Programmiersprachen (Programming Languages)

coordinates:	No. 185.208, VU, 3 ECTS
lecturer:	Franz Puntigam
web:	http://www.complang.tuwien.ac.at/franz/ps.html
usable for:	Software Engineering & Internet Computing Computational Intelligence
requirements:	experience in programming
registration:	in TISS until 2017-05-05 group building: 2017-05-05 during lecture
book:	Carlo Ghezzi, Mehdi Jazayeri. Programming Language Concepts. 3rd edition, John Wiley & Sons, 1998, ISBN 0-471-10426-4

Contents

overview

syntax, semantics, pragmatics
and implementation of languages

data types

(modularity and programming in the large)

control structures, exceptions and concurrency

object-oriented programming languages

functional programming languages

Overview

Languages in Software Development

tools: editor, compiler, simulator, debugger, . . .

languages for: program, specification, documentation, . . .

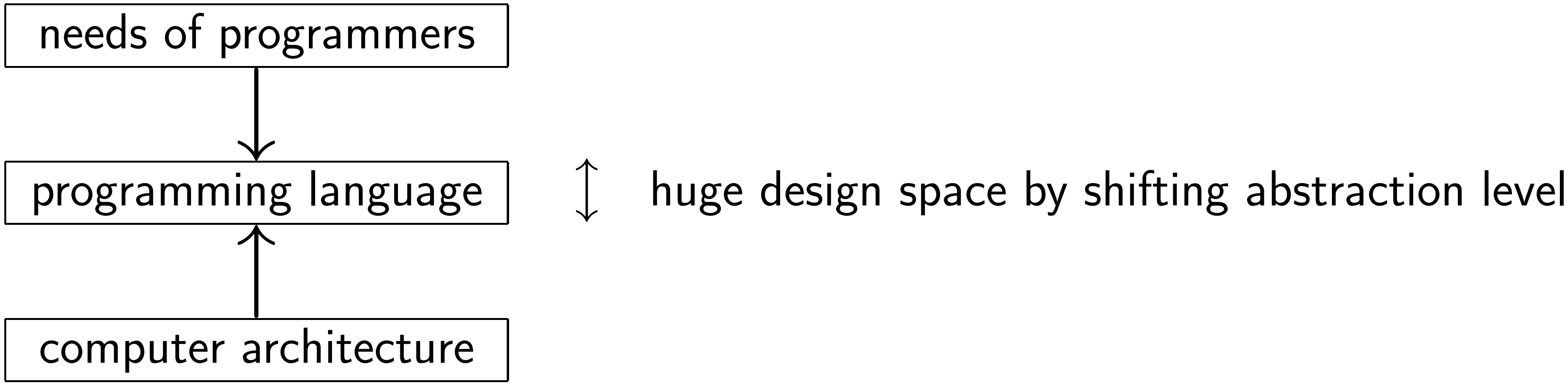
ideally: tools | use the same language,
ensure soundness and completeness,
give suggestions for improvement,
automate software development steps

however: languages are formal and custom-designed (bad integration)

Programming Paradigms (Ways of Thinking)

imperative declarative	(visibility of underlying machine model)
procedural functional applicative logic-oriented	(computational model, low-level abstraction)
module-based component-based object-oriented	(software organisation, high-level abstraction)
generic aspect-oriented ...	(software parameterization)

Design Space and Abstraction Level



. . . and there are many further dimensions in design space

History of Programming Languages

FORTRAN	1954–57	numeric computing
ALGOL 60	1958–60	numeric computin
COBOL	1959–60	business data processing
LISP	1956–62	symbolic computing, functional programming
PL/I	1963–64	general purpose
BASIC	1964	educational, interactive
SIMULA 67	1967	simulation
Pascal	1971	educational, general purpose
PROLOG	1972	artificial intelligence, logic-oriented programming
C	1972	systems programming
CLU	1974–77	ADT programming
Ada	1979	general purpose, embedded systems
Smalltalk	1971–80	personal computing, object-oriented programming
C++	1984	general purpose, object-oriented programming
Eiffel	1988	general purpose, object-oriented programming
Perl	1990	scripting language
Java	1995	network computing?, general purpose
...		what's the character of many newer languages?

Syntax, Semantics, Pragmatics

Syntactic and Semantic Rules

syntax:

rules describe appearance of language elements,
smallest syntactic units are characters

semantics:

rules describe meaning of language elements,
smallest syntactic units are rather abstract tokens

semantic rules usually much more complex than syntax rules

most important differences between languages are semantic ones

Semantic Properties of a Variable (Example)

name: used as abstraction of memory cell(s) and the value therein

type: kind of possible values in the variable, restricts usability of the variable

lifetime: time between allocation and deallocation of memory

scope: part of the programm where the variable is known

accessibility: variable may not be accessible everywhere where known, kind of access can be restricted (e.g., only readable)

visibility: variable can be hidden even if in scope and accessible

r-value: the current value in the variable

l-value: the location of the variable in memory

Values and References (Example Continued)

l-value is a reference to the variable,
r-value is the contents of the variable,
literal is a predefined value existing without variables

examples in C:

`x = y;`	r-value of `y` copied into memory referenced by l-value of `x`
`x = &y;`	l-value of `y` copied into memory referenced by l-value of `x`
`x = 3;`	literal 3 copied into memory referenced by l-value of `x`
`3 = y;`	error, literal 3 must not be used as l-value

Syntax Specification

$$
\begin{aligned}
&\langle\text{program}\rangle ::= \{\langle\text{statement}\rangle^*\}\\
&\langle\text{statement}\rangle ::= \langle\text{assignment}\rangle \mid \langle\text{conditional}\rangle \mid \langle\text{loop}\rangle\\
&\langle\text{assignment}\rangle ::= \langle\text{identifier}\rangle = \langle\text{expr}\rangle;\\
&\langle\text{conditional}\rangle ::= \texttt{if}\ \langle\text{expr}\rangle\ \{\langle\text{statement}\rangle^+\} \mid\\
&\qquad\qquad \texttt{if}\ \langle\text{expr}\rangle\ \{\langle\text{statement}\rangle^+\}\ \texttt{else}\ \{\langle\text{statement}\rangle^+\}\\
&\langle\text{loop}\rangle ::= \texttt{while}\ \langle\text{expr}\rangle\ \{\langle\text{statement}\rangle^+\}\\
&\langle\text{expr}\rangle ::= \langle\text{identifier}\rangle \mid \langle\text{number}\rangle \mid (\langle\text{expr}\rangle) \mid\\
&\qquad\qquad \langle\text{expr}\rangle\ \langle\text{operator}\rangle\ \langle\text{expr}\rangle
\end{aligned}
$$

$$
\begin{aligned}
&\langle\text{operator}\rangle ::= + \mid - \mid * \mid / \mid = \mid \neq \mid < \mid > \mid \leq \mid \geq\\
&\langle\text{identifier}\rangle ::= \langle\text{letter}\rangle\ \langle\text{ld}\rangle^*\\
&\langle\text{ld}\rangle ::= \langle\text{letter}\rangle \mid \langle\text{digit}\rangle\\
&\langle\text{number}\rangle ::= \langle\text{digit}\rangle^+\\
&\langle\text{letter}\rangle ::= \text{a} \mid \text{b} \mid \text{c} \mid \ldots \mid \text{z}\\
&\langle\text{digit}\rangle ::= 0 \mid 1 \mid 2 \mid \ldots \mid 9
\end{aligned}
$$

Abstract vs. Concrete Syntax, Pragmatics

```
while (x == y)
    x = ...;
```

```
while (x == y)
    {
        ...
    }
```

```
while x = y do
    begin
        ...
    end
```

```
if x = y then
    ...
end
```

```
if x = y then
    ...
else
    ...
end
```

```
while x = y do
    ...
end
```

Static and Dynamic Semantics

syntax defines well-formed programs,
semantic defines meaning of well-formed programs

some well-formed programms have no meaning,
example: **`while`** (1.75) { . . . }

meaningful well-formed programms usually distinguished from meaningless programs **before** program execution

static semantics defines which well-formed programs are meaningful

dynamic semantics defines effects of executing meaningful programs

Formal Semantics

semantics $\neq$ implementation

formal specification of semantics given in **meta-language**

formal semantics useful as a reference, but usually difficult to read

operational semantics: pseudo-implementation on virtual machine

axiomatic semantics: state transitions specified in logics

denotational semantics: state and state transitions specified by functions

Axiomatic Semantics

{**Precondition**} **Statement** {**Postcondition**}

$\{y \geq 0\}$ $x = y + 1;$ $\{x \geq 1$ and $y \geq 0\}$

{true}
if $x \geq y$ **then** max $=$ x; **else** max $=$ y;
$\{(\text{max} = x$ and $x \geq y)$ or $(\text{max} = y$ and $y > x)\}$

{P} S1; S2; {Q} if {P} S1; {R} and {R} S2; {Q}

{P} **while** B **do** L {P and not B} if {P} L {P} (P is a loop invariant)

Denotational Semantics

$$\text{dsem}(\text{S1; S2;}, \text{mem}) = \text{mem2} \quad \text{if} \quad \text{dsem}(\text{S1;}, \text{mem}) = \text{mem1} \quad \text{and}$$
$$\text{dsem}(\text{S2;}, \text{mem1}) = \text{mem2}$$

$$\text{dsem}(\texttt{if}\ \text{B}\ \texttt{then}\ \text{L1}\ \texttt{else}\ \text{L2}, \text{mem}) = \text{mem1} \quad \text{where}$$
$$\text{mem1} = \text{dsem}(\text{L1}, \text{mem}) \quad \text{if} \quad \text{mem}(\text{B}) = \text{true};$$
$$\text{mem1} = \text{dsem}(\text{L2}, \text{mem}) \quad \text{if} \quad \text{mem}(\text{B}) = \text{false}$$

$$\text{dsem}(\texttt{while}\ \text{B}\ \texttt{do}\ \text{L}, \text{mem}) =$$
$$\text{mem} \quad \text{if} \quad \text{mem}(\text{B}) = \text{false};$$
$$\text{dsem}(\texttt{while}\ \text{B}\ \texttt{do}\ \text{L}, \text{dsem}(\text{L}, \text{mem})) \quad \text{if} \quad \text{mem}(\text{B}) = \text{true}$$

Execution and Tools

interpreters and **processors** repeatedly execute the following sequence:

1. fetch next instruction
2. determine the actions to be executed
3. perform the actions

compilers translate programs from source languages to target languages (keeping the semantics)

linkers combine several modules into one unit

loaders relocate programs to fit them into memory (usually not necessary today because of memory management unit on processor)

Interpretation versus Compilation

pure compilation: target language is native language of processor
(fast execution, compiler often ensures type consistency)

pure interpretation: language directly executed by interpreter
(portable, often efficient use of memory, supports interactive development)

compilation and interpretation: translation from target language into intermediate language, intermediate language is interpreted (tradeoff)

just-in-time compilation: interpreter transparently uses compilation techniques to improve performance

repeated compilation: increases expressiveness of languages
(e.g., macros, templates, aspect-oriented programming techniques)

Binding

binding: setting the value of an attribute (name, type, scope, . . .) belonging to a language element (variable, expression, statement, . . .)

binding time: time when binding is established
(time of language definition, compiler implementation, compilation, execution)

stability of binding: is an established binding fixed or modifiable

static binding: fixed binding established before the execution begins

dynamic binding: modifiable binding established at run-time

Names and Scopes

a variable name is usually introduced by a **declaration**,
the scope ranges from the declaration to a language-defined end

dynamic scoping: scope ends at a new declaration for a variable of the same name (e.g., in Lisp and APL)

static (= lexical) scoping: lexical program structure determines scope

block-structured language: scope ends with block containing the declaration, declaration in inner block **hides** same names declared in outer block

Dynamic Scoping

```
{  /* Block A */
   int x;
   ...
}
...
```

```
{  /* Block B */
   int x;
   ...
}
...
```

```
{  /* Block C */
   ...
   x = ...;
   ...
}
```

depending on the execution sequence, `x` in `Block C` refers to either the variable declared in `Block A` or the variable declared in `Block B`

Static Scoping

```
#include <stdio.h>

main()
{
   int x, y, z;
   scanf("%d %d %d", &x, &y, &z);  /* read into x, y, z   */

   {          /* swap x und y                          */
      int z;  /* hides z declared in outer block       */
      z = x;  /* z from inner block, x from outer block */
      x = y;  /* x and y from outer block              */
      y = z;  /* y from outer block, z from inner block */
   }          /* scope of z from inner block ends      */

   printf("%d %d %d", x, y, z);    /* print (outer block) */
}
```

Static Typing

static typing: types are statically bound to variables

in Pascal: **var** x, y: integer;
c: character;

static typing is a basis for **static type checking**

x := c; is illegal

variable types can be specified explicitly, but that is not necessary

in FORTRAN the first letter of the variable name can determine the type,
ML and Haskell use **type inference** to determine the type

Dynamic Typing

dynamic typing: types are dynamically bound to variables (Lisp, Smalltalk), these variables are **polymorph** (have several types)

variable types change when assigning new values to the variables

in general **dynamic type checking** at run-time,
but in some cases **static type checking** possible (e.g., subtyping)

language is **untyped** if there is no type information at all,
rarely used in higher-level languages, usual in Assembler languages

l-Values

memory cells needed during whole lifetime of a variable or object

memory allocation: binding of a variable's l-value to a memory address

memory deallocation: cancelling the binding at the end of the scope

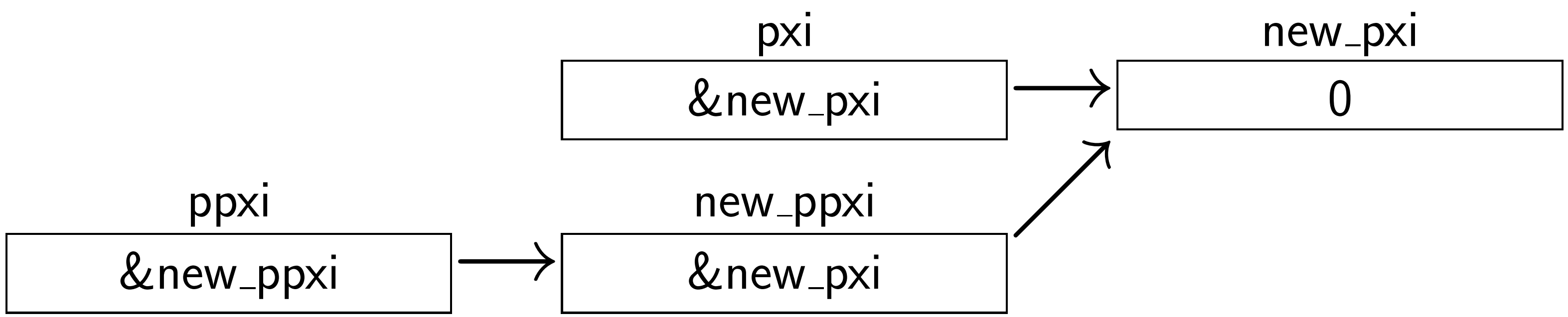
static allocation: memory allocated before and deallocated after execution

dynamic allocation: memory allocated and deallocated at run-time

dynamic allocation can be **explicit** or **implicit** (at the end of the scope)

Pointer

a **pointer** is an r-value that can be used as l-value




```
type PI = ^integer;
var pxi: PI;
...
new(pxi);
pxi^ = 0;
```

```
type PPI = ^PI;
var ppxi: PPI;
...
new(ppxi);
ppxi^ = pxi;
```

```
int x = 0;
int* px = &x;
int** ppx = &px;
int y = *px;
**ppx = 5;
```

Routine

routine specified by name, scope, type, l-value, r-value

head: name + type = **signature** (sometimes also parameter names)

fun: $T_1 \times T_2 \times \cdots \times T_n \to R$

body: local declarations + executable statements

r-value = body,
l-value = memory address of routine

routines are first-class objects if variables can contain routines
(or pointers to routines)

invocation: specifies actual parameters (= arguments)

i = sum(3);

Function and Pointer to Function in C

```
int sum(int n)
{
   int i, s;
   s = 0;
   for (i = 1; i <= n; i++)
      s = s + 1;
   return s;
}
```

```
int i;
int (*ps)(int);
...
ps = &sum;
i = (*ps)(5);
   /* i = sum(5); */
```

Declaration vs. Definition of Routines

```
int A(int x, int y);       /* declaration of A */
float B(int z)             /* definition of B  */
{  int w, u;
   ...
   w = A(z, u);            /* A usable here    */
   ...
};
int A(int x, int y)        /* definition of A  */
{  float t;
   ...
   t = B(x);               /* B usable here    */
   ...
};
```

Invocation of Routines

routine instance: code segment + activation record (or **stack frame**)

environment: local environment = objects in activation record
non-local environment = all other visible objects

recursion: new activation record created before execution of a routine ends

dynamic binding between code segment and activation record

formal parameter: parameter in the head of a routine

actual parameter: parameter (or argument) in an invocation

Parameter Binding

```
procedure Example(A: T1; B: T2 := W; C: T3);
```

`--` in Ada: procedure with default value for parameter `B`

```
Example(X, Y, Z);
```

`--` invocation; parameters are bound by position

```
Example(C => Z, A => X, B => Y);
```

`--` parameters are bound by name

```
Example(X, C => Z);
```

`--` parameters bound by position and name, default value is used

Generic Routine

```
template <class T> void swap(T& a, T& b)
/* generic routine in C++ swaps a und b */
{
  T temp = a;
  a = b;
  b = temp;
}
```

```
int i, j;
...
swap(i, j);
```

```
float f, g;
...
swap(f, g);
```

Overloading and Aliasing

overloading: one name refers to several entities

aliasing: several names refer to the same object

```
int i, j, k;
float a, b, c;
...
i = j + k;
a = b + c;
a = b + c + b();
a = b() + c + b(i);
```

```
int x = 0;
int *i = &x;
int *j = &x;
...
*i = 10;
```