

Compiler Construction

Lecture 22: Code Generation VII

(Static & Dynamic Data Structures)

Thomas Noll

Lehrstuhl für Informatik 2
(Software Modeling and Verification)

RWTH Aachen University

`noll@cs.rwth-aachen.de`

`http://www-i2.informatik.rwth-aachen.de/i2/cc08/`

Summer semester 2008

- 1 Repetition: Static Data Structures
- 2 Translation of Static Data Structures into AM Programs
- 3 A Translation Example
- 4 Dynamic Data Structures

Modified Syntax of EPL

Definition (Modified syntax of EPL)

The **modified syntax of EPL** is defined as follows (where $n \geq 1$):

$\mathbb{Z} :$ z (* z is an integer *)

$\mathbb{B} :$ $b ::= \text{true} \mid \text{false}$

$\mathbb{R} :$ r (* r is a real number *)

$\text{Con} :$ $c ::= z \mid b \mid r$

$\text{Ide} :$ I (* I is an identifier *)

$\text{Type} :$ $T ::= \text{bool} \mid \text{int} \mid \text{real} \mid I \mid \text{array}[z_1..z_2] \text{ of } T \mid$
 $\text{record } I_1:T_1;\dots;I_n:T_n \text{ end}$

$\text{Var} :$ $V ::= I \mid V[E] \mid V.I$

$\text{Exp} :$ $E ::= c \mid V \mid E_1 + E_2 \mid E_1 < E_2 \mid E_1 \text{ and } E_2 \mid \dots$

$\text{Cmd} :$ $C ::= V:=E \mid C_1;C_2 \mid$
 $\text{if } E \text{ then } C_1 \text{ else } C_2 \mid \text{while } E \text{ do } C$

$\text{Dcl} :$ $D ::= D_C \text{ } D_T \text{ } D_V$

$D_C ::= \varepsilon \mid \text{const } I_1:=c_1;\dots;I_n:=c_n;$

$D_T ::= \varepsilon \mid \text{type } I_1:=T_1;\dots;I_n:=T_n;$

$D_V ::= \varepsilon \mid \text{var } I_1:T_1;\dots;I_n:T_n;$

$\text{Pgm} :$ $P ::= D \ C$

The Modified Abstract Machine AM

Since (recursive) procedures are no longer supported, a procedure stack is not required anymore.

Definition (Modified abstract machine for EPL)

The **modified abstract machine for EPL (AM)** is defined by the **state space**

$$S := PC \times DS \times MS$$

with

- the **program counter** $PC := \mathbb{N}$,
- the **data stack** $DS := \mathbb{R}^*$, and
- the **main storage** $MS := \{\sigma \mid \sigma : \mathbb{N} \rightarrow \mathbb{R}\}$.

Definition (New AM instructions)

- **Procedure instructions** are no longer needed.
- **Transfer instructions** ($\text{LOAD}(dif, off)$, $\text{STORE}(dif, off)$) are replaced by the following instructions with the respective semantics $\llbracket O \rrbracket : S \dashrightarrow S$:

$$\begin{aligned}\llbracket \text{LOAD} \rrbracket(a, d : n, \sigma) &:= (a + 1, d : \sigma(n), \sigma) \\ &\quad \text{if } n \in \mathbb{N} \\ \llbracket \text{STORE} \rrbracket(a, d : n : r, \sigma) &:= (a + 1, d, \sigma[n \mapsto r]) \\ &\quad \text{if } n \in \mathbb{N}\end{aligned}$$

- Moreover the following **instruction for checking array bounds** is introduced:

$$\llbracket \text{CAB}(z_1, z_2) \rrbracket(a, d : z, \sigma) := \begin{cases} (a + 1, d : z, \sigma) & \text{if } z \in \{z_1, \dots, z_2\} \\ (0, d : \underbrace{\text{RTE}}_{\text{runtime error}}, \sigma) & \text{otherwise} \end{cases}$$

- 1 Repetition: Static Data Structures
- 2 Translation of Static Data Structures into AM Programs
- 3 A Translation Example
- 4 Dynamic Data Structures

Modifying the Symbol Table

$$\begin{aligned} Tab := \{st \mid st : Ide \dashrightarrow & (\{\mathbf{const}\} \times (\mathbb{B} \cup \mathbb{Z} \cup \mathbb{R})) \\ & \cup (\{\mathbf{var}\} \times Ide \times \mathbb{N}) \\ & \cup (\{\mathbf{type}\} \times \{\mathbf{bool}, \mathbf{int}, \mathbf{real}\} \times \{1\}) \\ & \cup (\{\mathbf{type}\} \times \{\mathbf{array}\} \times \mathbb{Z}^2 \times Ide \times \mathbb{N}) \\ & \cup (\{\mathbf{type}\} \times \{\mathbf{record}\} \times (Ide^2 \times \mathbb{N})^* \times \mathbb{N})\} \end{aligned}$$

Remarks:

- **Variable descriptor** ($\mathbf{var}, I, n$): type I , memory address n
- Last component of **type** entry: **memory requirement**
(base types: 1 “cell”)
- **Array descriptor** ($\mathbf{type}, \mathbf{array}, z_1, z_2, I, n$):
bounds z_1, z_2 , component type I
- **Record descriptor** ($\mathbf{type}, \mathbf{record}, I_1, J_1, o_1, \dots, I_l, J_l, o_l, n$): selector
 I_k , component type J_k , memory offset o_k
- “Indexed” table lookup: $st(I.I_k) := (J_k, o_k)$
if $st(I) = (\mathbf{type}, \mathbf{record}, \dots, I_k, J_k, o_k, \dots, n)$

Maintaining the Symbol Table I

The symbol table is again maintained by the function `update(D, st)` which specifies the update of symbol table *st* according to declaration *D*.

Maintaining the Symbol Table I

The symbol table is again maintained by the function `update( $D, st$ )` which specifies the update of symbol table st according to declaration D .

For the sake of simplicity we assume that $D = D_C D_T D_V \in Dcl$ is **flattened**, i.e., that every subtype is named by an identifier:

- If $D_T = \text{type } I_1 := T_1; \dots; I_n := T_n;$, then for every $k \in [n]$
 - $T_k \in \{\text{bool}, \text{int}, \text{real}\}$ or
 - $T_k \in \{I_1, \dots, I_{k-1}\}$ or
 - $T_k = \text{array}[z_1..z_2]$ of I_j where $j \in [k-1]$ or
 - $T_k = \text{record } J_1 : I_{j_1}; \dots; J_l : I_{j_l}$ end where $j_1, \dots, j_l \in [k-1]$
- For D_T as above, D_V must be of the form
 $D_V = \text{var } J_1 : I_{j_1}; \dots; J_k : I_{j_k};$ where $j_1, \dots, j_k \in [n]$

Maintaining the Symbol Table II

Definition 22.1 (Modified update function)

$\text{update} : Dcl \times Tab \dashrightarrow Tab$ is defined by

$$\text{update}(D_C \ D_T \ D_V, st) := \text{update}(D_V, \text{update}(D_T, \text{update}(D_C, st)))$$
$$\text{update}(\varepsilon, st) := st$$

Maintaining the Symbol Table II

Definition 22.1 (Modified update function)

update : $Dcl \times Tab \dashrightarrow Tab$ is defined by

$update(D_C \ D_T \ D_V, st) := update(D_V, update(D_T, update(D_C, st)))$

$update(\varepsilon, st) := st$

$update(\mathbf{const} \ I_1 := c_1; \dots; I_n := c_n; , st)$

$:= st[I_1 \mapsto (\mathbf{const}, c_1), \dots, I_n \mapsto (\mathbf{const}, c_n)]$

Maintaining the Symbol Table II

Definition 22.1 (Modified update function)

update : $Dcl \times Tab \dashrightarrow Tab$ is defined by

$update(D_C \ D_T \ D_V, st) := update(D_V, update(D_T, update(D_C, st)))$

$update(\varepsilon, st) := st$

$update(const \ I_1 := c_1; \dots; I_n := c_n; , st)$

$:= st[I_1 \mapsto (const, c_1), \dots, I_n \mapsto (const, c_n)]$

$update(type \ I := bool; D'_T, st) := update(type \ D'_T, st[I \mapsto (type, bool, 1)])$

$update(type \ I := int; D'_T, st) := update(type \ D'_T, st[I \mapsto (type, int, 1)])$

$update(type \ I := real; D'_T, st) := update(type \ D'_T, st[I \mapsto (type, real, 1)])$

$update(type \ I := J; D'_T, st) := update(type \ D'_T, st[I \mapsto st(J)])$

$update(type \ I := array[z_1 .. z_2] \ of \ J; D'_T, st)$

$:= update(type \ D'_T,$

$st[I \mapsto (type, array, z_1, z_2, J, k \cdot n)])$

if $st(J) = (type, \dots, n)$ and $k = z_2 - z_1 + 1$

$update(type \ I := record \ I_1 : J_1; \dots; I_l : J_l \ end; D'_T, st)$

$:= update(type \ D'_T, st[I \mapsto$

$(type, record, I_1, J_1, 0, I_2, J_2, n_1, \dots,$

$I_l, J_l, \sum_{i=1}^{l-1} n_i, \sum_{i=1}^l n_i)])$

if $st(J_i) = (type, \dots, n_i)$ for $i \in [l]$

Maintaining the Symbol Table II

Definition 22.1 (Modified update function)

update : $Dcl \times Tab \dashrightarrow Tab$ is defined by

$update(D_C \ D_T \ D_V, st) := update(D_V, update(D_T, update(D_C, st)))$

$update(\varepsilon, st) := st$

$update(const \ I_1 := c_1; \dots; I_n := c_n; , st)$

$:= st[I_1 \mapsto (const, c_1), \dots, I_n \mapsto (const, c_n)]$

$update(type \ I := bool; D'_T, st) := update(type \ D'_T, st[I \mapsto (type, bool, 1)])$

$update(type \ I := int; D'_T, st) := update(type \ D'_T, st[I \mapsto (type, int, 1)])$

$update(type \ I := real; D'_T, st) := update(type \ D'_T, st[I \mapsto (type, real, 1)])$

$update(type \ I := J; D'_T, st) := update(type \ D'_T, st[I \mapsto st(J)])$

$update(type \ I := array[z_1 .. z_2] \ of \ J; D'_T, st)$

$:= update(type \ D'_T,$

$st[I \mapsto (type, array, z_1, z_2, J, k \cdot n)])$

if $st(J) = (type, \dots, n)$ and $k = z_2 - z_1 + 1$

$update(type \ I := record \ I_1 : J_1; \dots; I_l : J_l \ end; D'_T, st)$

$:= update(type \ D'_T, st[I \mapsto$

$(type, record, I_1, J_1, 0, I_2, J_2, n_1, \dots,$

$I_l, J_l, \sum_{i=1}^{l-1} n_i, \sum_{i=1}^l n_i)])$

if $st(J_i) = (type, \dots, n_i)$ for $i \in [l]$

$update(var \ I_1 : J_1; \dots; I_n : J_n; , st) := st[I_1 \mapsto (var, J_1, 0), I_2 \mapsto (var, J_2, n_1), \dots,$

$I_n \mapsto (var, J_n, \sum_{i=1}^{n-1} n_i)]$

if $st(J_i) = (type, \dots, n_i)$ for $i \in [l]$

Example 22.2 (Modified update function)

```
Let  $D :=$  type Bool=bool; Int=int;  
      Array=array[1..20] of Bool;  
      Record=record S:Array; T:Int end;  
var x:Int; y:Array; z:Record;
```

Example 22.2 (Modified update function)

```
Let  $D :=$  type Bool=bool; Int=int;  
      Array=array[1..20] of Bool;  
      Record=record S:Array; T:Int end;  
var x:Int; y:Array; z:Record;
```

Then

$$\text{update}(D, \text{st}) = \text{st}[\begin{array}{l} \text{Bool} \mapsto (\text{type}, \text{bool}, 1), \\ \text{Int} \mapsto (\text{type}, \text{int}, 1), \\ \text{Array} \mapsto (\text{type}, \text{array}, 1, 20, \text{Bool}, 20), \\ \text{Record} \mapsto (\text{type}, \text{record}, \text{S}, \text{Array}, 0, \text{T}, \text{Int}, 20, 21), \\ x \mapsto (\text{var}, \text{Int}, 0), \\ y \mapsto (\text{var}, \text{Array}, 1), \\ z \mapsto (\text{var}, \text{Record}, 21) \end{array}]$$

Translation of Variables I

The translation employs the following auxiliary function to determine the type identifier of a given variable:

Definition 22.3 (vtype function)

The mapping

$$\text{vtype} : \text{Var} \times \text{Tab} \dashrightarrow \text{Ide}$$

is given by

$$\begin{aligned} \text{vtype}(I, \text{st}) &:= J \\ &\quad \text{if } \text{st}(I) = (\text{var}, J, n) \end{aligned}$$

$$\begin{aligned} \text{vtype}(V[E], \text{st}) &:= J \\ &\quad \text{if } \text{vtype}(V, \text{st}) = I \\ &\quad \text{and } \text{st}(I) = (\text{type}, \text{array}, z_1, z_2, J, n) \end{aligned}$$

$$\begin{aligned} \text{vtype}(V.I, \text{st}) &:= J \\ &\quad \text{if } \text{vtype}(V, \text{st}) = I' \text{ and } \text{st}(I'.I) = (J, o) \end{aligned}$$

Translation of Variables II

The function `vt` generates code for computing the memory address of a variable (and storing it on the data stack):

Definition 22.4 (Translation of variables)

The mapping

$$vt : Var \times Tab \dashrightarrow AM$$

is given by

```
vt(I, st) := LIT(n);
           if st(I) = (var, J, n)
vt(V[E], st) := vt(V, st)           % address of V
               et(E, st)             % array index
               CAB(z1, z2);         % bounds checking
               LIT(z1); SUB;        % index difference
               LIT(n); MULT;         % relative address
               ADD;                  % address of V[E]
           if vtype(V, st) = I and st(I) = (type, array, z1, z2, J, m)
           and st(J) = (type, ..., n)
vt(V.I, st) := vt(V, st)           % address of V
               LIT(o);              % offset
               ADD;                 % address of V.I
           if vtype(V, st) = I' and st(I'.I) = (J, o)
```

Definition 22.5 (Translation of expressions)

The mapping

$$et : Exp \times Tab \dashrightarrow AM$$

is given by

$$et(c, st) := LIT(c);$$

$$et(V, st) := \begin{cases} LIT(c); & \text{if } V \in Ide \text{ and } st(V) = (\text{const}, c) \\ vt(V, st) & \text{if } st(I) = (\text{var}, J, n) \\ LOAD; \end{cases}$$

$$et(E_1 + E_2, st) := \begin{array}{l} et(E_1, st) \\ et(E_2, st) \\ ADD; \end{array}$$

$$\vdots$$

Definition 22.6 (Translation of commands)

For the mapping

$$ct : Cmd \times Tab \dashrightarrow AM$$

only the handling of assignments needs to be adapted:

$$\begin{aligned} ct(V := E, st) &:= vt(V, st) \quad \% \text{ address of left-hand side} \\ &\quad et(E, st) \quad \% \text{ value of right-hand side} \\ &\quad STORE; \end{aligned}$$

Translation of Commands and Programs

Definition 22.6 (Translation of commands)

For the mapping

$$ct : Cmd \times Tab \dashrightarrow AM$$

only the handling of assignments needs to be adapted:

$$\begin{aligned} ct(V := E, st) &:= vt(V, st) \quad \% \text{ address of left-hand side} \\ &\quad et(E, st) \quad \% \text{ value of right-hand side} \\ &\quad STORE; \end{aligned}$$

Definition 22.7 (Translation of programs)

The mapping

$$trans : Pgm \dashrightarrow AM$$

is defined by

$$trans(D \ C) := ct(C, update(D, st_{\emptyset}))$$

- 1 Repetition: Static Data Structures
- 2 Translation of Static Data Structures into AM Programs
- 3 A Translation Example
- 4 Dynamic Data Structures

Example 22.8

```
 $P = \left. \begin{array}{l} \text{type Int=int; Array=array[1..10] of Int;} \\ \text{var a:Array; i:Int;} \\ \text{i:=1;} \\ \text{while i<=10 do} \\ \quad \text{a[i]:=i; i:=i+1;} \end{array} \right\} \begin{array}{l} D \\ \\ C \end{array}$ 
```

Example 22.8

$$\begin{aligned} P = & \text{type Int=int; Array=array[1..10] of Int;} \} D \\ & \text{var a:Array; i:Int;} \\ & \text{i:=1;} \\ & \text{while i<=10 do} \} C \\ & \quad \text{a[i]:=i; i:=i+1;} \end{aligned}$$
$$\text{trans}(P) = \text{ct}(C, \text{update}(D, \text{st}_\emptyset))$$

Example 22.8

$$\begin{array}{l} P = \text{type Int=int; Array=array[1..10] of Int;} \\ \quad \text{var a:Array; i:Int;} \\ \quad \text{i:=1;} \\ \quad \text{while i<=10 do} \\ \quad \quad \text{a[i]:=i; i:=i+1;} \end{array} \left. \vphantom{\begin{array}{l} P = \text{type Int=int; Array=array[1..10] of Int;} \\ \quad \text{var a:Array; i:Int;} \\ \quad \text{i:=1;} \\ \quad \text{while i<=10 do} \\ \quad \quad \text{a[i]:=i; i:=i+1;} \end{array}} \right\} D$$
$$\left. \vphantom{\begin{array}{l} P = \text{type Int=int; Array=array[1..10] of Int;} \\ \quad \text{var a:Array; i:Int;} \\ \quad \text{i:=1;} \\ \quad \text{while i<=10 do} \\ \quad \quad \text{a[i]:=i; i:=i+1;} \end{array}} \right\} C$$
$$\begin{aligned} \text{trans}(P) &= \text{ct}(C, \text{update}(D, \text{st}_\emptyset)) \\ \text{st} &:= \text{update}(D, \text{st}_\emptyset) \\ &= \text{st}_\emptyset[\quad \text{Int} \mapsto (\text{type}, \text{int}, 1), \\ &\quad \text{Array} \mapsto (\text{type}, \text{array}, 1, 10, \text{Int}, 10), \\ &\quad \text{a} \mapsto (\text{var}, \text{Array}, 0), \\ &\quad \text{i} \mapsto (\text{var}, \text{Int}, 10)] \end{aligned}$$

Example 22.8

$$\left. \begin{array}{l} P = \text{type Int=int; Array=array[1..10] of Int;} \\ \text{var a:Array; i:Int;} \end{array} \right\} D$$
$$\left. \begin{array}{l} \text{i:=1;} \\ \text{while i<=10 do} \\ \quad \text{a[i]:=i; i:=i+1;} \end{array} \right\} C$$
$$\begin{aligned} \text{trans}(P) &= \text{ct}(C, \text{update}(D, \text{st}_\emptyset)) \\ \text{st} &:= \text{update}(D, \text{st}_\emptyset) \\ &= \text{st}_\emptyset[\quad \text{Int} \mapsto (\text{type}, \text{int}, 1), \\ &\quad \text{Array} \mapsto (\text{type}, \text{array}, 1, 10, \text{Int}, 10), \\ &\quad \text{a} \mapsto (\text{var}, \text{Array}, 0), \\ &\quad \text{i} \mapsto (\text{var}, \text{Int}, 10)] \\ \text{ct}(C, \text{st}) &= \text{ct}(\text{i:=1}, \text{st}) \\ &\quad \text{ct}(\text{while i<=10 do a[i]:=i; i:=i+1}, \text{st}) \end{aligned}$$

Example 22.8

$$\left. \begin{array}{l} P = \text{type Int=int; Array=array[1..10] of Int;} \\ \quad \text{var a:Array; i:Int;} \\ \quad \text{i:=1;} \\ \quad \text{while i<=10 do} \\ \quad \quad \text{a[i]:=i; i:=i+1;} \end{array} \right\} \begin{array}{l} D \\ C \end{array}$$
$$\begin{aligned} \text{trans}(P) &= \text{ct}(C, \text{update}(D, \text{st}_\emptyset)) \\ \text{st} &:= \text{update}(D, \text{st}_\emptyset) \\ &= \text{st}_\emptyset[\quad \text{Int} \mapsto (\text{type}, \text{int}, 1), \\ &\quad \text{Array} \mapsto (\text{type}, \text{array}, 1, 10, \text{Int}, 10), \\ &\quad \text{a} \mapsto (\text{var}, \text{Array}, 0), \\ &\quad \text{i} \mapsto (\text{var}, \text{Int}, 10)] \end{aligned}$$
$$\begin{aligned} \text{ct}(C, \text{st}) &= \text{ct}(\text{i:=1}, \text{st}) \\ &\quad \text{ct}(\text{while i<=10 do a[i]:=i; i:=i+1}, \text{st}) \\ \text{ct}(\text{i:=1}, \text{st}) &= \text{vt}(\text{i}, \text{st}) \quad \% \text{ adr(i)} \\ &\quad \text{et}(1, \text{st}) \quad \% \text{ val(1)} \\ &\quad \text{STORE;} \\ &= \text{LIT}(10); \text{LIT}(1); \text{STORE;} \end{aligned}$$

Example 22.8 (continued)

```
ct(while i<=10 do a[i]:=i; i:=i+1,st)
    = a : et(i<=10,st)
          JFALSE(a');
          ct(a[i]:=i; i:=i+1,st)
          JMP(a);
    a' :
```

Example 22.8 (continued)

```
ct(while i<=10 do a[i]:=i; i:=i+1,st)
    = a : et(i<=10,st)
          JFALSE(a');
          ct(a[i]:=i; i:=i+1,st)
          JMP(a);
      a' :
et(i<=10,st) = LIT(10); LOAD; LIT(10); LE;
```

Example 22.8 (continued)

```
ct(while i<=10 do a[i]:=i; i:=i+1,st)
    = a : et(i<=10,st)
          JFALSE(a');
          ct(a[i]:=i; i:=i+1,st)
          JMP(a);
          a' :
et(i<=10,st) = LIT(10); LOAD; LIT(10); LE;
ct(a[i]:=i; i:=i+1,st) = ct(a[i]:=i,st) ct(i:=i+1,st)
```

Example 22.8 (continued)

```
ct(while i<=10 do a[i]:=i; i:=i+1,st)
    = a : et(i<=10,st)
          JFALSE(a');
          ct(a[i]:=i; i:=i+1,st)
          JMP(a);
          a' :
et(i<=10,st) = LIT(10); LOAD; LIT(10); LE;
ct(a[i]:=i; i:=i+1,st) = ct(a[i]:=i,st) ct(i:=i+1,st)
ct(a[i]:=i,st) = vt(a[i],st)  % adr(a[i])
                  et(i,st)    % val(i)
                  STORE;
```

Example 22.8 (continued)

```
ct(while i<=10 do a[i]:=i; i:=i+1,st)
    = a : et(i<=10,st)
          JFALSE(a');
          ct(a[i]:=i; i:=i+1,st)
          JMP(a);
          a' :
et(i<=10,st) = LIT(10); LOAD; LIT(10); LE;
ct(a[i]:=i; i:=i+1,st) = ct(a[i]:=i,st) ct(i:=i+1,st)
ct(a[i]:=i,st) = vt(a[i],st)    % adr(a[i])
                et(i,st)       % val(i)
                STORE;
vt(a[i],st) = vt(a,st)          % adr(a)
              et(i,st)          % val(i)
              CAB(1,10);        % bounds checking
              LIT(1); SUB;      % index diff.
              LIT(1); MULT;     % rel. address
              ADD;              % adr(a[i])
```

Example 22.8 (continued)

```
ct(while i<=10 do a[i]:=i; i:=i+1,st)
    = a : et(i<=10,st)
          JFALSE(a');
          ct(a[i]:=i; i:=i+1,st)
          JMP(a);
          a' :
et(i<=10,st) = LIT(10); LOAD; LIT(10); LE;
ct(a[i]:=i; i:=i+1,st) = ct(a[i]:=i,st) ct(i:=i+1,st)
ct(a[i]:=i,st) = vt(a[i],st)    % adr(a[i])
                et(i,st)       % val(i)
                STORE;
vt(a[i],st) = vt(a,st)          % adr(a)
                et(i,st)       % val(i)
                CAB(1,10);      % bounds checking
                LIT(1); SUB;    % index diff.
                LIT(1); MULT;   % rel. address
                ADD;            % adr(a[i])
vt(a,st) = LIT(0);
```

Example 22.8 (continued)

```
ct(while i<=10 do a[i]:=i; i:=i+1,st)
    = a : et(i<=10,st)
          JFALSE(a');
          ct(a[i]:=i; i:=i+1,st)
          JMP(a);
          a' :
et(i<=10,st) = LIT(10); LOAD; LIT(10); LE;
ct(a[i]:=i; i:=i+1,st) = ct(a[i]:=i,st) ct(i:=i+1,st)
ct(a[i]:=i,st) = vt(a[i],st)    % adr(a[i])
                et(i,st)       % val(i)
                STORE;
vt(a[i],st) = vt(a,st)          % adr(a)
              et(i,st)          % val(i)
              CAB(1,10);        % bounds checking
              LIT(1); SUB;      % index diff.
              LIT(1); MULT;     % rel. address
              ADD;              % adr(a[i])
vt(a,st) = LIT(0);
et(i,st) = LIT(10); LOAD;
```

Example 22.8 (continued)

```
ct(while i<=10 do a[i]:=i; i:=i+1,st)
    = a : et(i<=10,st)
          JFALSE(a');
          ct(a[i]:=i; i:=i+1,st)
          JMP(a);
          a' :
et(i<=10,st) = LIT(10); LOAD; LIT(10); LE;
ct(a[i]:=i; i:=i+1,st) = ct(a[i]:=i,st) ct(i:=i+1,st)
ct(a[i]:=i,st) = vt(a[i],st)  % adr(a[i])
                  et(i,st)    % val(i)
                  STORE;
vt(a[i],st) = vt(a,st)      % adr(a)
                  et(i,st)   % val(i)
                  CAB(1,10); % bounds checking
                  LIT(1); SUB; % index diff.
                  LIT(1); MULT; % rel. address
                  ADD;         % adr(a[i])
vt(a,st) = LIT(0);
et(i,st) = LIT(10); LOAD;
ct(i:=i+1,st) = LIT(10); LIT(10); LIT(1); ADD; STORE;
```

- 1 Repetition: Static Data Structures
- 2 Translation of Static Data Structures into AM Programs
- 3 A Translation Example
- 4 Dynamic Data Structures

Example 22.9 (Variant records in Pascal)

```
TYPE Coordinate = RECORD
    nr: INTEGER;
    CASE type: (cartesian, polar) OF
        cartesian: (x, y: REAL);
        polar: (r : REAL; phi: INTEGER )
    END
END;

VAR pt: Coordinate;
pt.type := cartesian; pt.x := 0.5; pt.y := 1.2;
```

Example 22.9 (Variant records in Pascal)

```
TYPE Coordinate = RECORD
    nr: INTEGER;
    CASE type: (cartesian, polar) OF
        cartesian: (x, y: REAL);
        polar: (r : REAL; phi: INTEGER )
    END
END;

VAR pt: Coordinate;
pt.type := cartesian; pt.x := 0.5; pt.y := 1.2;
```

Implementation:

- Allocate memory for “biggest” variant
- Share memory between variant fields

Example 22.10 (Dynamic arrays in Pascal)

```
FUNCTION Sum(VAR a: ARRAY OF REAL): REAL;  
  VAR  
    i: INTEGER; s: REAL;  
  BEGIN  
    s := 0.0; FOR i := 0 to HIGH(A) do s := s + a[i] END; Sum := s  
  END
```

Example 22.10 (Dynamic arrays in Pascal)

```
FUNCTION Sum(VAR a: ARRAY OF REAL): REAL;  
  VAR  
    i: INTEGER; s: REAL;  
  BEGIN  
    s := 0.0; FOR i := 0 to HIGH(A) do s := s + a[i] END; Sum := s  
  END
```

Implementation:

- Memory requirements unknown at compile time but determined by actual function/procedure parameters
 $\implies$ no heap required
- Use **array descriptor** with following fields as parameter value:
 - starting memory address of array
 - size of array
 - lower index of array (possibly fixed by 0)
 - upper index of array (actually redundant)
- Use data stack or **index register** to access array elements

Dynamic Memory Allocation I

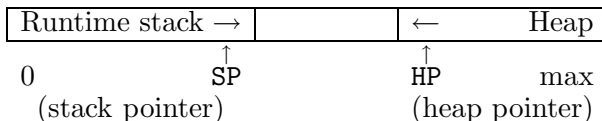
- Dynamically manipulated data structures (lists, trees, graphs, ...)
- So far: creation of (static) objects by **declaration**
- Now: creation of (dynamic) objects by **explicit memory allocation**
- Access by (implicit or explicit) **pointers**
- Deletion by **explicit deallocation** or **garbage collection**
(= automatic deallocation of unreachable objects)

Dynamic Memory Allocation I

- Dynamically manipulated data structures (lists, trees, graphs, ...)
- So far: creation of (static) objects by **declaration**
- Now: creation of (dynamic) objects by **explicit memory allocation**
- Access by (implicit or explicit) **pointers**
- Deletion by **explicit deallocation** or **garbage collection**
(= automatic deallocation of unreachable objects)
- Implementation: **runtime stack not sufficient**
(lifetime of objects generally exceeds lifetime of procedure calls)

⇒ new data structure: **heap**

- Simplest form of organization:



Dynamic Memory Allocation II

- New instruction: **NEW**
 - allocates n memory cells where n = topmost value of runtime stack
 - returns address of first cell
 - formal semantics:

```
if HP - <SP> > SP
  then HP := HP - <SP>; <SP> := HP
  else error("memory overflow")
```

Dynamic Memory Allocation II

- New instruction: **NEW**
 - allocates n memory cells where $n = \text{topmost value of runtime stack}$
 - returns address of first cell
 - formal semantics:

```
if HP - <SP> > SP
  then HP := HP - <SP>; <SP> := HP
  else error("memory overflow")
```

- But: collision check required for every operation which increases SP (e.g., expression evaluations)
- Efficient solution: add **extreme stack pointer** EP
 - points to topmost SP which will be used in the computation of current procedure
 - set by procedure entry code
 - statically computable at compile time
 - modified semantics of NEW:

```
if HP - <SP> > EP
  then HP := HP - <SP>; <SP> := HP
  else error("memory overflow")
```

- explicitly by programmer (e.g., Pascal's **dispose** or C's **free**) or ...
- automatically by runtime system (**garbage collection**; e.g., Java)
 - ① recursively mark all reachable objects
 - ② deallocate all unreachable objects

(careful design required for concurrent and/or realtime systems)
- management of deallocated memory areas by **free list**