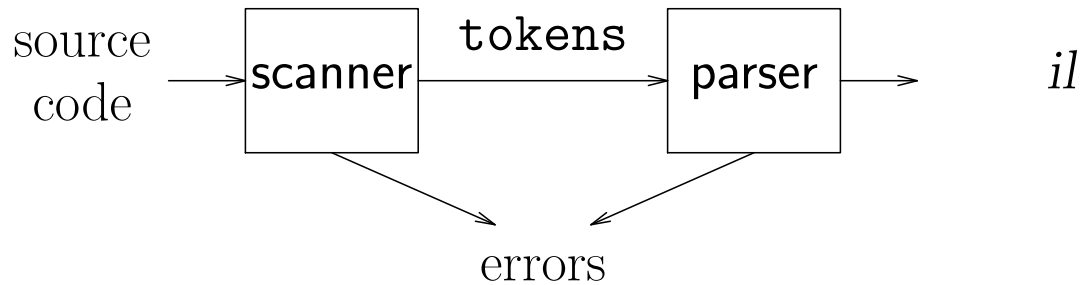


# Parsing

---



## Parser

- Checks input for grammatical correctness
- Determines syntax of token stream
- construct intermediate representation
- produce meaningful error messages

## Issues

- need mathematical model of syntax  $\Rightarrow$  grammar
- need algorithm for testing syntax  $\Rightarrow$  parsing

# Grammar

---

*Context-free syntax* is specified with a *grammar*.

$$\begin{aligned} \langle \text{sheep noise} \rangle & ::= \text{baa} \\ & \quad | \text{baa} \langle \text{sheep noise} \rangle \end{aligned}$$

The format is called *Backus-Naur form*. (BNF)

Formally, a grammar  $G = (S, N, T, P)$

$S$  is the *start symbol*

$N$  is a set of *non-terminal symbols*

$T$  is a set of *terminal symbols*

$P$  is a set of *productions* or *rewrite rules*

$$(P : N \rightarrow N \cup T)$$

## An Expression Grammar

---

*Context free syntax* can be put to better use.

P1		<goal>	::=	<expr>
P2		<expr>	::=	<expr> <op> <expr>
P3			::=	<b>number</b>
P4				<b>id</b>
P5		<op>	::=	+
P6				-

This grammar defines simple expressions with addition and subtraction over the tokens **id** and **number**.

$$S = \langle \text{goal} \rangle$$

$$T = \text{number}, \text{id}, +, -$$

$$N = \langle \text{goal} \rangle, \langle \text{expr} \rangle, \langle \text{op} \rangle$$

$$P = P1, P2, P3, P4, P5, P6$$

# Derivations

---

*We can view the productions of a grammar as rewriting rules.*

Using our example expression grammar

$$\begin{aligned} \langle \text{goal} \rangle &\Rightarrow \langle \mathbf{expr} \rangle \\ &\Rightarrow \langle \mathbf{expr} \rangle \langle \mathbf{op} \rangle \langle \text{expr} \rangle \\ &\Rightarrow \langle \mathbf{expr} \rangle \langle \mathbf{op} \rangle \langle \text{expr} \rangle \langle \mathbf{op} \rangle \langle \text{expr} \rangle \\ &\Rightarrow \langle \mathbf{id}, \mathbf{x} \rangle \langle \mathbf{op} \rangle \langle \text{expr} \rangle \langle \mathbf{op} \rangle \langle \text{expr} \rangle \\ &\Rightarrow \langle \mathbf{id}, \mathbf{x} \rangle + \langle \mathbf{expr} \rangle \langle \mathbf{op} \rangle \langle \text{expr} \rangle \\ &\Rightarrow \langle \mathbf{id}, \mathbf{x} \rangle + \langle \mathbf{num}, \mathbf{2} \rangle \langle \mathbf{op} \rangle \langle \text{expr} \rangle \\ &\Rightarrow \langle \mathbf{id}, \mathbf{x} \rangle + \langle \mathbf{num}, \mathbf{2} \rangle * \langle \mathbf{expr} \rangle \\ &\Rightarrow \langle \mathbf{id}, \mathbf{x} \rangle + \langle \mathbf{num}, \mathbf{2} \rangle * \langle \mathbf{id}, \mathbf{y} \rangle \end{aligned}$$

We have derived the string  $\mathbf{x} + \mathbf{2} * \mathbf{y}$ .

We denote this  $\langle \text{goal} \rangle \Rightarrow^* \mathbf{id} + \mathbf{num} * \mathbf{id}$ .

Such a sequence of rewrites is a *derivation* or a *parse*.

The process of discovering a derivation is called *parsing*.

# Derivations

---

*At each step, we chose a non-terminal to replace.*

*This choice can lead to different derivations.*

Two are of particular interest

*leftmost derivation*

the leftmost non-terminal is replaced at each step

*rightmost derivation*

the rightmost non-terminal is replaced at each step

*The previous example was a leftmost derivation.*

# Rightmost Derivation

---

For the string  $x + 2 * y$ :

$$\begin{aligned} \langle \text{goal} \rangle &\Rightarrow \langle \mathbf{expr} \rangle \\ &\Rightarrow \langle \text{expr} \rangle \langle \text{op} \rangle \langle \mathbf{expr} \rangle \\ &\Rightarrow \langle \text{expr} \rangle \langle \mathbf{op} \rangle \langle \text{id}, y \rangle \\ &\Rightarrow \langle \mathbf{expr} \rangle * \langle \text{id}, y \rangle \\ &\Rightarrow \langle \text{expr} \rangle \langle \text{op} \rangle \langle \mathbf{expr} \rangle * \langle \text{id}, y \rangle \\ &\Rightarrow \langle \text{expr} \rangle \langle \mathbf{op} \rangle \langle \text{num}, 2 \rangle * \langle \text{id}, y \rangle \\ &\Rightarrow \langle \mathbf{expr} \rangle + \langle \text{num}, 2 \rangle * \langle \text{id}, y \rangle \\ &\Rightarrow \langle \text{id}, x \rangle + \langle \text{num}, 2 \rangle * \langle \text{id}, y \rangle \end{aligned}$$

Again,  $\langle \text{goal} \rangle \Rightarrow^* \text{id} + \text{num} * \text{id}$ .

# Ambiguity

---

If a grammar has multiple leftmost derivations for a single sentential form, the grammar is *ambiguous*.

Similarly, a grammar with multiple rightmost derivations for a single sentential form is *ambiguous*.

**Example:** A second right-most derivation for the string  $x + 2 * y$ :

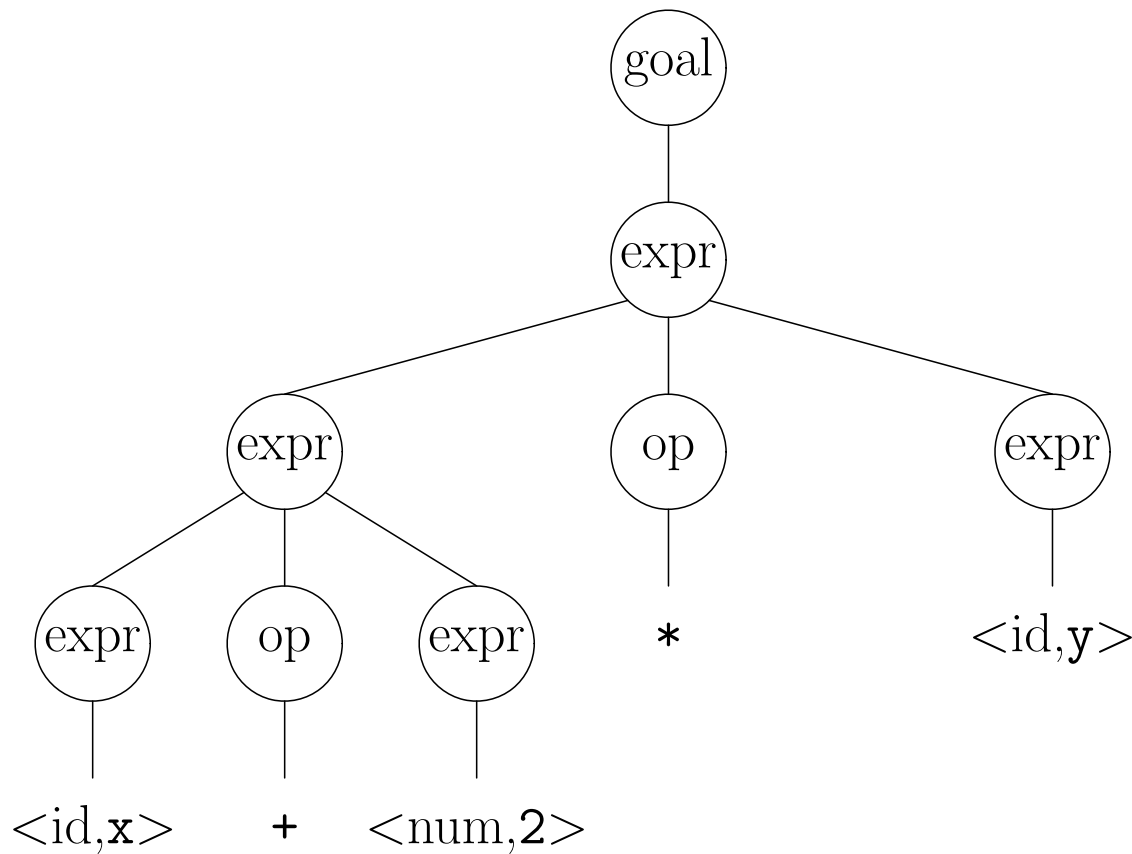
$\langle \text{goal} \rangle \Rightarrow \langle \mathbf{expr} \rangle$   
 $\Rightarrow \langle \text{expr} \rangle \langle \text{op} \rangle \langle \mathbf{expr} \rangle$   
 $\Rightarrow \langle \text{expr} \rangle \langle \text{op} \rangle \langle \text{expr} \rangle \langle \text{op} \rangle \langle \mathbf{expr} \rangle$   
 $\Rightarrow \langle \text{expr} \rangle \langle \text{op} \rangle \langle \text{expr} \rangle \langle \mathbf{op} \rangle \langle \text{id}, y \rangle$   
 $\Rightarrow \langle \text{expr} \rangle \langle \text{op} \rangle \langle \mathbf{expr} \rangle * \langle \text{id}, y \rangle$   
 $\Rightarrow \langle \text{expr} \rangle \langle \mathbf{op} \rangle \langle \text{num}, 2 \rangle * \langle \text{id}, y \rangle$   
 $\Rightarrow \langle \mathbf{expr} \rangle + \langle \text{num}, 2 \rangle * \langle \text{id}, y \rangle$   
 $\Rightarrow \langle \text{id}, x \rangle + \langle \text{num}, 2 \rangle * \langle \text{id}, y \rangle$

We may be able to eliminate ambiguities by rewriting the grammar.

## Parse tree

---

A parse can be represented by a tree, called a *parse tree* or a *syntax tree*, that represents each stage of the derivation



Each level of the tree represents one step in a derivation.

## Why use context-free grammars?

---

Many advantages:

- precise syntactic specification of a programming language
- easy to understand, avoids ad hoc definition
- easier to maintain, add new language features
- can automatically construct efficient parser
- parser construction reveals ambiguity, other difficulties
- imparts structure to language
- supports syntax-directed translation

## Grammars for regular languages

---

Can we place a restriction on the *form* of a grammar to ensure that it describes a regular language?

**Provable fact:**

For any RE  $r$ , there is a grammar  $g$  such that  $L(r) = L(g)$ .

The grammars that generate regular sets are called *regular grammars*.

**Definition:**

In a regular grammar, all productions have one of two forms:

1.  $A \rightarrow aB$
2.  $B \rightarrow a$

where  $A$ ,  $B$  are non-terminals and  $a$  is a terminal symbol.

These are also called *type 3* grammars (Chomsky).

## Scanning vs. parsing

---

*Where do we draw the line?*

$$term \rightarrow [a-zA-z] ([a-zA-z] \mid [0-9])^*$$
$$\mid 0 \mid [1-9][0-9]^*$$
$$op \rightarrow + \mid - \mid * \mid /$$
$$expr \rightarrow (term \ op)^* term$$

Regular expressions are used to classify.

- identifiers, numbers, keywords

Context-free grammars are used to count.

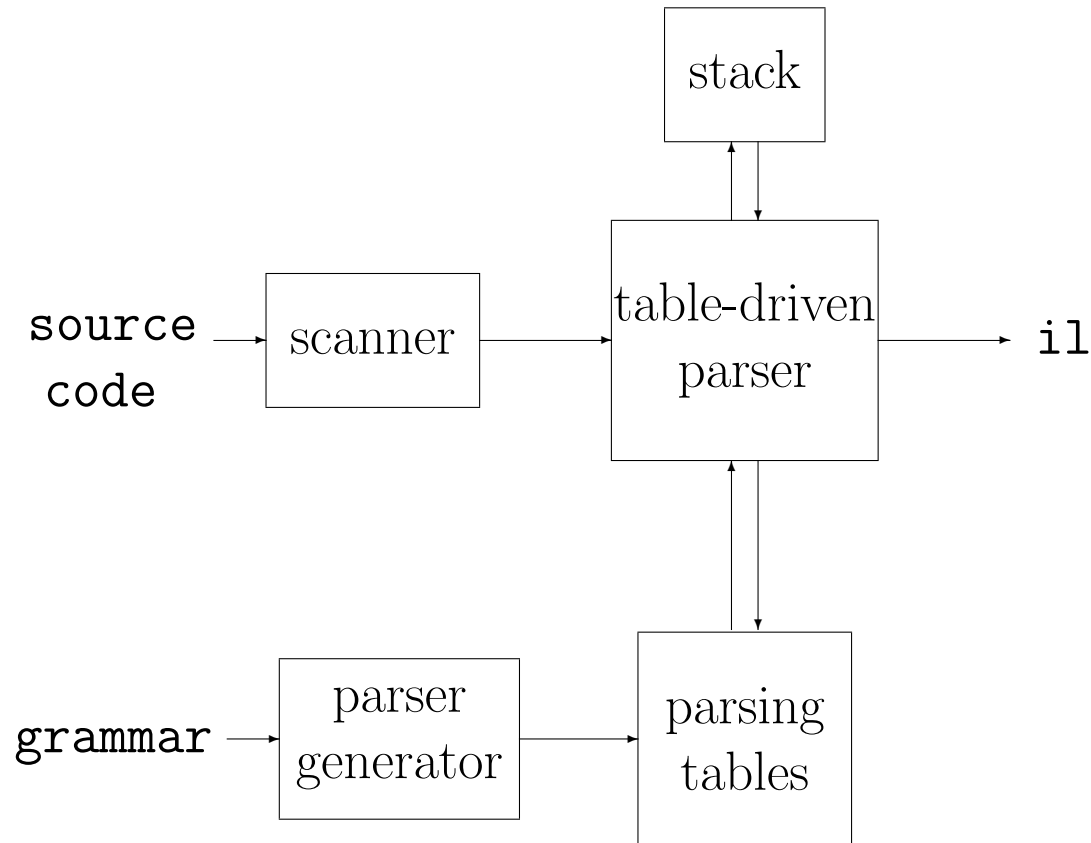
- brackets – ( ), begin – end, if – then – else
- imparting structure – expressions

Grammar for `cc` has 188 productions.

# Parser Construction

---

For many grammars, we can generate table-driven parsers which use a table (DFA) and *stack*



Parsers can also perform *actions* during each reduction to

- collect information for type checking, generate intermediate code