EECS 6083

Intro to Parsing

Context Free Grammars

Parsing

sequence of tokens $\xrightarrow{\text{parser}}$ syntax tree

- **Check the syntax (structure) of a program and create a tree representation of the program**
- **Programming languages have non-regular constructs.**
  - **Nesting**
  - **Recursion**
- **Context-Free Grammars are used to express the syntax for programming languages**

Syntax vs. Semantics

- **Syntax – structure of the program, expressed with grammatical rules.**
  - **Typically a Context Free Grammar (CFG) in Backus-Naur form (BNF) or Exteneded Backus-Naur form (EBNF).**
- **Semantics – meaning of the program, expressed with descriptive text or with *inference rules***
- **Consider the following English sentences:**

$$\underbrace{\text{I}}_{\text{subject}} \quad \underbrace{\text{fed}}_{\text{verb}} \quad \underbrace{\text{the elephant.}}_{\text{object}}$$

$$\underbrace{\text{I}}_{\text{subject}} \quad \underbrace{\text{fed}}_{\text{verb}} \quad \underbrace{\text{the brick wall}}_{\text{object}}$$

Both are syntactically correct, but the second is not semantically reasonable.

Context Free Grammars

- **Comprised of:**
  - a set of tokens or *terminal* symbols
  - a set of *non-terminal* symbols
  - a set of rules or *productions* which express the legal relationships between the symbols
  - A *start* or *goal* symbol
- **Example:**

(1) expr → expr - digit
(2) expr → expr + digit
(3) expr → digit
(4) digit → **0**|**1**|**2**|…|**9**

Terminals: -,+, **0,1,2,…,9**
Nonterminals: expr, digit
Start symbol: expr

Some Example CFGs

- **Palindromes over the alphabet {a, b, c}:**
  - (a palindrome is a word that has the same spelling backwards as forwards)
  - aabcbaa

Some Example CFGs (continued)

- **Palindromes over the alphabet {a, b, c}:**
  - (a palindrome is a word that has the same spelling backwards as forwards)
  - abba, c, abbcbcbba
- **CFG for Palindromes**

S→aSa

S→bSb

S→cSc

S→a

S→b

S→c

S→ε

terminal symbols: {a, b, c}

non-terminal symbols: {S}

Goal symbol: S

Some Example CFGs (continued)

- **Balanced Parenthesis and Square Brackets**
  - **E.g. ([[ ] (( ) [( )] [ ])])**

Some Example CFGs (continued)

- **Balanced Parenthesis and Square Brackets**
  - E.g. ([[ ] (( ) [( )] [ ])])
- **The CFG:**

B → (B)

| [B]

| BB

| ε

Checking for correct Syntax

- **Given a grammar for a language and a program how do you know if the syntax of the program is legal?**
- **A legal program can be *derived* from the start symbol of the grammar.**

Deriving a string

- The derivation begins with the start symbol
- At each step of a derivation the right hand side of a grammar rule is used to replace a non-terminal symbol.
- Continue replacing non-terminals until only terminal symbols remain

(1) expr -> expr - digit

(2) expr ->expr + digit

(3) expr -> digit

(4) digit -> 0|1|2|…|9

$$\text{expr} \xRightarrow{\text{Rule (1)}} \text{expr} - \text{digit} \xRightarrow{\text{Rule (4)}} \text{expr} - \mathbf{2} \xRightarrow{\text{Rule (2)}} \text{expr} + \text{digit} \mathbf{-} 2$$

$$\xRightarrow{\text{Rule (4)}} \text{expr} \mathbf{+ 8 - 2} \xRightarrow{\text{Rule (3)}} \text{digit} \mathbf{+ 8 - 2} \xRightarrow{\text{Rule (4)}} 3 + 8 - 2$$

Example Input:

3 + 8 – 2

Rightmost and leftmost derivations

- In a *rightmost derivation* the rightmost non-terminal is replaced at each step.
  - expr ⇒ expr – digit ⇒ expr – 2 ⇒ expr + digit – 2 ⇒ expr + 8 – 2 ⇒ digit + 8 – 2 ⇒ 3 + 8 – 2
  - corresponds to a postorder numbering in reverse of the internal nodes of the parse tree
- In a *leftmost derivation* the leftmost non-terminal is replaced at each step.
  - expr ⇒ expr – digit ⇒ expr + digit – digit ⇒ digit + digit – digit ⇒ 3 + digit –digit ⇒ 3 + 8 – digit ⇒ 3 + 8 – 2
  - corresponds to a preorder numbering of the nodes of a parse tree.

Parse tree

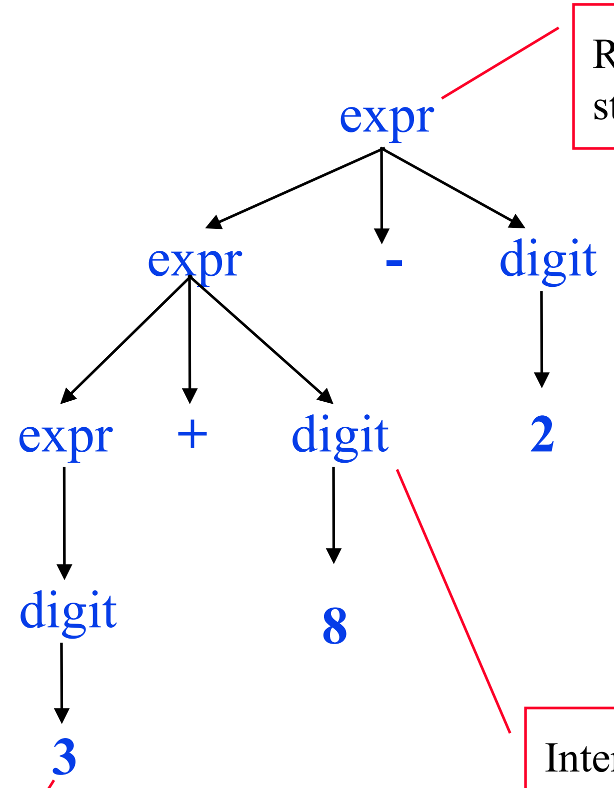
(1) expr -> expr - digit

(2) expr ->expr + digit

(3) expr -> digit

(4) digit -> 0|1|2|…|9

Example Input:
3 + 8 – 2



Root is the start symbol

Interior nodes are non-terminals

Leaves of parse tree are terminal symbols (tokens)

A More Useful Grammar

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

1	*Expr*	→	*Expr Op Expr*
2		|	number
3		|	id
4	*Op*	→	+
5		|	-
6		|	*
7		|	/

Rule	*Sentential Form*
—	*Expr*
1	*Expr Op Expr*
2	‹id,x› *Op Expr*
5	‹id,x› - *Expr*
1	‹id,x› - *Expr Op Expr*
2	‹id,x› - ‹num,2› *Op Expr*
6	‹id,x› - ‹num,2› * *Expr*
3	‹id,x› - ‹num,2› * ‹id,y›

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

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

The Two Derivations for *x* – 2 * *y*

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

Leftmost derivation

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

Rightmost derivation

In both cases, *Expr* ⇒* id – num * id

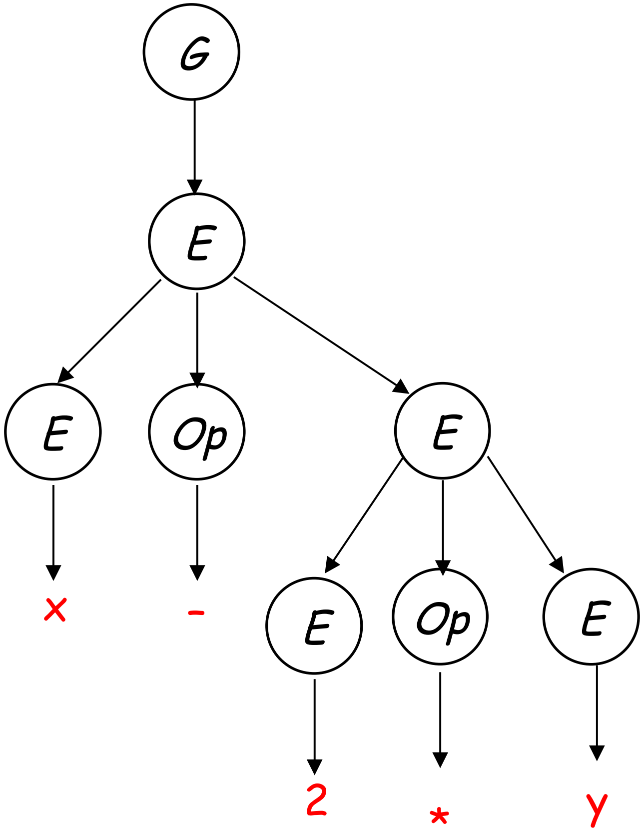
- The two derivations produce different parse trees
- The parse trees imply different evaluation orders!

Derivations and Parse Trees

Leftmost derivation

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

This evaluates as x - (2 * y)

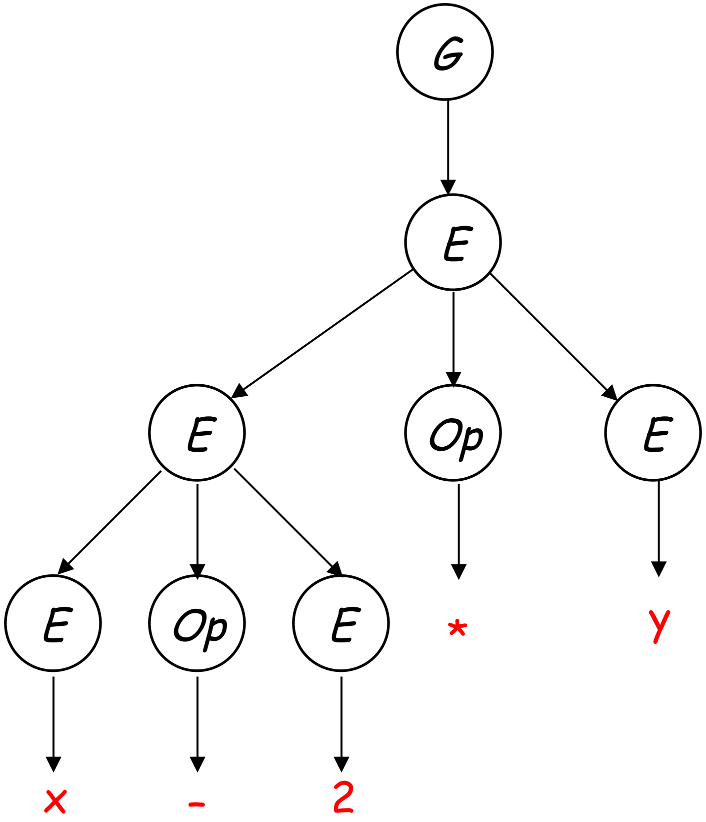


Derivations and Parse Trees

Rightmost derivation

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

This evaluates as (x - 2) * y



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:

	1	*Goal*	→	*Expr*
level two	2	*Expr*	→	*Expr* + *Term*
	3		|	*Expr* - *Term*
	4		|	*Term*
level one	5	*Term*	→	*Term* * *Factor*
	6		|	*Term* / *Factor*
	7		|	*Factor*
	8	*Factor*	→	number
	9		|	id

This grammar is slightly larger

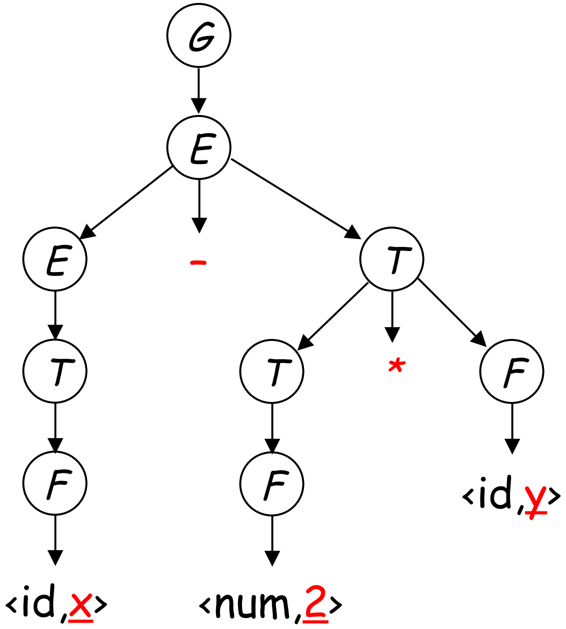
- Takes more rewriting to reach some of the terminal symbols
- Encodes expected precedence
- Produces same parse tree under leftmost & rightmost derivations

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

Derivations and Precedence

Rule	*Sentential Form*
—	*Goal*
1	*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

This produces x – (2 * y), along with an appropriate parse tree.

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

Ambiguous Grammars

Our original expression grammar had other problems

1	*Expr*	→	*Expr Op Expr*
2		|	number
3		|	id
4	*Op*	→	+
5		|	-
6		|	*
7		|	/

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

Original choice

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

New choice

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

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

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

Stmt → if *Expr* then *Stmt*
| if *Expr* then *Stmt* else *Stmt*
| … *other stmts* …

This ambiguity is entirely grammatical in nature

Ambiguity: Dangling Else example

stmt **→ if *expr* then *stmt***

| if *expr* then *stmt* else *stmt*

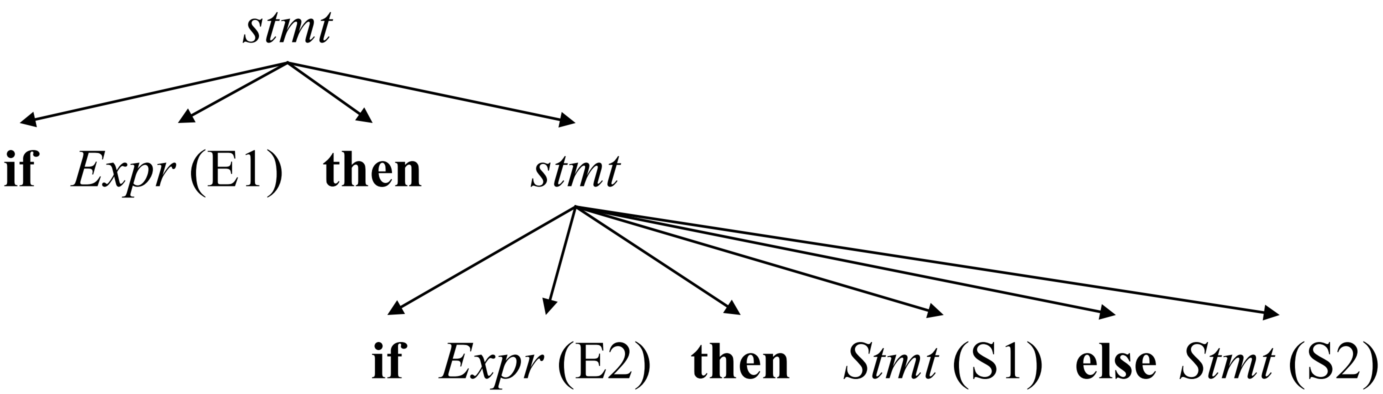
| other

Two parse trees for the legal sentence:

if E1 **then if** E2 **then** S1 **else** S2

Ambiguity: Dangling Else example (continued)

if E1 then (if E2 then S1 else S2)



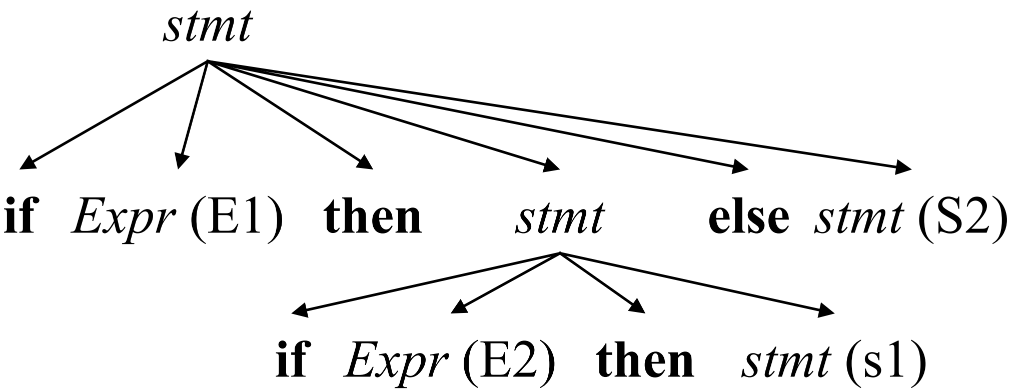
stmt

if *Expr* (E1) **then** *stmt*

if *Expr* (E2) **then** *Stmt* (S1) **else** *Stmt* (S2)

Ambiguity: Dangling Else example (continued)

if E1 then (if E2 then S1) else S2



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