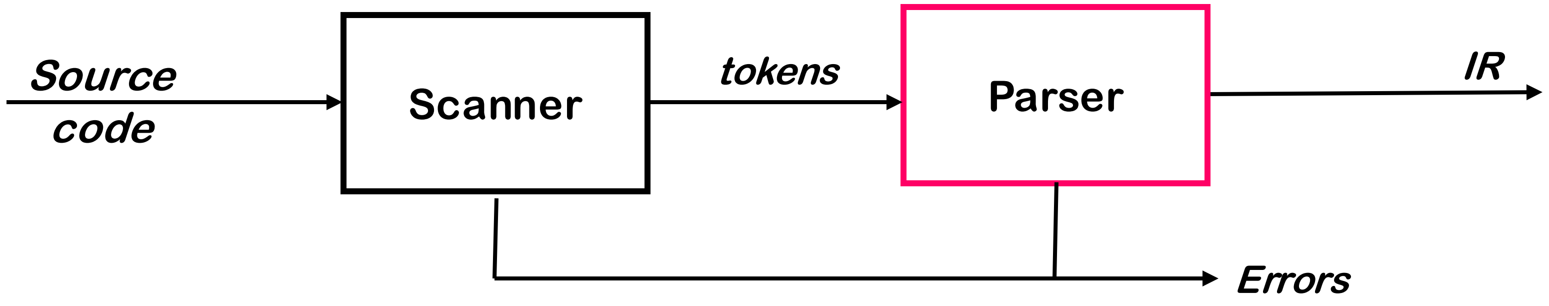
RICE

COMP 412
FALL 2010

Introduction to Parsing

Comp 412

The Front End



Parser

- Checks the stream of <u>words</u> and their <u>parts of speech</u> (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 chapter 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

We will define "context free" today. I am just deferring the definition for a couple of slides.

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 *nonterminal 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

From Lecture 1

Deriving Syntax

We can use the *SheepNoise* grammar to create sentences

- use the productions as *rewriting rules*

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)*
- *$L = \{ wcw^r \mid w \in \Sigma^* \}$*

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 <u>can</u> construct DFA's for

- Strings with alternating 0's and 1's
  $(\varepsilon \mid 1) (01)^* (\varepsilon \mid 0)$
- 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

$$\begin{array}{lll} \textit{Term} & \rightarrow & [\text{a-zA-Z}]\ ([\text{a-zA-Z}]\ |\ [\underline{0}\text{-}\underline{9}])^* \\ \textit{Op} & \rightarrow & \underline{+}\ |\ \underline{-}\ |\ \underline{*}\ |\ \underline{/} \\ \textit{Expr} & \rightarrow & (\ \textit{Term Op}\ )^*\ \textit{Term} \end{array}$$

$$([\text{a-zA-Z}]\ ([\text{a-zA-Z}]\ |\ [\underline{0}\text{-}\underline{9}])^*\ (\underline{+}\ |\ \underline{-}\ |\ \underline{*}\ |\ \underline{/}))^*\ [\text{a-zA-Z}]\ ([\text{a-zA-Z}]\ |\ [\underline{0}\text{-}\underline{9}])$$

Of course, this would generate a DFA …

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

⇒ Cannot add parenthesis, brackets, begin-end pairs, …

Context-free Grammars

What makes a grammar "context free"?

The SheepNoise grammar has a specific form:

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

Productions have a single nonterminal on the left hand side, which makes it impossible to encode left or right context.

$\Rightarrow$ The grammar is <u>context</u> free.

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

Notice that *L(SheepNoise)* is actually a regular language: $\underline{\text{baa}}^{+}$

Classic definition: any language that can be recognized by a push-down automaton is a context-free language.

A More Useful Grammar Than Sheep Noise

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

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

Rule	*Sentential Form*
—	*Expr*
0	*Expr Op Expr*
2	<id,<u>x</u>> *Op Expr*
4	<id,<u>x</u>> - *Expr*
0	<id,<u>x</u>> - *Expr Op Expr*
1	<id,<u>x</u>> - <num,<u>2</u>> *Op Expr*
5	<id,<u>x</u>> - <num,<u>2</u>> * *Expr*
2	<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* $\Rightarrow^*$ <u>id</u> – <u>num</u> * <u>id</u>

Derivations

The point of parsing is to construct a derivation

- At each step, we choose a nonterminal 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

Derivations

The point of parsing is to construct 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 γ_i is a sentential form
  - If γ contains only terminal symbols, γ is a sentence in *L(G)*
  - If γ contains 1 or more non-terminals, γ is a sentential form
- To get γ_i from γ_{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 β to get γ_i
  - In a leftmost derivation, it would be the first *NT* $A \in \gamma_{i-1}$

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

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

The Two Derivations for <u>x</u> – <u>2</u> * <u>y</u>

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

Leftmost derivation

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

Rightmost derivation

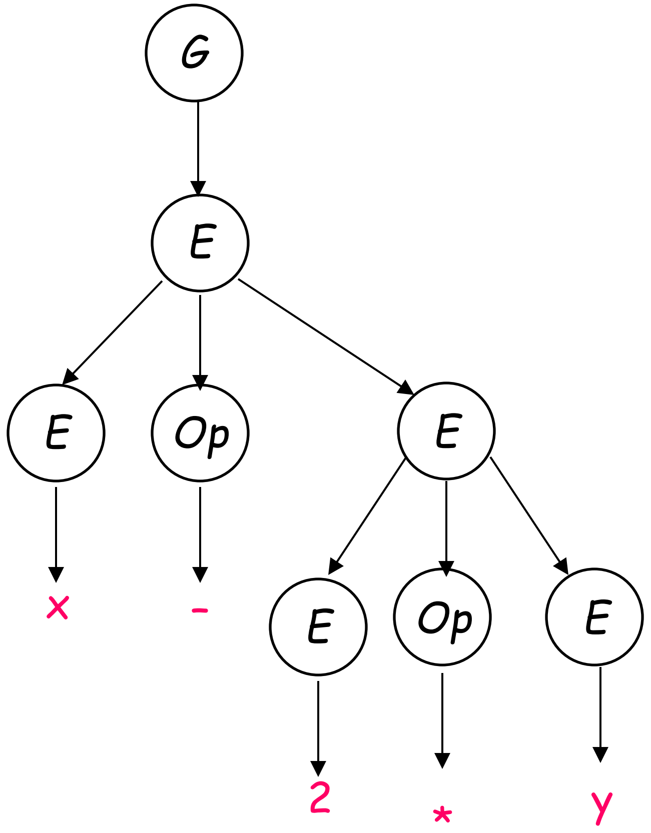
In both cases, *Expr* $\Rightarrow^*$ <u>id</u> – <u>num</u> * <u>id</u>
- The two derivations produce different parse trees
- The parse trees imply different evaluation orders!

Derivations and Parse Trees

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>*

This evaluates as x – (2 * y)

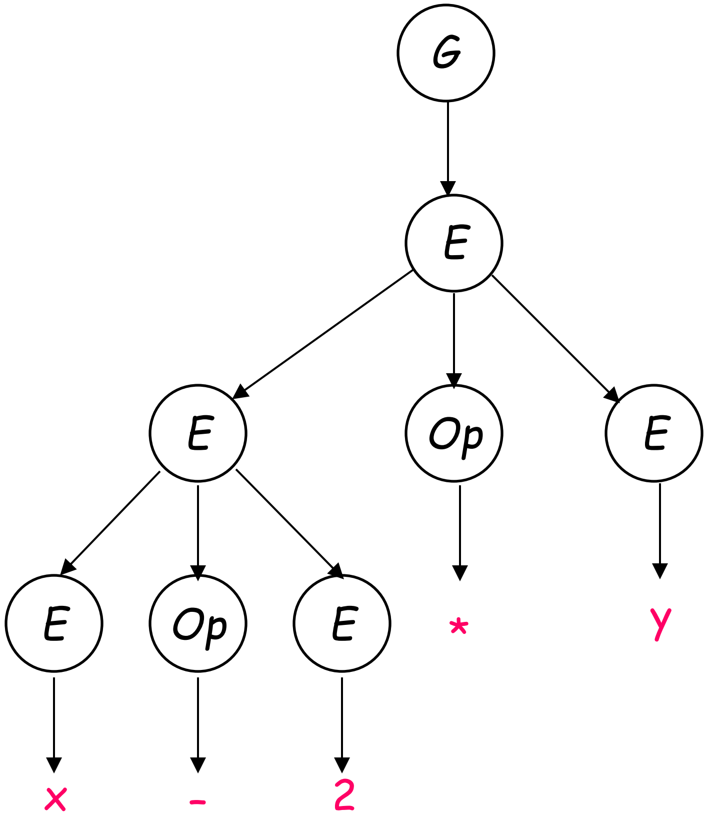


Derivations and Parse Trees

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>

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 <u>*precedence*</u>*, or implied order of evaluation*

To add precedence

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

For algebraic expressions

- Parentheses first (*level 1*)
- Multiplication and division, next (*level 2*)
- Subtraction and addition, last (*level 3*)

Derivations and Precedence

Adding the standard algebraic precedence produces:

	0	*Goal*	→	*Expr*
level 3	1	*Expr*	→	*Expr* + *Term*
	2		|	*Expr* - *Term*
	3		|	*Term*
level 2	4	*Term*	→	*Term* * *Factor*
	5		|	*Term* / *Factor*
	6		|	*Factor*
level 1	7	*Factor*	→	(*Expr*)
	8		|	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
- Correctness trumps the speed of the parser

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

Cannot handle precedence in an RE for expressions

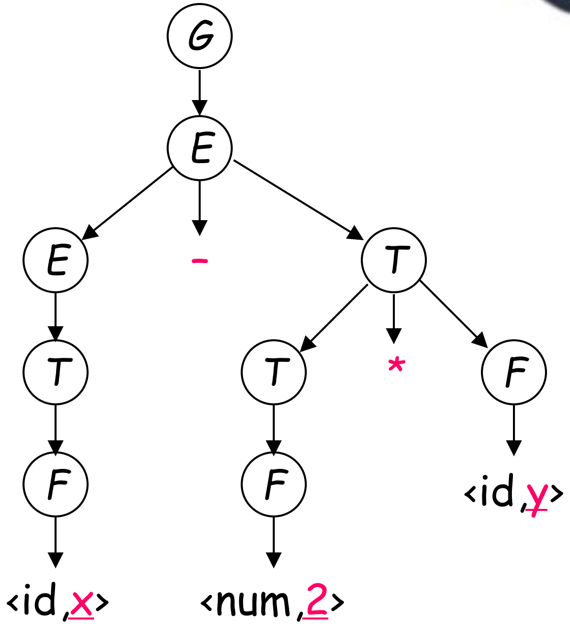
Introduced parentheses, too (beyond power of an RE)

One form of the "classic expression grammar"

Derivations and Precedence

Rule	*Sentential Form*
—	*Goal*
0	*Expr*
2	*Expr* - *Term*
4	*Expr* - *Term* * *Factor*
9	*Expr* - *Term* * <id,y>
6	*Expr* - *Factor* * <id,y>
8	*Expr* - <num,2> * <id,y>
3	*Term* - <num,2> * <id,y>
6	*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

Let's leap back to our original expression grammar.

It had other problems.

0	*Expr*	→	*Expr Op Expr*
1		|	number
2		|	id
3	*Op*	→	+
4		|	-
5		|	*
6		|	/

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>

Different choice than the first time

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

Two Leftmost Derivations for *x* – *2* * *y*

The Difference:

- Different productions chosen on the second step

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*
1	<id,x> - <num,2> * <id,y>

Original choice

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*
2	<id,x> - <num,2> * <id,y>

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
—	*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>

Original choice

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*
2	<id,x> - <num,2> * <id,y>

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

$$
\begin{aligned}
\textit{Stmt} \rightarrow\; & \underline{\text{if}}\;\; \textit{Expr}\;\; \underline{\text{then}}\;\; \textit{Stmt} \\
\mid\; & \underline{\text{if}}\;\; \textit{Expr}\;\; \underline{\text{then}}\;\; \textit{Stmt}\;\; \underline{\text{else}}\;\; \textit{Stmt} \\
\mid\; & \ldots\; \textit{other stmts}\; \ldots
\end{aligned}
$$

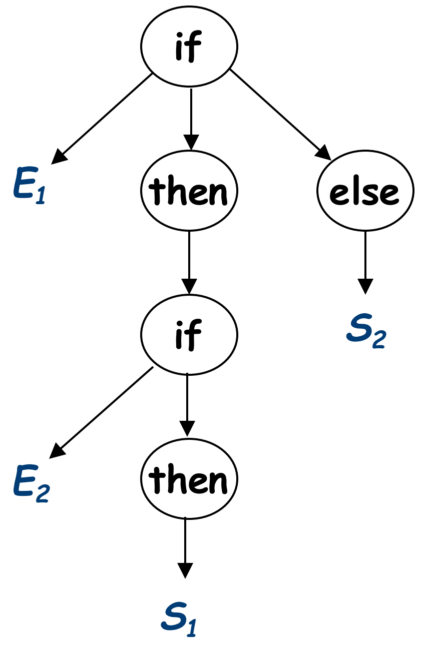
This ambiguity is inherent in the grammar

Ambiguity

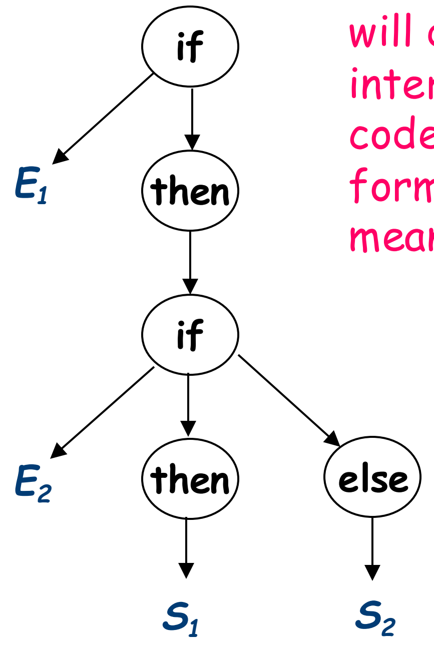
This sentential form has two derivations

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

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 <u>else</u> to innermost unmatched <u>if</u> (common sense rule)

0	*Stmt*	$\rightarrow$	<u>if</u> *Expr* <u>then</u> *Stmt*
1		|	<u>if</u> *Expr* <u>then</u> *WithElse* <u>else</u> *Stmt*
2		|	*Other Statements*
3	*WithElse*	$\rightarrow$	<u>if</u> *Expr* <u>then</u> *WithElse* <u>else</u> *WithElse*
4		|	*Other Statements*

With this grammar, example has only one rightmost derivation

Intuition: once into *WithElse*, we cannot generate an unmatched <u>else</u>
… a final <u>if</u> without an <u>else</u> can only come through rule 2 …

Ambiguity

<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 *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

†See Chapter 4