LANGUAGE DESCRIPTION

Clear and complete descriptions of a language are needed by programmers, implementers, and even language designers.

Nowadays, a language is typically described by a combination of formal syntax and informal semantics.

The syntax of a language specifies how programs in the language are built up; the semantics of the language specifies what programs mean.

dd/dd/dddd
01/02/2001

Organization of language descriptions:
– Tutorials
– Reference Manuals
– Formal Definitions
The formal syntax of a programming language usually consists of two layers:

**Lexical layer**

The lexical syntax of a language corresponds to the spelling of words in English.

It governs the formation of *numbers, symbols, identifiers, keywords*, etc.

**Grammar (Syntactic) layer**

The syntax of a language is described by a grammar, in particular a *context-free grammar*.

Notations for writing grammars include BNF, Extended BNF (EBNF), and syntax charts.

---

**EXPRESSIONS NOTATIONS**

Expressions such as $a+b\times c$ have been in use for centuries and were a starting point for the design of programming languages.

For example,

$$\left( -b + \sqrt{b^2 - 4 \times a \times c} \right) \div \left( 2 \times a \right)$$

can be written in Fortran as

$$(- b + \text{sqrt}(b**2 - 4.0**c))/(2.0*a)$$
EXPRESSIONS NOTATIONS (cont.)

Programming languages use a mix of notations:

– **Prefix Notation**: the operator is written first, followed by its operands, as in \(+ a b\).

– **Postfix Notation**: the operator is written last, preceded by its operands, as in \(a b +\).

– **Infix Notation**: the operator is written between its operands, as in \(a + b\).

– **Mixfix Notation**: some operations do not fit neatly into the prefix, postfix, and infix classification; i.e., symbols or keywords appear interspersed with the components of an expression. E.g., \(\text{if } a > b \text{ then } a \text{ else } b\).

Prefix and postfix notions are parenthesis-free.

PREFIX NOTATION

An expression in prefix notation is written as follows:

– The prefix notation for a constant or variable is the constant or variable itself.

– The application of a binary operator \(\circ \) to subexpressions \(E_1\) and \(E_2\) is written in prefix notation as \(\circ E_1 E_2\).

– The application of a k-ary operator \(\circ^k\) to subexpressions \(E_1, E_2, ..., E_k\) is written in prefix notation as \(\circ^k E_1 E_2 \ldots E_k\).
PREFIX NOTATION (cont.)

An advantage of prefix notation is that it is easy to decode (parse) during a left-to-right scan of an expression.

Examples:

\[ + \ x \ y \ (= \ x + y) \]
\[ * + \ x \ y \ z \ (= \ (x + y) * z) \]
\[ * + 20 \ 30 \ 60 \ (= \ (20 + 30) * 60 = 3000) \]
\[ * 20 + 30 \ 60 \ (= \ 20 * (30 + 60) = 1800) \]

List uses a variant of prefix:

\[ (\text{read} \ x) \]
\[ (\text{map} \ x \ y) \]

POSFIX NOTATION

An expression in posfix notation is written as follows:

– The posfix notation for a constant or variable is the constant or variable itself.
– The application of a binary operator \( \text{op} \) to subexpressions \( E_1 \) and \( E_2 \) is written in prefix notation as \( E_1 \ E_2 \ \text{op} \).
– The application of a \( k \)-ary operator \( \text{op}^k \) to subexpressions \( E_1, E_2, \ldots, E_k \) is written in prefix notation as \( E_1 \ E_2 \ldots \ E_k \ \text{op}^k \).

Examples:

\[ x \ y + \ (= \ x + y) \]
\[ x \ y + z * \ (= \ (x + y) * z) \]
\[ 20 \ 30 + 60 * \ (= \ (20 + 30) * 60 = 3000) \]
\[ 20 \ 30 \ 60 + * \ (= \ 20 * (30 + 60) = 1800) \]
POSFIX NOTATION (cont.)

An advantage of posfix notations is that they can be mechanically evaluated with the help of a stack.

<table>
<thead>
<tr>
<th>Expression (postfix)</th>
<th>stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>7 7<em>4 2</em>3*–</td>
<td>7</td>
</tr>
<tr>
<td>7<em>4 2</em>3*–</td>
<td>7 7</td>
</tr>
<tr>
<td><em>4 2</em>3*–</td>
<td>49</td>
</tr>
<tr>
<td>4 2<em>3</em>–</td>
<td>49 4</td>
</tr>
<tr>
<td>2<em>3</em>–</td>
<td>49 4 2</td>
</tr>
<tr>
<td><em>3</em>–</td>
<td>49 8</td>
</tr>
<tr>
<td>3*–</td>
<td>49 8 3</td>
</tr>
<tr>
<td>*–</td>
<td>49 24</td>
</tr>
<tr>
<td>–</td>
<td>25</td>
</tr>
</tbody>
</table>

INFIX NOTATION

In infix notation, (binary) operators appear between their operands.

E.g., x + y*z

An advantage of infix notation is that it is familiar and hence easy to read.
However, additional concepts needed for resolving ambiguities.
**INFIX NOTATION (cont.)**

**Precedence:** An operator at a higher precedence level takes its operands before an operator at a lower precedence level. For example, assuming as usual that the operator * has higher precedence than +, then
\[ a + b \cdot c = a + (b \cdot c) \]

**Associativity:** An operator is **left associative** if subexpressions containing multiple occurrences of the operator are grouped from left to right.
\[ 4-2-1 = (4-2)-1 = 2-1 = 1 \]

An operator is **right associative** if subexpressions containing multiple occurrences of the operator are grouped from right to left.
\[ 2^{3^4} = 2^{(3^4)} = 2^{81} \]

---

**ABSTRACT SYNTAX**

The **abstract syntax** of a language identifies the meaningful components of each construct in the language. The meaningful components of an expression are the operators and their operands in the expression. Their structure can be conveniently represented by a tree, where an operator and its operands are represented by a node and its children (subtrees).
ABSTRACT SYNTAX TREE

Trees showing the operator/operand structure of an expression are called abstract syntax trees, because they show the syntactic structure of an expression independent of the notation in which the expression was originally written.

\[ b \times b - 4 \times a \times c \]

Obtain Expressions From Abstract Syntax Tree

\[ \begin{array}{c}
\ast \\
\ast \\
b \\
b \\
\ast \\
c \\
4 \\
a \\
\end{array} \]

**Prefix:** root, left-subtree, right-subtree

\[ b \times b - 4 \times a \times c \]

**Infix:** left-subtree, root, right-subtree

\[ b \times b - 4 \times a \times c \]

**Postfix:** left-subtree, right-subtree, root

\[ b \times 4 \times a \times c \times - \]
LEXICAL SYNTAX

Keywords like `if` and symbols like `<=` are treated as units in a programming language, just as words are treated as units in English.

The syntax of a programming language is specified in terms of units called tokens or terminals.

A lexical syntax for a language specifies the correspondence between the written representation of the language and the tokens or terminal in a grammar for the language.

The lexical units of a program are identifiers, keywords, operators, and punctuation symbols.

Informal description usually suffices for specifying the lexical syntax of a language; real numbers are one possible exception.

---

LEXICAL SYNTAX (cont.)

The mathematical symbols for some common operations and their representations in Pascal and C.

<table>
<thead>
<tr>
<th>binary operation</th>
<th>symbol</th>
<th>Pascal</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>less than</td>
<td><code>&lt;</code></td>
<td><code>&lt;</code></td>
<td><code>&lt;</code></td>
</tr>
<tr>
<td>less than or equal</td>
<td><code>&lt;=</code></td>
<td><code>&lt;=</code></td>
<td><code>&lt;=</code></td>
</tr>
<tr>
<td>equal</td>
<td><code>==</code></td>
<td><code>==</code></td>
<td><code>==</code></td>
</tr>
<tr>
<td>not equal</td>
<td><code>&lt;&gt;</code></td>
<td><code>&lt;&gt;</code></td>
<td><code>!=</code></td>
</tr>
<tr>
<td>greater than</td>
<td><code>&gt;</code></td>
<td><code>&gt;</code></td>
<td><code>&gt;</code></td>
</tr>
<tr>
<td>greater than or equal</td>
<td><code>&gt;=</code></td>
<td><code>&gt;=</code></td>
<td><code>&gt;=</code></td>
</tr>
<tr>
<td>add</td>
<td><code>+</code></td>
<td><code>+</code></td>
<td><code>+</code></td>
</tr>
<tr>
<td>subtract</td>
<td><code>-</code></td>
<td><code>-</code></td>
<td><code>-</code></td>
</tr>
<tr>
<td>multiply</td>
<td><code>*</code></td>
<td><code>*</code></td>
<td><code>*</code></td>
</tr>
<tr>
<td>divide, reals</td>
<td><code>/</code></td>
<td><code>/</code></td>
<td><code>/</code></td>
</tr>
<tr>
<td>divide, integers</td>
<td><code>div</code></td>
<td><code>div</code></td>
<td><code>/</code></td>
</tr>
<tr>
<td>remainder, integer</td>
<td><code>mod</code></td>
<td><code>mod</code></td>
<td><code>%</code></td>
</tr>
</tbody>
</table>
CONTEXT-FREE GRAMMARS

The concrete syntax of a language describes its written representation, including lexical details such as the placement of keywords and punctuation marks. Context-free grammars are a formalism for specifying concrete syntax.

A context-free grammar, or simply grammar, has four parts:
- A set of tokens or terminals.
- A set of nonterminals.
- A set of productions (production rules) for identifying the components of a construct. Each production has a nonterminal as its left side and a string over the sets of terminals and nonterminals as its right side.
- A nonterminal chosen as the starting nonterminal.

Example:

\[ E \rightarrow E + T \]
\[ E \rightarrow T \]
\[ T \rightarrow T \ast F \]
\[ T \rightarrow F \]
\[ F \rightarrow ( E ) \]
\[ T \rightarrow \text{id} \]

Another way of presenting the rules:

\[ S \Rightarrow^* ( )( ) \]

\[ S := e | S S | ( S ) \]
BNF (Backus-Naur Form)

Variant of CFG where nonterminals are enclosed by \( \langle \rangle \). Note that the concept of a context-free grammar, consisting of terminals, nonterminals, productions, and a starting nonterminal, is independent of the notation used to write grammars. BNF is one such notation.

Ex. BNF rules for real numbers:

\[
\langle \text{real-number} \rangle \::= \langle \text{digital-sequence} \rangle . \langle \text{digit-sequence} \rangle \\
\langle \text{digit-sequence} \rangle \::= \langle \text{digit} \rangle | \langle \text{digit} \rangle \langle \text{digit-sequence} \rangle \\
\langle \text{digit} \rangle \::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
\]

Alternatively,

\[
\langle \text{real-number} \rangle \::= \langle \text{integer-part} \rangle . \langle \text{fraction} \rangle \\
\langle \text{integer-part} \rangle \::= \langle \text{empty} \rangle | \langle \text{digit-sequence} \rangle \\
\langle \text{fraction} \rangle \::= \langle \text{digit-sequence} \rangle \\
\langle \text{digit-sequence} \rangle \::= \langle \text{digit} \rangle | \langle \text{digit} \rangle \langle \text{digit-sequence} \rangle \\
\langle \text{digit} \rangle \::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
\]

DERIVATIONS

A derivation consists of a sequence of strings, beginning with the starting nonterminal. Each successive string is obtained by replacing a nonterminal by the right side of one of its productions. A derivation ends with a string consisting entirely of terminals.

Example:

\[
\langle \text{real-number} \rangle \Rightarrow \langle \text{integer-part} \rangle . \langle \text{fraction} \rangle \\
\Rightarrow \langle \text{integer-part} \rangle \langle \text{digit} \rangle . \langle \text{fraction} \rangle \\
\Rightarrow \langle \text{digit} \rangle \langle \text{digit} \rangle . \langle \text{fraction} \rangle \\
\Rightarrow 2 \langle \text{digit} \rangle . \langle \text{fraction} \rangle \\
\Rightarrow 21 . \langle \text{fraction} \rangle \\
\Rightarrow 21 . \langle \text{digit} \rangle \langle \text{fraction} \rangle \\
\Rightarrow 21.8 \langle \text{fraction} \rangle \\
\Rightarrow 21.8 \langle \text{digit} \rangle \\
\Rightarrow 21.89
\]
PARSE TREES

A grammar for a language imposes a hierarchical structure, called a parse tree, on programs in the language.

A parse tree shows how a string can be built:
- Each leaf is labeled with a terminal or 〈empty〉.
- Each nonleaf node is labeled with a nonterminal.
- The label of a nonleaf node is the left side of some production and the labels of the children of the node, from left to right, form the rightside of that production.
- The root is labeled with the starting nonterminal.

A parse tree generates the string formed by reading the terminals at its leaves from left to right.

---

Example: A Parse Tree

A parse tree w.r.t. the above grammar for generating 123.789.
SYNTACTIC AMBIGUITY

A grammar for a language is syntactically ambiguous, or simply ambiguous, if some string in its language has more than one parse tree.

Programming languages can usually be described by unambiguous grammars.

If ambiguities exist, they are resolved by establishing conventions that rule out all but one parse tree for each string.

AN AMBIGUOUS GRAMMAR

\[ E ::= E - E | 0 | 1 \]

The string \(1 - 0 - 1\), for instance, has two parse trees.

![Parse trees for the string 1 - 0 - 1]
A well-known example of syntactic ambiguity is the **dangling-else ambiguity**.

An ambiguous grammar:

\[
S ::= \text{if } E \text{ then } S \\
S ::= \text{if } E \text{ then } S \text{ else } S
\]

The string `if E1 then if E2 then S1 else S2` has two parse trees; the `else` can be matched with either `if`.

![Parse Trees](image_url)

The dangling-else ambiguity is typically resolved by matching an `else` with the nearest unmatched `if`.

---

A grammar for a language is usually designed to reflect the **abstract syntax**.

A well-designed grammar can make it easy to pick out the meaningful components of a construct.

With a well-designed grammar, parse trees are similar enough to abstract syntax trees that the grammar can be used to organize a language description or a program that exploits the syntax.
ASSOCIATIVITY AND PRECEDENCE

A partial table of binary operators in C, in the order of increasing precedence. The assignment operator is right associative; all the others are left associative.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Precedence</th>
</tr>
</thead>
<tbody>
<tr>
<td>assignment</td>
<td>=</td>
</tr>
<tr>
<td>logical or</td>
<td></td>
</tr>
<tr>
<td>logical and</td>
<td>&amp;&amp;</td>
</tr>
<tr>
<td>inclusive or</td>
<td></td>
</tr>
<tr>
<td>exclusive or</td>
<td>-</td>
</tr>
<tr>
<td>and</td>
<td>&amp;</td>
</tr>
<tr>
<td>equality</td>
<td>== !=</td>
</tr>
<tr>
<td>relational</td>
<td>&lt; &lt;= &gt;= &gt;</td>
</tr>
<tr>
<td>shift</td>
<td>&lt;&lt; &gt;&gt;</td>
</tr>
<tr>
<td>additive</td>
<td>+ -</td>
</tr>
<tr>
<td>multiplicative</td>
<td>* / %</td>
</tr>
</tbody>
</table>

EXTENDED BNF (EBNF)

EBNF is an extension of BNF that allows lists and optional elements to be specified.

- Braces, { and }, represent zero or more repetitions.
- Brackets, [ and ], represent an optional construct.
- A vertical bar | represents a choice.
- Parentheses, ( and ), are used for grouping.

\[
\begin{align*}
\langle \text{expression} \rangle & ::= \langle \text{term} \rangle \{ \langle \text{term} \rangle \} \\
\langle \text{term} \rangle & ::= \langle \text{factor} \rangle \{ \langle \text{factor} \rangle \} \\
\langle \text{factor} \rangle & ::= (\langle \text{expression} \rangle \langle \text{expression} \rangle) | \text{name} | \text{number}
\end{align*}
\]
SYNTAX CHARTS

Besides their visual appeal, an advantage of syntax charts is that all of the nonterminals in a chart are meaningful. With BNF, it is sometimes necessary to make up auxiliary nonterminals to achieve the effect of alternative paths and loops in a syntax chart.