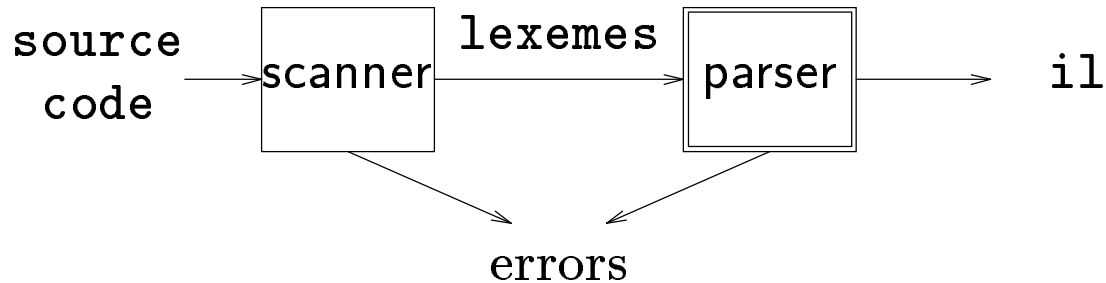


The role of the parser



Parser

- perform context-free syntax analysis
- guide the context-sensitive analysis
- construct an intermediate representation
- produce meaningful error messages
- attempt error correction

For the next three weeks, we will look at parser construction.

Syntax analysis

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

Formally, a context-free grammar G is a four-tuple (T, NT, S, P)

T is the set of terminal symbols in the grammar.

For our purposes, the set of terminals is equivalent to the set of tokens returned by the lexical analyzer.

NT is a set of syntactic variables that denote sets of (sub)strings occurring in the language. These are used to impose a structure on the grammar.

S is a distinguished nonterminal ($S \in NT$) that denotes the entire set of strings in $L(G)$. This is sometimes called a *goal symbol*. S cannot appear on the right hand side of some production.

P is a set of productions that specify the way that terminals and non-terminals can be combined to form strings in the language. Each production must have a single non-terminal on its left hand side.

Syntax analysis

Grammars are often written in BNF, or Backus-Naur form.

```
1 | <goal> ::= <expr>
2 | <expr> ::= <expr> <op> <expr>
3 |           | number
4 |           | id
5 | <op> ::= +
6 |           | -
7 |           | *
8 |           | /
```

This grammar gives simple expressions over numbers and identifiers.

In a BNF for a grammar, we represent

- a) non-terminals with brackets or capital letters,
- b) terminals with `typewriter` font or underline,
- c) productions as in the example.

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. $A \rightarrow aA$
2. $A \rightarrow a$

where A is a non-terminal and a is a terminal symbol.

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

Scanning vs. parsing

Where do we draw the line?

$$\begin{aligned} \textit{term} &\rightarrow [\text{a-zA-z}] ([\text{a-zA-z}] \mid [0-9])^* \\ &\mid 0 \mid [1-9][0-9]^* \end{aligned}$$
$$\textit{op} \rightarrow + \mid - \mid * \mid /$$
$$\textit{expr} \rightarrow (\textit{term op})^* \textit{term}$$

Regular expressions are used to classify.

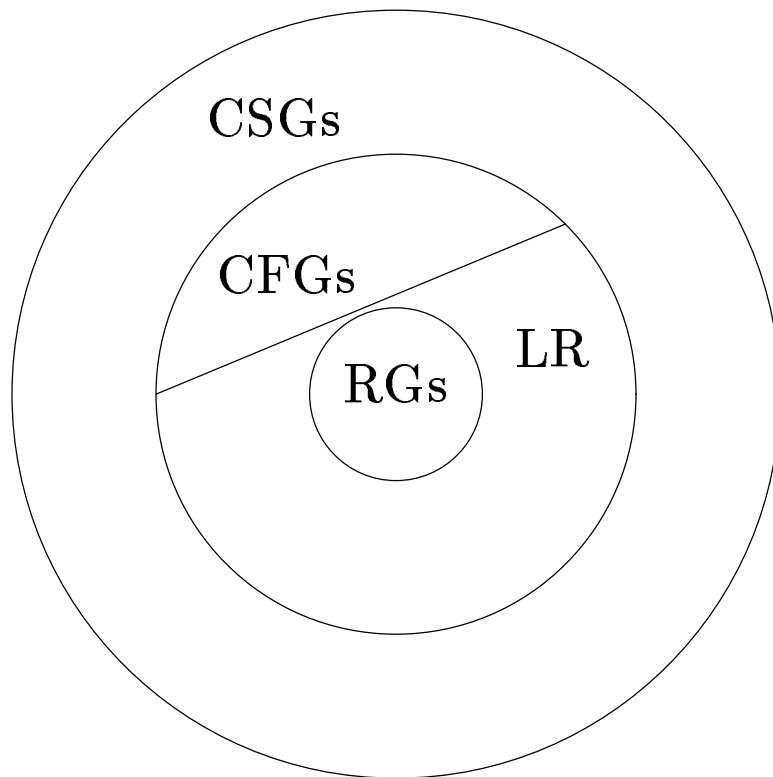
- identifiers, numbers, keywords

Context-free grammars are used to count.

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

Grammar for cc has 188 productions.

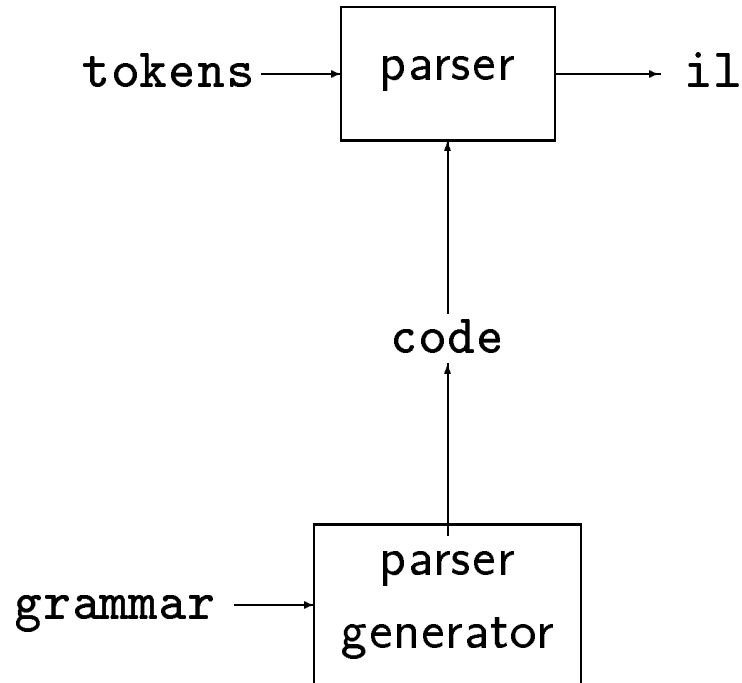
Complexity of parsing



Complexity:

Regular grammars	<i>dfas</i>	$O(n)$
LR grammars	Knuth's algorithm	$O(n)$
Arbitrary CFGs	Early's algorithm	$O(n^3)$
Arbitrary CSGs	<i>ibas</i>	<i>P-SPACE</i> <i>COMPLETE</i>

Parsing — the big picture



Our goal is a flexible parser generator system.

Derivations

We can view the productions of a cfg as rewriting rules.

Using our example

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

We have derived the sentence $x + 2 * y$.

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

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 example was a leftmost derivation.

Rightmost Derivation

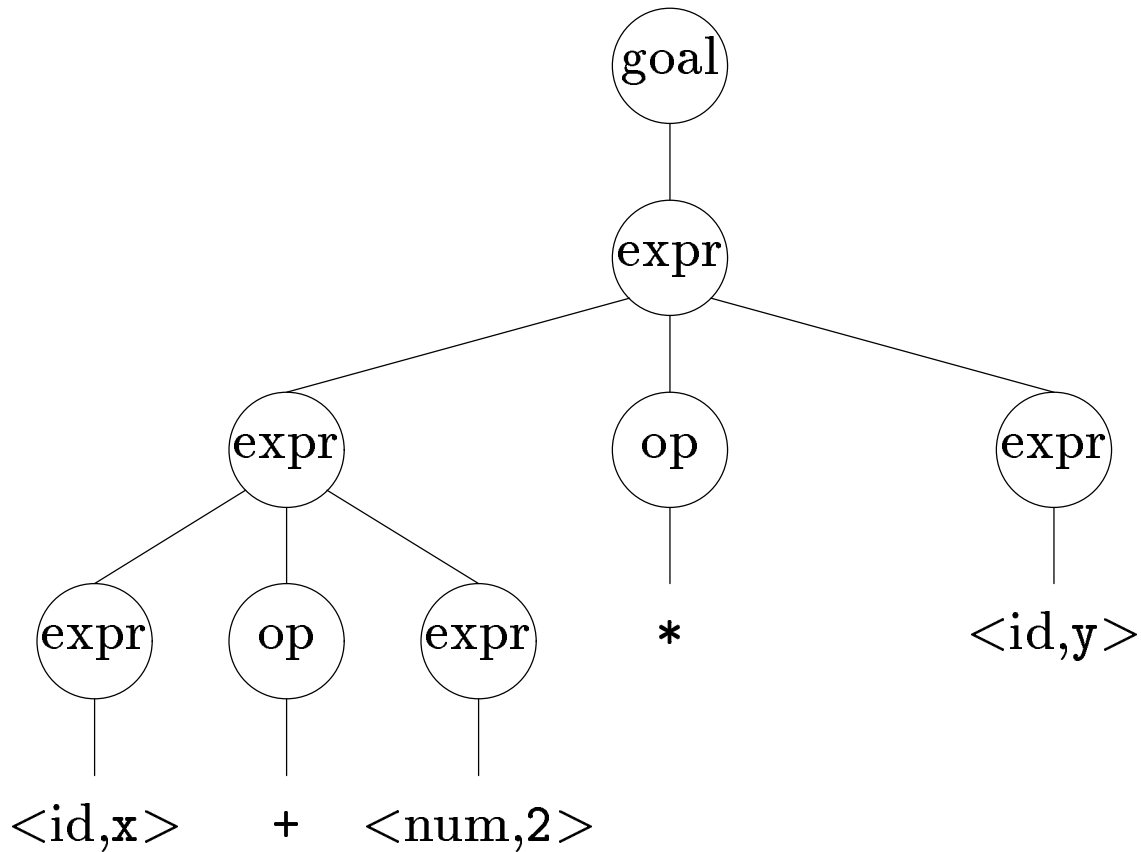
For the string $x + 2 * y$:

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

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

Precedence

Let's look at the parse tree.



Treewalk evaluation would give the “wrong” answer.

*(x + 2) * y instead of x + (2 * y)*

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 takes additional machinery

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 enforces a precedence on the derivation

- terms *must* be derived from expressions
- forces the “correct” tree

Precedence

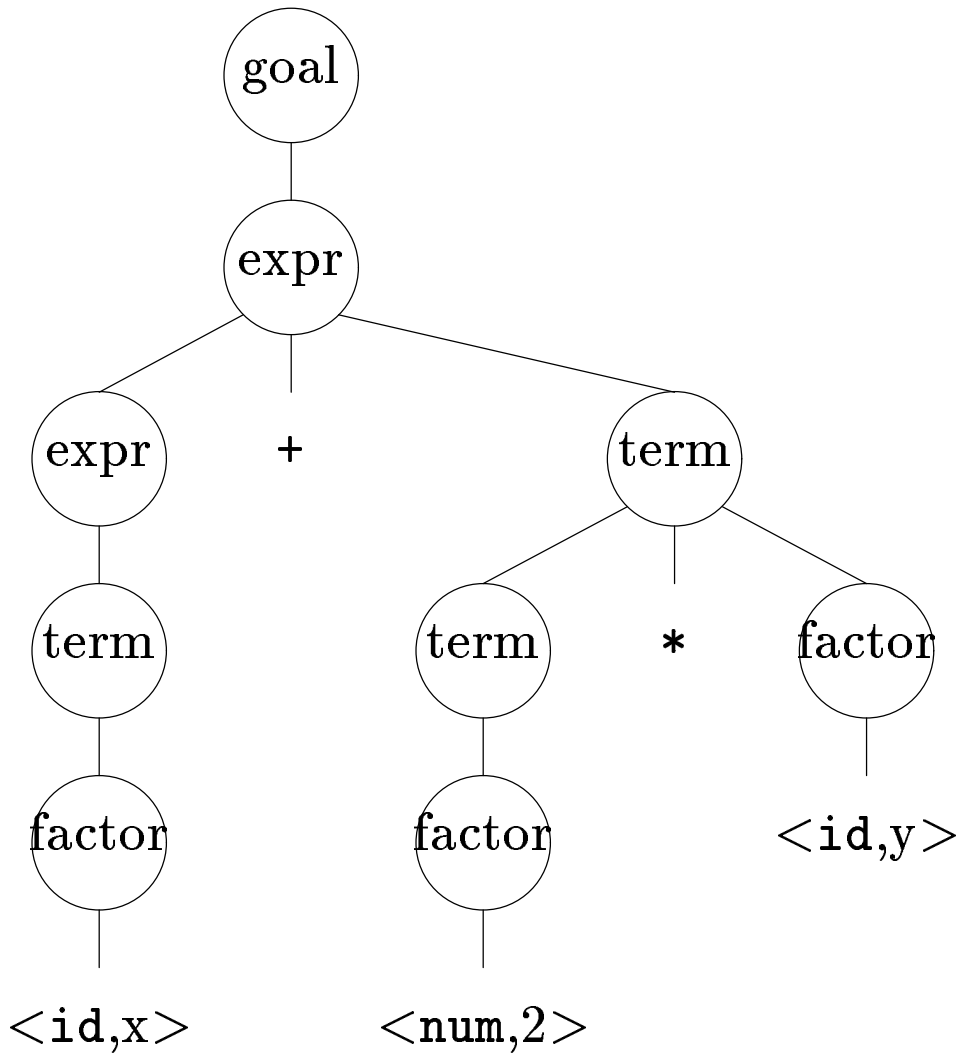
Now, for the string $x + 2 * y$:

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

Again, $\langle \text{goal} \rangle \Rightarrow^* \text{id} + \text{num} * \text{id}$, but this time, we build the desired tree.

Precedence

This time, we get the desired parse tree.



*Treewalk evaluation computes $x + (2 * y)$.*

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

```
<stmt> ::= if <expr> then <stmt>
          | if <expr> then <stmt> else <stmt>
          | other stmts
```

Consider deriving the sentential form:

if E_1 then if E_2 then S_1 else S_2

It has two derivations.

This ambiguity is purely grammatical.

It is a *context-free* ambiguity.

Ambiguity

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

```
<stmt> ::= <ms>
         | <us>
<ms>   ::= if <expr> then <ms> else <ms>
         | other stmts
<us>   ::= if <expr> then <stmt>
         | if <expr> then <ms> else <us>
```

This grammar generates the same language as the ambiguous grammar, but applies the common sense rule

match each else with the closest unmatched then

This is pretty clearly the language designer's intent.

Ambiguity

Ambiguity generally refers to a confusion in the context-free specification.

Context-sensitive confusions can arise from *overloading*.

$a = f(17)$

In many Algol-like languages, f could be either a function or a subscripted variable.

Disambiguating this statement requires context.

- need *values* of declarations
- not *context free*
- really an issue of *type*

Rather than complicate parsing, we will handle this separately.