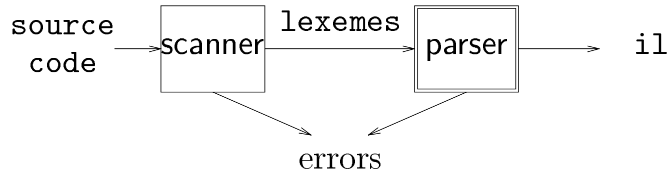
Parsing



Parser

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

Issues

- need mathematical model of syntax ⇒ grammar
- need algorithm for testing syntax ⇒ parsing

Grammars

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

Example

1	SheepNoise	::=	SheepNoise `baa`
2		|	`baa`

This context-free grammar (CFG) defines the set of noises sheep normally make.

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

- S is the start symbol
- N is a set of non-terminal symbols
- T is a set of terminal symbols (tokens)
- P is a set of productions (rewrite rules)

Syntax analysis

Grammars are often written in Backus-Naur Form (BNF)

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.

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`.

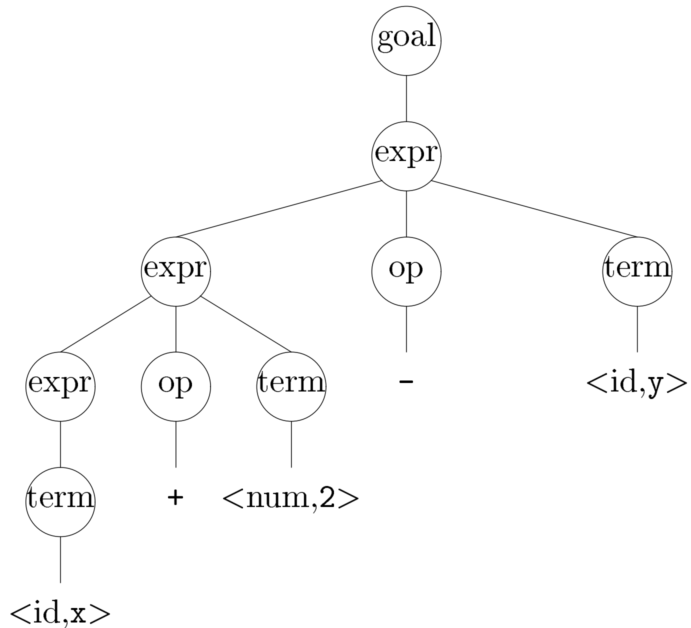
We denote this <goal> $\Rightarrow^*$ `id + num * id`.

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

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

Parse tree

A parse can be represented by a tree, called a *parse tree* or a *syntax tree.*



The parse tree represents each stage of the derivation

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

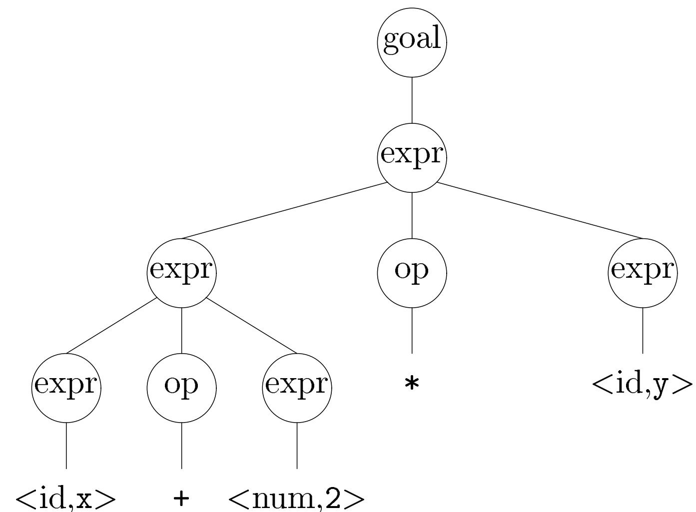
For the string `x + 2 * y`:

<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> $\Rightarrow^*$ `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

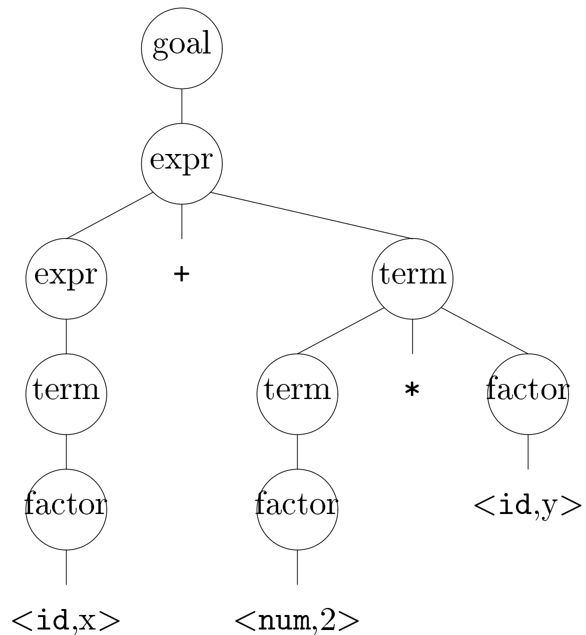
Now, for the string `x + 2 * y`:

<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> $\Rightarrow^*$ `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` 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.