


Formal
Methods
& Tools


University of Twente
The Netherlands


Run-Time Organization

Vertalerbouw **HC5**

VB HC5

Arend Rensink
kamer: Zilverling 5090
telefoon: 4862
email: rensink@cs.utwente.nl


Original design: Theo Ruys
University of Twente
Department of Computer Science
Formal Methods & Tools


© Theo Ruys, Arend Rensink

What have you seen last time?

- Antlr string grammars
 - Lexer and parser rules
 - Visualisation of syntax trees
- Antlr tree grammars
 - Checker: use tree walking to build/check symbol table
 - Interpreter: use tree walking to execute program
- Antlr tips on associativity, precedence, greedy parsing
 - (Not covered during lecture; see slides)


VB HC 1
Ch. 1 - Introduction
2


© Theo Ruys, Arend Rensink

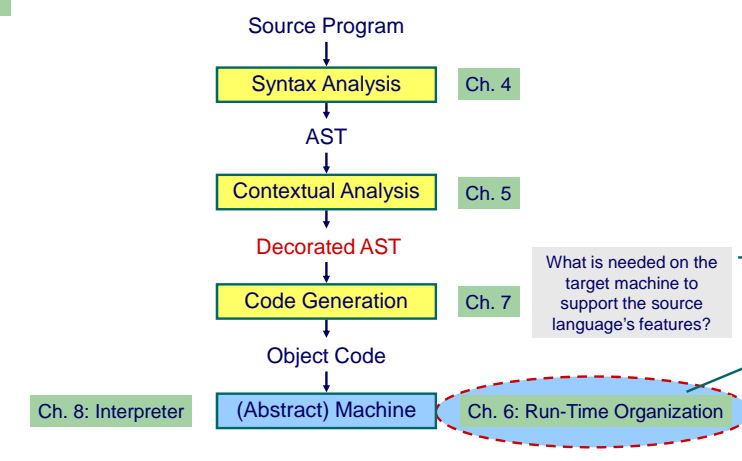
Overview of Lecture 5

- Ch 6 – **Run-Time Organization**
 - 6.1 Data representation
 - 6.2 Expression evaluation
 - 6.3 Static storage allocation
 - 6.4 Stack storage allocation
 - 6.5 Routines
 - 6.6 Heap storage allocation
 - 6.7 Run-time organization of OO languages
 - 6.8 TAM Abstract Machine

VB HC 5
Ch. 6 - Run-Time Organization
3


© Theo Ruys, Arend Rensink

Compiler Phases



```

graph TD
    SP[Source Program] --> SA[Syntax Analysis]
    SA --> AST[AST]
    AST --> CA[Contextual Analysis]
    CA --> DAST[Decorated AST]
    DAST --> CG[Code Generation]
    CG --> OC[Object Code]
    OC --> AM["(Abstract) Machine"]
    AM --> I[Interpreter]
  
```

Ch. 4: Syntax Analysis

Ch. 5: Contextual Analysis

Ch. 7: Code Generation

Ch. 8: Interpreter

Ch. 6: Run-Time Organization

What is needed on the target machine to support the source language's features?

VB HC 5
Ch. 6 - Run-Time Organization
4

© Theo Ruys, Arend Rensink

Run-time Organization (1)

- A compiler translates a *high-level language program* into an equivalent *low-level language program*.
- Run-time organization is concerned with representing *high-level structures* in terms of a typical *low-level machine's* **memory architecture** and **machine instructions**.

VB HC 5 Ch. 6 - Run-Time Organization 5

© Theo Ruys, Arend Rensink

Run-time Organization (2)

- Key issues** in run-time organization:
 - Data representation**
 - representing the values of each type of the source language
 - Expression evaluation**
 - evaluation of expressions, intermediate results
 - Storage allocation**
 - storing variables: global, local and heap
 - Routines**
 - implementing procedures, functions, parameter passing
 - Run-time organization of **OO languages**
 - objects, methods, inheritance

VB HC 5 Ch. 6 - Run-Time Organization 6

© Theo Ruys, Arend Rensink

Data representation

- Uniqueness**
 - each value should always have the same representation
- Non-confusion**
 - Different values of a type should have distinct representations
 - (not true for representation of real numbers)
- Constant-size**
 - All values of a type should occupy the same amount of space
 - (not always true for representation of UTF8 characters)
- Indirection:**
 - direct:** content of a variable x can be directly accessed
 - indirect:** content of a variable x has to be accessed via a pointer/handle

⇒ **key design** decision in run-time organization

VB HC 5 Ch. 6 - Run-Time Organization 7

© Theo Ruys, Arend Rensink

Indirection

Direct representation

x: bit pattern of x

- efficient access** (no need to follow pointers)
- efficient storage** (stack rather than heap)
- Pascal, C/C++

Indirect representation

x: → bit pattern of x
pointer

- for types with **varying size of representation**
 - dynamic arrays
 - recursive types
 - objects
- pointers/handles have **constant representation**
- ML, Haskell, Prolog

Many language implementations use mixed strategy!

VB HC 5 Ch. 6 - Run-Time Organization 8

© Theo Ruys, Arend Rensink

Primitive Types

- Notation:
 - $\#[T]$ number of different values in T
 - $\text{size}[T]$ minimal size (in bits) to represent a value of T
- Primitive type: type that cannot be decomposed into smaller types.
 - Triangle: Integer Char Boolean

	$\#[T]$	$\text{size}[T]$	representation
Boolean	2	1	0 and 1
Integer	2^{16} or 2^{32}	16 / 32	2-complement
Char	2^8 or 2^{16}	8 / 16	ASCII/Unicode
float	infinite*	32 / 64	approximation

VB HC 5 Ch. 6 - Run-Time Organization 9

© Theo Ruys, Arend Rensink

Records (1)

```

type Date ~ record
  y : Integer,
  m : Integer,
  d : Integer
end;
type Details ~ record
  female : Boolean,
  dob : Date,
  status : Char
end;
var today: Date;
var my: Details

```

The usual representation of a record value is the concatenation of individual representations of each of its component values.

Here addressing of the memory is on word-level (e.g., two bytes). Hence, a boolean is represented by a word.

today.y

today.m

today.d

my.female

my.dob.y

my.dob.m

my.dob.d

my.status

VB HC 5 Ch. 6 - Run-Time Organization 10

© Theo Ruys, Arend Rensink

Records (2)

- Most real machines have alignment restrictions, e.g.
 - memory can only be addressed as words (4/8 bytes)
 - memory addressing on word-level is faster
- Storing records in memory might be suboptimal (w.r.t. space).

```

type Date ~ record
  y : Integer,
  m : Integer,
  d : Integer
end;
type Details ~ record
  female : Boolean,
  dob : Date,
  status : Char
end;
var today: Date;
var my: Details

```

4 byte (32-bit) addressing

f			
y ₁	y ₂		
m ₁	m ₂		
d ₁	d ₂		
s			

1 byte (8-bit) addressing

f
y ₁
y ₂
m ₁
m ₂
d ₁
d ₂
s

VB HC 5 Ch. 6 - Run-Time Organization 11

© Theo Ruys, Arend Rensink

Disjoint unions

- A disjoint union is like a record, but the elements never exist at the same time. A type tag determines which of the elements is currently valid.

```

type Number =
  record
    case (discrete: Boolean) of
      true: (i: Integer);
      false: (r: Real)
    end;
var num: Number

```

num.discrete

num.i

num.discrete

num.r

VB HC 5 Ch. 6 - Run-Time Organization 12

© Theo Ruys, Arend Rensink

Arrays (1)

An **array** is a *composite data type* (like a record).

An **array value** consists of multiple values *of the same type* (unlike a record).

- static arrays**: size (i.e. number of elements) is known at compile time.
- dynamic arrays**: size can not be known at compile time (e.g. the number of elements may vary at run-time).
 - Not in Triangle

```

type Name = array 4 of Char;
var me: Name;
var full: array 2 of Name;

```

me[0]	'n'
me[1]	'o'
me[2]	'o'
me[3]	't'

full[0][0]	'm'
full[0][1]	'i'
full[0][2]	'e'
full[0][3]	's'
full[1][0]	't'
full[1][1]	'e'
full[1][2]	'u'
full[1][3]	'n'

VB HC 5 Ch. 6 - Run-Time Organization 13

© Theo Ruys, Arend Rensink

Arrays (2)

- Dynamic arrays**: size is not known at compile time
 - Has to be stored as part of value

```

char[] buffer;
buffer = new char[len];

```

(Java)

buffer.length 8
buffer.start 8

possible implementation 1

8	buffer.length
'C'	buffer[0]
'o'	buffer[1]
'm'	buffer[2]
'p'	buffer[3]
'i'	buffer[4]
'i'	buffer[5]
'e'	buffer[6]
'r'	buffer[7]

possible implementation 2

'C'	buffer[0]
'o'	buffer[1]
'm'	buffer[2]
'p'	buffer[3]
'i'	buffer[4]
'i'	buffer[5]
'e'	buffer[6]
'r'	buffer[7]

VB HC 5 Ch. 6 - Run-Time Organization 14

© Theo Ruys, Arend Rensink

Recursive Types

- A **recursive type** is a type which is defined in terms of itself.
 - Values of recursive type T have **components** that are themselves of type T.

```

type List = ^Node;
Node = record
  data: integer;
  next: List;
end;
var li: List;

```

Pascal-notation for: **pointer to**.
Pointer usually occupies a machine word (e.g.:16/32/64 bits.)

li → 11 → 5 → 2009 → •

VB HC 5 Ch. 6 - Run-Time Organization 15

© Theo Ruys, Arend Rensink

Run-time Organization

- Key issues** in run-time organization:
 - Data representation**
 - representing the values of each type of the source language
 - Expression evaluation**
 - evaluation of expressions, intermediate results
 - Storage allocation**
 - storing variables: global, local and heap
 - Routines**
 - implementing procedures, functions, parameter passing
 - Run-time organization of **OO languages**
 - objects, methods, inheritance

VB HC 5 Ch. 6 - Run-Time Organization 16

© Theo Ruys, Arend Rensink

Expression Evaluation

- Consider the following Triangle expressions

$$(0 \leq i) \wedge (i < n)$$

$$a*a + 2*a*b - 4*a*c$$
- The low-level machine typically has instructions for addition, multiplication, division, etc.
 - These instructions work on two operands at a time
- Where to store the intermediate results?
 - Stack machine: post-fix evaluation on the stack (easy)
 - Register machine: using registers (more work)
- The Triangle Abstract Machine (TAM) is a stack machine.

VB HC 5
Ch. 6 - Run-Time Organization
17

© Theo Ruys, Arend Rensink

Stack machines

- Typical instructions

Instr.	Meaning
STORE a	Pop the top value off the stack and store it at address a.
LOAD a	Fetch a value from address a and push it on to the stack.
LOADL n	Push the literal value n onto the stack.
ADD	Replace the two top values on the stack by their sum.
SUB	Replace the two top values on the stack by their difference.
MUL	Replace the two top values on the stack by their product.

VB HC 5
Ch. 6 - Run-Time Organization
18

© Theo Ruys, Arend Rensink

Stack machine: Example

d := a*a + 2*a*b - 4*a*c;

LOAD a
LOAD a
MUL
LOADL 2
LOAD a
MUL
LOAD b
MUL
ADD
LOADL 4
LOAD a
MUL
LOAD c
MUL
SUB
STORE d

a

a

a*a

a*a

a

2

2*a

2*a*b

a

2

2*a

2*a*b

a

2

2*a

2*a*b

VB HC 5
Ch. 6 - Run-Time Organization
19

© Theo Ruys, Arend Rensink

Register machines

- Register machine:
 - A register machine has a fixed number of registers R1, R2, ... to store intermediate values.
- Typical instructions:

Instr.	Meaning
STORE Ri a	Store the value in Ri into memory location a.
LOAD Ri a	Load the value on memory location a into Ri.
MULT Ri x	Multiply the values in Ri and x and store the result in Ri (overwriting the old value).
SUB Ri x	Subtract the value in x from Ri and store the result in Ri.
...	

where x is either a register Ri, an address a, or a literal value L

VB HC 5
Ch. 6 - Run-Time Organization
20

5

© Theo Ruys, Arend Rensink

Expression Evaluation (5)

- Register machine code is **efficient**.
- But **compilation** to a register machine is rather **complex**:
 - We have to manage the allocation of registers (try to reuse/minimize number of registers).
 - We must assign a specific register to each intermediate result.
 - What if there are not enough registers for evaluating an expression?

Example:

```
d := a*a + 2*a*b - 4*a*c;
```

```
LOAD R1 a ; R1: a
MULT R1 a ; R1: a*a
LOAD R2 2 ; R2: 2
MULT R2 a ; R2: 2*a
MULT R2 b ; R2: 2*a*b
ADD R1 R2 ; R1: a*a+2*a*b
LOAD R2 4 ; R2: 4
MULT R2 a ; R2: 4*a
MULT R2 c ; R2: 4*a*c
SUB R1 R2 ; R1: a*a + ...
STORE R1 d ; store result
```

VB HC 5 Ch. 6 - Run-Time Organization 21

© Theo Ruys, Arend Rensink

Run-time Organization


- Key issues** in run-time organization:
 - Data representation**
 - representing the values of each type of the source language
 - Expression evaluation**
 - evaluation of expressions, intermediate results
 - Storage allocation**
 - storing variables: global, local and heap
 - Routines**
 - implementing procedures, functions, parameter passing
 - Run-time organization of **OO languages**
 - objects, methods, inheritance

VB HC 5 Ch. 6 - Run-Time Organization 22

© Theo Ruys, Arend Rensink

Storage

- Memory** on target machine
 - data store**: stack and heap
 - size of addressable words is 16, 32 or 64 bits
 - code store**: instructions
 - size of instruction words is of less importance
- In **TAM**
 - data store**: 16-bit data words
 - all primitive Triangle types represented by **one 16 bit word**
 - code store**: 32-bit code words



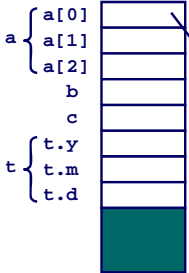
VB HC 5 Ch. 6 - Run-Time Organization 23

© Theo Ruys, Arend Rensink

Allocation for global variables

- Global variables**
 - Compiler can **compute** exactly **how much memory** is needed for **global** variables.
 - Compiler can **allocate memory** for each global variable.

```
let
  type Date = record
    y: Integer,
    m: Integer,
    d: Integer
  end;
  var a: array 3 of Integer;
  var b: Boolean;
  var c: Char;
  var t: Date
in
  ...
```



Does not have to be the first memory loc.

VB HC 5 Ch. 6 - Run-Time Organization 24

© Theo Ruys, Arend Rensink

Allocation for local variables

- Local variable **v**
 - exists \neq accessible
 - Declared inside a procedure
 - Exists during the activation of the procedure: *lifetime* of **v**

```

let
  var b: Boolean;
  var c: Char
in
  proc Y() ~
    let var d: Integer
    in ...

  proc Z() ~
    let var e: Integer
    in ... Y(); ...
  in begin
    ... Y(); ...; Z(); ...
  end
end
  
```

global variables; lifetime: throughout the program

local variable of **Y**: when procedure **Y** is active

local variable of **Z**: when procedure **Z** is active

As there can be recursive activations of a procedure, there can also be multiple 'copies' of local variables

VB HC 5 Ch. 6 - Run-Time Organization 25

© Theo Ruys, Arend Rensink

Stack storage allocation

- Observations
 - Global variables: exist throughout the program's run-time
 - Local variables: lifetime properly nested \Rightarrow stack storage allocation
- Stack frame (activation record)
 - Each procedure has a stack frame, for
 - local variables
 - administration data (return address, dynamic/static link)
 - parameters
 - When the procedure is called, a stack frame is allocated
 - When the procedure has ended, the stack frame is popped

VB HC 5 Ch. 6 - Run-Time Organization 26

© Theo Ruys, Arend Rensink

Stack organisation in TAM

```

let ...
in proc Y() ~
  proc Z() ~ .. Y()
in .. Y(); Z();
end
  
```

after start

program calls **Y**

Y has returned

program calls **Z**

Z calls **Y**

Y has returned

Z has returned

dynamic link

Special registers

SB	Stack Base
LB	Local Base
ST	Stack Top

VB HC 5 Ch. 6 - Run-Time Organization 27

© Theo Ruys, Arend Rensink

Stack frames

The dynamic link is a pointer to the base of the underlying frame (from which the current frame was called). It is the old content of LB, which will be restored at the end of the procedure.

frame

link data

static link
dynamic link
return address

local variables

Code address to which control will be returned at the end of the procedure.

VB HC 5 Ch. 6 - Run-Time Organization 28

© Theo Ruys, Arend Rensink

Stack frame allocation

- Calling a procedure

```

let ...
in
  proc Y() ~
  proc Z() ~
    in ... Y() ...
  in ...
  
```

When Y is called:

- allocate stack frame for Y
- make dynamic link point to "old" LB
- update LB (becomes the "old" ST)
- save return address (R) in the frame

When Y returns:

- update ST (to point to Y's LB)
- update LB using Y's dynamic link ("old LB")
- Set program counter to (R)

VB HC 5 Ch. 6 - Run-Time Organization 29

© Theo Ruys, Arend Rensink

Memory instructions

- General memory instructions
 - LOAD d[reg]** push mem[reg+d]
 - push the value at address **d** relative to the contents of **reg** (e.g., **SB** or **LB**) onto the stack.
 - STORE d[reg]** pop mem[reg+d]
 - pop the value on top of the stack to the address **d** relative to the contents of **reg** (e.g., **SB** or **LB**).
- Applied to variables:
 - Global variables are in the **SB** frame
 - LOAD d[SB]** and **STORE d[SB]**
 - Local variables are in the **LB** frame
 - LOAD d[LB]** and **STORE d[LB]**

VB HC 5 Ch. 6 - Run-Time Organization 30

© Theo Ruys, Arend Rensink

Addressing variables (1)

- Addressing variables

```

let
  var a: array 3 of Char;
  var b: Boolean;
  var c: Char
in
  proc Y() ~
    let var d: Integer;
    var e: Integer
  in ...

  proc Z() ~
    let var f: Integer;
    var g: Char
  in
    ... Y(); ...
  in begin
    ... Y(); ...; Z(); ...
  end
end
  
```

link data

frame		
static link	dynamic link	return address

local variables

var	size	address
a	3	0[SB]
b	1	3[SB]
c	1	4[SB]
d	1	3[LB]
e	1	4[LB]
f	1	3[LB]
g	1	4[LB]

Due to the link data (3 words) in a procedure's frame, the first local variable is at relative address 3.

VB HC 5 Ch. 6 - Run-Time Organization 31

© Theo Ruys, Arend Rensink

Addressing variables

```

program
  D
  procedure P
    D
    procedure PP
      D
      proc PPP
        begin C end
      begin C end
    end
  procedure Q
    D
    begin C end
  begin C end
end
  
```

- Nested block structure
 - PPP can access variables of PPP, PP, P and the global variables
 - Using **d[SB]** and **d[LB]**, we can only access local variables in PPP and the global variables
 - But if we know where PPP's enclosing procedures PP and P are on the stack, we can access their local variables as well.
 - Static link:** pointer to the frame of the enclosing procedure

VB HC 5 Ch. 6 - Run-Time Organization 32

© Theo Ruys, Arend Rensink

Static links

```

let
...
proc P() ~
let
...
proc Q() ~
let
...
proc R() ~
let
in ... S()

in ... R(); ...

proc S() ~
let ...
in ... P(); ...

in ... Q(); ...

in ... P(); ...

```

Using the static link of R, we can access the variables of Q.
Using the static link of Q, we can access the variables of P.

Static link: pointer to enclosing procedure

dynamic link
static link

VB HC 5 Ch. 6 - Run-Time Organization 33

© Theo Ruys, Arend Rensink

Static links in TAM

```

let proc P()
in let proc Q()
in let proc R() ...

```

The static link is the first word of a stack frame, i.e., the location that LB points at. When R is the current frame:

- contents(LB) will give the enclosing frame of R, i.e., Q
- contents(contents(LB)) will give the enclosing frame of Q, i.e., P

display registers	SB	LB	L1	L2	L3	L4	...
			contents(LB)	contents(L1)	contents(L2)	contents(L3)	...
	points to frame with global variables	points to topmost frame R	points to frame R' which encloses R	points to frame R'' which enclose R'	points to frame R''' which encloses R''	points to frame R'''' which encloses R'''	...

VB HC 5 Ch. 6 - Run-Time Organization 34

© Theo Ruys, Arend Rensink

Stack storage allocation (11)

```

let ! level 1
var a: Integer;
proc P() ~
let ! level 2
var b: Integer;
proc Q() ~
let ! level 3
var c: Integer;
proc R() ~
let ! level 4
var d: Integer;
in ...
in ...
in ...
in ...

```

Within R, all variables a, b, c, and d are accessible. From the context analysis phase, we know the scope levels of these variables.

	level	scope	address
a	1	global	0[SB]
b	2	level(R)-2	3[L2]
c	3	level(R)-1	3[L1]
d	4	local	3[LB]

Scopes are available at compile time! Hence, the addresses of all variables can be computed at compile time.

VB HC 5 Ch. 6 - Run-Time Organization 35

© Theo Ruys, Arend Rensink

Run-time Organization

- Key issues in run-time organization:
 - Data representation
 - representing the values of each type of the source language
 - Expression evaluation
 - evaluation of expressions, intermediate results
 - Storage allocation
 - storing variables: global, local and heap
 - Routines
 - implementing procedures, functions, parameter passing
 - Run-time organization of OO languages
 - objects, methods, inheritance

VB HC 5 Ch. 6 - Run-Time Organization 36

© Theo Ruys, Arend Rensink

Routines

- Routines are the assembly language equivalent of procedures/functions/methods
- Low-level routine instructions:
 - CALL *r*** push **current address** to the call stack and jump to instruction ***r***
 - RETURN** pop **address** from the call stack and transfer control to this saved address
- Issues to consider when emulating HL procedures:
 - Calling a routine: passing arguments
 - Returning from a routine: Returning a value
 - setting up the routine's **static link**, L1, L2, etc.

⇒ This defines a "routine protocol" ("calling convention")

Dictated by target machine.

VB HC 5 Ch. 6 - Run-Time Organization 37

© Theo Ruys, Arend Rensink

Calling convention in TAM

- Routine Protocol** for machines with a **stack** (often used):
 - caller** passes arguments on top of the stack
 - routine is called; **callee** uses arguments
 - callee** replaces the arguments by the return value
 - ⇒ no bound on the number and size of the arguments
- TAM instructions
 - CALL(*reg*) *addr*** i.e., the enclosing scope
 - calls the routine at ***addr***, using the value in ***reg*** as static link
 - arguments are already on the stack
 - RETURN(*n*) *d***
 - pops an ***n*-word result** from the stack
 - pops the **topmost frame**
 - pops ***d* words of the arguments**
 - then pushes the **result** back

VB HC 5 Ch. 6 - Run-Time Organization 38

© Theo Ruys, Arend Rensink

Value parameters

- Parameters** (arguments): information between caller/callee
 - actual parameters** are used by the caller when the procedure is called
 - formal parameters** are used within the procedure
 - actual and formal parameters correspond one-to-one
- Value-parameters: passing by value** behave like local variables within the procedure
 - only a copy of the parameter is passed
 - implementation: push the value onto the stack

```
let proc sum(i:Integer, j:Integer) ~ putint(i+j);
var x: Integer
in begin
  x := 27; sum(x, 27)
end
```

VB HC 5 Ch. 6 - Run-Time Organization 39

© Theo Ruys, Arend Rensink

Variable parameters

- var-parameters: passing by reference**
 - the variable itself is passed: when the *formal parameter* changes, the *actual parameter* is changed as well
 - actual parameter* has to be a variable
 - implementation:
 - caller** pushes address of variable onto the stack
 - callee** uses indirection to get the value out of the passed argument (i.e., the address)

```
let proc S(var n:Integer, i:Integer) ~ n:=n+i;
var b: record
  y:integer, m:Integer, d:Integer
end
in begin
  b := {y~2003, m ~ 4, d ~ 10};
  S(var b.m, 6)
end
```

Instructions to load and store indirect variables:
LOADA d[reg]
STOREA d[reg]

VB HC 5 Ch. 6 - Run-Time Organization 40

© Theo Ruys, Arend Rensink

Frame layout

- Frame layout (revisited)

arguments are addressed relative to LB with negative offsets

Arguments for current procedure which were put there by the caller.

Link data

Local data, grows and shrinks during execution.

LB

ST

VB HC 5 Ch. 6 - Run-Time Organization 41

© Theo Ruys, Arend Rensink

Routines (6)

```

let var g: Integer;
func F(m: Integer, n: Integer)
: Integer ~ m*n ;
proc W(i:Integer) ~
let const s ~ i*i
in begin
putint(F(i,s));
putint(F(s,s))
end
in begin
getint(var g);
W(g+1)
end

```

g is a var-parameter

g+1 is a value-parameter

PUSH	1	- expand globals to make space for g
LOADA	0[SB]	- push the absolute address of g
CALL	getint	- read an integer into g
LOAD	0[SB]	- push the value of g
CALL	succ	- add 1 (built-in primitive operation)
CALL(SB)	W	- call W (using SB as the static link)
POP	1	- remove globals
HALT		- end the program

VB HC 5 Ch. 6 - Run-Time Organization 42

© Theo Ruys, Arend Rensink

Routines (7)

Note that arguments are directly below the current frame.

```

func F(m: Integer, n: Integer)
: Integer ~ m*n ;
proc W(i:Integer) ~
let const s ~ i*i
in begin
putint(F(i,s));
putint(F(s,s))
end

```

W:	LOAD	-1[LB]	- push the value of i
	LOAD	-1[LB]	- push the value of i
	CALL	mult	- multiply, the result will be the value of s
	LOAD	-1[LB]	- push the value of i
	LOAD	3[LB]	- push the value of s
	CALL(SB)	F	- call F (using SB as static link)
	CALL	putint	- write the value returned
	LOAD	3[LB]	- push the value of s
	LOAD	3[LB]	- push the value of s
	CALL(SB)	F	- call F (using SB as static link)
	CALL	putint	- write the value returned
	RETURN	(0) 1	- return, replacing the 1-word arg by a 0-word result
F:	LOAD	-2[LB]	- push the value of m
	LOAD	-1[LB]	- push the value of n
	CALL	mult	- multiply
	RETURN	(1) 2	- return, replacing the 2-word args by 1-word result

VB HC 5 Ch. 6 - Run-Time Organization 43

© Theo Ruys, Arend Rensink

Procedure parameters

- Procedures/functions as "first-class values"
 - procedures/functions passed as arguments
 - e.g., passing a `lessthan` function to a `sorting algorithm`
 - implementation:
 - routine is represented by a (start address, static link)-pair, typically called its **closure**
 - caller pushes procedure's **closure** on the stack

```

let
func twice(func doit(Integer x): Integer, i: Integer): Integer ~
doit(doit(i));
func double(Integer d) ~ d*2;
var x: Integer
in begin
x := twice(double, 10);
end

```

To use procedural/functional parameters, we have to use TAM's **CALLI** (call-indirect) instruction.

VB HC 5 Ch. 6 - Run-Time Organization 44

What have you seen today?

- Data representation
 - Representation of values for each source language type
- Expression evaluation
 - Register or stack machines
- Storage allocation
 - Global, local and heap
- Routines
 - Calling conventions
 - Parameter passing