Compiler Construction

Lecture 17: Code Generation II (The Compiler)

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/cc12/

Summer Semester 2012

Online Registration for Seminars and Practical Courses (Praktika) in Winter Term 2012/13

Who?

Students of:
- Master Courses
- Bachelor Informatik (~~ProSeminar~~!)

Where?

web-info8.informatik.rwth-aachen.de/apse

When?

18.06.2012 - 01.07.2012

Outline

Syntax of EPL

Definition (Syntax of EPL)

The syntax of EPL is defined as follows:

$$
\begin{array}{lll}
\mathbb{Z}: & z & (*\ z \text{ is an integer } *) \\
\mathit{Ide}: & I & (*\ I \text{ is an identifier } *) \\
\mathit{AExp}: & A ::= z \mid I \mid A_1 + A_2 \mid \ldots & \\
\mathit{BExp}: & B ::= A_1 < A_2 \mid \texttt{not}\ B \mid B_1\ \texttt{and}\ B_2 \mid B_1\ \texttt{or}\ B_2 & \\
\mathit{Cmd}: & C ::= I\ \texttt{:=}\ A \mid C_1\,\texttt{;}\,C_2 \mid \texttt{if}\ B\ \texttt{then}\ C_1\ \texttt{else}\ C_2 \mid & \\
 & \quad \texttt{while}\ B\ \texttt{do}\ C \mid I\,\texttt{()} & \\
\mathit{Dcl}: & D ::= D_C\ D_V\ D_P & \\
 & D_C ::= \varepsilon \mid \texttt{const}\ I_1\ \texttt{:=}\ z_1, \ldots, I_n\ \texttt{:=}\ z_n\texttt{;} & \\
 & D_V ::= \varepsilon \mid \texttt{var}\ I_1, \ldots, I_n\texttt{;} & \\
 & D_P ::= \varepsilon \mid \texttt{proc}\ I_1\texttt{;}K_1\texttt{;} \ldots \texttt{;proc}\ I_n\texttt{;}K_n\texttt{;} & \\
\mathit{Blk}: & K ::= D\ C & \\
\mathit{Pgm}: & P ::= \texttt{in/out}\ I_1, \ldots, I_n\texttt{;}K\,\texttt{.} &
\end{array}
$$

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 $s = (l, d, p) \in S$ is given by

- a program label $l \in PC$,
- a data stack $d = d.r : \ldots : d.1 \in DS$, and
- a procedure stack $p = p.1 : \ldots : p.t \in PS$.

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

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

where

static link sl: points to frame of surrounding declaration environment
$\Longrightarrow$ used to access non-local variables

dynamic link dl: points to previous frame (i.e., of calling procedure)
$\Longrightarrow$ used to remove topmost frame after termination of procedure call

return address ra: program label after termination of procedure call
$\Longrightarrow$ used to continue program execution after termination of procedure call

local variables v_i: values of locally declared variables

Structure of Procedure Stack IV

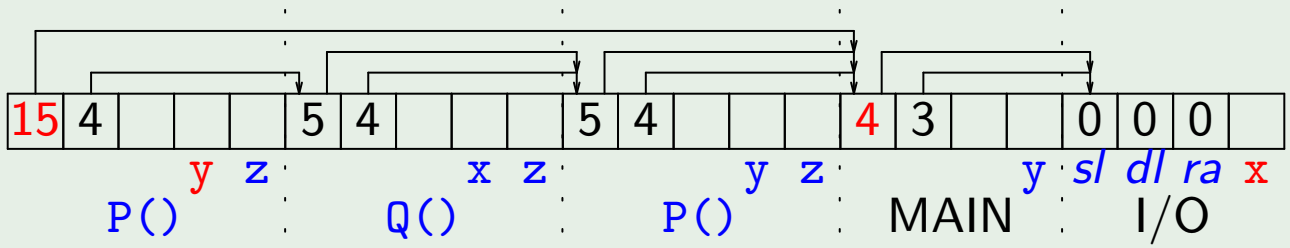
Observation:

- 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 16.9)

```
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 $\Longrightarrow$ *dif* $= 2$ P uses y $\Longrightarrow$ *dif* $= 0$

Outline

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 the new frame and jumps to the given address
  (= starting address of procedure)
- `RET` removes the topmost frame and returns to the calling site

Definition 17.1 (Semantics of procedure instructions)

The semantics of a procedure instruction O, $[\![O]\!] : S \dashrightarrow S$, is defined as follows:

$$[\![\texttt{CALL}(ca, dif, loc)]\!](l, d, p)$$
$$:= (ca, d, \underbrace{(\mathrm{base}(p, dif) + loc + 2)}_{sl} : \underbrace{(loc + 2)}_{dl} : \underbrace{(l + 1)}_{ra} : \underbrace{0 : \ldots : 0}_{loc \text{ times}} : p)$$

$$[\![\texttt{RET}]\!](l, d, p.1 : \ldots : p.t)$$
$$:= (\underbrace{p.3}_{ra}, d, p.(\underbrace{p.2}_{dl} + 2) : \ldots : p.t) \qquad \text{if } t \geq p.2 + 2$$

Semantics of Transfer Instructions

- $\texttt{LOAD}(dif, off)$ and $\texttt{STORE}(dif, off)$ 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
- $\texttt{LIT}(z)$ loads the literal constant $z \in \mathbb{Z}$

Definition 17.2 (Semantics of transfer instructions)

The semantics of a transfer instruction O, $[\![O]\!] : S \dashrightarrow S$, is defined as follows:

$$[\![\texttt{LOAD}(dif, off)]\!](l, d, p) := (l + 1, d : p.(\text{base}(p, dif) + off + 2), p)$$
$$[\![\texttt{STORE}(dif, off)]\!](l, d : z, p) := (l + 1, d, p[\text{base}(p, dif) + off + 2 \mapsto z])$$
$$[\![\texttt{LIT}(z)]\!](l, d, p) := (l + 1, d : z, p)$$

AM Programs and Their Semantics

Definition 17.3 (Semantics of AM programs)

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

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

The set of all AM programs is denoted by AM.

The semantics of AM programs is determined by

$$[\![.]\!] : AM \times S \dashrightarrow S$$

with

$$[\![P]\!](l, d, p) := \begin{cases} [\![P]\!]([\![O_l]\!](l, d, p)) & \text{if } l \in [k] \\ (l, d, p) & \text{otherwise} \end{cases}$$

Outline

Structure of Symbol Table

Goal: define translation mapping $\mathrm{trans} : \mathit{Pgm} \dashrightarrow \mathit{AM}$

The translation employs a symbol table:

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

whose entries are created by declarations:

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

Maintaining the Symbol Table

The symbol table is maintained by the function $\mathrm{update}(D, \mathrm{st}, l)$ which specifies the update of symbol table st according to declaration D (with respect to current level l):

Definition 17.4 (update function)

$$\mathrm{update} : \mathit{Dcl} \times \mathit{Tab} \times \mathit{Lev} \dashrightarrow \mathit{Tab}$$

is defined by

$$\begin{aligned}
&\mathrm{update}(D_C\ D_V\ D_P, \mathrm{st}, l)\\
&\quad := \mathrm{update}(D_P, \mathrm{update}(D_V, \mathrm{update}(D_C, \mathrm{st}, l), l), l)\\
&\qquad \text{if all identifiers in } D_C\ D_V\ D_P \text{ different}\\
&\mathrm{update}(\varepsilon, \mathrm{st}, l)\\
&\quad := \mathrm{st}\\
&\mathrm{update}(\texttt{const}\ I_1\ \texttt{:=}\ z_1\texttt{,}\ \ldots\texttt{,}\ I_n\ \texttt{:=}\ z_n\texttt{;}, \mathrm{st}, l)\\
&\quad := \mathrm{st}[I_1 \mapsto (\texttt{const}, z_1), \ldots, I_n \mapsto (\texttt{const}, z_n)]\\
&\mathrm{update}(\texttt{var}\ I_1\texttt{,}\ \ldots\texttt{,}\ I_n\texttt{;}, \mathrm{st}, l)\\
&\quad := \mathrm{st}[I_1 \mapsto (\texttt{var}, l, 1), \ldots, I_n \mapsto (\texttt{var}, l, n)]\\
&\mathrm{update}(\texttt{proc}\ I_1\texttt{;}K_1\texttt{;}\ \ldots\texttt{;proc}\ I_n\texttt{;}K_n\texttt{;}, \mathrm{st}, l)\\
&\quad := \mathrm{st}[I_1 \mapsto (\texttt{proc}, a_1, l, \mathrm{size}(K_1)), \ldots, I_n \mapsto (\texttt{proc}, a_n, l, \mathrm{size}(K_n))]\\
&\qquad \text{with ``fresh'' addresses } a_1, \ldots, a_n\\
&\qquad \text{where } \mathrm{size}(D_C\ \texttt{var}\ I_1\texttt{,}\ \ldots\texttt{,}\ I_n\texttt{;}\ D_P C) := n
\end{aligned}$$

The Initial Symbol Table

Reminder: an EPL program $P = \texttt{in/out}\ I_1, \ldots, I_n; K\,. \in Pgm$ has a semantics of type $\mathbb{Z}^n \dashrightarrow \mathbb{Z}^n$.

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

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

Thus the corresponding initial symbol table has n entries:

$$\mathrm{st}_{I/O}(I_j) := (\texttt{var}, 0, j) \qquad \text{for every } j \in [n]$$

Outline

Translation of Programs

Translation of $\texttt{in/out}\ I_1, \ldots, I_n; D\ C.$:

1. Create MAIN frame for executing C
2. Stop program execution after return

Definition 17.5 (Translation of programs)

The mapping

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

is defined by

$$\begin{aligned}\text{trans}(\texttt{in/out}\ I_1, \ldots, I_n; K.) := {} & 1 : \texttt{CALL}(a, 0, \text{size}(K)); \\ & 2 : \texttt{JMP}(0); \\ & \text{kt}(K, \text{st}_{I/O}, a, 1)\end{aligned}$$

Outline

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 – recursion!)
3. Create code for C (using the updated symbol table)

Definition 17.6 (Translation of blocks)

The mapping

$$\mathrm{kt} : \mathit{Blk} \times \mathit{Tab} \times \mathit{PC} \times \mathit{Lev} \dashrightarrow \mathit{AM}$$

("block translation") is defined by

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

Outline

Translation of Declarations

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

Definition 17.7 (Translation of declarations)

The mapping

$$\mathrm{dt} : \mathit{Dcl} \times \mathit{Tab} \times \mathit{Lev} \dashrightarrow \mathit{AM}$$

("declaration translation") is defined by

$$\begin{aligned}
&\mathrm{dt}(D_C\ D_V\ D_P, \mathrm{st}, l) \\
&\quad := \quad \mathrm{dt}(D_P, \mathrm{st}, l) \\
&\mathrm{dt}(\varepsilon, \mathrm{st}, l) \\
&\quad := \quad \varepsilon \\
&\mathrm{dt}(\texttt{proc}\ I_1\texttt{;}K_1\texttt{;}\ldots\texttt{;proc}\ I_n\texttt{;}K_n\texttt{;}, \mathrm{st}, l) \\
&\quad := \quad \mathrm{kt}(K_1, \mathrm{st}, a_1, l+1) \\
&\qquad\qquad \vdots \\
&\qquad\qquad \mathrm{kt}(K_n, \mathrm{st}, a_n, l+1) \\
&\qquad\qquad \text{where } \mathrm{st}(I_j) = (\texttt{proc}, a_j, \ldots, \ldots) \text{ for every } j \in [n]
\end{aligned}$$

Outline

Translation of Commands

Definition 17.8 (Translation of commands)

The mapping

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

("command translation") is defined by

$$\begin{aligned}
\mathrm{ct}(I := A, \mathrm{st}, a, l) &:= \mathrm{at}(A, \mathrm{st}, a, l) \\
&\quad a' : \texttt{STORE}(l - \mathit{lev}, \mathit{off}); \\
&\quad \text{if } \mathrm{st}(I) = (\texttt{var}, \mathit{lev}, \mathit{off}) \\
\mathrm{ct}(I(), \mathrm{st}, a, l) &:= a : \texttt{CALL}(\mathit{ca}, l - \mathit{lev}, \mathit{loc}); \\
&\quad \text{if } \mathrm{st}(I) = (\texttt{proc}, \mathit{ca}, \mathit{lev}, \mathit{loc}) \\
\mathrm{ct}(C_1; C_2, \mathrm{st}, a, l) &:= \mathrm{ct}(C_1, \mathrm{st}, a, l) \\
&\quad \mathrm{ct}(C_2, \mathrm{st}, a', l) \\
\mathrm{ct}(\texttt{if } B \texttt{ then } C_1 \texttt{ else } C_2, \mathrm{st}, a, l) &:= \mathrm{bt}(B, \mathrm{st}, a, l) \\
&\quad a' : \texttt{JFALSE}(a''); \\
&\quad \mathrm{ct}(C_1, \mathrm{st}, a' + 1, l) \\
&\quad a'' - 1 : \texttt{JMP}(a'''); \\
&\quad \mathrm{ct}(C_2, \mathrm{st}, a'', l) \\
&\quad a''' : \\
\mathrm{ct}(\texttt{while } B \texttt{ do } C, \mathrm{st}, a, l) &:= \mathrm{bt}(B, \mathrm{st}, a, l) \\
&\quad a' : \texttt{JFALSE}(a'' + 1); \\
&\quad \mathrm{ct}(C, \mathrm{st}, a' + 1, l) \\
&\quad a'' : \texttt{JMP}(a);
\end{aligned}$$

Outline

Translation of Boolean Expressions

Definition 17.9 (Translation of Boolean expressions)

The mapping

$$\mathrm{bt} : \mathit{BExp} \times \mathit{Tab} \times \mathit{PC} \times \mathit{Lev} \dashrightarrow \mathit{AM}$$

("Boolean expression translation") is defined by

$$\begin{array}{rcl}
\mathrm{bt}(A_1 < A_2, \mathrm{st}, a, l) & := & \mathrm{at}(A_1, \mathrm{st}, a, l) \\
& & \mathrm{at}(A_2, \mathrm{st}, a', l) \\
& & a'' : \texttt{LT;} \\
\mathrm{bt}(\texttt{not } B, \mathrm{st}, a, l) & := & \mathrm{bt}(B, \mathrm{st}, a, l) \\
& & a' : \texttt{NOT;} \\
\mathrm{bt}(B_1 \texttt{ and } B_2, \mathrm{st}, a, l) & := & \mathrm{bt}(B_1, \mathrm{st}, a, l) \\
& & \mathrm{bt}(B_2, \mathrm{st}, a', l) \\
& & a'' : \texttt{AND;} \\
\mathrm{bt}(B_1 \texttt{ or } B_2, \mathrm{st}, a, l) & := & \mathrm{bt}(B_1, \mathrm{st}, a, l) \\
& & \mathrm{bt}(B_2, \mathrm{st}, a', l) \\
& & a'' : \texttt{OR;}
\end{array}$$

Translation of Arithmetic Expressions

Definition 17.10 (Translation of arithmetic expressions)

The mapping

$$\mathrm{at} : \mathit{AExp} \times \mathit{Tab} \times \mathit{PC} \times \mathit{Lev} \dashrightarrow \mathit{AM}$$

("arithmetic expression translation") is defined by

$$\begin{aligned}
\mathrm{at}(z, \mathrm{st}, a, l) &:= a : \texttt{LIT}(z); \\
\mathrm{at}(l, \mathrm{st}, a, l) &:= \begin{cases} a : \texttt{LIT}(z); & \text{if } \mathrm{st}(l) = (\texttt{const}, z) \\ a : \texttt{LOAD}(l - \mathit{lev}, \mathit{off}); & \text{if } \mathrm{st}(l) = (\texttt{var}, \mathit{lev}, \mathit{off}) \end{cases} \\
\mathrm{at}(A_1 + A_2, \mathrm{st}, a, l) &:= \mathrm{at}(A_1, \mathrm{st}, a, l) \\
&\quad\ \mathrm{at}(A_2, \mathrm{st}, a', l) \\
&\quad\ a'' : \texttt{ADD};
\end{aligned}$$

Outline

Example: Factorial Function I

Example 17.11 (Factorial function; cf. Example 16.3)

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.
```

$\mathrm{trans}(\texttt{in/out}\ I_1, \ldots, I_n; K.) :=$

$1 : \texttt{CALL}(a, 0, \mathrm{size}(K)); \mathrm{kt}(D\ C, \mathrm{st}, a, l) :=$

$2 : \texttt{JMP}(0);$

$\mathrm{kt}(K, \mathrm{st}_{I/O}, a, 1)$

$\mathrm{dt}(D, \mathrm{update}(D, \mathrm{st}, l), l)$ $\quad \mathrm{update}(\texttt{var}\ I_1, \ldots, I_n;, \mathrm{st}, l) :=$

$\mathrm{ct}(C, \mathrm{update}(D, \mathrm{st}, l), a, l)$ $\quad \mathrm{st}[I_1 \mapsto (\texttt{var}, l, 1), \ldots, I_n \mapsto (\texttt{var}, l, n)]$

$a' : \texttt{RET};$ $\quad \mathrm{update}(\texttt{proc}\ I_1; K_1; \ldots; \texttt{proc}\ I_n; K_n;, \mathrm{st}, l) :=$

$\mathrm{st}[I_1 \mapsto (\texttt{proc}, a_1, l, \mathrm{size}(K_1)), \ldots, I_n \mapsto (\texttt{proc}, a_n, l, \mathrm{size}(K_n))]$

$\mathrm{kt}(K_1, \mathrm{st}, a_1, l+1)$ $\quad \mathrm{ct}(\texttt{if}\ B\ \texttt{then}\ C_1\ \texttt{else}\ C_2, \mathrm{st}, a, l) :=$

$\vdots$

Intermediate code:

$\mathrm{trans}(\texttt{in/out}\ x; K.)$

$1 : \texttt{CALL}(a_0, 0, 1);$

$2 : \texttt{JMP}(0);$

$\mathrm{kt}(K, \mathrm{st}_{I/O}, a_0, 1)$

$1 : \texttt{CALL}(a_0, 0, 1);$

$2 : \texttt{JMP}(0);$

$\mathrm{dt}(D, \mathrm{update}(D, \mathrm{st}_{I/}$

$\mathrm{ct}(C, \mathrm{update}(D, \mathrm{st}_{I/}$

$a_2 : \texttt{RET};$

$1 : \texttt{CALL}(a_0, 0, 1);$

$2 : \texttt{JMP}(0);$

$\mathrm{dt}(D, \mathrm{st}', 1)$

$\mathrm{ct}(C, \mathrm{st}', a_0, 1)$

$a_2 : \texttt{RET};$

$1 : \texttt{CALL}(a_0, 0, 1);$

$2 : \texttt{JMP}(0);$

$\mathrm{kt}(K_F, \mathrm{st}', a_1, 2)$

$\mathrm{ct}(C, \mathrm{st}', a_0, 1)$

$a_2 : \texttt{RET};$

$1 : \texttt{CALL}(a_0, 0, 1);$

$2 : \texttt{JMP}(0);$

$\mathrm{ct}(C_F, \mathrm{st}', a_1, 2)$

$a_3 : \texttt{RET};$

Example: Factorial Function II

Example 17.11 (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;
```

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

```
 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;
```

Example: Factorial Function III

Example 17.11 (Factorial function; continued)

Computation for $\mathtt{x} = 2$:

```
 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;
```

PC	*DS*	*PS*
1	ε	0 : 0 : 0 : 2
17	ε	4 : 3 : 2 : 0 : 0 : 0 : 0 : 2
18	1	4 : 3 : 2 : 0 : 0 : 0 : 0 : 2
19	ε	4 : 3 : 2 : 1 : 0 : 0 : 0 : 2
3	ε	3 : 2 : 20 : 4 : 3 : 2 : 1 : 0 : 0 : 0 : 2
4	2	3 : 2 : 20 : 4 : 3 : 2 : 1 : 0 : 0 : 0 : 2
5	2 : 1	3 : 2 : 20 : 4 : 3 : 2 : 1 : 0 : 0 : 0 : 2
6	1	3 : 2 : 20 : 4 : 3 : 2 : 1 : 0 : 0 : 0 : 2
7	ε	3 : 2 : 20 : 4 : 3 : 2 : 1 : 0 : 0 : 0 : 2
8	1	3 : 2 : 20 : 4 : 3 : 2 : 1 : 0 : 0 : 0 : 2
9	1 : 2	3 : 2 : 20 : 4 : 3 : 2 : 1 : 0 : 0 : 0 : 2
10	2	3 : 2 : 20 : 4 : 3 : 2 : 1 : 0 : 0 : 0 : 2
11	ε	3 : 2 : 20 : 4 : 3 : 2 : 2 : 0 : 0 : 0 : 2
12	2	3 : 2 : 20 : 4 : 3 : 2 : 2 : 0 : 0 : 0 : 2
13	2 : 1	3 : 2 : 20 : 4 : 3 : 2 : 2 : 0 : 0 : 0 : 2
14	1	3 : 2 : 20 : 4 : 3 : 2 : 2 : 0 : 0 : 0 : 2
15	ε	3 : 2 : 20 : 4 : 3 : 2 : 2 : 0 : 0 : 0 : 1
3	ε	6 : 2 : 16 : 3 : 2 : 20 : 4 : 3 : 2 : 2 : 0 : 0 : 0 : 1
4	1	6 : 2 : 16 : 3 : 2 : 20 : 4 : 3 : 2 : 2 : 0 : 0 : 0 : 1
5	1 : 1	6 : 2 : 16 : 3 : 2 : 20 : 4 : 3 : 2 : 2 : 0 : 0 : 0 : 1
6	0	6 : 2 : 16 : 3 : 2 : 20 : 4 : 3 : 2 : 2 : 0 : 0 : 0 : 1
16	ε	6 : 2 : 16 : 3 : 2 : 20 : 4 : 3 : 2 : 2 : 0 : 0 : 0 : 1
16	ε	3 : 2 : 20 : 4 : 3 : 2 : 2 : 0 : 0 : 0 : 1
20	ε	4 : 3 : 2 : 2 : 0 : 0 : 0 : 1
21	2	4 : 3 : 2 : 2 : 0 : 0 : 0 : 1
22	ε	4 : 3 : 2 : 2 : 0 : 0 : 0 : 2
2	ε	0 : 0 : 0 : 2
0	ε	0 : 0 : 0 : 2

Outline

Correctness of the Translation

Theorem 17.12 (Correctness of translation)

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

$$\begin{aligned} & \mathfrak{M}[\![P]\!](z_1, \ldots, z_n) = (z'_1, \ldots, z'_n) \\ \iff & [\![\mathrm{trans}(P)]\!](1, \varepsilon, 0 : 0 : 0 : z_1 : \ldots : z_n) = (0, \varepsilon, 0 : 0 : 0 : z'_1 : \ldots : z'_n) \end{aligned}$$

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 □

Outline

Outlook

1. More about code generation:
   - Handling of procedure parameters
   - Static & dynamic data structures
   - Compiler backend (register allocation, instruction selection & placement, ...)
   - Code analysis & optimization
     $\Longrightarrow$ *Static Program Analysis* in Winter semester
2. June 27/28, July 4: Computational Differentiation [Naumann]
3. July 11: preparation of exam (question time)
4. July 12: 1st exam