



Compiler Construction

Lecture 17: Code Generation III (Implementation of Static Data Structures)

Summer Semester 2016

Thomas Noll

Software Modeling and Verification Group

RWTH Aachen University

<https://moves.rwth-aachen.de/teaching/ss-16/cc/>

Remaining Lectures

- Today: 17 – Code Generation III (Implementation of Static Data Structures)
- Thursday: 18 – Code Generation IV (Implementation of Dynamic Data Structures)
- 19 July: 19 – Code Generation V (Compiler Backend)
- [19 July, 12:15–13:45: Q&A session]
- [21 July: cancelled]

Recap: Syntax of EPL

Outline of Lecture 17

Recap: Syntax of EPL

Implementation of Data Structures

Static Data Structures

Modifying the Abstract Machine

Modifying the Symbol Table

The Translation

A Translation Example

Recap: Syntax of EPL

Syntax of EPL

Definition (Syntax of EPL)

The **syntax of EPL** is defined as follows:

$$\begin{aligned} \mathbb{Z} : & \quad z && \text{(* } z \text{ is an integer *)} \\ Ide : & \quad I && \text{(* } I \text{ is an identifier *)} \\ AExp : & \quad A ::= z \mid I \mid A_1 + A_2 \mid \dots \\ BExp : & \quad B ::= A_1 < A_2 \mid \text{not } B \mid B_1 \text{ and } B_2 \mid B_1 \text{ or } B_2 \\ Cmd : & \quad C ::= I := A \mid C_1 ; C_2 \mid \text{if } B \text{ then } C_1 \text{ else } C_2 \mid \text{while } B \text{ do } C \mid I() \\ Dcl : & \quad D ::= D_C D_V D_P \\ & \quad D_C ::= \varepsilon \mid \text{const } l_1 := z_1, \dots, l_n := z_n; \\ & \quad D_V ::= \varepsilon \mid \text{var } l_1, \dots, l_n; \\ & \quad D_P ::= \varepsilon \mid \text{proc } l_1 ; K_1 ; \dots ; \text{proc } l_n ; K_n; \\ Blk : & \quad K ::= D C \\ Pgm : & \quad P ::= \text{in/out } l_1, \dots, l_n ; K. \end{aligned}$$

Implementation of Data Structures

Outline of Lecture 17

Recap: Syntax of EPL

Implementation of Data Structures

Static Data Structures

Modifying the Abstract Machine

Modifying the Symbol Table

The Translation

A Translation Example

Implementation of Data Structures

Implementation of Data Structures

Source code: **data structures** = arrays, records, lists, trees, ...

⇒ **structured** state space, variables with components

Implementation of Data Structures

Implementation of Data Structures

Source code: **data structures** = arrays, records, lists, trees, ...

⇒ **structured** state space, variables with components

Abstract machine: **linear** memory structure, cells for storing atomic data

Implementation of Data Structures

Implementation of Data Structures

Source code: **data structures** = arrays, records, lists, trees, ...

⇒ **structured** state space, variables with components

Abstract machine: **linear** memory structure, cells for storing atomic data

Translation: **mapping** of structured state space to linear memory

(⇒ **address computation**)

- **static** data structures: memory requirements known at compile time
- **dynamic** data structures: memory requirements runtime dependent
 - ⇒ heap, pointers, garbage collection, ...

Implementation of Data Structures

Implementation of Data Structures

Source code: **data structures** = arrays, records, lists, trees, ...

⇒ **structured** state space, variables with components

Abstract machine: **linear** memory structure, cells for storing atomic data

Translation: **mapping** of structured state space to linear memory

(⇒ **address computation**)

- **static** data structures: memory requirements known at compile time
 - **dynamic** data structures: memory requirements runtime dependent
- ⇒ heap, pointers, garbage collection, ...

First step:

- static data structures (arrays and records)
- inductive type definitions
- no blocks or procedures (only for simplification; “orthogonal” extension)

Static Data Structures

Outline of Lecture 17

Recap: Syntax of EPL

Implementation of Data Structures

Static Data Structures

Modifying the Abstract Machine

Modifying the Symbol Table

The Translation

A Translation Example

Static Data Structures

Modified Syntax of EPL

Definition 17.1 (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} : $b ::= \text{true} \mid \text{false}$ (* b is a Boolean *)
 \mathbb{R} : r (* r is a real number *)
Con : $c ::= z \mid b \mid r$ (* c is a constant *)
Ide : I, J (* I, J are identifiers *)
Type : $T ::= \text{bool} \mid \text{int} \mid \text{real} \mid I \mid \text{array}[z_1..z_2] \text{ of } T \mid$
 $\text{record } l_1 : T_1; \dots; l_n : T_n \text{ end}$
Var : $V ::= I \mid V[E] \mid V.I$
Exp : $E ::= c \mid V \mid E_1 + E_2 \mid E_1 < E_2 \mid E_1 \text{ and } E_2 \mid \dots$
Cmd : $C ::= V := E \mid C_1; C_2 \mid \text{if } E \text{ then } C_1 \text{ else } C_2 \mid \text{while } E \text{ do } C$
Dcl : $D ::= D_C D_T D_V$
 $D_C ::= \varepsilon \mid \text{const } l_1 := c_1; \dots; l_n := c_n;$
 $D_T ::= \varepsilon \mid \text{type } l_1 := T_1; \dots; l_n := T_n;$
 $D_V ::= \varepsilon \mid \text{var } l_1 : T_1; \dots; l_n : T_n;$
Pgm : $P ::= D C$

Static Semantics I

- All identifiers in a declaration D have to be **different**.

Static Semantics I

- All identifiers in a declaration D have to be **different**.
- In $T = \text{record } l_1 : T_1 ; \dots ; l_n : T_n \text{ end}$, all selectors l_j must be **different**.

Static Semantics I

- All identifiers in a declaration D have to be **different**.
- In $T = \text{record } l_1 : T_1 ; \dots ; l_n : T_n \text{ end}$, all selectors l_j must be **different**.
- In $T = \text{array}[z_1 .. z_2]$ of T , $z_1 \leq z_2$.

Static Semantics I

- All identifiers in a declaration D have to be **different**.
- In $T = \text{record } l_1 : T_1 ; \dots ; l_n : T_n \text{ end}$, all selectors l_j must be **different**.
- In $T = \text{array}[z_1 .. z_2]$ of T , $z_1 \leq z_2$.
- Type definitions must **not be recursive**:
if $D_T = \text{type } l_1 := T_1 ; \dots ; l_n := T_n$; and type identifier l occurs in T_j , then $l \in \{l_1, \dots, l_{j-1}\}$.

Static Semantics I

- All identifiers in a declaration D have to be **different**.
- In $T = \text{record } l_1 : T_1 ; \dots ; l_n : T_n \text{ end}$, all selectors l_j must be **different**.
- In $T = \text{array}[z_1 .. z_2]$ of T , $z_1 \leq z_2$.
- Type definitions must **not be recursive**:
if $D_T = \text{type } l_1 := T_1 ; \dots ; l_n := T_n$; and type identifier l occurs in T_j , then $l \in \{l_1, \dots, l_{j-1}\}$.
- All type identifiers used in in a variable declaration D_V must be **declared** in D_T .

Static Semantics I

- All identifiers in a declaration D have to be **different**.
- In $T = \text{record } l_1 : T_1 ; \dots ; l_n : T_n \text{ end}$, all selectors l_j must be **different**.
- In $T = \text{array}[z_1 .. z_2]$ of T , $z_1 \leq z_2$.
- Type definitions must **not be recursive**:
if $D_T = \text{type } l_1 := T_1 ; \dots ; l_n := T_n$; and type identifier l occurs in T_j , then $l \in \{l_1, \dots, l_{j-1}\}$.
- All type identifiers used in in a variable declaration D_V must be **declared** in D_T .
- Every identifier used in a command C must be **declared** in D (as a constant or variable).

Static Semantics I

- All identifiers in a declaration D have to be **different**.
- In $T = \text{record } l_1 : T_1 ; \dots ; l_n : T_n \text{ end}$, all selectors l_j must be **different**.
- In $T = \text{array}[z_1 .. z_2]$ of T , $z_1 \leq z_2$.
- Type definitions must **not be recursive**:
if $D_T = \text{type } l_1 := T_1 ; \dots ; l_n := T_n$; and type identifier l occurs in T_j , then $l \in \{l_1, \dots, l_{j-1}\}$.
- All type identifiers used in in a variable declaration D_V must be **declared** in D_T .
- Every identifier used in a command C must be **declared** in D (as a constant or variable).
- Variables in expressions and assignments have a **base type** (`bool/int/real`; possibly via type identifiers).

Static Semantics II

- Array **indices** must have type `int`.

Static Semantics II

- Array **indices** must have type `int`.
- **Execution conditions** (`while`) and **branching expressions** (`if`) must have type `bool`.

Static Semantics II

- Array **indices** must have type `int`.
- **Execution conditions** (`while`) and **branching expressions** (`if`) must have type `bool`.
- The types of the left-hand side and of the right-hand side types of an assignment must be **compatible**.

Static Semantics II

- Array **indices** must have type `int`.
- **Execution conditions** (`while`) and **branching expressions** (`if`) must have type `bool`.
- The types of the left-hand side and of the right-hand side types of an assignment must be **compatible**.
- **Type compatibility**: $\mathbb{Z} \subseteq \mathbb{R}$ in mathematics, but not on computers (different representation)
⇒ **type casts**

weak typing: implicit casting by compiler (`2.5 + 1`, `1 + "42"`)

⇒ risk of undetected “real” errors;
for **programming-in-the-small** (script languages)

strong typing: explicit casting by programmer

⇒ enhanced software reliability;
for **programming-in-the-large**

Static Data Structures

Static Semantics II

- Array **indices** must have type `int`.
- **Execution conditions** (`while`) and **branching expressions** (`if`) must have type `bool`.
- The types of the left-hand side and of the right-hand side types of an assignment must be **compatible**.
- **Type compatibility**: $\mathbb{Z} \subseteq \mathbb{R}$ in mathematics, but not on computers (different representation)
 \implies **type casts**

weak typing: implicit casting by compiler (`2.5 + 1`, `1 + "42"`)

\implies risk of undetected “real” errors;
for **programming-in-the-small** (script languages)

strong typing: explicit casting by programmer

\implies enhanced software reliability;
for **programming-in-the-large**

- Instantiation of operators/functions/procedures/... for different parameter types:
polymorphism or **overloading**

`+` : `int` \times `int` \rightarrow `int` `+` : `real` \times `real` \rightarrow `real`

Modifying the Abstract Machine

Outline of Lecture 17

Recap: Syntax of EPL

Implementation of Data Structures

Static Data Structures

Modifying the Abstract Machine

Modifying the Symbol Table

The Translation

A Translation Example

Modifying the Abstract Machine

The Modified Abstract Machine AM

- Additional **main storage** for keeping data values
- **Procedure stack** not required anymore (as procedures no longer supported)

Definition 17.2 (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** $DS := \mathbb{R}^*$ (**true** $\hat{=}$ 1, **false** $\hat{=}$ 0; top to the right), and
- the **main storage** $MS := \{\sigma \mid \sigma : \mathbb{N} \rightarrow \mathbb{R}\}$.

Modifying the Abstract Machine

New AM Instructions

Definition 17.3 (New AM instructions)

- **Procedure instructions** are no longer needed.

Modifying the Abstract Machine

New AM Instructions

Definition 17.3 (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 $\llbracket \cdot \rrbracket : S \dashrightarrow S$:

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

Modifying the Abstract Machine

New AM Instructions

Definition 17.3 (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 $\llbracket \cdot \rrbracket : S \dashrightarrow S$:

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

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

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

Modifying the Symbol Table

Outline of Lecture 17

Recap: Syntax of EPL

Implementation of Data Structures

Static Data Structures

Modifying the Abstract Machine

Modifying the Symbol Table

The Translation

A Translation Example

Modifying the Symbol Table

Modifying the Symbol Table

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

Remarks:

- **Variable descriptor** (var, l, m): type l , memory address m
- Last component m of **type** entry: **memory requirement** (base types: 1 “cell”)
- **Array descriptor** ($\text{type}, \text{array}, z_1, z_2, l, m$):
 - bounds z_1, z_2
 - component type identifier l
- **Record descriptor** ($\text{type}, \text{record}, l_1, J_1, o_1, \dots, l_n, J_n, o_n, m$):
 - selector l_k
 - component type identifier J_k
 - memory offset o_k
- “Indexed” table lookup: $\text{st}(l.l_k) := (J_k, o_k)$ if $\text{st}(l) = (\text{type}, \text{record}, \dots, l_k, J_k, o_k, \dots, m)$

Modifying the Symbol Table

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 .

Modifying the Symbol Table

Maintaining the Symbol Table I

The symbol table is again maintained by the function $\text{update}(D, \text{st})$ 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 $D_T = \text{type } l_1 := T_1; \dots; l_n := T_n;$, then for every $k \in [n]$
 - $T_k \in \{\text{bool}, \text{int}, \text{real}\}$ or
 - $T_k \in \{l_1, \dots, l_{k-1}\}$ or
 - $T_k = \text{array}[z_1..z_2]$ of l_j where $j \in [k-1]$ or
 - $T_k = \text{record } J_1 : l_{j_1}; \dots; J_l : l_{j_l}$ end where $j_1, \dots, j_l \in [k-1]$
- For D_T as above, D_V must be of the form $D_V = \text{var } J_1 : l_{j_1}; \dots; J_k : l_{j_k};$ where $j_1, \dots, j_k \in [n]$

Modifying the Symbol Table

Maintaining the Symbol Table II

Definition 17.4 (Modified update function)

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

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

Modifying the Symbol Table

Maintaining the Symbol Table II

Definition 17.4 (Modified update function)

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

$$\begin{aligned} & \text{update}(D_C D_T D_V, st) \\ & \quad := \text{update}(D_V, \text{update}(D_T, \text{update}(D_C, st))) \\ & \text{update}(\varepsilon, st) \\ & \quad := st \\ & \text{update}(\text{const } l_1 := c_1; \dots; l_n := c_n; , st) \\ & \quad := st[l_1 \mapsto (\text{const}, c_1), \dots, l_n \mapsto (\text{const}, c_n)] \end{aligned}$$

Modifying the Symbol Table

Maintaining the Symbol Table II

Definition 17.4 (Modified update function)

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

$$\begin{aligned} & \text{update}(D_C D_T D_V, st) \\ & := \text{update}(D_V, \text{update}(D_T, \text{update}(D_C, st))) \\ & \text{update}(\varepsilon, st) \\ & := st \\ & \text{update}(\text{const } l_1 := c_1; \dots; l_n := c_n; , st) \\ & := st[l_1 \mapsto (\text{const}, c_1), \dots, l_n \mapsto (\text{const}, c_n)] \\ & \text{update}(\text{var } l_1 : J_1; \dots; l_n : J_n; , st) \\ & := st[l_1 \mapsto (\text{var}, J_1, o_1), \dots, l_n \mapsto (\text{var}, J_n, o_n)] \\ & \quad \text{if } st(J_k) = (\text{type}, \dots, m_k) \text{ and } o_k := \sum_{i=1}^{k-1} m_i \text{ for } k \in [n] \end{aligned}$$

Modifying the Symbol Table

Maintaining the Symbol Table III

Definition 17.4 (Modified update function; continued)

$$\begin{aligned} & \text{update}(\text{type } l := \text{bool}; D'_T, \text{st}) \\ & := \text{update}(\text{type } D'_T, \text{st}[l \mapsto (\text{type}, \text{bool}, 1)]) \\ & \text{update}(\text{type } l := \text{int}; D'_T, \text{st}) \\ & := \text{update}(\text{type } D'_T, \text{st}[l \mapsto (\text{type}, \text{int}, 1)]) \\ & \text{update}(\text{type } l := \text{real}; D'_T, \text{st}) \\ & := \text{update}(\text{type } D'_T, \text{st}[l \mapsto (\text{type}, \text{real}, 1)]) \\ & \text{update}(\text{type } l := J; D'_T, \text{st}) \\ & := \text{update}(\text{type } D'_T, \text{st}[l \mapsto \text{st}(J)]) \\ & \text{update}(\text{type } l := \text{array}[z_1..z_2] \text{ of } J; D'_T, \text{st}) \\ & := \text{update}(\text{type } D'_T, \text{st}[l \mapsto (\text{type}, \text{array}, z_1, z_2, J, n \cdot m)]) \\ & \quad \text{if } \text{st}(J) = (\text{type}, \dots, m) \text{ and } n := z_2 - z_1 + 1 \\ & \text{update}(\text{type } l := \text{record } l_1:J_1; \dots; l_n:J_n \text{ end}; D'_T, \text{st}) \\ & := \text{update}(\text{type } D'_T, \text{st}[l \mapsto (\text{type}, \text{record}, l_1, J_1, o_1, \dots, l_n, J_n, o_n, o_{n+1})]) \\ & \quad \text{if } \text{st}(J_k) = (\text{type}, \dots, m_k) \text{ and } o_k := \sum_{i=1}^{k-1} m_i \text{ for } k \in [n] \end{aligned}$$

Modifying the Symbol Table

Maintaining the Symbol Table IV

Example 17.5 (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;
```

Modifying the Symbol Table

Maintaining the Symbol Table IV

Example 17.5 (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

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

The Translation

Outline of Lecture 17

Recap: Syntax of EPL

Implementation of Data Structures

Static Data Structures

Modifying the Abstract Machine

Modifying the Symbol Table

The Translation

A Translation Example

The Translation

Translation of Variables I

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

Definition 17.6 (vtype function)

The mapping

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

is given by

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

The Translation

Translation of Variables II

Function `vt` generates code for computing the memory address of a variable on the data stack:

Definition 17.7 (Translation of variables)

The mapping $vt : Var \times Tab \dashrightarrow AM$ is given by

```
vt(I, st) := LIT(m);           if st(I) = (var, J, m)
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]
vt(V.I, st) := vt(V, st)       % address of V
                  LIT(o);       % offset
                  ADD;          % address of V.I
                  if vtype(V, st) = I', st(I'.I) = (J, o)
```

The Translation

Translation of Expressions

Definition 17.8 (Translation of expressions)

The mapping

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

is given by

$$\begin{aligned} et(c, st) &:= LIT(c); \\ et(V, st) &:= \begin{cases} LIT(c); & \text{if } V \in Ide \text{ and } st(V) = (const, c) \\ vt(V, st) & \text{otherwise} \\ LOAD; \end{cases} \\ et(E_1 + E_2, st) &:= \begin{array}{l} et(E_1, st) \\ et(E_2, st) \\ ADD; \end{array} \\ &\vdots \end{aligned}$$

The Translation

Translation of Commands and Programs

Definition 17.9 (Translation of commands)

For the mapping

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

only the handling of assignments needs to be adapted:

```
ct( $V := E$ , st) := vt( $V$ , st)  % address of left-hand side
                  et( $E$ , st)  % value of right-hand side
STORE;
```

The Translation

Translation of Commands and Programs

Definition 17.9 (Translation of commands)

For the mapping

$$ct : \mathit{Cmd} \times \mathit{Tab} \dashrightarrow \mathit{AM}$$

only the handling of assignments needs to be adapted:

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

Definition 17.10 (Translation of programs)

The mapping

$$\text{trans} : \mathit{Pgm} \dashrightarrow \mathit{AM}$$

is defined by

$$\text{trans}(D \ C) := ct(C, \text{update}(D, st_\emptyset)) \quad \text{where } st_\emptyset(l) \text{ is undefined for every } l \in \mathit{Ide}$$

A Translation Example

Outline of Lecture 17

Recap: Syntax of EPL

Implementation of Data Structures

Static Data Structures

Modifying the Abstract Machine

Modifying the Symbol Table

The Translation

A Translation Example

A Translation Example

Translation Example I

Example 17.11

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



```
 $\left. \begin{array}{l} \text{i:=1;} \\ \text{while i<=10 do} \\ \quad \text{a[i]:=i; i:=i+1;} \end{array} \right\} C$ 
```

A Translation Example

Translation Example I

Example 17.11

```

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

$$\left. \begin{array}{l} \text{i:=1;} \\ \text{while i<=10 do} \\ \quad \text{a[i]:=i; i:=i+1;} \end{array} \right\} C$$
  

$$\text{trans}(P) = \text{ct}(C, \text{update}(D, \text{st}_\emptyset))$$

```

A Translation Example

Translation Example I

Example 17.11

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

A Translation Example

Translation Example I

Example 17.11

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

A Translation Example

Translation Example I

Example 17.11

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

A Translation Example

Translation Example II

Example 17.11 (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' :
```

A Translation Example

Translation Example II

Example 17.11 (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;
```

A Translation Example

Translation Example II

Example 17.11 (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)
```

A Translation Example

Translation Example II

Example 17.11 (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) % address of a[i]
                et(i, st) % value of i
                STORE;
```

A Translation Example

Translation Example II

Example 17.11 (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) % address of a[i]
                et(i, st)      % value of i
                STORE;
vt(a[i], st) = vt(a, st)      % address of a
               et(i, st)      % value of i
               CAB(1, 10);    % bounds checking
               LIT(1); SUB;   % index difference
               LIT(1); MULT;  % relative address
               ADD;           % address of a[i]
```

A Translation Example

Translation Example II

Example 17.11 (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) % address of a[i]
                et(i, st)      % value of i
                STORE;
vt(a[i], st) = vt(a, st)      % address of a
                et(i, st)      % value of i
                CAB(1, 10);    % bounds checking
                LIT(1); SUB;   % index difference
                LIT(1); MULT;  % relative address
                ADD;           % address of a[i]
vt(a, st) = LIT(0);
```


A Translation Example

Translation Example II

Example 17.11 (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) % address of a[i]
                et(i, st) % value of i
                STORE;
vt(a[i], st) = vt(a, st) % address of a
              et(i, st) % value of i
              CAB(1, 10); % bounds checking
              LIT(1); SUB; % index difference
              LIT(1); MULT; % relative address
              ADD; % address of a[i]
vt(a, st) = LIT(0);
et(i, st) = LIT(10); LOAD;
```

A Translation Example

Translation Example II

Example 17.11 (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) % address of a[i]
                et(i, st) % value of i
                STORE;
vt(a[i], st) = vt(a, st) % address of a
              et(i, st) % value of i
              CAB(1, 10); % bounds checking
              LIT(1); SUB; % index difference
              LIT(1); MULT; % relative address
              ADD; % address of a[i]
vt(a, st) = LIT(0);
et(i, st) = LIT(10); LOAD;
ct(i:=i+1, st) = LIT(10); LIT(10); LOAD; LIT(1); ADD; STORE;
```