trees imply different evaluation orders!



Derivations and Parse Trees

Leftmost derivation

Rule	Sentential Form
—	<i>Expr</i>
1	<i>Expr Op Expr</i>
3	$\langle id, \underline{x} \rangle Op Expr$
5	$\langle id, \underline{x} \rangle - Expr$
1	$\langle id, \underline{x} \rangle - Expr Op Expr$
2	$\langle id, \underline{x} \rangle - \langle num, \underline{2} \rangle Op Expr$
6	$\langle id, \underline{x} \rangle - \langle num, \underline{2} \rangle * Expr$
3	$\langle id, \underline{x} \rangle - \langle num, \underline{2} \rangle * \langle id, \underline{y} \rangle$



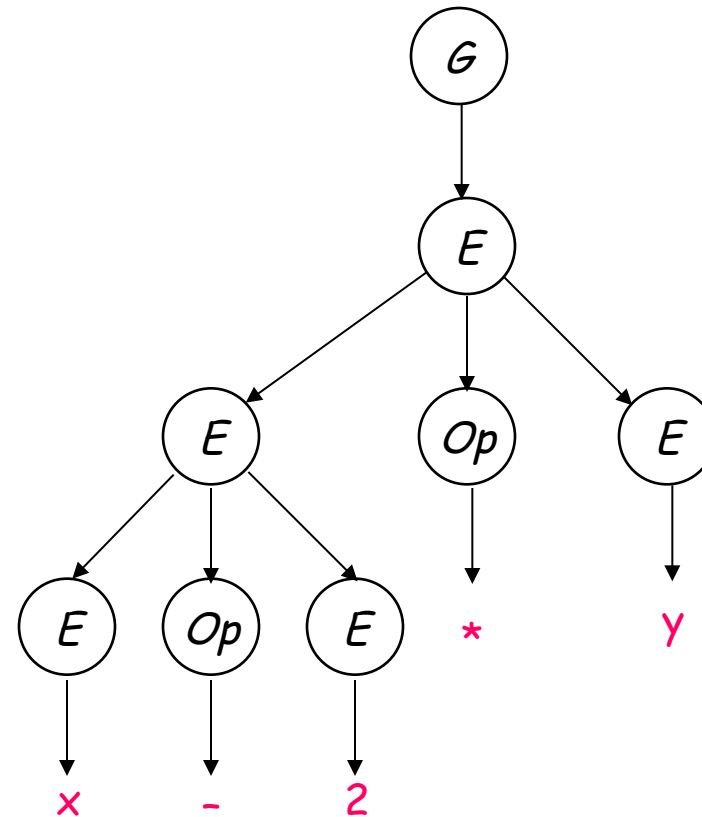
This evaluates as $\underline{x} - (\underline{2} * \underline{y})$



Derivations and Parse Trees

Rightmost derivation

Rule	Sentential Form
—	<i>Expr</i>
1	<i>Expr Op Expr</i>
3	<i>Expr Op</i> $\langle id, y \rangle$
6	<i>Expr</i> * $\langle id, y \rangle$
1	<i>Expr Op Expr</i> * $\langle id, y \rangle$
2	<i>Expr Op</i> $\langle num, 2 \rangle$ * $\langle id, y \rangle$
5	<i>Expr</i> - $\langle num, 2 \rangle$ * $\langle id, y \rangle$
3	$\langle id, x \rangle$ - $\langle num, 2 \rangle$ * $\langle id, y \rangle$



This evaluates as $(x - 2) * y$

This ambiguity is NOT good

Derivations and Precedence



*These two derivations point out a problem with the grammar:
It has no notion of precedence, or implied order of evaluation*

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)*



Derivations and Precedence

Adding the standard algebraic precedence produces:

level two	1	<i>Goal</i>	→	<i>Expr</i>
	2	<i>Expr</i>	→	<i>Expr</i> + <i>Term</i>
	3			<i>Expr</i> - <i>Term</i>
	4			<i>Term</i>
level one	5	<i>Term</i>	→	<i>Term</i> * <i>Factor</i>
	6			<i>Term</i> / <i>Factor</i>
	7			<i>Factor</i>
	8	<i>Factor</i>	→	<u>number</u>
	9			<u>id</u>
	10			<u>(Expr)</u>

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 $x - 2 * y$*

Cannot handle precedence in an RE for expressions

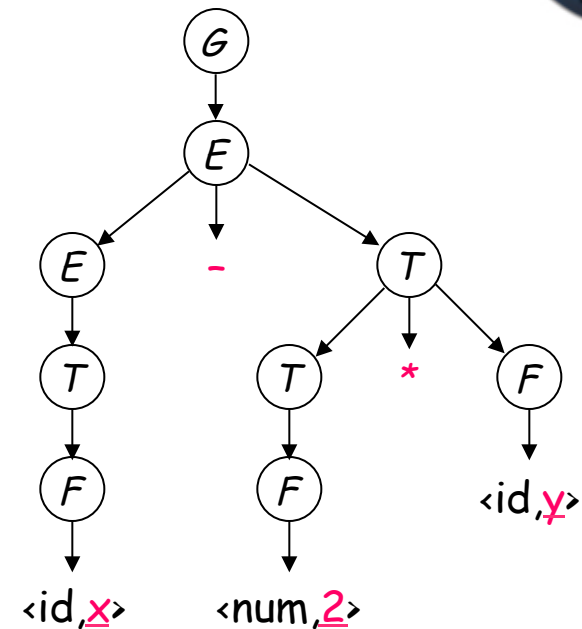
Introduced parentheses, too (beyond power of an RE)



Derivations and Precedence

Rule	Sentential Form
—	Goal
2	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

It derives $x - (2 * y)$, along with an appropriate parse tree.

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



Ambiguous Grammars

Our original expression grammar had other problems

1	$Expr$	\rightarrow	$Expr Op Expr$
2			<u>number</u>
3			<u>id</u>
4	Op	\rightarrow	+
5			-
6			*
7			/

Rule	Sentential Form
—	$Expr$
1	$Expr Op Expr$
③	$\langle id, x \rangle Op Expr$
5	$\langle id, x \rangle - Expr$
1	$\langle id, x \rangle - Expr Op Expr$
2	$\langle id, x \rangle - \langle num, 2 \rangle Op Expr$
6	$\langle id, x \rangle - \langle num, 2 \rangle * Expr$
3	$\langle id, x \rangle - \langle num, 2 \rangle * \langle id, y \rangle$

- This grammar allows multiple leftmost derivations for $x - 2 * y$
- Hard to automate derivation if > 1 choice
- The grammar is *ambiguous*

Different choice
than the first time



Two Leftmost Derivations for $x - 2 * y$

The Difference:

- Different productions chosen on the second step

Rule	Sentential Form
—	<i>Expr</i>
1	<i>Expr Op Expr</i>
③	$\langle id, x \rangle Op Expr$
5	$\langle id, x \rangle - Expr$
1	$\langle id, x \rangle - Expr Op Expr$
2	$\langle id, x \rangle - \langle num, 2 \rangle Op Expr$
6	$\langle id, x \rangle - \langle num, 2 \rangle * Expr$
3	$\langle id, x \rangle - \langle num, 2 \rangle * \langle id, y \rangle$

Original choice

Rule	Sentential Form
—	<i>Expr</i>
1	<i>Expr Op Expr</i>
①	<i>Expr Op Expr Op Expr</i>
3	$\langle id, x \rangle Op Expr Op Expr$
5	$\langle id, x \rangle - Expr Op Expr$
2	$\langle id, x \rangle - \langle num, 2 \rangle Op Expr$
6	$\langle id, x \rangle - \langle num, 2 \rangle * Expr$
3	$\langle id, x \rangle - \langle num, 2 \rangle * \langle id, y \rangle$

New choice

- Both derivations succeed in producing $x - 2 * y$



Two Leftmost Derivations for $x - 2 * y$

The Difference:

- Different productions chosen on the second step

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

Original choice

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

New choice

Different choices in same situation, again

Remember nondeterminism?



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
 - However, they must have the same parse tree!

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

$$\begin{array}{l} \text{Stmt} \rightarrow \underline{\text{if}} \text{ Expr } \underline{\text{then}} \text{ Stmt} \\ \quad \quad | \underline{\text{if}} \text{ Expr } \underline{\text{then}} \text{ Stmt } \underline{\text{else}} \text{ Stmt} \\ \quad \quad | \dots \text{ other stmts } \dots \end{array}$$

This ambiguity is inherent in the grammar

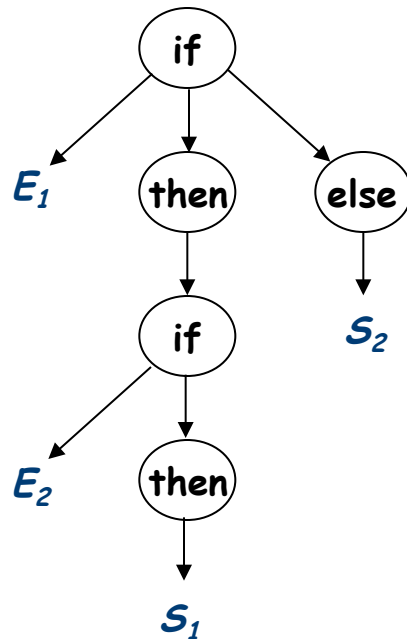


Ambiguity

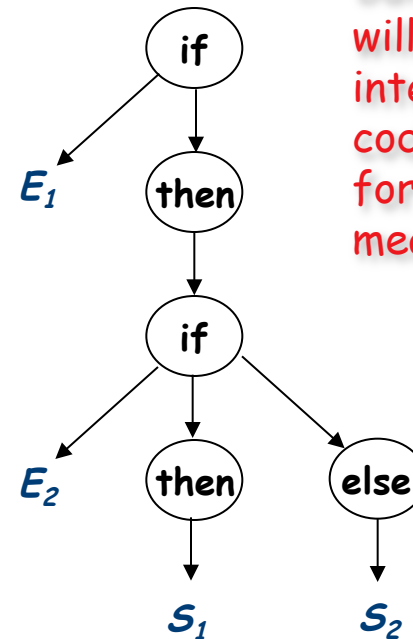
This sentential form has two derivations

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

Part of the problem is that the structure built by the parser will determine the interpretation of the code, and these two forms have different meanings!



*production 2, then
production 1*



*production 1, then
production 2*

Ambiguity

The grammar forces the structure to match the desired meaning.



Removing the ambiguity

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

1	<i>Stmt</i>	→	<u>if</u> <i>Expr</i> <u>then</u> <i>Stmt</i>
2			<u>if</u> <i>Expr</i> <u>then</u> <i>WithElse</i> <u>else</u> <i>Stmt</i>
3			<i>Other Statements</i>
4	<i>WithElse</i>	→	<u>if</u> <i>Expr</i> <u>then</u> <i>WithElse</i> <u>else</u> <i>WithElse</i>
5			<i>Other Statements</i>

With this grammar, example has only one rightmost derivation

Intuition: once into *WithElse*, we cannot generate an unmatched else
... a final if without an else can only come through rule 2 ...



Ambiguity

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

Rule	Sentential Form
—	$Stmt$
1	<u>if</u> $Expr$ <u>then</u> $Stmt$
2	<u>if</u> $Expr$ <u>then</u> <u>if</u> $Expr$ <u>then</u> $WithElse$ <u>else</u> $Stmt$
3	<u>if</u> $Expr$ <u>then</u> <u>if</u> $Expr$ <u>then</u> $WithElse$ <u>else</u> S_2
5	<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 $Exprs$

This grammar has only one rightmost derivation for the example



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 variable

Disambiguating this one requires context

- Need values of declarations
- 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
- 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