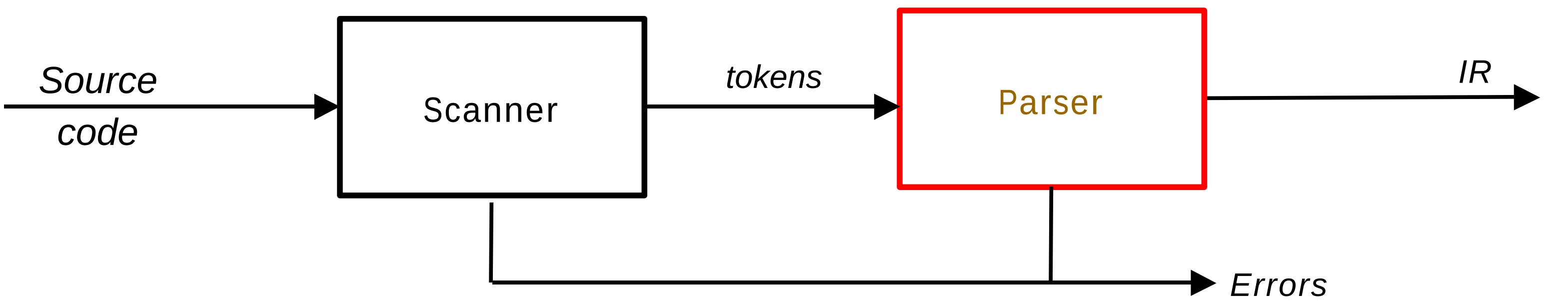
Introduction to Parsing

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
- May build 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

1. Context-free grammars and derivations
2. Top-down parsing
   - Hand-coded recursive descent parsers
3. Bottom-up parsing
   - Generated LR(1) parsers

Specifying Syntax with a Grammar

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

$$\textit{SheepNoise} \rightarrow \textit{SheepNoise}\ \underline{\text{baa}}$$
$$\mid\ \underline{\text{baa}}$$

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*

Rule	*Sentential Form*
—	*SheepNoise*
2	baa

Rule	*Sentential Form*
—	*SheepNoise*
1	*SheepNoise* baa
2	baa baa

Rule	*Sentential Form*
—	*SheepNoise*
1	*SheepNoise* baa
1	*SheepNoise* baa baa
2	baa baa baa

And so on ...

This example quickly runs out of intellectual steam ...

A More Useful Grammar

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

1	*Goal*	→	*Expr*
2	*Expr*	→	*Expr Op Expr*
3		|	<u>number</u>
4		|	<u>id</u>
5	*Op*	→	+
6		|	-
7		|	*
8		|	/

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

We denote this: *Expr* Þ * <u>id</u> - <u>num</u> * <u>id</u>

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

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 a *rightmost* derivation
- Interestingly, it turns out to be different

The Two Derivations for x - 2 * y

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

Leftmost derivation

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

Rightmost derivation

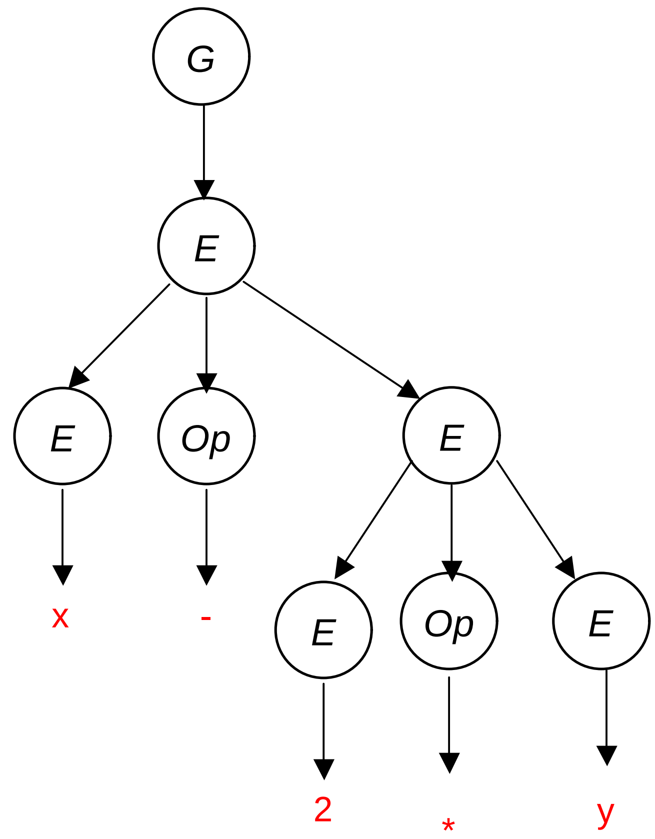
In both cases, *Expr* $\Rightarrow^*$ id - num * id

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

Derivations and Parse Trees

Leftmost derivation

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

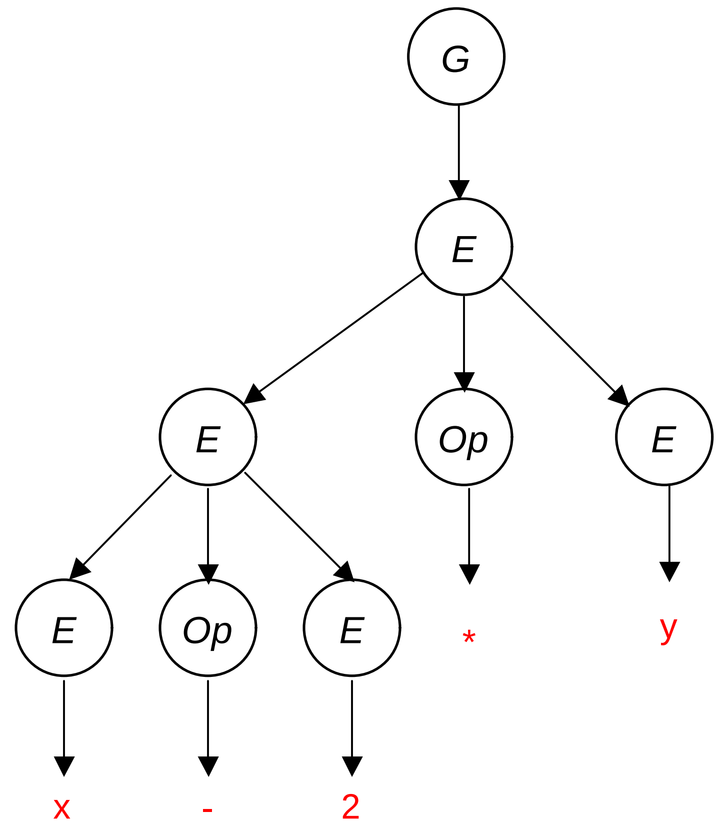


This evaluates as x - (2 * y)

Derivations and Parse Trees

Rightmost derivation

Rule	Sentential Form
—	*Expr*
2	*Expr Op Expr*
4	*Expr Op* <id, y>
7	*Expr* * <id, y>
2	*Expr Op Expr* * <id, y>
3	*Expr Op* <num,2> * <id, y>
6	*Expr* - <num,2> * <id, y>
4	<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 parser to recognize high precedence subexpressions first

For algebraic expressions

- Multiplication and division, first
- Subtraction and addition, next

Derivations and Precedence

Adding the standard algebraic precedence produces:

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

This grammar is slightly larger

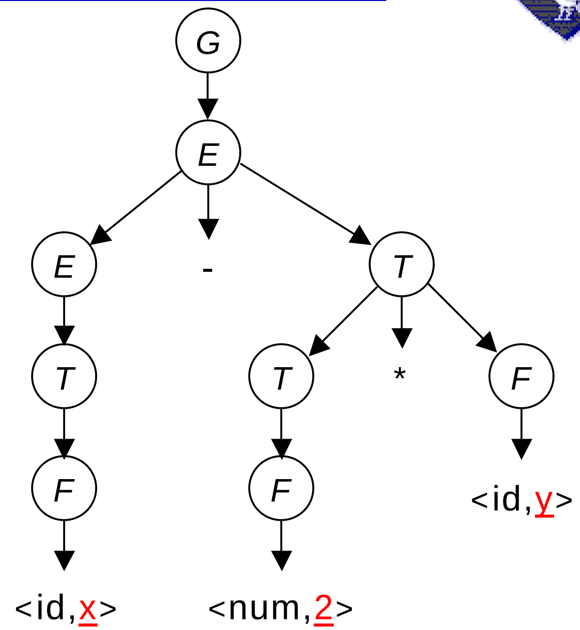
- Takes more rewriting to reach some of the terminal symbols
- Encodes expected precedence
- Produces same parse tree under leftmost & rightmost derivations

Let's see how it parses our example

Derivations and Precedence

Rule	*Sentential Form*
—	*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	*Goal*	→	*Expr*
2	*Expr*	→	*Expr Op Expr*
3		|	number
4		|	id
5	*Op*	→	+
6		|	-
7		|	*
8		|	/

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

- 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

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

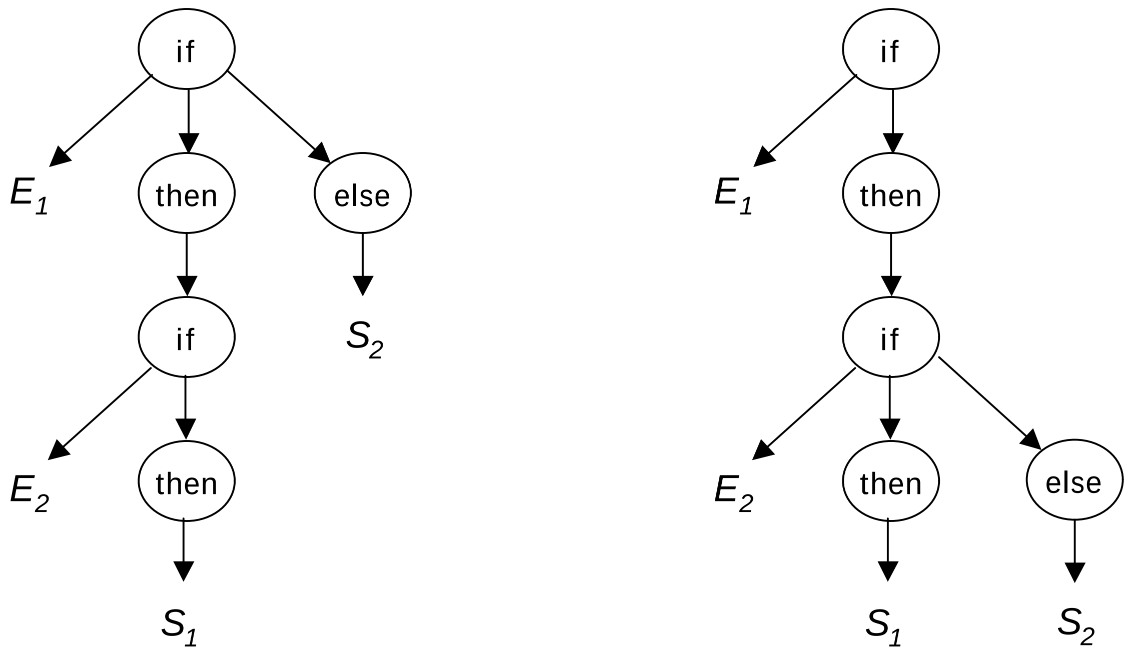
Stmt ® if *Expr* then *Stmt*
| if *Expr* then *Stmt* else *Stmt*
| *… other stmts …*

This ambiguity is entirely grammatical in nature

Ambiguity

This sentential form has two derivations

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



production 2, then production 1

production 1, then production 2