

Language descriptions

- Tutorials: guided tours of a language.
- Reference manuals: describe the syntax and semantics of languages in English.
- Formal definition: Precise description of the syntax and semantics of a language. It is aimed at specialists.

Formal - based on mathematics.

Expressions

- Starting point for a language.
- $(-b + \text{sqrt}(b * b - 4.0 * a * c)) / (2.0 * a)$
- Different notations for an expression.
- **Prefix notation** (Example: + 1 2)
- **Postfix notation** (Example: 1 2 +)
- **Infix notation** (Example: 1 + 2)
- **Mixfix notation**: Operations that do not fit into the prefix, infix and postfix classifications. (Example: if-then-else)

- **Examples:**

– Prefix:

* + 20 30 60 =

* 20 + 30 60 =

– Postfix:

20 30 + 60 * =

20 30 60 + * =

How to evaluate a prefix or postfix expression?

Infix notation

- How do you compute $5 * 7 + 3 - 1$?
- Is it $(5 * 7) + (3 - 1)$?
- Is it $5 * (7 + 3) - 1$?
- To deal with the ambiguity: Use parentheses or use a precedence on operators.

Example: $\{*, /\} > \{+, -\}$.

The expression is: $(5 * 7) + 3 - 1$.

$*$ and $/$ have the same precedence. $+$ and $-$ have the same precedence.

- An operator is said to be **left-associative** if subexpressions containing the same operator or operator with the same precedence are grouped from left to right to be decoded.

Examples:

– $-$ is left-associative. $4 - 2 - 1$ is $(4 - 2) - 1$ and not $4 - (2 - 1)$.

– $5 * 7 + 3 - 1$ is $((5 * 7) + 3) - 1$.

– $+$, $*$ and $/$ are also left-associative.

- An operator is said to be **right-associative** if subexpressions containing the same operator or operator with the same precedence are grouped from right to left to be decoded.

Examples:

- Exponentiation is right-associative. 2^{3^4} is $2^{(2^3)}$.
- The assignment symbol is right-associative.

Abstract syntax tree

- The **abstract syntax** of a language identifies the meaningful components of each construct in the language. It captures intent, independent of notation.

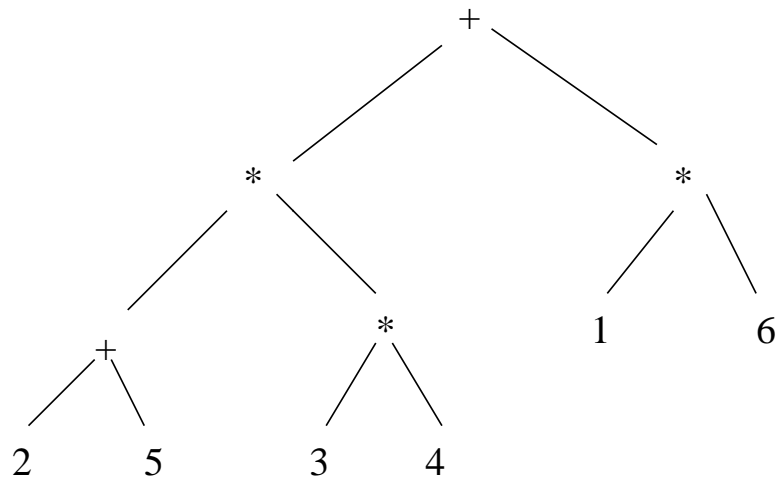
Examples: $a + b$, $+ a b$ and $a b +$ have the same abstract syntax.

- Described by an **abstract syntax tree**.

Example: $a + b$, $+ a b$ and $a b +$ are represented by the same abstract syntax tree.

Expression Trees

Rooted trees can be used to represent arithmetic formulas.



The root of the tree represents the final value of the computation.

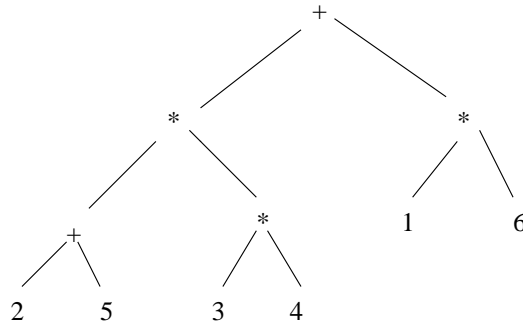
The leaves represent the operands, and the intermediate nodes the operators.

This tree represents the equation $(2+5)*(3+4)+(1*6)$ using conventional arithmetic notation, or $\{2\ 5\ +\ 3\ 4\ *\ 1\ 6\ *\ +\}$ using reverse Polish notation (as on an HP calculator) (postfix notation).

These formulas can be constructed by appropriately **traversing** the expression tree, i.e. visiting the nodes in the correct order.

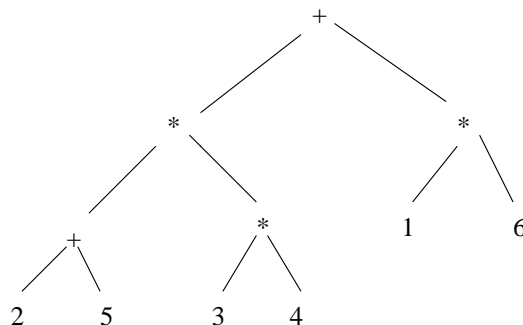
There are three natural orders to visit the nodes of the tree, each of which walks up and down the tree in a **recursive** manner:

- **In-order** – visit all the left subtree before visiting the root node, then visit the right subtree.



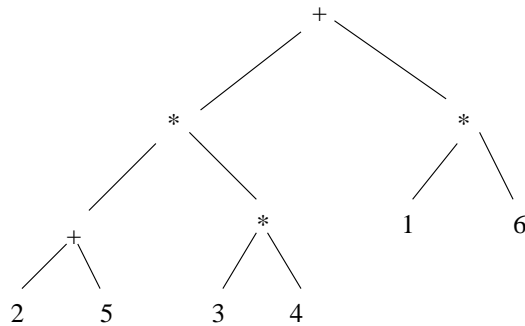
Infix notation : $2 + 5 * 3 + 4 + 1 * 6$

- **Post-order** – visit the left subtree and right subtrees completely before visiting the root node.



Postfix notation : $25 + 34 * * 16 * +$

- **Pre-order** – visit the root node before visiting the left subtree and right subtrees.



Prefix notation : $+ * + 2 5 * 3 4 * 1 6$

- A fourth natural order, **breadth-first** traversal, traverses level-by-level, $\{+ * * + * 1 6 2 5 3 4\}$.

This involves jumping around from one subtree to another, and hence is not good for evaluating expressions, although breadth-first traversal does have numerous applications in computer science.

if-the-else Tree

- How to represent *if $a > b$ then a else b* ?

Lexical syntax

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

Example: Java has 40 to 50 tokens (if, then, while, =, +, symbol, number...).

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

Example: Assume the expression

$$b * b - 4 * a * c$$

in a programming language.

Its lexical syntax is:

$$\textit{symbol}_b * \textit{symbol}_b - \textit{number}_4 * \textit{symbol}_a * \textit{symbol}_c$$

Context-free grammars

- Context-free grammars are the notation for specifying **concrete syntax**.
- The most common notation is the **BNF** (Backus-Naur form) notation.
- A grammar is a set of **production rules**.
- \langle , \rangle , $::=$ and $|$ are **meta-symbols** to defined BNF rules.
- A **production rule** is of the form:

`left-hand-side ::= ‘definition’`

- The left-hand-side is the name of a grammatical category. It is a nonterminal.
- $::=$ means “is defined”.
- The right-hand-side is a definition that specifies the grammatical structure of the symbol appearing on the left-hand-side of the rule.

- The right-hand-side of a rule can be a **terminal** or a **nonterminal** object.
 - **Terminal objects** are the tokens of the language. Terminals are not defined by other grammar rules.
 - **Nonterminal objects** are defined by other rules of grammar.Nonterminal objects are written inside angle brackets.
| separates two alternative definitions of a nonterminal. It means OR.
- There is a special nonterminal symbol called the **goal symbol** (starting nonterminal).
- Designing a grammar is not easy (ambiguities)! (see later)

Examples of grammars

- $\langle \text{digit} \rangle ::= 0|1|2|3|4|5|6|7|8|9$

$\langle \text{digit} \rangle$ is a nonterminal.

0, ..., 9 are terminals.

- $\langle \text{integer} \rangle ::= \langle \text{digit} \rangle \mid \langle \text{digit} \rangle \langle \text{integer} \rangle$
 $\langle \text{signed integer} \rangle ::= \langle \text{sign} \rangle \langle \text{integer} \rangle$
 $\langle \text{sign} \rangle ::= +|-|^{\wedge}$

- Subset of Java:

$\langle \text{assignment statement} \rangle ::= \langle \text{variable} \rangle = \langle \text{expression} \rangle$
 $\langle \text{expression} \rangle ::= \langle \text{variable} \rangle \mid \langle \text{variable} \rangle + \langle \text{variable} \rangle$
 $\langle \text{variable} \rangle ::= x|y|z$

- Subset of the English language!

$\langle \text{sentence} \rangle ::= \langle \text{noun} \rangle \langle \text{verb} \rangle$
 $\langle \text{noun} \rangle ::= \text{bees} \mid \text{dogs}$
 $\langle \text{verb} \rangle ::= \text{buzz} \mid \text{bite}$

What is the grammar?

- Write a BNF grammar that describes boolean expressions of the form *var op var*.
 - *var* can be *x*, *y* or *z*.
 - *op* can be *==*, *<* or *>*.
 - The parentheses are part of the expression.

```
<expression> ::= (<var><op><var>) | <var><op><var>
<var> ::= x|y|z
<op> ::= < | == | >
```

Parse tree

- A **parse tree** is designed according to a **grammar**.
 - Each leaf is labeled with a terminal or $\langle \textit{empty} \rangle$ (the empty string).
 - Each non leaf node is labeled with a nonterminal.
 - Each non leaf node label is the left side of a production rule and the labels of the children of the node, from left to right, form the right hand side of that production.
 - The root is labeled with the goal symbol.
- **A string is in the language if and only if it is generated by some parse tree.**
- To construct a parse tree:
 - Top-down approach
 - Bottom-up approach

Examples

- $\langle \text{integer} \rangle ::= \langle \text{digit} \rangle \mid \langle \text{digit} \rangle \langle \text{integer} \rangle$
 $\langle \text{signed integer} \rangle ::= \langle \text{sign} \rangle \langle \text{integer} \rangle$
 $\langle \text{sign} \rangle ::= + \mid - \mid ^$

Parse tree of +112

- Subset of Java:

$\langle \text{assignment statement} \rangle ::= \langle \text{variable} \rangle = \langle \text{expression} \rangle$
 $\langle \text{expression} \rangle ::= \langle \text{variable} \rangle \mid \langle \text{variable} \rangle + \langle \text{variable} \rangle$
 $\langle \text{variable} \rangle ::= x \mid y \mid z$

Parse tree of $x = y + z$

- Subset of the English language!

$\langle \text{sentence} \rangle ::= \langle \text{noun} \rangle \langle \text{verb} \rangle$
 $\langle \text{noun} \rangle ::= \text{bees} \mid \text{dogs}$
 $\langle \text{verb} \rangle ::= \text{buzz} \mid \text{bite}$

Is “bees bite” in the language described by the previous grammar?

Syntactic ambiguity

- A grammar is said **ambiguous** if some string in its language has more than one parse tree.
- **Example 1:** The following grammar is ambiguous.

$\langle E \rangle ::= \langle E \rangle - \langle E \rangle \mid 0 \mid 1$

1-0-1 has 2 different parse trees.

- **Example 2: dangling-else ambiguity:** The following grammar is ambiguous.

$\langle S \rangle ::= \text{if } E \text{ then } S \mid \text{if } E \text{ then } S \text{ else } S$

if E1 then if E2 then S1 else S2 has 2 different parse trees.

- **Example 3:** The following grammar is ambiguous.

$\langle \text{assignment statement} \rangle ::= \langle \text{variable} \rangle = \langle \text{expression} \rangle$
 $\langle \text{expression} \rangle ::= \langle \text{variable} \rangle \mid \langle \text{expression} \rangle + \langle \text{expression} \rangle$
 $\langle \text{variable} \rangle ::= x \mid y \mid z$

$x = x + y + z$ has 2 different parse trees.

Solving ambiguity

- To solve the ambiguity we can use the properties of some symbols of the grammar.

– associative symbols, precedence on symbols

Note: When designing the grammar it is also useful to take into account the properties of the symbols.

- **Example 1:**

$\langle E \rangle ::= \langle E \rangle - \langle E \rangle \mid 0 \mid 1$

To solve ambiguity we use the fact that $-$ is left-associative.

- **Example 2:**

$\langle S \rangle ::= \text{if } E \text{ then } S \mid \text{if } E \text{ then } S \text{ else } S$

if E1 then if E2 then S1 else S2 has 2 different parse trees.

To solve ambiguity we match an else with the nearest unmatched if.

- **Example 3:**

```
<assignment statement> ::= <variable>=<expression>  
<expression> ::= <variable> | <expression>+<expression>  
<variable> ::= x|y|z
```

To solve ambiguity we use the fact that + is left-associative.

From abstract to concrete syntax

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

The parse trees are to be as close as possible to the abstract syntax tree. We add a label to each node of the parse tree representing the token it refers to.

- **Example:**

$\langle E \rangle ::= \langle E \rangle - \langle E \rangle \mid 0 \mid 1$

Concrete tree of 1-0-1

Variants of grammars

- EBNF (Extended BNF)
- Syntax charts are a graphical notation for grammars.

Compilation process

Overview

Compilers

- High-level languages must be translated to machine language prior to execution.
- This is done using a software called a **compiler**.
- **Compilers** are complex software to design and implement.
- Why?

To one high-level language statement correspond many machine language or assembly language statements.

High-level languages are **one-to-many**

Whereas:

To one assembly language statement correspond one machine language statements.

Assembly languages are **one-to-one**.

Example

High level language

1 instruction

`a = b + c - d;`



Assembly language

4 instructions

`LOAD B,R`

`ADD C,R`

`SUBSTRACT R,D,R`

`STORE R,A`



Machine language

4 instructions

Sequence of 0 and 1

Language

- **Syntax:** Linguistic analysis of the *structure* of the program.
- **Semantics:** Meaning of each statement of the program and of the program.
- Translation must be **correct**.
The machine language program is a **correct** translation of the high-level language program. (They do the same thing).
- The translated code must be **efficient** and **concise** (speed and size of the compiled program).

Compilation process

- **Phase I: Lexical Analysis**
- **Phase II: Parsing**
- **Phase III: Semantics analysis and code generation**
- **Phase IV: Code optimization**
- *Sourceprogram* →
Scanner (or lexical analyzer) →
Parser →
Code Generator →
Optimizer →
Object program
- **Example:** File.java → ... → File.class (bytecode)
→ machine language

Phase I

Lexical analysis

- *Grouping letters into tokens.*
Blanks and non essential characters are discarded.
- **Example:** Java has 40 to 50 tokens (if, then, while, =, +, symbol, number...).
- Tokens are **classified** by type. A number is assigned to each class. This encoding permits us to test the syntax easily.
- **Example:** 1 for all symbols, 2 for the numbers, 3 to =, 4 to +...

- **Lexical analyzer or scanner:**
 - Input: A high-level program
 - Output: A set of tokens and a classification

- **Algorithm:**

Discard blanks
until a nonblank character is found

Group characters together
until either a blank or a end character is found

- **Example:**

- *Input:* $a = b + 319 - \text{delta};$
- *Output:*

a	1
=	3
b	1
+	4
319	2
delta	1
;	6

Phase II

Parsing

- The **parser** represents the program as a **parse tree**.
- If a compiler is able to diagram the program by a parse tree with the goal of the grammar as root, it concludes that the statement is structurally correct.

- **Algorithm of a parser:**

If by repeating the application of the grammar rules a parser converts the sequence of input tokens into the goal symbol, then the program is syntactically correct otherwise it is not.

Phase III

Semantics and code generation

- Consider the following grammar.

```
<sentence> ::= <noun><verb>  
<noun> ::= dog | man  
<verb> ::= bit
```

“Man bit” is syntactically complete but it makes no sense!

This problem is dealt with the phase III of the compilation process.

- In Java:

```
char a = 'c';  
double b = 1.2;  
int sum = 0;
```

```
sum = a+b;
```

What does it mean to add a character to a real number?

Is this accepted or not?

Semantics and code generation

- Use of the parse tree.
- During the semantics and code generation phase the compiler:
 - analyzes the **meaning** of the tokens and
 - tries to understand the **actions** they performed.This is **semantics analysis**.
At this point **semantics errors** are detected.
- The compiler also generates the proper sequence of machine language instructions to carry out these actions.
This is **code generation**.

Declaration of a variable

- `<declaration> ::= <type><variable>;`
`<type> ::= int`
`<variable> ::= x | y | z`
- `int x;`
- Parse tree of `int x;` :
- **Construction of the semantics records of `int x;`:**
 - * Semantics records are associated with each *nonterminal symbol* in the grammar.
 - * A **semantic record** is a *data structure* that stores information of a nonterminal and its data type.
 - Semantics record:

x	int
---	-----
 - The compiler adds a link (in memory) from the nonterminal in the parse tree to the semantics record of this nonterminal.

- **Semantics analysis:**

A pass over the parse tree determines whether all branches are semantically valid. If so, then the compiler generates machine language instructions otherwise there is a semantic errors and the process is stopped.

Here there is no semantics errors.

- **Code generation:**

int x; gives the default value 0 to x.

x: .DATA 0

$$x = y + z$$

- `<assignment statement> ::= <variable>=<expression>`
`<expression> ::=`
 `<variable> | <expression> + <expression>`
`<variable> ::= x|y|z`
- Assume x, y and z are declared as int.
Parse tree of $x = y + z$:

- Consider $y + z$

`<expression>+<expression>`

- **Construction of the semantics records:**

Introduction of a variable called *temp*, name generated by the compiler.

- **Semantics analysis:**

No semantics errors.

Addition is well-defined for integers and returns an integer.

- **Code generation:**

```
LOAD Y, R
ADD Z, R
STORE R, TEMP
TEMP: . DATA 0
```

- Consider $x = y + z$

`<variable> = <expression>`

- **Construction of the semantics records:**

Use of the variable called *temp*.

- **Semantics analysis:**

No semantics errors.

An integer = An integer is valid.

- **Code generation:**

```
LOAD TEMP, R
STORE R, X
```

- A semantic record is created for `< assignment statement >`.
The same as the semantic record of `< variable >`.

Overall code generation

- $x = y + z$
- LOAD Y, R
ADD Z, R
STORE R, TEMP
LOAD TEMP, R
STORE R, X
TEMP: . DATA 0

Phase IV

Code optimization

- The generated code must be efficient in space and time.
- Assume that one instruction is executed in 1 micro-second except ADD and SUBTRACT takes 2 micro-seconds and MULTIPLY and DIV takes 3 micro-seconds.
- What is the time needed to execute the following program?

```
INCREMENT X  
INCREMENT X  
INCREMENT X
```

- What is the time needed to execute the following program?

```
LOAD X, R  
ADD THREE, R  
STORE R, X  
THREE: .DATA 3
```

- Are these programs equivalent? Which one is the most efficient?

Different optimizations

- **Constant evaluation:** Evaluation of expression during compilation instead of execution.
- **Strength reduction:** A slow operation is replaced by a faster one.
- **Eliminating unnecessary operations.**

Constant evaluation

- Consider $x = 1 + 1$.

- LOAD ONE, R
ADD ONE, R
STORE R, X
ONE: .DATA 1

is equivalent to

```
LOAD TWO, R  
STORE R, X  
TWO: .DATA 2
```

Strength reduction

- Consider $x = 2 * x$.

- LOAD X, R
MULTIPLY TWO, R
STORE R, X
TWO: .DATA 2

is equivalent to

```
LOAD X, R  
ADD X, R  
STORE R, X
```

Elimination of unnecessary operations

- Consider the following code:

x=y

z=y

- LOAD Y, R
STORE X, R
LOAD Y, R
STORE R, Z

is equivalent to

LOAD Y, R
STORE R, X
STORE R, Z

Conclusion

- Topics we study:
 - Syntax - abstract, lexical, concrete
 - Grammars - BNF grammars
 - Parse tree
- We survey the compilation process.
 - No real details.
 - Very complex.