

Syntax and Semantics

Syntax

For a programming language it is the form of its expressions, statements and program units.

e.g. : **if** (*< expr >*) *< statement >*

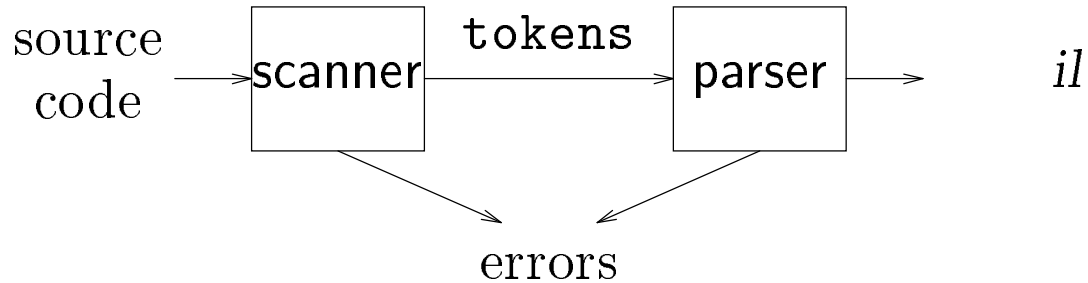
Semantics

For a programming language it is the meaning of its expressions, statements and program units.

Meaning of the example statement is that if the current values of the expression is true, the embedded statement is selected for execution.

Syntax and semantics are closely related.

Front end

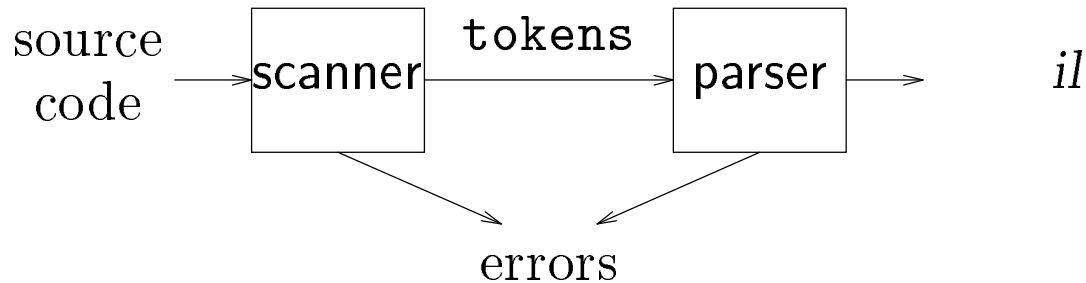


Responsibilities:

- recognize legal procedure
- report errors
- produce *il*
- preliminary storage map
- shape the code for the back end

Much of front end construction can be automated

Scanner



Scanner

- maps characters into *tokens* – the basic unit of syntax

`x = x + y;`

becomes

`<id, x> = <id, x> + <id, y> ;`

- character string for a *token* is a *lexeme*
- typical tokens: *number, id, +, -, *, /, do, end*
- eliminates white space (*tabs, blanks, comments*)
- a key issue is speed

Regular Expressions

Patterns are often specified as *regular languages*.
Notations used to describe a regular language (or a regular set) include both *regular expressions* and *regular grammars*.

identifier

$letter \rightarrow (a \mid b \mid c \mid \dots \mid z \mid A \mid B \mid C \mid \dots \mid Z)$

$digit \rightarrow (0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9)$

$id \rightarrow letter (letter \mid digit)^*$

numbers

$integer \rightarrow (+ \mid - \mid \epsilon) (0 \mid (1 \mid 2 \mid 3 \mid \dots \mid 9) (digit)^*)$

$decimal \rightarrow integer . (digit)^*$

$real \rightarrow (integer \mid decimal) E (+ \mid -) (digit)^*$

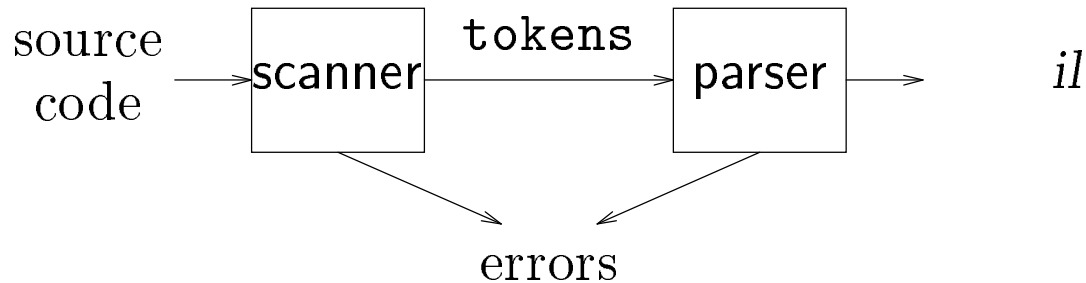
$complex \rightarrow "(real \text{ , } real)"$

Numbers can get much more complicated

Most programming language tokens can be described with regular expressions.

We can use regular expressions to automatically build scanners.

Parser



Parser:

- recognize context-free syntax
- guide context-sensitive analysis
- construct *il(s)*
- produce meaningful error messages
- attempt error correction

Parser generators mechanize much of the work

Grammar

Context-free syntax is specified with a *grammar*. It is called a *context-free grammar*.

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

This grammar defines the set of noises that a sheep makes under normal circumstances.

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*

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

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.

Extended BNF notation : Adds some convenience notation.

- Optional element identified by, [..]
- Choice of alternatives identified by, (a|b)
- 0 or more times identified by, {..}*

Derivations

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

Using an example

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

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

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

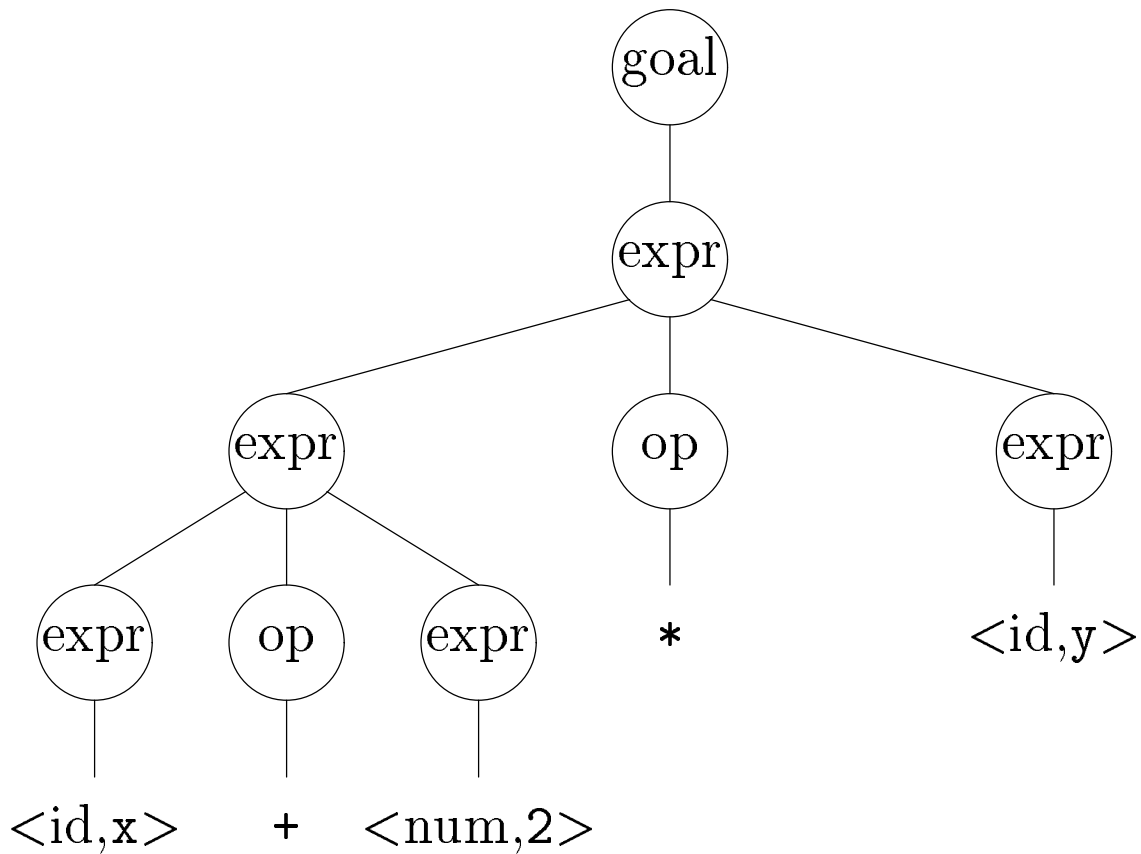
Such a sequence of rewrites is a derivation or a parse.

The process of discovering a derivation is called parsing.

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

Precedence

Let's look at the parse tree.



Treewalk evaluation would give the “wrong” answer.

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

Precedence

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

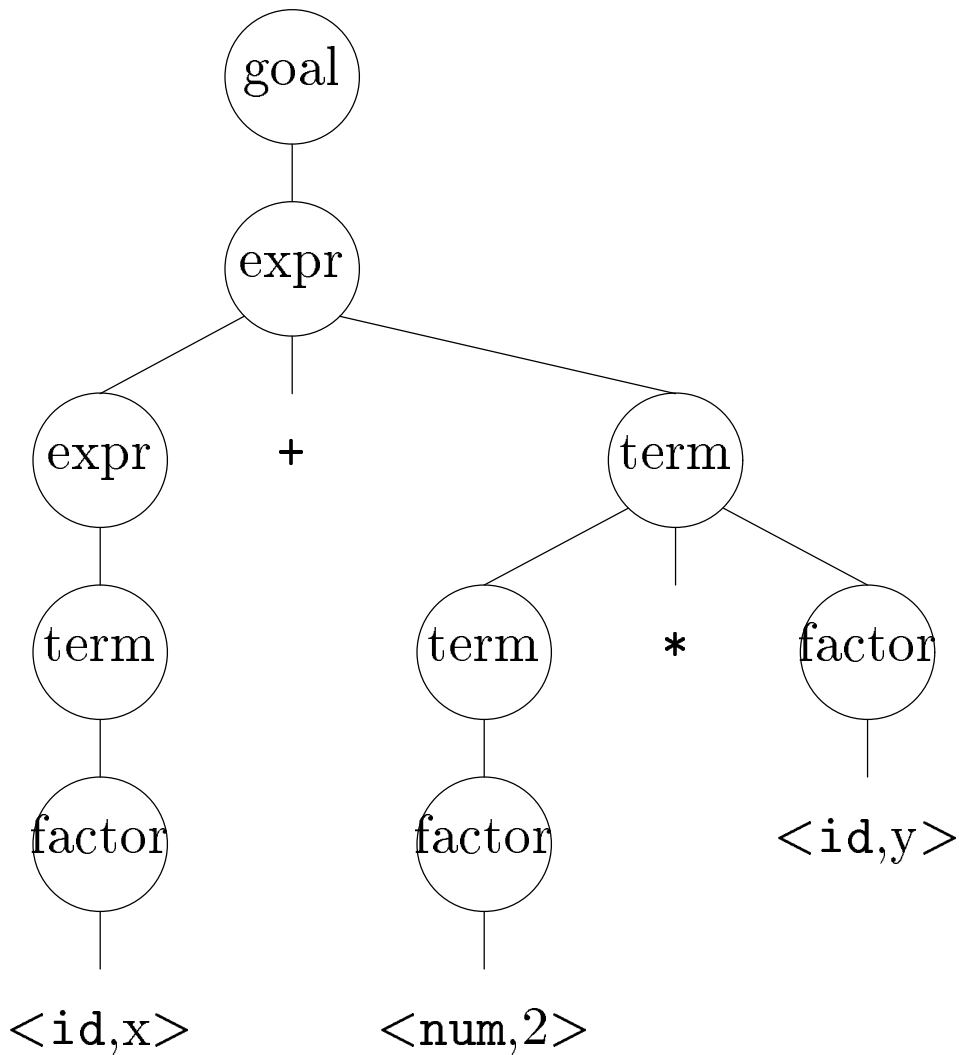
Now, for the string $x + 2 * y$:

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

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 derivations for a single sentential form, the grammar 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*

Static Semantics

Some things are difficult or impossible to specify in BNF

Example problems :

- Difficult : $\langle int \rangle := \langle float \rangle$ is illegal but opposite is legal.
- Impossible : All variables must be declared before they're used.

Static semantics of a language does not deal with the meaning of the program; it deals with legal forms of the programming language. It cannot be described with BNF.

We can use attribute grammars to describe static semantics of a language

Attribute grammars

Attribute grammar

- generalization of context-free grammar
- each grammar symbol has an associated set of attributes
- augment grammar with rules that define values
- high-level specification, independent of evaluation scheme

Dependences between attributes

- values are computed from constants & other attributes
- *synthesized attribute* – value computed from children
- *inherited attribute* – value computed from siblings & parent

Circularity is a serious problem

Attribute grammars

Synthesized attributes

derives its value from constants and children

- only synthesized attributes \Rightarrow S-attributed grammar
- S-attributed grammars can be evaluated in one bottom-up pass

S-attributed grammar is good match to LR parsing

Inherited attributes

derives its value from constants, siblings, and parent

- used to express context (*context-sensitive checking*)
- can *always* rewrite to avoid inherited attributes
- inherited attribute rules are “more natural”

Mechanical translation of inherited attributes is more problematic

We want to use both kinds of attribute

Attribute grammars

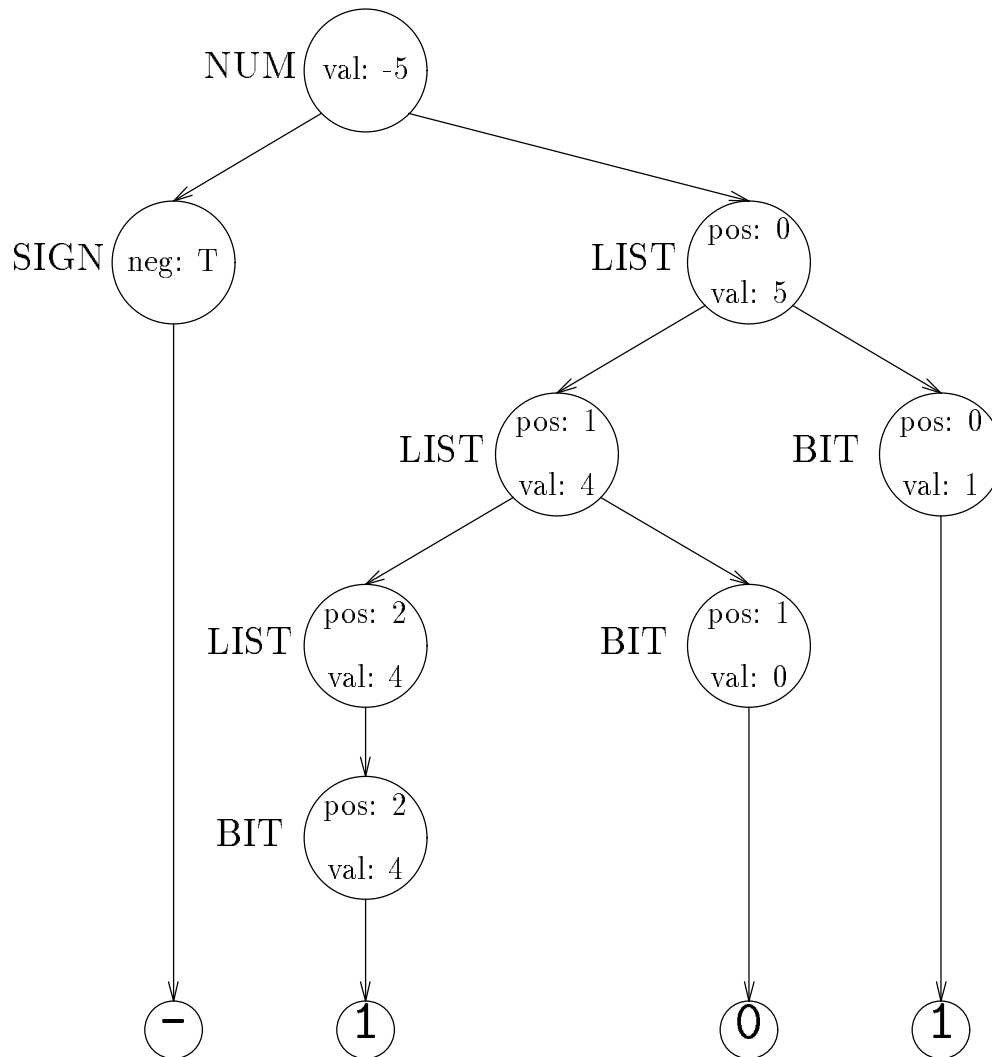
Example

A grammar to evaluate signed binary numbers

<i>Production</i>	<i>Evaluation Rules</i>
1 NUM ::= SIGN LIST	LIST.pos \leftarrow 0 NUM.val \leftarrow if SIGN.neg then -LIST.val else LIST.val
2 SIGN ::= +	SIGN.neg \leftarrow false
3 SIGN ::= -	SIGN.neg \leftarrow true
4 LIST ::= BIT	BIT.pos \leftarrow LIST.pos LIST.val \leftarrow BIT.val
5 LIST ₀ ::= LIST ₁ BIT	LIST ₁ .pos \leftarrow LIST ₀ .pos + 1 BIT.pos \leftarrow LIST ₀ .pos LIST ₀ .val \leftarrow LIST ₁ .val + BIT.val
6 BIT ::= 0	BIT.val \leftarrow 0
7 BIT ::= 1	BIT.val \leftarrow 2 ^{BIT.pos}

Attribute grammars

Example



- val and neg are *synthesized* attributes
- pos is an *inherited* attribute