Principles of Programming Languages

II.2 Imperative Programming Model

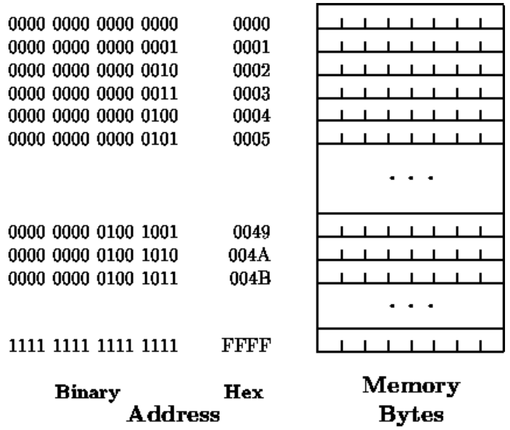
Dr. Herbert Prähofer
Institute for System Software
Johannes Kepler University Linz

Imperative Programming Model

- Abstraction of von Neumann computers
  - programs
    - sequence of statements
    - incl. control structures
  - memory
    - variables, pointers and references for addressing memory locations
    - data structures for structuring memory

```
START: JUMP    LOOP
RSV:   0000
LSV:   0000
RMP:   0000
LMP:   0000
OFF:   0000
ON:    0100
LOOP:  LOAD    1      RSV
       LOAD    2      LSV
       SUB     1   2  3
       LOAD    1      OFF
       LOAD    2      ON
       BRANCH  3      RGT
LFT:   STORE   2      RMP
       STORE   1      LMP
       JUMP    LOOP
RGT:   STORE   2      LMP
       STORE   1      RMP
       JUMP    LOOP
```

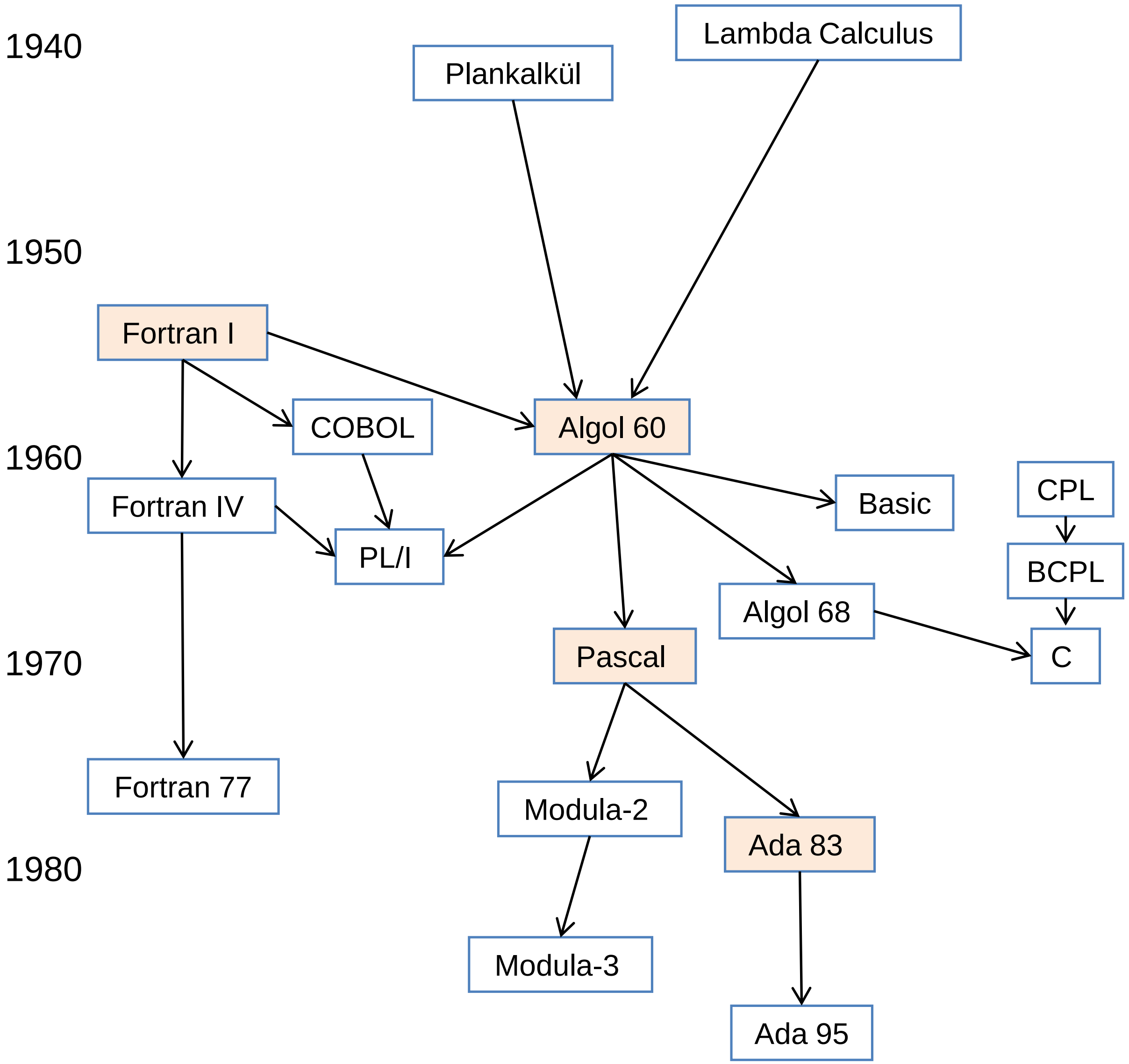
- Program execution based on state transitions
  - values in memory represent state
  - assignment causes state changes in memory



II.2 Imperative Languages

- History
- Conceptional Model and Operational Semantics
- Summary

History of Imperative Languages

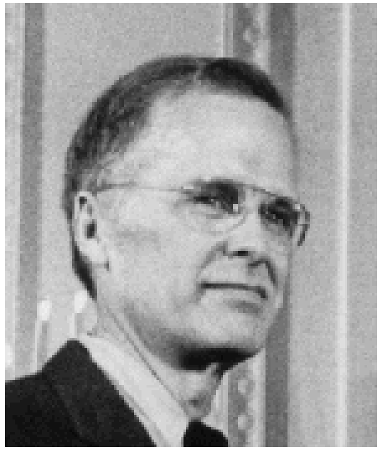


Historical milestones

- ☐ 1945: German Konrad Zuse develops Plankalkül
- ☐ 1954: 1st high-level programming language by John Backus at IBM
- ☐ 1960: international team defines language specification Algol 60
- ☐ 1964: IBM published PL/I with the goal to become a common language for scientific and business programming; PL/I finally failed its goals
- ☐ 1970: Niklaus Wirth releases Pascal language together with an implementation
- ☐ 1972-73: Dennis Ritchie defined C as the programming language for the Unix OS
- ☐ 1983: Ada was released to become the standard language of the US Department of Defence

Fortran

- Formula Translating System
- Developed 1954 under lead of John Backus at IBM.
- 1st higher programming language commonly used
- For mathematical scientific computing
- Close to target computer system IBM 704 (to allow efficient code)



John Backus

Basic language features

- ☐ Numeric data types : INTEGER, REAL, COMPLEX
- ☐ Arrays with indexing, e.g., A(I, J)
- ☐ Arithmetic expressions (formulae), e.g. I * J + K
- ☐ Control structures IF and DO
- ☐ GOTOs
- ☐ Subroutines (procedures)
- ☐ Input and output commands

Specific properties

- ☐ Call-by-reference parameter passing
- ☐ Implicit declarations of variables based on naming convention

Fortran Example

Average of values in an array

```
    SUBROUTINE AVRG(A, N, RES)
      REAL A(0:N), RES
      SUM = 0.0
      DO 1, I = 0, N-1
        SUM = SUM + A(I)
1     CONTINUE
      IF N.GT.0 GOTO 2
      RES = 0.0
      RETURN
2     RES = SUM / N
      RETURN
    END
```

Call-by-reference parameter passing

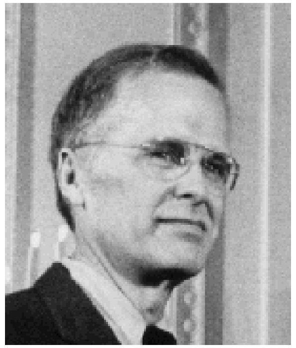
loop from 1 bis N-1

IF comparing number and jump if true

Comparison operators .EQ., .NE., .LT., ...

Implicit declarations of SUM und I
- I to N: INTEGER
- otherwise: REAL

Algol 60



John Backus
IBM

Friedrich Bauer
TU München

John McCarthy
MIT

Peter Naur
RZ Kopenhagen

Alan Perlis
CMU

Heinz Rutishauser
ETH Zürich

Algorithmic Language 1960

- Committee under lead of ACM und GAMM (Ges. für angewandte Math. u. Mechanik).
- One of the most influential languages in history

Concepts

- Block structure
- Nested statements
- Recursion
- Boolean expressions
- Static scoping and dynamic extent
- Parameter passing
  - call-by-name
  - call-by-value
- Procedure parameters (form of lambda functions)
- Dynamic arrays (allocated on stack)
- Variable declarations required
- Strong static typing
- Run-time checks
- Formal syntax definition in BNF
- Simple and concise (language report 16 pages)

Algol Example

```
comment A sample program;
begin
  real procedure apply (f) to: (a) within: (low, high);
    real procedure f;
    array a;
    integer low, high;
  begin
    real sum; integer i;
    sum := 0;
    for i := low step 1 until high do
      sum := sum + f(a[i]);
    apply = sum;
  end;

  real procedure square(x);
    value x;
    integer x;
  begin square := x * x
  end;

  integer array b[0:100];
  ...
  res := apply (square) to: (b) within: (0, 10);
  ...
end
```

Composed procedure names

procedure as parameters

Array of varying size
(depending on parameter)

return value

call-by-value

procedure parameter

call-by-name

Algol Call-by-Name

see Lambda Calculus

- Idea of **call-by-name** was **inlining**
  ➔ substituting parameters by argument expressions
  ➔ also includes **call-by-reference**

```
procedure Inc  (n);
   integer n;
   begin
      n := n + 1;
   end;
```

```
Inc (x);
```

Substitution
n by **x**

```
x := x + 1;
```

```
Inc (a[k]);
```

Substitution
n by **a[k]**

```
a[k] := a[k] + 1;
```

Pascal

Niklaus Wirth

1968-72 by Niklaus Wirth at ETH Zürich
Originally as educational language
Some cleanup and extension of Algol60

One of the first languages for microcomputers (e.g. Apple II).

Still alive in Borland Delphi

Innovations

- New types
  - Enumerations, Subranges
  - Records (with variants)
  - Sets, Files
  - Typsafe pointers
- Control statements for structured programming (while, for, repeat, case)
- Named constants
- Dynamic memory allocations
- Call-by-reference and call-by-value
- Bytecode (P-Code)

Restrictions (original release)

- No dynamic arrays
- Declarations before usage
- Separate declaration sections in procedure
- No string type

Pascal Example

```
program Sample (input, output);
   const len = 100;
   type Table = array [0 .. len-1] of integer;
   var tab: Table;
      i, val, pos: integer;

   procedure Find(tab: Table; x: integer; var pos: integer);
      var i, j, m: integer;
   begin
      i := 0; j := len-1; pos := -1;
      while i <= j do begin
         m = (i + j) div 2;
         if tab[m] = x then begin pos := m; return end
         else if x < tab[m] then j := m - 1
         else { x > tab[m] } i := m + 1
      end
   end;

begin
   for i := 0 to len-1 do read(tab[i]);
   read(val);
   Find(tab, val, pos);
   writeln("index = ", pos);
end.
```

Constant declaration
Type declaration
Variable declaration

Reference parameter

Control structure for
structured programming

Comment

ADA

Jean Ichbiah

- 1979 - 1982 developed by Jean Ichbiah (Honeywell Bull)
- Winner of call of US Department of Defense (DoD) for a universal language for embedded systems
- Named after Ada, Countess of Lovelace (1815 - 1852), working with Lord Byran as a „first programmer“

Innovations

- Module concept (packages)
- Parallel processes (Tasks)
- Message communication
- Exception handling
- Generics

Most of it was already known from other languages

Characteristics

- Very complex language
- Compiler have to be validated
- Standardized by DoD (no dialects allowed)
- Still in use within DoD

Ada Example [1/2]

- Module concept
  - Packages with type and procedure definitions
  - Data encapsulation by public and private sections plus implementation

```
package Stacks is
  type Stack is limited private;
  procedure Push(s: in out Stack; x: in Integer);
  procedure Pop(s: in out Stack; x: out Integer);
  function  Contains(s: in Stack; x: in Integer) return boolean;
private
  type Values is array (1..100) of integer;
  type Stack is
    record
      data: Values;
      top: integer range 0..100 := 0;
    end record;
end Stacks;
```

package declaration

data type (definition is hidden)

public procedureinterfaces

private section
with hidden data type definitions

ADA EXAMPLE [2/2]

```
package body Stacks is

  procedure Push(s: in out Stack; x: in integer) is
  begin
    s.top := s.top + 1;
    s.data(s.top) := x;
  end Push;

  procedure Pop(s: in out Stack; x: out integer) is
  begin
    x := s.data(s.top);
    s.top := s.top - 1;
  end Pop;

  function Contains(s: in Stack; x: in integer)
    return boolean is
  begin
    for i in 1..s.top loop
      if x = s.data(i) then return true; end if;
    end loop;
    return false;
  end Contains;

end Stacks;
```

package implementation

package import

Application

```
with Stacks;
use Stacks;
procedure Test is
  s: Stack;
begin
  Push(s, 3); ...
  Pop(s, x); ...
end;
```

II.2 Imperative Languages

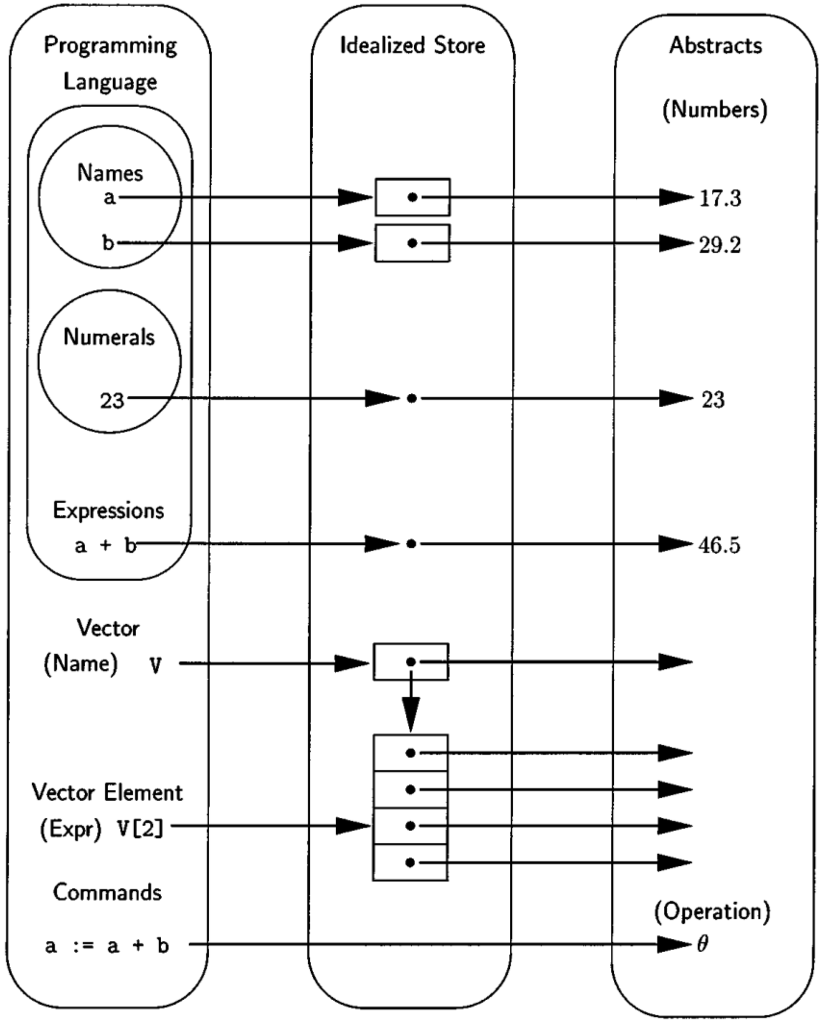
- History
- Conceptional Model and Operational Semantics
- Summary

Conceptional Model

Based on:
Christopher Strachey. Fundamental Concepts in Programming Languages.
Higher Order Symbol. Comput. 13, 2000.

Basic concepts and terms

- Value
  - element of the domain of values
- Store
  - stores values in locations
- Address:
  - some location in store which can hold a content
- State
  - A mapping of addresses to current values
- Name:
  - an identifier referring to a location in store
- Literal:
  - language representation of a value
- Expression:
  - something which can be evaluated to a value
- Statement:
  - something which can be executed
  - and possibly causes a state change
- Assignment statement:
  - a statement changing the content in a location in the store



Core of Imperative Languages

Syntax

- Literals, e.g. for integers, Booleans, etc.

$$L = L_{\mathrm{Int}} \mid L_{\mathrm{Bool}} \mid \dots \;.$$
$$L_{Int} = \dots \mid "-2" \mid "-1" \mid "0" \mid "1" \mid "2", \dots \;.$$
$$L_{Bool} = \; "\mathbf{true}" \mid "\mathbf{false}" \;.$$

- Variable names (identifiers)

$$N = Letter \{ \, (Letter \mid Digit) \, \} \;.$$
$$Letter = "a" \mid "b" \mid \dots \;. \qquad Digit = "0" \mid \dots \;.$$

- Expressions

$$Expr = L \mid N \mid Uop\ Expr \mid Bop\ Expr\ Expr$$
$$Uop \; = "not" \mid "neg"$$
$$Bop \; = "+" \mid "-" \mid "*" \mid "/" \mid "and" \mid "or" \;.$$

- Statements

$Stmt = \; "SKIP" \quad \mid$	no-op statement
$L\text{-}Expr \; ":=" \; Expr \quad \mid$	assignment
$"IF" \; Expr \; Stmt \; "ELSE" \; Stmt \quad \mid$	if-statement
$"WHILE" \; Expr \; Stmt \quad \mid$	while-statement
$Stmt \; ; \; Stmt \;.$	statement sequence

Core of Imperative Languages: Assignments

An assignment

left hand side (LHS)

right hand side (RHS)

L-Expr ":=" *Expr*

changes the content of the store at a location in store

L-Expr

L-Expr = *N* | … .

- an expression giving a location in store
- result is so-called ***L-value***

Expr

- an expression giving a value
- also called *R-value*

Example:

```
x := x + 1
```

Examples of L-Exprs:

```
x
a[i]
m[i][j+i]
p.name
…
```

Conceptional Model: Basic Functions

- Value domains D is the set of all values, i.e.,

$$D = \mathit{Integers} \cup \text{Bool} \cup \ldots$$

- Literals from L represent values of value domains D

$$\mathit{value}: L \rightarrow D$$

e.g., Literal 1 represents integer value 1

- Address space

$$A \subset \{0, 1, 2, \ldots\}$$

- Program state as mapping from addresses to values

$$\mathit{state}: A \rightarrow D$$

- Mapping of L-Expr to addresses

$$\mathit{addr}: \mathit{L\text{-}Expr} \rightarrow A$$

- Reading a value from store with an L-Expr

$$\mathit{read}: \mathit{State} \times \mathit{L\text{-}Expr} \rightarrow D$$
$$\mathit{read}(\mathit{state}, n) = \mathit{state}(\mathit{addr}(n))$$

- Update store at position given by L-Expr

$$\mathit{updt}: \mathit{State} \times \mathit{L\text{-}Expr} \times D \rightarrow \mathit{State}$$
$$\mathit{updt}(\mathit{state}, n, v) = \mathit{state}'$$
$$\text{with} \quad \begin{array}{ll} \mathit{state}'(a) = v & \text{if } \mathit{addr}(n) = a \\ \mathit{state}'(a) = \mathit{state}(a) & \text{otherwise} \end{array}$$

Semantics of Expressions

Defined by Lambda Calculus

- Evaluation of expressions to value

$$\boldsymbol{eval}: \boldsymbol{Expr} \times \boldsymbol{State} \to \boldsymbol{D}$$

$$eval(e, state) = \begin{cases} value(e) & \text{if } e \in L \\ read(e, state) & \text{if } e \in L\text{-}Expr \\ app(bop, eval(a, state), eval(b, state)) & \text{if } e = bop\ a\ b \\ app(up, eval(a, state)) & \text{if } e = uop\ a \end{cases}$$

Strict execution semantics

- with

$$app: Bop \times D \times D \to D$$
$$app: Uop \times D \to D$$

is function application (beta-reduction) as defined by Lambda Calculus

Semantics of Programming Languages

There are different ways to specify the semantics of programming languages

- **Operational Semantics**
  - ☐ program specified as a transition system

more amenable for imperative languages

- **Denotational Semantics**
  - ☐ program specified as a function from input to output
  - ☐ based on semantic domains

- **Axiomatic Semantics**
  - ☐ program specified based on Hoare clauses
  - ☐ with pre- and postconditions

Structural Operational Semantics

Also called: ***Small-step semantics***

Transition rules

- Effect of statement execution on program state and statement sequence

$$\langle S_1, state_1 \rangle \to \langle S_2, state_2 \rangle$$

transition

Execution of some statement S_1 in state $state_1$ results in some statement S_2 and state $state_2$

- Rule with premise and conclusion

$$\frac{\langle S_1, state_1 \rangle \to \langle S_2, state_2 \rangle}{\langle S_3, state_3 \rangle \to \langle S_4, state_4 \rangle}$$

If $\langle S_1, state_1 \rangle$ transits to $\langle S_2, state_2 \rangle$ then $\langle S_3, state_3 \rangle$ transits to $\langle S_4, state_4 \rangle$

Operational Semantics: Transition Rules

Syntax | Semantics

Stmt =

"SKIP" |

$\langle \mathbf{SKIP}, state \rangle \rightarrow \langle \varnothing, state \rangle$

with ∅ is empty program

L-Expr ":=" *Expr* |

$\langle n \ := e, state \rangle \rightarrow \langle \varnothing, updt(state, n, eval(e, state)) \rangle$

"IF" *Expr Stmt* "ELSE" *Stmt* |

$\langle \mathbf{IF}\ B\ S_1\ \mathbf{ELSE}\ S_2, state \rangle \rightarrow \langle S_1, state \rangle$ if $eval(B, state)$

$\langle \mathbf{IF}\ B\ S_1\ \mathbf{ELSE}\ S_2, state \rangle \rightarrow \langle S_2, state \rangle$ if $\neg eval(B, state)$

note statement sequence!

"WHILE" *Expr Stmt* |

$\langle \mathbf{WHILE}\ B\ S, state \rangle \rightarrow \langle S; \mathbf{WHILE}\ B\ S, state \rangle$ if $eval(B, state)$

$\langle \mathbf{WHILE}\ B\ S, state \rangle \rightarrow \langle \varnothing, state \rangle$ if $\neg eval(B, state)$

Stmt ; *Stmt*

.

$$\frac{\langle S_1, state_1 \rangle \rightarrow \langle S_2, state_2 \rangle}{\langle S_1; S, state_1 \rangle \rightarrow \langle S_2; S, state_2 \rangle}$$

$\langle \varnothing\ ; S, state_1 \rangle \rightarrow \langle S, state_1 \rangle$

JKU

Operational Semantics

Function exec realizes operational semantics

$\boldsymbol{exec} : \boldsymbol{Stmt} \times State \to \boldsymbol{Stmt} \times State$
$exec(S_1, state_1) = (S_2, state_2)$ **with** $\langle S_1, state_1 \rangle \to \langle S_2, state_2 \rangle$ **according to above rules**

Conceptional Model: Procedures

Procedures as extensions of lambda-functions with statements

- Data type **Unit** with single value **()**

$$D = Integers \cup Bool \cup \ldots \cup Unit$$
$$Unit = \{\, () \,\}$$

Unit corresponds to void

- Return statement

$$Stmt = \ \ldots \mid \text{"RET "}\ Expr\,.$$

- Procedures as lambda-functions with statement body

$$Proc = (\lambda\ v : t_1\ .\ S) : t_1 \text{->} t_2 \qquad S \in Stmt$$

- Statements as expressions

$$Expr = \ \ldots \mid \text{"("}\ Proc\ Expr\ \text{")"} \mid Stmt\,.$$

Statemetns can return values !

- Expressions as statements

$$Stmt = \ \ldots \mid Expr\,.$$

Expressions can be executed as statements !

Conceptional Model: Procedures

- Evaluation of statements as expressions

$\boldsymbol{eval}: \boldsymbol{Expr} \times \boldsymbol{State} \rightarrow \boldsymbol{D}$

$eval(e, state) =$

$\quad \ldots$

$\quad eval(s, state) \quad = ()$ for $s \in \{$ SKIP, :=, IF, WHILE $\}$

$\quad eval(S_1; S, state_1) \quad = eval(S, state_2)$ where $exec(S_1; S, state_1) = (S_2; S, state_2)$

$\quad eval(RET\ e, state) \quad = eval(e, state)$

$\quad eval(((\lambda\ x.\ s)\ A), state) = eval(s\,[A/x], state)$

Beta-reduction for parameter passing

- Execution of expressions as statements

$\boldsymbol{exec}: \boldsymbol{Stmt} \times \boldsymbol{State} \rightarrow \boldsymbol{S} \times \boldsymbol{State}$

Beta-reduction for parameter passing

$\ldots$

$exec(RET\ e, state) = exec(e, state)$

$exec(((\lambda\ x.\ s)\ A), state) = exec(s\,[A/x], state)$

$exec(n := e, state_1) = (\varnothing, upd\boxed{t(state_2, n, eval(e, state_1)})$ where $exec(e, state_1) = (\varnothing, state_2)$

$\ldots$

Evaluation of expression e can have side effects resulting in new $state_2$!

Conceptional Model: Procedures

Example: procedure incr

$$\mathbf{incr} = \lambda\, x\ .\ x := x + 1\ ;\ \mathrm{RET}\ x$$

- Procedure application

$$y := 1\ ;$$

$$z := \mathrm{incr}\ y$$

- Execution + evaluation

$(z := \boxed{\mathrm{incr}\ y}, \{(y, 1), \ldots\})$ ➔ eval: *incr*

➔ $(z := \boxed{(\lambda\, x\ .\ x := x + 1;\ \mathrm{RET}\ x)\ \ y}, \{(y, 1), \ldots\})$ ➔ beta-reduction (call-by-name)

➔ $(z :=\ y := \boxed{y} + 1\ ;\ \mathrm{RET}\ y\ , \{(y, 1), \ldots\})$ ➔ eval: y

➔ $(z := (\,y := \boxed{1 + 1}\ ;\ \mathrm{RET}\ y\,), \{(y, 1), \ldots\})$ ➔ eval: $\mathbf{1 + 1}$

➔ $(z := (\boxed{y := 2};\ \mathrm{RET}\ y\,), \{(y, 1), \ldots\})$ ➔ exec: $y := \mathbf{2}$

➔ $(z := \boxed{\mathrm{RET}\ 2}, \{(y, 2), \ldots\})$ ➔ exec+eval: $\mathbf{RET\ 2}$

➔ $(\boxed{z := 2}, \{(y, 2), \ldots\})$ ➔ exec: $z := \mathbf{2}$

➔ $(\emptyset, \{(y, 2), (z, 2), \ldots\})$

Referential Transparency

An important property of an expression is

Referential Transparency

*An expression is **referential transparent** if the value of an expression*
*which contains **sub-expressions** is **ONLY DEPENDENT** on the **values** of the sub-expressions!*

Any other property such as

- its internal structure,
- the number and nature of its components,
- the order in which they are evaluated

!
From left to right,
or right to left
or in parallel

are irrelevant

➔ Referential transparent expressions are not dependent on side effects!

Referential Transparency: Functional vs. Imperative

Function plus1

$$plus1 = \lambda\, x\, .\, x + 1$$

- Two equal function applications

$$(plus1\ x) + (plus1\ x)$$

Function applications
- in any order
  - left to right or opposite
  - on demand
  - parallel
- memorization (cache for computed values)

Procedure incr

$$incr = \lambda\, x\, .\ x := x + 1\ ;\ RET\ x$$

- Two equal procedure calls give different results

$$(incr\ x) + (incr\ x)$$

Procedure calls
- inherent sequential
- no parallel execution possible
- no memorization possible

II.2 Imperative Languages

- History
- Conceptional Model and Operational Semantics
- Summary

SUMMARY

- Imperative programming model distinguishes between
  - statements
  - expressions

- Execution
  - statements result in state transitions (changes in memory)
  - expressions result in values

- Procedure with return values combine statements and expressions

- Referential transparency is an important property of expressions
  - Referential transparent expressions are independent of side effects
  - Referential transparency allows :
    - evaluation of expressions in any order and in parallel
    - memorization of function values