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. $A \rightarrow aB$
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?

$$\text{term} \rightarrow [\text{a-zA-Z}]([\text{a-zA-Z}] | [0\text{-}9])^*$$
$$| \; 0 \; | \; [1\text{-}9][0\text{-}9]^*$$
$$\text{op} \rightarrow + \mid - \mid * \mid /$$
$$\text{expr} \rightarrow (\text{term op})^* \text{ term}$$

Regular expressions are used to classify

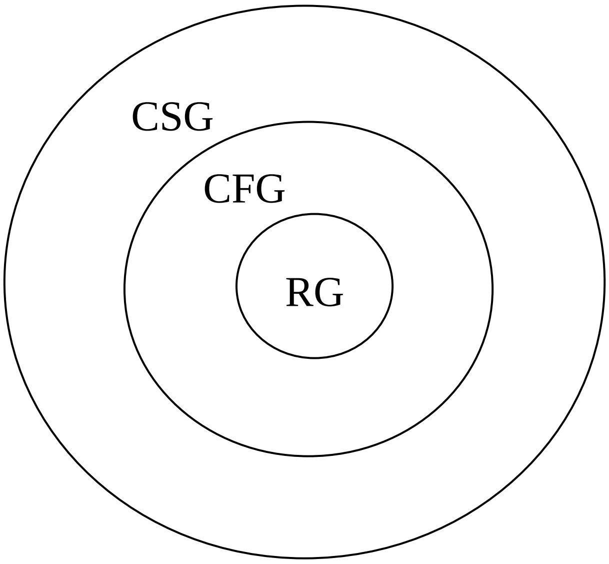
- Identifiers, numbers, keywords

Context-free grammars are used to count

- Brackets – (), begin – end, if-then-else
- Imparting structure –expression

Grammars for cc has 188 productions.

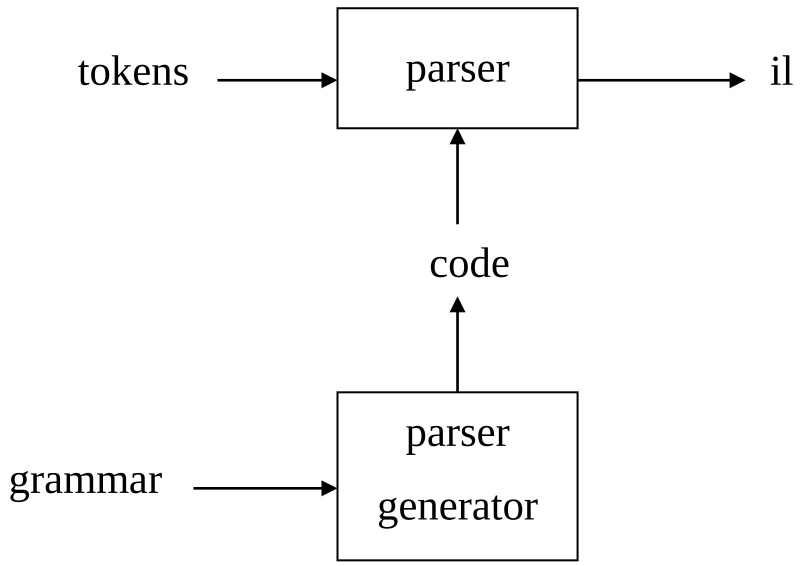
Complexity of parsing



Complexity:

Regular grammars	dfas	O(n)
Arbitrary CFGs	Early’s algorithm	$O(n^3)$
Arbitrary CSGs	lbas	P-SPACE COMPLETE

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

<goal> => <expr>
=> <expr> <op> <expr>
=> <expr> <op> <expr> <op> <expr>
=> <id,x> <op> <expr> <op> <expr>
=> <id,x> + <expr> <op> <expr>
=> <id,x> + <num, 2> <op> <expr>
=> <id,x> + <num, 2> * <expr>
=> <id,x> + <num, 2> * <id, y>

We have derived the sentence x + 2 * y.
We denote this <goal> $\Rightarrow^*$ id + num * 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 right ost non-terminal is replaced at each step

The example was a leftmost derivation.

Rightmost derivation

For the string

<goal> => <expr>

=> <expr> <op> <expr>

=> <expr> <op> <id, y>

=> <expr> * <id, y>

=> <expr> <op> <expr> * <id, y>

=> <expr> <op> <num, 2> * <id, y>

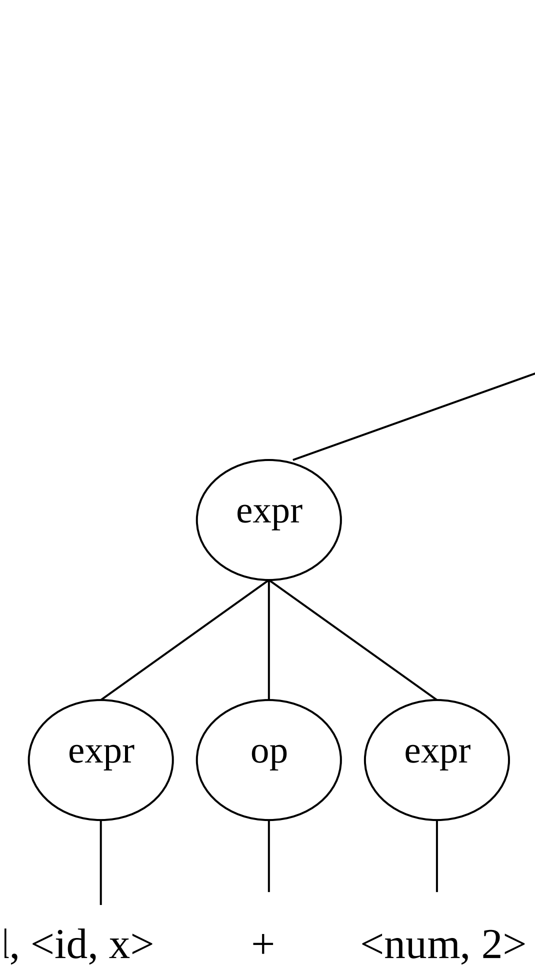
=> <expr> + <num, 2> * <id, y>

=> <id,x> + <num, 2> * <id, y>

Again, <goal> =>* id + num * 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

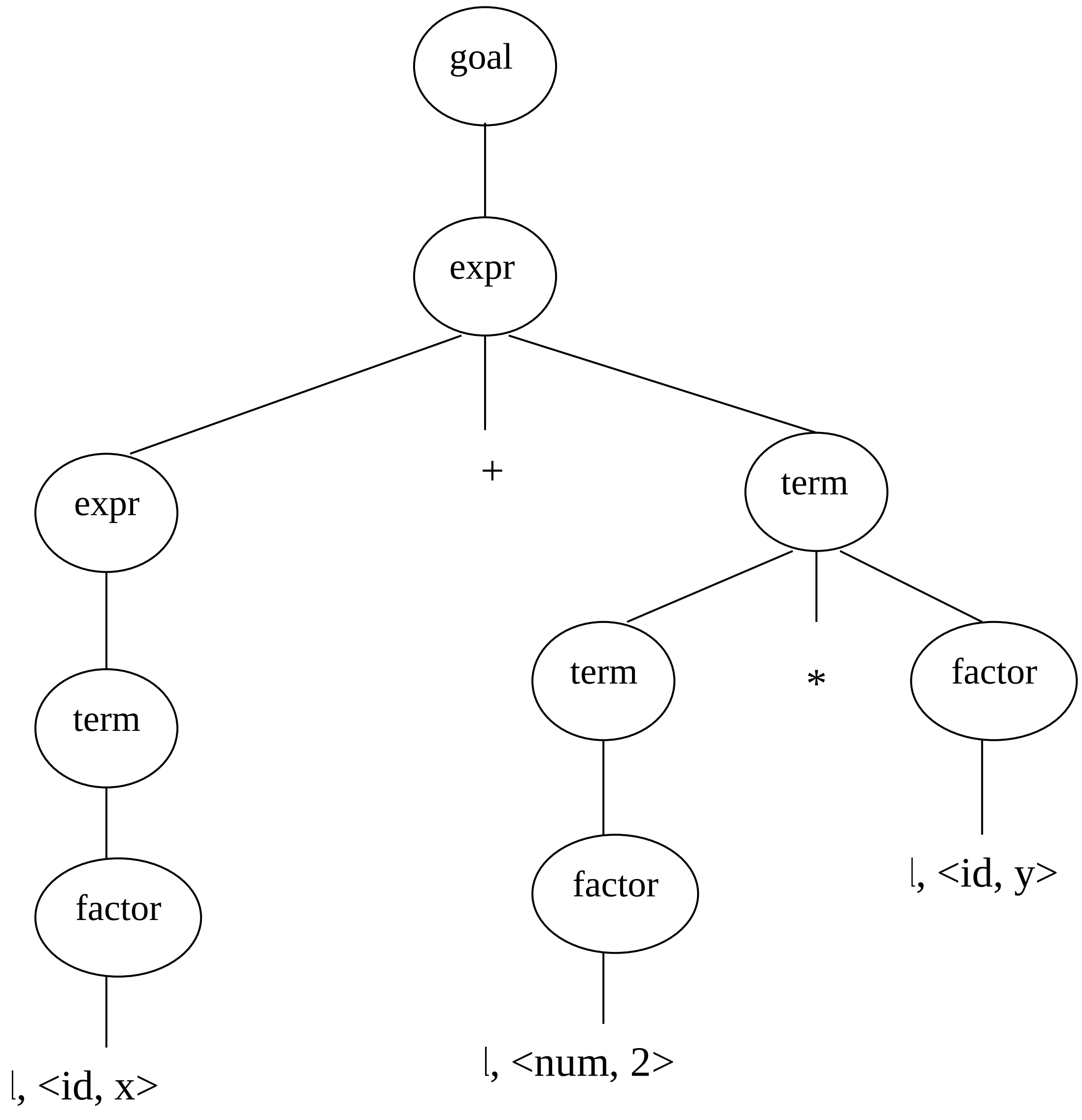
Now, for the string x + 2 * y:

<goal> => <expr>
    => <expr> + <term>
    => <expr> + <term> * <factor>
    => <expr> + <term> * < id, y >
    => <expr> + <factor> * < id, y >
    => <expr> + <num, 2> * <id, y>
    => <term> + <num, 2> * <id, y>
    => <factor> + <num, 2> * <id, y>
    => <id,x> + <num, 2> * <id, y>

Again, <goal> =>* id + num * 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 can 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.