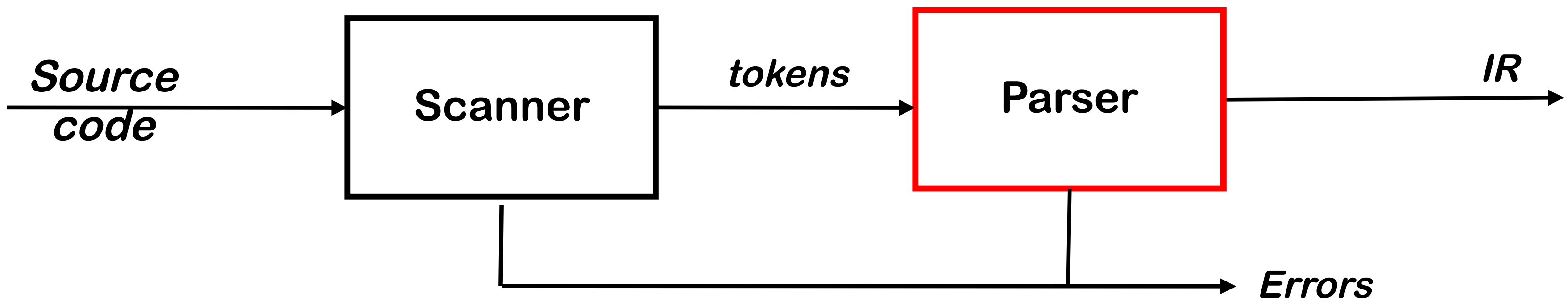
Introduction to Parsing
Part I

The Front End



Parser

- Checks the stream of words and their parts of speech (produced by the scanner) for grammatical correctness
- 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)*

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 (CFG)

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

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

Context-Free Grammar

It is written in a variant of Backus–Naur form, BNF notation

Formally, a grammar is a four tuple, *G = (S,NT,T,P)*

- *S* is the *start (or goal) symbol*
- *N* T 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*

  Production rules follow format $NT \rightarrow (NT \cup T)^{+}$

Specifying Syntax with a Grammar

$$SheepNoise \rightarrow SheepNoise\ \underline{\text{baa}}$$
$$|\ \underline{\text{baa}}$$

What are the:

S:

NT:

T:

P:

Specifying Syntax with a Grammar

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

What are the:

S: SheepNoise

NT: SheepNoise

T: baa

P: SheepNoise → SheepNoise baa

SheepNoise → baa

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

And so on ...

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		|	/

What are the NT and T?

Derivation Example

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

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

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

The Two Derivations for x – 2 * y

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

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

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

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>

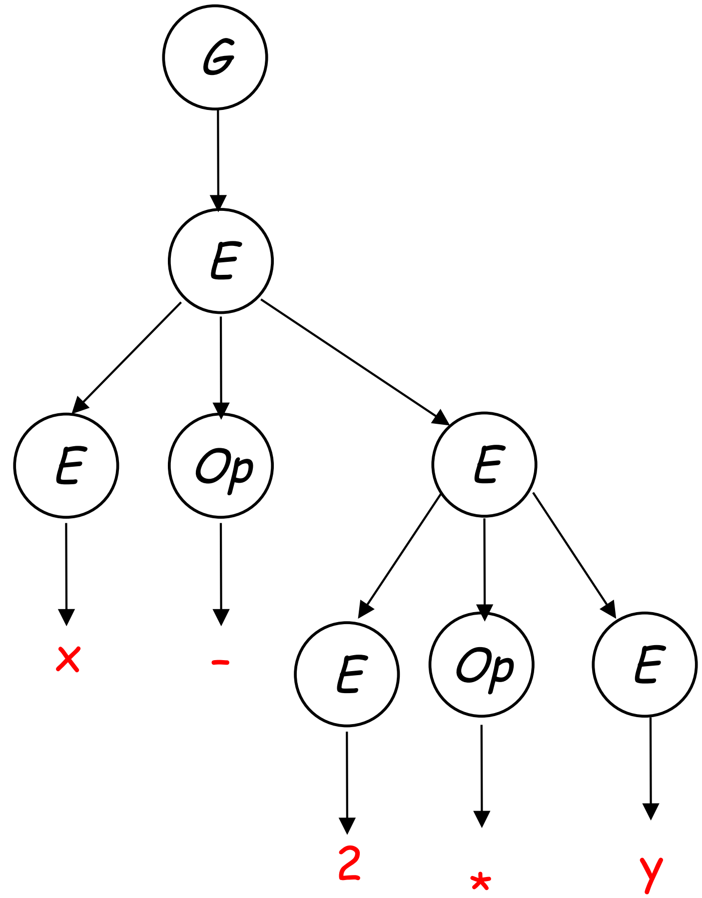
Let's do the parse tree on the board

This evaluates as x - (2 * y)

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>

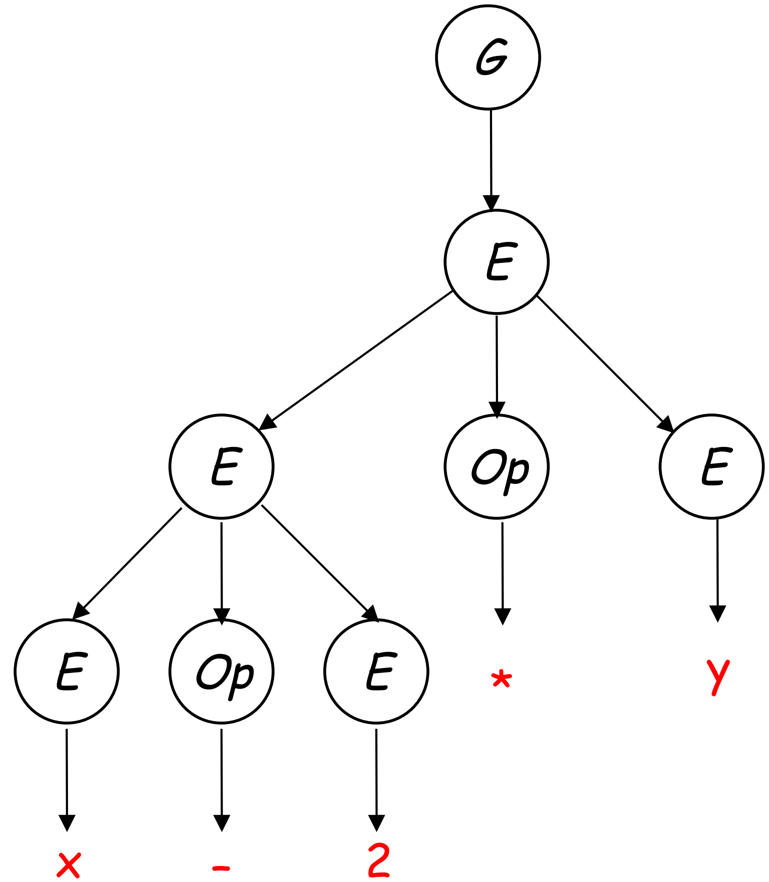
Let's do the parse tree on the board

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*)