Language description methods

Contents

Syntax and semantics

Points

- Form and meaning of programming languages
- Types of processing
- Types of languages

Syntax of a language determines how programs are built from elementary units (keywords, identifiers, numbers, brackets, and so on). A syntactically correct program may still not be acceptable, or it may work in a way that we do not want (or do not expect).

Formal syntax is a system for describing exactly the structure of program. Such systems include grammars, BNF, syntactic diagrams (syntax graphs).

A **grammar** is a finite description of an infinite language.

There are infinitely many different programs, but every program is finite and must be recognized in finite time.

Semantics of a language determines the meaning of elementary units and their combinations: how the meaning of a program derives from the meaning of its components.

Semantic description methods

- **Operational semantics**

  Simple lower-level operations explain how higher-level statements are performed.
- **Denotational semantics**

  A program computes a function, a mapping

  data ⟶ results
- **Axiomatic semantics**

  A program establishes a relation

  data ⟷ results

Formal semantics: experimental description languages, extended grammars exist, but are not in much use (too difficult for most users?).

Lexical analysis is the pre-processing of a file with a program: recognize units larger than single characters (keywords, identifiers, numbers, brackets, and so on!). This makes translators usefully modularized.

Syntactic analysis, based on grammars, means either recognition (program correct/not correct) or parsing (a representation of the syntactic structure is built for a correct program).

It is the essential part of any implementation of a programming language.

Syntactic generation, also based on grammars, is the flip side of analysis: it runs from a syntactic structure to a program. Generation is important in language technology, though not much in programming languages.

A **language** can be defined formally as a set of sentences which are sequences of elementary pieces, built according to certain rules. In a programming language, a complete program is such a "sentence".

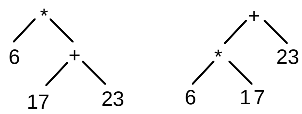
A hierarchy of languages:

regular <
context-free <
context-sensitive <
recursively enumerable.

Programming languages usually have context-free syntax and context-sensitive semantics.

Context-freeness (only during syntactic analysis) means that a fragment does not depend on other fragments, e.g. an occurrence of a variable is not related to its declaration .

« ‡ ◊ ¤ Δ ¤ ◊ ‡ »



Grammars

Points

A formal grammar has four components.

- Terminal symbols = language elements (e.g. Pascal symbols or English words).
- Non-terminal symbols = auxiliary symbols, denoting classes of constructions (e.g. loop_statement, Boolean_expression).
- The goal (start) symbol denotes any sentence.
- Productions = rewriting rules ("this structure has such and such components"), used to recognize or generate sentences.

Two ways of rewriting:

- ° from the start symbol, **produce** more and more specific approximations of a sentence, replacing non-terminals with their definitions;
- ° **reduce** the sentence into more and more general forms, replacing definitions with non-terminals, and reach the goal symbol (the same!).

- Productions are what makes a grammar context-free, context-sensitive and so on.

☒ Example: a grammar of expressions

- terminal symbols: + - * ( ) x y
- non-terminal symbols: ‹expr› ‹term› ‹factor› ‹var›

Notation: angle bracket distinguish non-terminals from terminals.

- start/goal symbol: ‹expr›
- productions:

Notation: "LHS → RHS" means "the Left-Hand Side consists of things on the Right-Hand Side".

The bar | separates alternative Right-Hand Sides with the same Left-Hand Side.

‹expr› → ‹term› |
‹expr› + ‹term› |
‹expr› - ‹term›
‹term› → ‹factor› |
‹term› * ‹factor›
‹factor› → ‹var› | (‹expr›)
‹var› → x | y

For any sentence σ, productions can be applied in two directions.

(1) Top-down: derive σ from the start symbol. σ will then be an example of expression.

(2) Bottom-up: fold σ into the initial symbol.

Let us take the sequence of terminal symbols

(x - y) * x + y

(it **is** an expression, but we must first show it is).

Consider two derivations. On each line, the underscored part is involved in rewriting the previous line according to some grammar production.

A derivation top-down:
‹expr› ⇒
‹expr› + ‹term› ⇒
‹term› + ‹term› ⇒
‹term› * ‹factor› + ‹term› ⇒
‹factor› * ‹factor› + ‹term› ⇒
(‹expr›) * ‹factor› + ‹term› ⇒
(‹expr› - ‹term›) * ‹factor› + ‹term› ⇒
(‹term› - ‹term›) * ‹factor› + ‹term› ⇒
(‹factor› - ‹term›) * ‹factor› + ‹term› ⇒
(‹var› - ‹term›) * ‹factor› + ‹term› ⇒
(x - ‹term›) * ‹factor› + ‹term› ⇒
(x - ‹factor›) * ‹factor› + ‹term› ⇒
(x - ‹var›) * ‹factor› + ‹term› ⇒
(x - y) * ‹factor› + ‹term› ⇒
(x - y) * ‹var› + ‹term› ⇒
(x - y) * x + ‹term› ⇒
(x - y) * x + ‹factor› ⇒
(x - y) * x + ‹var› ⇒
(x - y) * x + y

A derivation bottom-up:
(x - y) * x + y ⇒
(‹var› - y) * x + y ⇒
(‹factor› - y) * x + y ⇒
(‹term› - y) * x + y ⇒
(‹expr› - y) * x + y ⇒
(‹expr› - ‹var›) * x + y ⇒
(‹expr› - ‹factor›) * x + y ⇒
(‹expr› - ‹term›) * x + y ⇒
(‹expr›) * x + y ⇒
‹factor› * x + y ⇒
‹term› * x + y ⇒
‹term› * ‹var› + y ⇒
‹term› * ‹factor› + y ⇒
‹term› + y ⇒
‹expr› + y ⇒
‹expr› + ‹var› ⇒
‹expr› + ‹factor› ⇒
‹expr› + ‹term› ⇒
‹expr›

In both derivations guessing is required: which production should be applied now?

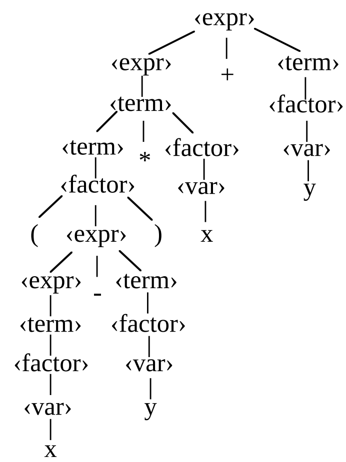
Strategies of choice are at the heart of <u>parsing algorithms</u>. Ideally, we would always guess correctly. Less ideally, we may have to try a production, fail, and return to try another.

Both processes recognize the given sequence of symbols

(x - y) * x + y

as an expression that is <u>well-formed</u> according to our grammar.

The results of both derivations can be both summarized in the same tree (<u>abstract syntax tree</u>, <u>parse tree</u>):



Note that the order in which productions have been applied during derivations is not shown in this tree.

Extended BNF (EBNF) is not more expressive power, but allows shorter grammars.

(1) <u>Optional elements.</u>
Put an optional fragment in square brackets.

☒ For example, we can rewrite this:

- ‹conditional_statement› →
  if ‹condition› **then** ‹statement›
    end if |
  if ‹condition› **then** ‹statement›
    else ‹statement› **end if**

into the following shorter (equivalent!) form:

- ‹conditional_statement› →
  if ‹condition› **then** ‹statement›
    [**else** ‹statement›] **end if**

(2) <u>Repetition</u>: put in curly brackets elements repeated zero or more times.

☒ For example:

- ‹digits› → ‹digit› | ‹digit› ‹digits›

may be equivalently written as

‹digits› → ‹digit› { ‹digit› }

Using the recursive production, a derivation for the integer 197 could be this:

‹digits› ⇒ ‹digit› ‹digits› ⇒ ‹digit› ‹digit› ‹digits› ⇒
‹digit› ‹digit› ‹digit› ⇒
1 ‹digit› ‹digit› ⇒ 1 9 ‹digit› ⇒ 1 9 7

Using repetitions, we can “guess” <u>one</u> of the possible extended forms:

‹digits› ⇒ ‹digit›
‹digits› ⇒ ‹digit› ‹digit›
‹digits› ⇒ ‹digit› ‹digit› ‹digit›
......

and shorten the derivation:

‹digits› ⇒ ‹digit› ‹digit› ‹digit› ⇒
1 ‹digit› ‹digit› ⇒ 1 9 ‹digit› ⇒ 1 9 7

(3) Multiple choice.

Put elements of which one must be used in round brackets, separated by bars.

☒ For example:

‹expr› → ‹term› | ‹expr› (+ | -) ‹term›

☒ In EBNF, our grammar of expressions:

‹expr› → ‹term› | ‹expr› + ‹term› | ‹expr› - ‹term›
‹term› → ‹factor› | ‹term› * ‹factor›
‹factor› → ‹var› | (‹expr›)
‹var› → x | y

could look as follows:

‹expr› → { ‹term› (+ | -) } ‹term›
‹term› → { ‹factor› * } ‹factor›
‹factor› → ‹var› | (‹expr›)
‹var› → x | y

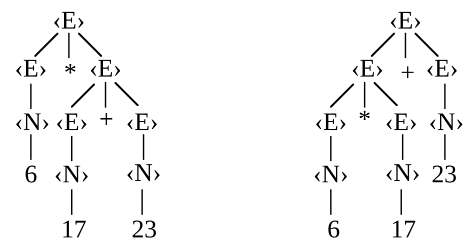
A grammar is **ambiguous** when there is an expression defined by this grammar, which has more than one structurally different derivation tree.

☒ Example: a grammar of arithmetic expressions

‹E› → ‹E› + ‹E› | ‹E› * ‹E› | ‹N›

where ‹N› denotes any unsigned integer.

The expression 6 * 17 + 23 has two different derivation trees:



These trees represent two different ways of computing the value of the expression!

Ambiguity should be avoided.

In our last example, we should have a usual two-level definition instead of a definition with + and * at the same level:

expressions (that's ‹E›) consist of terms (that's ‹T›) which consist of numbers (that's ‹N›).

‹E› → { ‹T› + } ‹T›
‹T› → { ‹N› * } ‹N›

Grammars—summary