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

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

needs of programmers

programming language

computer architecture

↑ huge design space by shifting abstraction level

... and there are many further dimensions in design space
History of Programming Languages

<table>
<thead>
<tr>
<th>Language</th>
<th>Year</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>FORTRAN</td>
<td>1954–57</td>
<td>numeric computing</td>
</tr>
<tr>
<td>ALGOL 60</td>
<td>1958–60</td>
<td>numeric computing</td>
</tr>
<tr>
<td>COBOL</td>
<td>1959–60</td>
<td>business data processing</td>
</tr>
<tr>
<td>LISP</td>
<td>1956–62</td>
<td>symbolic computing, functional programming</td>
</tr>
<tr>
<td>PL/I</td>
<td>1963–64</td>
<td>general purpose</td>
</tr>
<tr>
<td>BASIC</td>
<td>1964</td>
<td>educational, interactive</td>
</tr>
<tr>
<td>SIMULA 67</td>
<td>1967</td>
<td>simulation</td>
</tr>
<tr>
<td>Pascal</td>
<td>1971</td>
<td>educational, general purpose</td>
</tr>
<tr>
<td>PROLOG</td>
<td>1972</td>
<td>artificial intelligence, logic-oriented programming</td>
</tr>
<tr>
<td>C</td>
<td>1972</td>
<td>systems programming</td>
</tr>
<tr>
<td>CLU</td>
<td>1974–77</td>
<td>ADT programming</td>
</tr>
<tr>
<td>Ada</td>
<td>1979</td>
<td>general purpose, embedded systems</td>
</tr>
<tr>
<td>Smalltalk</td>
<td>1971–80</td>
<td>personal computing, object-oriented programming</td>
</tr>
<tr>
<td>C++</td>
<td>1984</td>
<td>general purpose, object-oriented programming</td>
</tr>
<tr>
<td>Eiffel</td>
<td>1988</td>
<td>general purpose, object-oriented programming</td>
</tr>
<tr>
<td>Perl</td>
<td>1990</td>
<td>scripting language</td>
</tr>
<tr>
<td>Java</td>
<td>1995</td>
<td>network computing?, general purpose</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td>what’s the character of many newer languages?</td>
</tr>
</tbody>
</table>
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 program 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;  \hspace{1cm} r-value of y copied into memory referenced by l-value of x
x = &y; \hspace{1cm} l-value of y copied into memory referenced by l-value of x
x = 3;  \hspace{1cm} literal 3 copied into memory referenced by l-value of x
3 = y;  \hspace{1cm} error, literal 3 must not be used as l-value
Syntax Specification

\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 ::= \text{if} \ \langle \text{expr} \rangle \ \{ \langle \text{statement} \rangle^* \} \mid
\text{if} \ \langle \text{expr} \rangle \ \{ \langle \text{statement} \rangle^* \} \ \text{else} \ \{ \langle \text{statement} \rangle^* \}
\langle \text{loop} \rangle ::= \text{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 
\langle \text{expr} \rangle \langle \text{operator} \rangle \langle \text{expr} \rangle
\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 ::= a \mid b \mid c \mid \ldots \mid z
\langle \text{digit} \rangle ::= 0 \mid 1 \mid 2 \mid \ldots \mid 9
Abstract vs. Concrete Syntax, Pragmatics

\[ \text{while}(x == y) \]
\[ x = \ldots; \]
\[ \text{if } x = y \text{ then} \]
\[ \ldots \]
\[ \text{end} \]

\[ \text{while}(x == y) \]
\[ \{ \]
\[ \ldots \]
\[ \} \]

\[ \text{begin} \]
\[ \ldots \]
\[ \text{end} \]

\[ \text{while } x = y \text{ do} \]
\[ \ldots \]
\[ \text{end} \]

\[ \text{else} \]
\[ \ldots \]
\[ \text{end} \]
Static and Dynamic Semantics

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

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

meaningful well-formed programms usually distinguished from meaningless
programs \texttt{before} program execution

\textbf{static semantics} defines which well-formed programs are meaningful
\textbf{dynamic semantics} defines effects of executing meaningful programs
Formal Semantics

semantics \neq \text{implementation}

formal specification of semantics given in \textit{meta-language}

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

\textbf{operational semantics:} pseudo-implementation on virtual machine

\textbf{axiomatic semantics:} state transitions specified in logics

\textbf{denotational semantics:} state and state transitions specified by functions
Axiomatic Semantics

{Precondition} Statement {Postcondition}

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

{true}
if x \geq y then max = x; else max = y;
{(max = x \text{ and } x \geq y) \text{ or } (max = y \text{ 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} \quad (P \text{ is a loop invariant})
Denotational Semantics

dsem(S1; S2; mem) = mem2 if dsem(S1; mem) = mem1 and
                dsem(S2; mem1) = mem2

dsem(if B then L1 else L2, mem) = mem1 where
                mem1 = dsem(L1, mem) if mem(B) = true;
                mem1 = dsem(L2, mem) if mem(B) = false

dsem(while B do L, mem) =
                mem if mem(B) = false;
                dsem(while B do L, dsem(L, mem)) if mem(B) = 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: 
```pascal
var x, y: integer;
c: character;
```

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

```pascal
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
I-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.

```plaintext
type PI = ^integer;
var pxi: PI;

... new(pxi);
pxi ^= 0;

int x = 0;
int* px = &x;
```

```plaintext
type PPI = ^PI;
var ppxi: PPI;

... new(ppxi);
ppxi ^= pxi;

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);
```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;  
float f, g;  
...
swap(i, j);  
swap(f, g);
Overloading and Aliasing

**overloading:** one name refers to several entities

**aliasing:** several names refer to the same object

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

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