



# Compiler Construction

Lecture 16: Code Generation II (The Translation)

Summer Semester 2017

Thomas Noll

Software Modeling and Verification Group

RWTH Aachen University

<https://moves.rwth-aachen.de/teaching/ss-17/cc/>

# Recap: Intermediate Code

## Syntax of EPL

### Definition (Syntax of EPL)

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

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

## Recap: Intermediate Code

---

### The Abstract Machine AM

#### Definition (Abstract machine for EPL)

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

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

with

- the **program counter**  $PC := \mathbb{N}$ ,
- the **data stack**  $DS := \mathbb{Z}^*$  (top of stack to the right), and
- the **procedure stack** (or: **runtime stack**)  $PS := \mathbb{Z}^*$  (top of stack to the left).

Thus a state  $s = (pc, d, p) \in S$  is given by

- a program counter  $pc \in PC$ ,
- a data stack  $d = d.r : \dots : d.1 \in DS$ , and
- a procedure stack  $p = p.1 : \dots : p.t \in PS$ .

# Recap: Intermediate Code

---

## AM Instructions

### Definition (AM instructions)

The set of **AM instructions** is divided into

arithmetic instructions: ADD, MULT, ...

Boolean instructions: NOT, AND, OR, LT, ...

jumping instructions: JMP(*ca*), JFALSE(*ca*) (*ca* ∈ PC)

procedure instructions: CALL(*ca*, *dif*, *loc*) (*ca* ∈ PC, *dif*, *loc* ∈ ℕ), RET

transfer instructions: LOAD(*dif*, *off*), STORE(*dif*, *off*) (*dif*, *off* ∈ ℕ), LIT(*z*) (*z* ∈ ℤ)

## Recap: Intermediate Code

---

### Structure of Procedure Stack I

The semantics of procedure and transfer instructions requires a particular structure of the procedure stack  $p \in PS$ : it must be composed of **frames** (or: **activation records**) of the form

$$sl : dl : ra : v_1 : \dots : v_k$$

where

**static link  $sl$** : points to frame of surrounding declaration environment

- used to access non-local variables

**dynamic link  $dl$** : points to previous frame (i.e., of calling procedure)

- used to remove topmost frame after termination of procedure call

**return address  $ra$** : program counter after termination of procedure call

- used to continue program execution after termination of procedure call

**local variables  $v_i$** : values of locally declared variables

# Recap: Intermediate Code

## Structure of Procedure Stack II

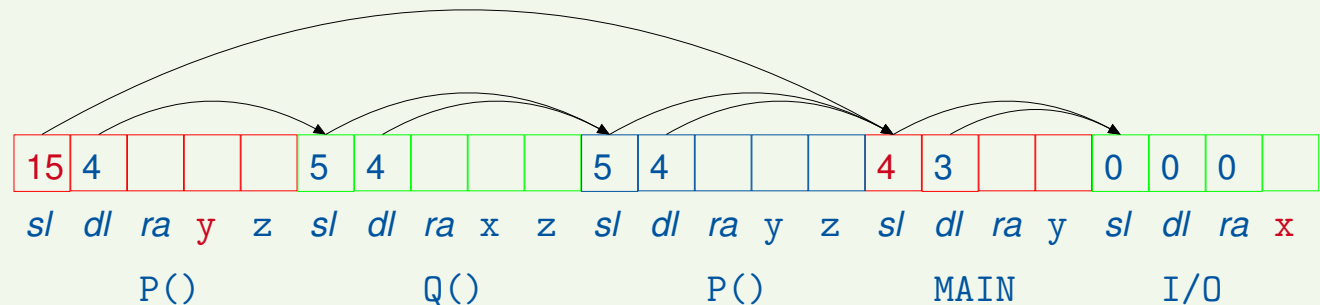
### Observation:

- The usage of a variable in a procedure body refers to its **innermost declaration**.
- If the level difference between the usage and the declaration is *dif*, then a **chain of *dif* static links** has to be followed to access the corresponding frame.

### Example (cf. Example 15.9)

```
in/out x;  
const c = 10;  
var y;  
proc P;  
  var y, z;  
  proc Q;  
    var x, z;  
    [... P() ...]  
  [... x ... y ... Q() ...]  
proc R;  
  [... P() ...]  
  [... P() ...].
```

Procedure stack after second call of P:



P uses x  $\implies dif = 2$

P uses y  $\implies dif = 0$

# Semantics of Procedure and Transfer Instructions

## The `base` Function

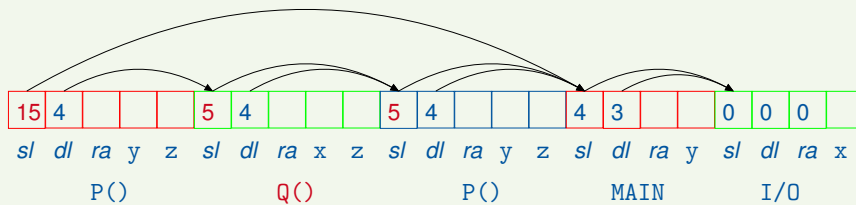
Computes static link information upon procedure call, given procedure stack and level difference

### Definition 16.1 (base function)

The function  $\text{base} : PS \times \mathbb{N} \dashrightarrow \mathbb{N}$  is given by

$$\begin{aligned} \text{base}(p, 0) &:= 1 \\ \text{base}(p, dif + 1) &:= \text{base}(p, dif) + p.\text{base}(p, dif) \end{aligned}$$

### Example 16.2 (cf. Example 16.4)



In the second call of `P` (from `Q`):  $dif = 2$

$$\begin{aligned} &\text{base}(p, 0) = 1 \\ \implies &\text{base}(p, 1) = 1 + p.1 = 6 \\ \implies &\text{base}(p, 2) = 6 + p.6 = 11 \\ \implies &sl = \text{base}(p, 2) + \underbrace{2}_{dl,ra} + \underbrace{2}_{y,z} = 15 \end{aligned}$$

# Semantics of Procedure and Transfer Instructions

## Semantics of Procedure Instructions

- **CALL** ( $ca, dif, loc$ ) with
  - code address  $ca \in PC$
  - level difference  $dif \in \mathbb{N}$
  - number of local variables  $loc \in \mathbb{N}$creates new frame and jumps to given address (= starting address of procedure)
- **RET** removes the topmost frame and returns to the calling site

### Definition 16.3 (Semantics of procedure instructions)

The semantics of a procedure instruction  $O$ ,  $\llbracket O \rrbracket : S \dashrightarrow S$ , is defined as follows:

$$\llbracket \text{CALL}(ca, dif, loc) \rrbracket (pc, d, p) \\ := (ca, d, \underbrace{\text{base}(p, dif) + loc + 2}_{sl}) : \underbrace{(loc + 2)}_{dl} : \underbrace{(pc + 1)}_{ra} : \underbrace{0 : \dots : 0}_{\times loc} : p$$

$$\llbracket \text{RET} \rrbracket (pc, d, p.1 : \dots : p.t) \\ := (\underbrace{p.3}_{ra}, d, p.(\underbrace{p.2}_{dl} + 2) : \dots : p.t) \quad \text{if } t \geq p.2 + 2$$



# Semantics of Procedure and Transfer Instructions

## Semantics of Transfer Instructions

- $\text{LOAD}(dif, off)$  and  $\text{STORE}(dif, off)$  with

- level difference  $dif \in \mathbb{N}$
- variable offset  $off \in \mathbb{N}$

respectively load and store variable values between data and procedure stack, following a chain of  $dif$  static links

- $\text{LIT}(z)$  loads the literal constant  $z \in \mathbb{Z}$

### Definition 16.4 (Semantics of transfer instructions)

The **semantics of a transfer instruction**  $O$ ,  $\llbracket O \rrbracket : S \dashrightarrow S$ , is defined as follows:

$$\begin{aligned}\llbracket \text{LOAD}(dif, off) \rrbracket (pc, d, p) &:= (pc + 1, d : p.(\text{base}(p, dif) + off + 2), p) \\ \llbracket \text{STORE}(dif, off) \rrbracket (pc, d : z, p) &:= (pc + 1, d, p[\text{base}(p, dif) + off + 2 \mapsto z]) \\ \llbracket \text{LIT}(z) \rrbracket (pc, d, p) &:= (pc + 1, d : z, p)\end{aligned}$$

# Semantics of Procedure and Transfer Instructions

## AM Programs and Their Semantics

### Definition 16.5 (Semantics of AM programs)

An **AM program** is a sequence of  $k \geq 1$  labelled AM instructions:

$$P = 1 : O_1; \dots; k : O_k$$

The set of all AM programs is denoted by  $AM$ .

The **semantics of AM programs** is determined by

$$\llbracket \cdot \rrbracket : AM \times S \dashrightarrow S$$

with

$$\llbracket P \rrbracket(pc, d, p) := \begin{cases} \llbracket P \rrbracket(\llbracket O_{pc} \rrbracket(pc, d, p)) & \text{if } \llbracket O_{pc} \rrbracket(pc, d, p) \text{ defined} \\ (pc, d, p) & \text{otherwise} \end{cases}$$

(in particular,  $\llbracket P \rrbracket(pc, d, p) = (pc, d, p)$  if  $pc \notin [k]$ )

# The Symbol Table

---

## Structure of Symbol Table

**Goal:** define **translation mapping**  $\text{trans} : Pgm \dashrightarrow AM$

The translation employs a **symbol table**:

$$\begin{aligned} Tab := \{st \mid st : Ide \dashrightarrow & (\{const\} \times \mathbb{Z}) \\ & \cup (\{var\} \times Lev \times Off) \\ & \cup (\{proc\} \times PC \times Lev \times Size)\} \end{aligned}$$

whose entries are created by declarations:

- constant declarations:  $(const, z)$ 
  - **value**  $z \in \mathbb{Z}$
- variable declarations:  $(var, lev, off)$ 
  - **declaration level**  $lev \in Lev := \mathbb{N}$  ( $0 \cong I/O$ ,  $1 \cong MAIN$ , ...)
  - **offset**  $off \in Off := \mathbb{N}$
  - offset and difference between usage and declaration level determine procedure stack entry
- procedure declarations:  $(proc, ca, lev, loc)$ 
  - **code address**  $ca \in PC$
  - **declaration level**  $lev \in Lev$
  - **number of local variables**  $loc \in Size := \mathbb{N}$

# The Symbol Table

## Maintaining the Symbol Table

Function  $\text{update}(D, \text{st}, \text{lev})$  specifies update of symbol table  $\text{st}$  according to declaration  $D$  (with respect to current level  $\text{lev}$ ):

### Definition 16.6 (update function)

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

$$\begin{aligned} & \text{update}(D_C D_V D_P, \text{st}, \text{lev}) \\ & := \text{update}(D_P, \text{update}(D_V, \text{update}(D_C, \text{st}, \text{lev}), \text{lev}), \text{lev}) \\ & \quad \text{if all identifiers in } D_C D_V D_P \text{ different} \end{aligned}$$
$$\text{update}(\varepsilon, \text{st}, \text{lev}) := \text{st}$$
$$\begin{aligned} & \text{update}(\text{const } l_1 := z_1, \dots, l_n := z_n; , \text{st}, \text{lev}) \\ & := \text{st}[l_1 \mapsto (\text{const}, z_1), \dots, l_n \mapsto (\text{const}, z_n)] \end{aligned}$$
$$\begin{aligned} & \text{update}(\text{var } l_1, \dots, l_n; , \text{st}, \text{lev}) \\ & := \text{st}[l_1 \mapsto (\text{var}, \text{lev}, 1), \dots, l_n \mapsto (\text{var}, \text{lev}, n)] \end{aligned}$$
$$\begin{aligned} & \text{update}(\text{proc } l_1; K_1; \dots; \text{proc } l_n; K_n; , \text{st}, \text{lev}) \\ & := \text{st}[l_1 \mapsto (\text{proc}, a_1, \text{lev}, \text{size}(K_1)), \dots, l_n \mapsto (\text{proc}, a_n, \text{lev}, \text{size}(K_n))] \\ & \quad \text{with "fresh" addresses } a_1, \dots, a_n \text{ where } \text{size}(D_C \text{ var } l_1, \dots, l_n; D_P C) := n \end{aligned}$$

# The Symbol Table

---

## The Initial Symbol Table

**Reminder:** an EPL program  $P = \text{in/out } l_1, \dots, l_n; K. \in \text{Pgm}$  has a **semantics** of type  $\mathbb{Z}^n \dashrightarrow \mathbb{Z}^n$ .

Given input values  $(z_1, \dots, z_n) \in \mathbb{Z}^n$ , we choose the **initial state**

$$s := (1, \varepsilon, \underbrace{0 : 0 : 0 : z_1 : \dots : z_n}_{\text{I/O frame}}) \in S = PC \times DS \times PS$$

Thus the corresponding **initial symbol table** has  $n$  entries:

$$\text{st}_{I/O}(l_j) := (\text{var}, \underbrace{0}_{\text{lev}}, \underbrace{j}_{\text{off}}) \quad \text{for every } j \in [n]$$

# Translation of Programs

---

## Translation of Programs

Translation of `in/out  $l_1, \dots, l_n; D C$` :

1. Create MAIN frame for executing  $C$  (remember: `CALL( $ca, dif, loc$ )`)
2. Stop program execution after return

### Definition 16.7 (Translation of programs)

The mapping

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

is defined by

$$\begin{aligned} \text{trans}(\text{in/out } l_1, \dots, l_n; K.) &:= 1 : \text{CALL}(a, 0, \text{size}(K)); \\ &2 : \text{JMP}(0); \\ &\text{kt}(K, \text{st}_{I/O}, a, 1) \end{aligned}$$

with “fresh” address  $a$

# Translation of Blocks

---

## Translation of Blocks

Translation of  $D C$ :

1. Update symbol table according to  $D$
2. Create code for procedures declared in  $D$   
(using the updated symbol table to handle recursion)
3. Create code for  $C$  (using the updated symbol table)

### Definition 16.8 (Translation of blocks)

The mapping

$$kt : Blk \times Tab \times PC \times Lev \dashrightarrow AM$$

(“block translation”) is defined by

$$\begin{aligned} kt(D C, st, a, lev) &:= dt(D, \text{update}(D, st, lev), lev) \\ &\quad ct(C, \text{update}(D, st, lev), a, lev) \\ &\quad a' : RET; \end{aligned}$$

with “fresh” address  $a'$

# Translation of Declarations

---

## Translation of Declarations

Translation of  $D$ : generate code for the procedures declared in  $D$

### Definition 16.9 (Translation of declarations)

The mapping

$$dt : Dcl \times Tab \times Lev \dashrightarrow AM$$

(“declaration translation”) is defined by

$$dt(D_C D_V D_P, st, lev) := dt(D_P, st, lev)$$

$$dt(\varepsilon, st, lev) := \varepsilon$$

$$dt(\text{proc } l_1; K_1; \dots; \text{proc } l_n; K_n; , st, lev) := kt(K_1, st, a_1, lev + 1)$$

$$\vdots$$

$$kt(K_n, st, a_n, lev + 1)$$

where  $st(l_j) = (\text{proc}, a_j, \dots, \dots)$

for every  $j \in [n]$



# Translation of Commands

## Translation of Commands

### Definition 16.10 (Translation of commands)

The mapping

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

(“command translation”) is defined by

$$ct(l := A, st, a, lev) := at(A, st, a, lev); a' : \text{STORE}(lev - lev', off); \\ \text{if } st(l) = (\text{var}, lev', off)$$

$$ct(l(), st, a, lev) := a : \text{CALL}(ca, lev - lev', loc); \\ \text{if } st(l) = (\text{proc}, ca, lev', loc)$$

$$ct(C_1; C_2, st, a, lev) := ct(C_1, st, a, lev); ct(C_2, st, a', lev)$$

$$ct(\text{if } B \text{ then } C_1 \text{ else } C_2, st, a, lev) := bt(B, st, a, lev); a' : \text{JFALSE}(a''); \\ ct(C_1, st, a' + 1, lev); a'' - 1 : \text{JMP}(a'''); \\ ct(C_2, st, a'', lev); a''' :$$

$$ct(\text{while } B \text{ do } C, st, a, lev) := bt(B, st, a, lev); a' : \text{JFALSE}(a'' + 1); \\ ct(C, st, a' + 1, lev); a'' : \text{JMP}(a);$$

# Translation of Expressions

---

## Translation of Boolean Expressions

### Definition 16.11 (Translation of Boolean expressions)

The mapping

$$bt : BExp \times Tab \times PC \times Lev \dashrightarrow AM$$

(“Boolean expression translation”) is defined by

$$bt(A_1 < A_2, st, a, lev) := at(A_1, st, a, lev); at(A_2, st, a', lev); a'' : LT;$$

$$bt(\text{not } B, st, a, lev) := bt(B, st, a, lev); a' : NOT;$$

$$bt(B_1 \text{ and } B_2, st, a, lev) := bt(B_1, st, a, lev); bt(B_2, st, a', lev); a'' : AND;$$

$$bt(B_1 \text{ or } B_2, st, a, lev) := bt(B_1, st, a, lev); bt(B_2, st, a', lev); a'' : OR;$$

# Translation of Expressions

## Translation of Arithmetic Expressions

### Definition 16.12 (Translation of arithmetic expressions)

The mapping

$$at : AExp \times Tab \times PC \times Lev \dashrightarrow AM$$

(“arithmetic expression translation”) is defined by

$$at(z, st, a, lev) := a : LIT(z) ;$$

$$at(l, st, a, lev) := \begin{cases} a : LIT(z) ; & \text{if } st(l) = (\text{const}, z) \\ a : LOAD(lev - lev', off) ; & \text{if } st(l) = (\text{var}, lev', off) \end{cases}$$

$$at(A_1 + A_2, st, a, lev) := at(A_1, st, a, lev) \\ at(A_2, st, a', lev) \\ a'' : ADD ;$$

# A Translation Example

## Example: Factorial Function I

### Example 16.13 (Factorial function; cf. Example 15.3)

#### Source code:

```
in/out x;
var y;
proc F;
  if x > 1 then
    y := y * x;
    x := x - 1;
    F()
  y := 1;
  F();
  x := y.
```

$\text{trans}(\text{in/out } l_1, \dots, l_n; K.) :=$

```
1 : CALL(a, 0, size(K)); kt(D C, st, a, lev) :=
2 : JMP(0);
```

```
kt(K, stI/O, a, 1)
```

```
dt(D, update(D, st, lev), lev) update(var l1, ..., ln; st, lev) :=
```

```
ct(C, update(D, st, lev), a, lev) st[l1 ↦ (var, lev, 1), ..., ln ↦ (var, lev, n)]
```

```
a' : RET;
```

```
kt(K1, st, a1, lev + 1)
```

```
⋮
```

```
kt(Kn, st, an, lev + 1)
```

where  $\text{st}(l_j) = (\text{proc } a_j \dots)$  for every  $j \in [n]$

```
bt(B, st, a, lev) bt(A1 > A2, st, a, lev) :=
```

```
a' : JFALSE(a);
```

```
ct(C1, st, a' + 1, lev)
```

#### Intermediate code:

```
trans(in/out x; K.) 1 : CALL(a0, 0, 1);
```

```
2 : JMP(0);
```

```
kt(K, stI/O, a0, 1)
```

```
1 : CALL(a0, 0, 1);
```

```
2 : JMP(0);
```

```
dt(D, update(D, stI/O, 1), 1)
```

```
ct(C, update(D, stI/O, 1), a0, 1)
```

```
a2 : RET;
```

```
1 : CALL(a0, 0, 1);
```

```
2 : JMP(0);
```

```
dt(D, st', 1)
```

```
ct(C, st', a0, 1)
```

```
a2 : RET;
```

```
1 : CALL(a0, 0, 1);
```

```
2 : JMP(0);
```

dt(proc l<sub>1</sub>; K<sub>1</sub>; ...)

```
kt(KF, st', a1, 2)
```

```
ct(C, st', a0, 1)
```

```
a2 : RET;
```

```
1 : CALL(a0, 0, 1);
```

```
2 : JMP(0);
```

```
ct(CF, st', a1, 2)
```

```
a3 : RET;
```

```
ct(C, st', a0, 1)
```

```
1 : CALL(a0, 0, 1);
```

```
2 : JMP(0);
```

# A Translation Example

## Example: Factorial Function II

### Example 16.13 (Factorial function; continued)

#### Code with symbolic addresses:

```
1 : CALL(a0,0,1);
2 : JMP(0);
a1 : LOAD(2,1);
      LIT(1);
      GT;
a4 : JFALSE(a3);
      LOAD(1,1);
      LOAD(2,1);
      MULT;
      STORE(1,1);
      LOAD(2,1);
      LIT(1);
      SUB;
      STORE(2,1);
      CALL(a1,1,0);
a3 : RET;
a0 : LIT(1);
      STORE(0,1);
      CALL(a1,0,0);
      LOAD(0,1);
      STORE(1,1);
a2 : RET;
```

#### Linearised ( $a_0 = 17, a_1 = 3, a_2 = 22, a_3 = 16, a_4 = 6$ ):

```
1 : CALL(17,0,1);
2 : JMP(0);
3 : LOAD(2,1);
4 : LIT(1);
5 : GT;
6 : JFALSE(16);
7 : LOAD(1,1);
8 : LOAD(2,1);
9 : MULT;
10 : STORE(1,1);
11 : LOAD(2,1);
12 : LIT(1);
13 : SUB;
14 : STORE(2,1);
15 : CALL(3,1,0);
16 : RET;
17 : LIT(1);
18 : STORE(0,1);
19 : CALL(3,0,0);
20 : LOAD(0,1);
21 : STORE(1,1);
22 : RET;
```

# A Translation Example

## Example: Factorial Function III

### Example 16.13 (Factorial function; continued)

Computation for  $x = 2$ :

	PC	DS	PS
1 : CALL(17,0,1);	1	$\epsilon$	0 : 0 : 0 : 2
2 : JMP(0);	17	$\epsilon$	4 : 3 : 2 : 0 : 0 : 0 : 0 : 2
3 : LOAD(2,1);	18	1	4 : 3 : 2 : 0 : 0 : 0 : 0 : 2
4 : LIT(1);	19	$\epsilon$	4 : 3 : 2 : 1 : 0 : 0 : 0 : 2
5 : GT;	3	$\epsilon$	3 : 2 : 20 : 4 : 3 : 2 : 1 : 0 : 0 : 0 : 2
6 : JFALSE(16);	4	2	3 : 2 : 20 : 4 : 3 : 2 : 1 : 0 : 0 : 0 : 2
7 : LOAD(1,1);	5	2 : 1	3 : 2 : 20 : 4 : 3 : 2 : 1 : 0 : 0 : 0 : 2
8 : LOAD(2,1);	6	1	3 : 2 : 20 : 4 : 3 : 2 : 1 : 0 : 0 : 0 : 2
9 : MULT;	7	$\epsilon$	3 : 2 : 20 : 4 : 3 : 2 : 1 : 0 : 0 : 0 : 2
10 : STORE(1,1);	8	1	3 : 2 : 20 : 4 : 3 : 2 : 1 : 0 : 0 : 0 : 2
11 : LOAD(2,1);	9	1 : 2	3 : 2 : 20 : 4 : 3 : 2 : 1 : 0 : 0 : 0 : 2
12 : LIT(1);	10	2	3 : 2 : 20 : 4 : 3 : 2 : 1 : 0 : 0 : 0 : 2
13 : SUB;	11	$\epsilon$	3 : 2 : 20 : 4 : 3 : 2 : 2 : 0 : 0 : 0 : 2
14 : STORE(2,1);	12	2	3 : 2 : 20 : 4 : 3 : 2 : 2 : 0 : 0 : 0 : 2
15 : CALL(3,1,0);	13	2 : 1	3 : 2 : 20 : 4 : 3 : 2 : 2 : 0 : 0 : 0 : 2
16 : RET;	14	1	3 : 2 : 20 : 4 : 3 : 2 : 2 : 0 : 0 : 0 : 2
17 : LIT(1);	15	$\epsilon$	3 : 2 : 20 : 4 : 3 : 2 : 2 : 0 : 0 : 0 : 1
18 : STORE(0,1);	3	$\epsilon$	6 : 2 : 16 : 3 : 2 : 20 : 4 : 3 : 2 : 2 : 0 : 0 : 0 : 1
19 : CALL(3,0,0);	4	1	6 : 2 : 16 : 3 : 2 : 20 : 4 : 3 : 2 : 2 : 0 : 0 : 0 : 1
20 : LOAD(0,1);	5	1 : 1	6 : 2 : 16 : 3 : 2 : 20 : 4 : 3 : 2 : 2 : 0 : 0 : 0 : 1
21 : STORE(1,1);	6	0	6 : 2 : 16 : 3 : 2 : 20 : 4 : 3 : 2 : 2 : 0 : 0 : 0 : 1
22 : RET;	16	$\epsilon$	6 : 2 : 16 : 3 : 2 : 20 : 4 : 3 : 2 : 2 : 0 : 0 : 0 : 1
	16	$\epsilon$	3 : 2 : 20 : 4 : 3 : 2 : 2 : 0 : 0 : 0 : 1
	20	$\epsilon$	4 : 3 : 2 : 2 : 0 : 0 : 0 : 1
	21	2	4 : 3 : 2 : 2 : 0 : 0 : 0 : 1
	22	$\epsilon$	4 : 3 : 2 : 2 : 0 : 0 : 0 : 2
	2	$\epsilon$	0 : 0 : 0 : 2
	0	$\epsilon$	0 : 0 : 0 : 2

# Correctness of the Translation

---

## Correctness of the Translation

### Theorem 16.14 (Correctness of translation)

For every  $P \in \text{Pgm}$ ,  $n \in \mathbb{N}$ , and  $(z_1, \dots, z_n), (z'_1, \dots, z'_n) \in \mathbb{Z}^n$ :

$$\begin{aligned} & \llbracket P \rrbracket(z_1, \dots, z_n) = (z'_1, \dots, z'_n) \\ \iff & \llbracket \text{trans}(P) \rrbracket(1, \varepsilon, 0 : 0 : 0 : z_1 : \dots : z_n) = (0, \varepsilon, 0 : 0 : 0 : z'_1 : \dots : z'_n) \end{aligned}$$

### Proof.

see M. Mohnen: *A Compiler Correctness Proof for the Static Link Technique by means of Evolving Algebras*, Fundamenta Informaticae 29(3), 1997, pp. 257–303  $\square$