source language $SL$ $\rightarrow$ target language $TL$

- translation

- input $s \in SL$ (determined by scanner, parser, and attribute evaluators)

- translation based on (attributed) abstract syntax tree:
  syntax directed translation
  (attributes can also play a role (semantics))

- translation should be semantics preserving
translating infix into postfix expressions

source language infix expressions
target language postfix expressions (reverse Polish)
(c.f. computation of expressions using a stack)

\[

e \rightarrow e + t
\]
\[
 e \rightarrow t
\]
\[
 t \rightarrow t \ast f
\]
\[
 t \rightarrow f
\]
\[
 f \rightarrow (e)
\]
\[
 f \rightarrow p | q | r
\]
\[
 e' \rightarrow e' t' +
\]
\[
 e' \rightarrow t'
\]
\[
 t' \rightarrow t' f' \ast
\]
\[
 t' \rightarrow f'
\]
\[
 f' \rightarrow e' \quad (parentheses \text{ unnecessary})
\]
\[
 f' \rightarrow p | q | r
\]
parallel derivations (using corresponding rules)

<table>
<thead>
<tr>
<th>Expression</th>
<th>Derivation</th>
</tr>
</thead>
<tbody>
<tr>
<td>( E )</td>
<td>( \Rightarrow E + T )</td>
</tr>
<tr>
<td>( E ) ( \Rightarrow 2 )</td>
<td>( \Rightarrow E' + T )</td>
</tr>
<tr>
<td>( T ) ( \Rightarrow 2 )</td>
<td>( \Rightarrow T' F' + T' )</td>
</tr>
<tr>
<td>( (E) ) ( \Rightarrow 2 )</td>
<td>( \Rightarrow E' F' + T' )</td>
</tr>
<tr>
<td>( (E + T) ) ( \Rightarrow )</td>
<td>( \Rightarrow E' + F' + T' )</td>
</tr>
<tr>
<td>( (p + q) ) ( \Rightarrow )</td>
<td>( \Rightarrow p q F' + T' )</td>
</tr>
<tr>
<td>( (p + q) ) ( \Rightarrow )</td>
<td>( \Rightarrow p q r T' )</td>
</tr>
<tr>
<td>( (p + q) ) ( \Rightarrow )</td>
<td>( \Rightarrow p q r T' F' + + )</td>
</tr>
<tr>
<td>( (p + q) ) ( \Rightarrow )</td>
<td>( \Rightarrow p q + r T' F' + + )</td>
</tr>
<tr>
<td>( (p + q) ) ( \Rightarrow )</td>
<td>( \Rightarrow p q + r T' F' + + )</td>
</tr>
<tr>
<td>( (p + q) ) ( \Rightarrow )</td>
<td>( \Rightarrow p q + r T' F' + + )</td>
</tr>
<tr>
<td>( (p + q) ) ( \Rightarrow )</td>
<td>( \Rightarrow p q + r T' F' + + )</td>
</tr>
<tr>
<td>( (p + q) ) ( \Rightarrow )</td>
<td>( \Rightarrow p q + r T' F' + + )</td>
</tr>
<tr>
<td>( (p + q) ) ( \Rightarrow )</td>
<td>( \Rightarrow p q + r T' F' + + )</td>
</tr>
<tr>
<td>( (p + q) ) ( \Rightarrow )</td>
<td>( \Rightarrow p q + r T' F' + + )</td>
</tr>
<tr>
<td>( (p + q) ) ( \Rightarrow )</td>
<td>( \Rightarrow p q + r T' F' + + )</td>
</tr>
<tr>
<td>( (p + q) ) ( \Rightarrow )</td>
<td>( \Rightarrow p q + r T' F' + + )</td>
</tr>
<tr>
<td>( (p + q) ) ( \Rightarrow )</td>
<td>( \Rightarrow p q + r T' F' + + )</td>
</tr>
<tr>
<td>( (p + q) ) ( \Rightarrow )</td>
<td>( \Rightarrow p q + r T' F' + + )</td>
</tr>
<tr>
<td>( (p + q) ) ( \Rightarrow )</td>
<td>( \Rightarrow p q + r T' F' + + )</td>
</tr>
</tbody>
</table>

\( \tau_E(e + t) = \tau_E(e) \tau_T(t) + \)

(translation function for each nonterminal)
translation functions (in terms of concrete syntax) (1)

$$\tau_E : L(E) \rightarrow \Gamma^*$$

$L(E)$ restricted by context conditions

output alphabet $\Gamma = \{p, q, r, +, *\}$  (alphabet of target language)

interpretation of $\tau_E(e)$:

parse $e$ according to grammar (start symbol $E$) yielding (unique) parse tree $t_e$

type of the root of $t_e$ determines translation $\tau_E(e)$ in terms of translations of the subtrees

$$\tau_E(e + t) = \tau_E(e)\tau_T(t) +$$
$$\tau_E(t) = \tau_T(t)$$

$$\tau_T(t \ast f) = \tau_T(t)\tau_F(f)\ast$$
$$\tau_T(f) = \tau_F(f)$$

$$\tau_F((e)) = \tau_E(e)$$

$$\tau_F(n) = n \quad n \in \{p, q, r\}$$
\[
\tau_E((p + q) * r + q * r) \\
= \tau_E((p + q) * r) \tau_T(r) + \\
= \tau_T((p + q) * r) \tau_T(q) \tau_F(r) + \\
= \tau_T((p + q) \tau_F(r) \tau_T(q) \tau_F(r) + \\
= p q + r * q r * +
\]
translating functions (in terms of abstract syntax)

signature (describing abstract syntax)

| sorts:       | $E, OP, ID$       |
| operators:   | $bin : E \times OP \times E \rightarrow E$ |
|              | $var : ID \rightarrow E$ |
|              | $+, * : \rightarrow OP$ |
|              | $p, q, r : \rightarrow ID$ |

syntax

| $E$  | $\rightarrow_{bin} E \ OP \ E$ |
| $E$  | $\rightarrow_{var} ID$ |
| $OP$ | $\rightarrow + | *$ |
| $ID$ | $\rightarrow p | q | r$ |

translation function

$$\tau_E(\ bin(e_1, o, e_2)) = \tau_E(e_1) \tau_E(e_2) o \quad o \in \{+,*\}$$
$$\tau_E(\ var(n)) = n \quad n \in \{p,q,r\}$$

more readable (possibly ambiguous)

$$\tau_E(e_1 \ o \ e_2) = \tau_E(e_1) \tau_E(e_2) o$$
$$\tau_E(n) = n$$
signature (describing abstract syntax)

sorts: \( E, OP, ID \)

operators: \( bin : E \times OP \times E \rightarrow E \)
\( var : ID \rightarrow E \)
\( +, \ast : \rightarrow OP \)
\( p, q, r : \rightarrow ID \)

variant (translation function for each sort; more general)

\[
\begin{align*}
\tau_E(\ bin(e_1, o, e_2)) &= \tau_E(e_1)\tau_E(e_2)\tau_{OP}(o) \\
\tau_E(\ var(n)) &= \tau_{ID}(n) \\
\tau_{OP}(o) &= o & o \in \{+, \ast\} \\
\tau_{ID}(n) &= n & n \in \{p, q, r\}
\end{align*}
\]
(concrete syntax)

- grammar \( G = (N, \Sigma, P, S) \)
- target language \( Z \)
- for each nonterminal \( A \in N \) translation function
  \[ \tau_A : L(A) \rightarrow Z \]
  (domain \( L(A) \) possibly restricted by context conditions, additional parameters possible (information on context))
- for each production rule
  \[ p : A \rightarrow x_0A_1x_1 \cdots x_{n-1}A_nx_n \ (x_i \in \Sigma^*, A_i \in N) \]
  a function
  \[ \sigma_p : Z^n \rightarrow Z \]
(concrete syntax, continued)

- if $x \in L(A)$ and

$$A \Rightarrow x_0A_1x_1 \cdots x_{n-1}A_nx_n \Rightarrow^* x_0y_1x_1 \cdots x_{n-1}y_nx_n = x$$

(implying $A_i \Rightarrow^* y_i \ (1 \leq i \leq n))$ then

$$\tau_A(x) = \sigma_p(\tau_{A_1}(y_1), \cdots, \tau_{A_n}(y_n))$$

transitions of subtrees

- examples of SDTS: T-recipe, recursive descent parsing scheme
translation infix to postfix expressions

\[ p : E \rightarrow E + T \]

\[ Z = \Gamma^* \quad \Gamma = \{p, q, r, +, *\} \]

\[ \sigma_p(z_1, z_2) = z_1z_2 + \]

\[ \tau_E(e + t) = \sigma_p(\tau_E(e), \tau_T(t)) = \tau_E(e)\tau_T(t) + \]
(abstract syntax)

- signature \((S, O)\) (\(S\) set of sorts, \(O\) set of operators)
- target language \(Z\)
- for each sort \(s \in S\) translation function

\[
\tau_s : \text{Term}_s \rightarrow Z
\]

where \(\text{Term}_s\) is the set of all terms of sort \(s\)

- for each operator \(o : s_1 \times s_2 \times \cdots \times s_n \rightarrow s\) a function

\[
\sigma_o : Z^n \rightarrow Z
\]

- for each operator \(o : s_1 \times s_2 \times \cdots \times s_n \rightarrow s\) and all terms \(t_1 : s_1, t_2 : s_2, \ldots, \) and \(t_n : s_n\) term \(o(t_1, t_2, \ldots, t_n)\) has sort \(s\), and

\[
\tau_s(o(t_1, t_2, \ldots, t_n)) = \sigma_o(\tau_{s_1}(t_1), \tau_{s_2}(t_2), \ldots, \tau_{s_n}(t_n))
\]

- note that translation function \(\tau_s\) has a variant for each operator with result sort \(s\)
translation infix to postfix expressions

\[ \text{bin} : E \times OP \times E \rightarrow E \]

\[ Z = \Gamma^* \quad \Gamma = \{ p, q, r, +, * \} \]

\[ \sigma_{\text{bin}}(z_1, z_2, z_3) = z_1z_3z_2 \]

\[ \tau_E(\text{bin}(e_1, op, e_2)) = \sigma_{\text{bin}}(\tau_E(e_1), \tau_{OP}(op), \tau_E(e_2)) \]

\[ \tau_{OP}(op) = op \]

notation

for readability the argument of the translation function will usually be written in concrete syntax, i.e.

\[ \tau_{STAT}(if\,1(e, ss)) \]

is written as

\[ \tau_{STAT}(\text{if } e \text{ then } ss \text{ fi}) \]

(in some cases this notation may be ambiguous)
**Concrete Syntax**

Result type of translation function dependent on nonterminal

\[ \tau_A : L(A) \rightarrow Z_A \]

\[ \sigma_p : Z_{A_1} \times Z_{A_2} \times \cdots \times Z_{A_n} \rightarrow Z_A \]

Set \( Z_A \) may also be a Cartesian product

Target language: \( Z_S \) (\( S \) start symbol)

**Abstract Syntax**

Result type of translation function dependent on sort

\[ \tau_s : \text{Term}_s \rightarrow Z_s \]

\[ \sigma_o : s_1 \times s_2 \times \cdots \times s_n \rightarrow s \]

\[ \sigma_o : Z_{s_1} \times Z_{s_2} \times \cdots \times Z_{s_n} \rightarrow Z_s \]

Set \( Z_s \) may also be a Cartesian product

Target language: \( Z_{s_0} \) (\( s_0 \) principal sort)
target language TL

assembly language of a simple and somewhat idealized stack machine

- no registers (except instruction pointer)
- (unbounded) expression stack (contains values)
- separate data and program memory
- data memory consists of cells capable of storing exactly one value of one of the base types

operational semantics

meaning of instructions in TL is expressed in terms of their effect on the state of the stack machine

formal description of the stack machine is given by means of a TL-interpreter
representation of values in TL

integers : $iv(z)$ \hspace{1cm} (\(z \in \mathbb{Z}\))
reals : $rv(r)$ \hspace{1cm} (\(r \in \mathbb{R}\))
booleans : $bv(v)$ \hspace{1cm} (\(b \in \{0, 1\}\))
characters : $cv(c)$ \hspace{1cm} (\(c \in \text{Char}\))

$iv(3), iv(-17)$ \hspace{1cm} $rv(12.83E-13)$
$bv(\text{true})$ \hspace{1cm} $cv(\text{'}?\text{'}), cv(\text{'}p\text{'})$

Value = set of all value representations

data memory addresses

$\text{MemAddress} = \{0, \cdots, \text{Memsize}\}$

labels (for jump instruction)

$\text{Plabel} = \text{set of labels}$
instruction set \textit{Instr}

- arithmetic: ADD, SUB, \ldots
- relational: EQL, NEQ, LEQ, LESS, \ldots
- boolean: AND, OR, NEG
- data memory: \texttt{STO}(m), \texttt{LDV}(m)
- constants: \texttt{LDC}(c)
- stack manipulation: DUP, POP
- input/output: GET, PUT
- range checking: \texttt{CHK}(iv(z_1), iv(z_2))
- jumping: FJP(\textit{l}), TJP(\textit{l}), UJP(\textit{l}), XJP
- termination: HLT

where \( m \in \text{MemAddress}, \ c \in \text{Value}, \ z_1, z_2 \in \mathbb{Z}, \ l \in \text{Plabel} \)
target language TL = \((Instr \cup Plabel)^*\)

notation of TL programs

\[\cdots \oplus \langle LDV(1) \rangle \oplus \langle l_1 \rangle \oplus \langle LDV(2) \rangle \oplus \langle ADD \rangle \oplus \cdots\]

\[\cdots ; LDV(1); \ l_1 : LDV(2); \ ADD; \cdots\]

\[: \]
LDV(1)

\[l_1 : \ LDV(2)\]
ADD

\[: \]
the TL-interpreter inspects the instructions of a TL-program \( p \) and performs the corresponding actions.

**Interpreter state:** two set of variables

- **Data variables**
  
  \[ M : \text{array}[\text{MemAddress}] \text{ of Value} \]
  
  (data memory consisting of Value cells)

  \[ es : \text{Value}^* \]
  
  (expression stack)

  \[ inp : \text{Value}^* \]
  
  (input (file))

  \[ out : \text{Value}^* \]
  
  (output (file))

- **Control variables**

  \[ P : \text{array}[\text{ProgAddress}] \text{ of Instr} \]
  
  (program memory)

  \[ \text{ProgAddress} = \{0, \ldots, \text{Progsize}\} \]

  \[ Lmap : \text{Plabel} \rightarrow \text{ProgAddress} \]
  
  (mapping of (used) labels to program memory addresses)

  \[ ip : \text{ProgAddress} \]
  
  (instruction pointer)
• representation of a TL-program within interpreter

\[
\cdots ; \text{LDV}(1); \; l_1 : \text{LDV}(2); \; \text{ADD}; \; \text{UJP}(l_2); \cdots
\]

\[
\begin{align*}
P[35] &= \text{LDV}(1) & Lmap(l_1) &= 36 \\
P[36] &= \text{LDV}(2) & Lmap(l_2) &= \ldots \\
P[37] &= \text{ADD} & \quad & \\
P[38] &= \text{UJP}(l_2) & \\
\vdots & & \\
\end{align*}
\]
var run : bool; in : Instr; ...

es, ip, run : = ⟨⟨⟩⟩, 0, true;
{ assumption: P[0] contains first instruction to be executed }
do run → in : = P[ip]; ip : = ip + 1;
   case in of
      SUB :
         v1, v2, es : = Top(es), Top(Pop(es)), Pop(Pop(es));
         if v1, v2 :: iv(x1), iv(x2) → es : = ⟨⟨iv(x2 - x1)⟩⟩ ⊕ es
            | v1, v2 :: rv(y1), rv(y2) → es : = ⟨⟨rv(y2 - y1)⟩⟩ ⊕ es
      fi
      ADD : ...

• stack operations (only defined for nonempty stacks)
  \[ Top(⟨⟨v⟩⟩ ⊕ es) = v \quad Pop(⟨⟨v⟩⟩ ⊕ es) = es \]

• pattern matching guards if \( v :: iv(x) \rightarrow S \) ...
  interpretation of guard: “\( v \) is of the form \( iv(x) \)”
  (also introduces and binds an integer variable \( x \) that may only
  be used in statements \( S \) )
SUB:
\[ v_1, v_2, es := \text{Top}(es), \text{Top}(\text{Pop}(es)), \text{Pop}(\text{Pop}(es)); \]
\[ \text{if } v_1, v_2 :: iv(x_1), iv(x_2) \rightarrow es := \langle \langle iv(x_2 - x_1) \rangle \rangle \oplus es \]

- computations are performed on the expression stack

\[
\begin{cases}
es = \begin{array}{c}
iv(x_1) \\
v_1
\end{array} \\
iv(x_2) \\
ES
\end{cases}
\]

SUB
\[
\begin{cases}
es = \begin{array}{c}
v_2
\end{array} \\
iv(x_2 - x_1)
\end{cases}
\]

\[
\begin{cases}
\text{ip} = IP \\
\text{ip} = IP + 1
\end{cases}
\]
case in of

: LEQ : \( v_1, v_2, es : = \text{Top}(es), \text{Top}(\text{Pop}(es)), \text{Pop}(\text{Pop}(es)) \);
  \textbf{if} \( v_1, v_2 :: iv(x_1), iv(x_2) \rightarrow es : = \langle \langle \text{bv}(x_2 \leq x_1) \rangle \rangle \oplus es \)
  \textbf{if} \( v_1, v_2 :: rv(y_1), rv(y_2) \rightarrow es : = \langle \langle \text{bv}(y_2 \leq y_1) \rangle \rangle \oplus es \)
  \textbf{if} \( v_1, v_2 :: bv(b_1), bv(b_2) \rightarrow es : = \langle \langle \text{bv}(b_2 \leq b_1) \rangle \rangle \oplus es \)
  \textbf{if} \( v_1, v_2 :: cv(c_1), cv(c_2) \rightarrow es : = \langle \langle \text{bv}(c_2 \leq c_1) \rangle \rangle \oplus es \)
  \textbf{fi}

LESS : …. 

: 

NEG : \( v, es : = \text{Top}(es), \text{Pop}(es) \);
  \textbf{if} \( v :: iv(x) \rightarrow es : = \langle \langle iv(-x) \rangle \rangle \oplus es \)
  \textbf{if} \( v :: rv(y) \rightarrow es : = \langle \langle rv(-y) \rangle \rangle \oplus es \)
  \textbf{if} \( v :: bv(b) \rightarrow es : = \langle \langle \text{bv}(-b) \rangle \rangle \oplus es \)
  \textbf{fi}

: 


case in of
:
STO(m) : M[m], es : = Top(es), Pop(es)

LDV(m) : es : = ⟨⟨M[m]⟩⟩ ⊕ es

LDC(c) : es : = ⟨⟨c⟩⟩ ⊕ es

DUP : es : = ⟨⟨Top(es)⟩⟩ ⊕ es

POP : es : = Pop(es)

PUT : es, out : = Pop(es), out ⊕ ⟨⟨Top(es)⟩⟩

GET : es, inp : = ⟨⟨First(inp)⟩⟩ ⊕ es, Rest(inp)
case in of

TJP(l) : \( v \), es : = Top(es), Pop(es);
    if \( v :: bv(b) \) →
        if \( b \rightarrow ip : = Lmap(l) \)
        [ ¬b → skip
        fi
    fi
FJP(l) : ...
UJP(l) : ip : = Lmap(l)
XJP : v, es : = Top(es), Pop(es);
    if \( v :: iv(x) \rightarrow ip : = ip - 1 + x \) fi
    \{ -1 since \( ip \) has already been increased by 1 \}
HLT : run : = false
esac

• if \( P[ip] \) is undefined than a \textit{runtime error} occurs.
abstract syntax

\[ EXPR \rightarrow_{bin} EXPR \quad OP_2 \quad EXPR \]
\[ EXPR \rightarrow_{un} \quad OP_1 \quad EXPR \]
\[ EXPR \rightarrow_{intcon} \quad INTREP \]
\[ EXPR \rightarrow_{boolcon} \quad true | false \]
\[ EXPR \rightarrow_{vare} \quad VAR \]
\[ VAR \rightarrow \quad ID \]
\[ OP_2 \rightarrow \quad + | - | \leq | \wedge | \cdots \]
\[ OP_1 \rightarrow \quad - | \text{not} \]
in fact translation of term

\[ \text{bin( } \text{bin( vare( var( x ))), +, intcon(3)), } *, \ vare( \text{var( y ))) \text{)} \]

additional information needed:

data memory locations (addresses) where the values of variables \( x \) and \( y \) are stored, say addresses 31 and 43

\[
\tau_E((x + 3) \times y) = \left\{ \begin{array}{c}
\tau_E(x + 3) = \left\{ \begin{array}{c}
\tau_E(x) \\
\tau_E(3) \\
\tau_{OP_2}(+) \\
\end{array} \right.
\end{array} \right.
\]

\[
\tau_E(y) = \text{LDV(43)}
\]

\[
\tau_{OP_2}(\times) = \text{MUL}
\]

\[
\tau_{E}(3) \text{ = LDV(31)}
\]

\[
\text{LDC(iv(3)) = ADD}
\]

\[
\text{LDV(43) = MUL}
\]

- no separate translation function for \( VAR \)
  (will be introduced for the translation of array variables)
\( \tau_{EXPR} : Term_{EXPR} \times Adrf \rightarrow TL \)

\[ Adrf = \text{Name} \rightarrow \text{MemAddress} \]

if \( \alpha \in Adrf \) then

\[ \alpha(x) = \text{“address in data memory } M \text{ of } x” \]

notation with address function

as additional parameter

\[ \tau_{EXPR}(e)\alpha = \text{“TL-program that, when executed, puts the} \]

value of \( e \) on top of the expression stack \( es \)”

( data memory unchanged, \( ip = ip_{old} + \text{length}(\tau_{EXPR}(e)\alpha) \) )

\[ \{ es = ES \} \tau_{EXPR}(e)\alpha \{ es = \langle \text{“value of } e \rangle \oplus ES \} \]
expressions: translation function (1)

\[ \tau_{EXPR}(e_1 \ op \ e_2)\alpha = \tau_{EXPR}(\text{bin}(e_1, op, e_2))\alpha \]
\[ = \tau_{EXPR}(e_1)\alpha \]
\[ \tau_{EXPR}(e_2)\alpha \]
\[ \tau_{OP_2}(op) \]
\[ e_1, e_2 \in \text{Term}_{EXPR} \]
\[ op \in \text{Term}_{OP_2} \]

\[ \tau_{OP_2}(+) = \text{ADD} \]
\[ \tau_{OP_2}(\wedge) = \text{AND} \]
\[ \tau_{OP_2}(-) = \text{SUB} \]
\[ \tau_{OP_2}(\Rightarrow) = \text{NEG}; \text{AND}; \text{NEG} \]
\[ \tau_{OP_2}(\leq) = \text{LEQ} \]

(NB \( e_1 \Rightarrow e_2 = \neg e_1 \lor e_2 = \neg(e_1 \wedge \neg e_2) \))

\[ \tau_{EXPR}(op \ e)\alpha = \tau_{EXPR}(\text{un}(op, e))\alpha \]
\[ = \tau_{EXPR}(e)\alpha \]
\[ \tau_{OP_1}(op) \]
\[ e \in \text{Term}_{EXPR} \]
\[ op \in \text{Term}_{OP_1} \]

\[ \tau_{OP_1}(-) = \text{NEG} \]
\[ \tau_{OP_1}(\text{not}) = \text{NEG} \]
\[
\tau_{\text{EXPR}}(v)\alpha = \tau_{\text{EXPR}}(\text{vare}(v))\alpha \\
= \text{LDV}(\alpha(v)) \quad v \in \text{Term}_{\text{VAR}}
\]
(no array types and variables!)

\[
\tau_{\text{EXPR}}(z)\alpha = \tau_{\text{EXPR}}(\text{intcon}(z))\alpha \\
= \text{LDC}(\text{iv}(z)) \quad z \in \mathbb{Z}
\]

\[
\tau_{\text{EXPR}}(b)\alpha = \tau_{\text{EXPR}}(\text{boolcon}(b))\alpha \\
= \text{LDC}(\text{bv}(b)) \quad b \in \{\text{false, true}\}
\]

in general

\[
\tau_{\text{EXPR}}(c)\alpha = \text{LDC}(c')
\]

where \(c' \in \text{Value}\) is the machine representation of constant \(c\).
example: translation of $x - 3 \leq 2 \ast y$

(assume $\alpha(x) = 15$ and $\alpha(y) = 3$)

$$\tau_{EXPR}(x - 3 \leq 2 \ast y)\alpha$$

$$= \tau_{EXPR}(x - 3)\alpha$$
$$\tau_{EXPR}(2 \ast y)\alpha$$

LEQ

$$= \tau_{EXPR}(x)\alpha$$
$$\tau_{EXPR}(3)\alpha$$

SUB
$$\tau_{EXPR}(2)\alpha$$
$$\tau_{EXPR}(y)\alpha$$

MUL
LEQ

= LDV(15)
LDC(iv(3))

SUB
LDC(iv(2))
LDV(3)
MUL
LEQ
translation functions
\[ \tau_{\text{STATS}} : \ Term_{\text{STATS}} \times \ Adrf \to TL \]
\[ \tau_{\text{STAT}} : \ Term_{\text{STAT}} \times \ Adrf \to TL \]

\( ss \in Term_{\text{STATS}} \)

\( n_0, n_1, \ldots, n_k \) the variables occurring in \( ss \)

\( \alpha \in Adrf, \{n_0, n_1, \cdots, n_k\} \subseteq \text{dom}(\alpha) \)

\( TL\)-program \( \tau_{\text{STATS}}(ss)\alpha \) has the same effect on \( inp, out, \) and data memory cells \( \alpha(n_0), \alpha(n_1), \ldots, \) and \( \alpha(n_k) \) as statement list \( ss \) has on \( inp, out, \) and variables \( n_0, n_1, \ldots, \) and \( n_k \)

\( (\tau_{\text{STATS}}(ss)\alpha \) has no net effect on the expression stack)
- $STAT \rightarrow \text{skip}$

$$\tau_{STAT}(\text{skip})\alpha = \varepsilon$$

- $STAT \rightarrow \text{abort}$

$$\tau_{STAT}(\text{abort})\alpha = \text{HLT}$$

- $STAT \rightarrow \text{VAR := EXPR}$

$$\tau_{STAT}(v := e)\alpha = \tau_{EXPR}(e)\alpha \quad \tau_{STO}(\alpha(v))$$
concurrent assignment \( \text{STAT} \rightarrow \text{VARS} := \text{EXPRS} \)

\[ \tau_{\text{STAT}}(v_1, v_2, \cdots, v_n := e_1, e_2, \cdots, e_n)\alpha = \]

\[ \begin{align*}
\tau_{\text{EXPR}}(e_1)\alpha \\
\tau_{\text{EXPR}}(e_2)\alpha \\
\vdots \\
\tau_{\text{EXPR}}(e_n)\alpha
\end{align*} \]

\( \text{STO}(\alpha(v_n)) \)
\( \vdots \)
\( \text{STO}(\alpha(v_2)) \)
\( \text{STO}(\alpha(v_1)) \)

no effect on data memory

reverse order
- \( STAT \rightarrow \textbf{if} \ EXPR \ \textbf{then} \ STATS \ \textbf{fi} \)

\[
\tau_{STAT}(\textbf{if} \ e \ \textbf{then} \ ss \ \textbf{fi})\alpha = \tau_{EXPR}(e)\alpha
\]

- \( FJP(l) \)
- \( \tau_{STATS}(ss)\alpha \)

\( l : \)

where \( l \) is a fresh label (this assumes a mechanism for obtaining fresh labels in a code generator)

- \( STAT \rightarrow \textbf{if} \ EXPR \ \textbf{then} \ STATS \ \textbf{else} \ STATS \ \textbf{fi} \)

\[
\tau_{STAT}(\textbf{if} \ e \ \textbf{then} \ tss \ \textbf{else} \ ess \ \textbf{fi})\alpha = \\
\tau_{EXPR}(e)\alpha
\]

- \( FJP(l_{\text{else}}) \)
- \( \tau_{STATS}(tss)\alpha \)
- \( UJP(l_{\text{end}}) \)

\( l_{\text{else}} : \tau_{STATS}(ess)\alpha \)

\( l_{\text{end}} : \)

where \( l_{\text{else}} \) and \( l_{\text{end}} \) are fresh labels
\[ \text{\textbf{STAT} } \rightarrow \textbf{while } \text{EXPR } \textbf{do } \text{STATS} \textbf{ od} \]

\[ \tau_{\text{STAT}}(\textbf{while } e \textbf{ do } ss \textbf{ od})_\alpha = \]

\[ l_{\text{begin}} : \quad \tau_{\text{EXPR}}(e)_\alpha \]
\[ \quad \text{FJP}(l_{\text{end}}) \]
\[ \quad \tau_{\text{STATS}}(ss)_\alpha \]
\[ \quad \text{UJP}(l_{\text{begin}}) \]

\[ l_{\text{end}} : \]

or

\[ \quad \text{UJP}(l_{\text{guard}}) \]

\[ l_{\text{body}} : \quad \tau_{\text{STATS}}(ss)_\alpha \]

\[ l_{\text{guard}} : \quad \tau_{\text{EXPR}}(e)_\alpha \]
\[ \quad \text{TJP}(l_{\text{body}}) \]

\[ x = \text{number of instructions in } \tau_{\text{EXPR}}(e)_\alpha \text{ and } \tau_{\text{STATS}}(ss)_\alpha \]

execution of one step of the repetition takes

- \( 2 + x \) instructions in the 1st translation
- \( 1 + x \) instructions in the 2nd translation

the 2nd translation exploits the fact that after executing body \( ss \) the guard \( e \) must always be evaluated (fixed transition)
syntax

\[
\begin{align*}
PROG & \rightarrow \left[\left[ DECS \mid STATS \right]\right] \\
DECS & \rightarrow \text{var } ID : TYPE \{, ID : TYPE\} \\
TYPE & \rightarrow \text{int} \mid \text{bool} \mid \text{real} \mid \text{char}
\end{align*}
\]

translation function

\[
\tau_{PROG} : \text{Term}_{PROG} \rightarrow TL
\]

\[
\tau_{PROG}\left(\left[\left[ \text{var } n_0 : t_0, n_1 : t_1, \cdots, n_k : t_k \mid ss \right]\right]\right)
\]

\[
= \tau_{STATS}(ss)\alpha
\]

HLT

where address function \(\alpha\) is given by

\[
\begin{align*}
\text{dom}(\alpha) & = \{n_0, n_1, \cdots, n_k\} \\
\alpha(n_i) & = i \quad (0 \leq i \leq k)
\end{align*}
\]
\[\tau_{\text{PROG}}(\|[\textbf{var} \ n_0 : t_0, n_1 : t_1, \cdots, n_k : t_k \ | \ ss]\|)\]
\[= \tau_{\text{STATS}}(ss)\alpha\]
\[\text{HLT}\]
\[\alpha(n_i) = i \ (0 \leq i \leq k)\]

- \(\alpha\) maps the variables to cells 0,1,…,\(k\) of data memory \(M\)
- each variable is of a base type and each value of a base type can be stored in one cell of \(M\)
- \(k \leq \text{Memsize}\) (data memory sufficiently large)
- other choices are possible, but preferably a mapping to a series of subsequent addresses in \(M\)
- if array types and variables are included a different mapping has to be employed
translation of array variables: addresses

\( v : \text{array}[1 : 10] \text{ of array}[5 : 7] \text{ of } int \)

in total \( v \) has \( 10 \times 3 \) elements of type \( int \)

\( v[7] \) has 3 elements of type \( int \):
  \( v[7][5], \, v[7][6], \, v[7][7] \)

mapping on \( adjacent \) memory cells

\[
\begin{array}{ccccccccc}
\hline
\hline
\hline
\end{array}
\]

address \( \alpha(v) \)

\[
\alpha(v) + (1 - 1) \times 3 \quad \alpha(v) + (2 - 1) \times 3 \quad \alpha(v) + (3 - 1) \times 3
\]

address of \( v[i][j] = \alpha(v) + (i - 1) \times 3 + j - 5 \)

address \( v[i] \)
representation size of types/variables

\[ \text{size} : \text{Type} \rightarrow \mathbb{N} \]

\[ \text{size}(t) = \text{number of memory cells needed to represent a value/variable of type } t \]

\[ \text{size}(\text{int}) = \text{size}(\text{real}) = \text{size}(\text{bool}) = \text{size}(\text{char}) = 1 \]

(one memory cell needed per base type value)

\[ \text{size}((\text{array}[l : u] \text{ of } t) = (u - l + 1) \times \text{size}(t) \]
address function $\alpha : Name \rightarrow MemAddress$

$v \in dom(\alpha)$, $v$ has type $t$

- $t$ is a base type: $v$ is stored in cell $\alpha(v)$
- $t = \text{array}[l : u] \text{ of } t_{elt}$:
  - $v$ is stored in cells
    \[ \alpha(v), \alpha(v) + 1, \ldots, \alpha(v) + \text{size}(t) - 1 \]
  - for $l \leq i \leq u$ $v[i]$ is stored in cells
    \[ \alpha(v) + (i - l) \times \text{size}(t_{elt}), \ldots \alpha(v) + (i + 1 - l) \times \text{size}(t_{elt}) - 1 \]
consistent address function $\alpha$

$n_1, n_2 \in \text{dom}(\alpha), n_1 \neq n_2$

$n_1 : t_1, n_2 : t_2$

address sets

\[
\{\alpha(n_1), \alpha(n_1) + 1, \ldots, \alpha(n_1) + \text{size}(t_1) - 1\}
\]

and

\[
\{\alpha(n_2), \alpha(n_2) + 1, \ldots, \alpha(n_2) + \text{size}(t_2) - 1\}
\]

are disjoint
translation of programs with differently sized types/variables 42/89

\[
\tau_{\text{PROG}}(\left[\text{var } n_0 : t_0, n_1 : t_1, \ldots, n_k : t_k \mid s \right]) = \tau_{\text{STATS}}(s)\alpha \\
\text{HLT}
\]

where address function \(\alpha\) is given by

- \(\text{dom}(\alpha) = \{n_0, n_1, \ldots, n_k\}\)
- \(\alpha(n_i) = (\sum_{j \leq j < i} \text{size}(t_j)) \quad (0 \leq i \leq k)\)
  
(address of variable \(n_i\) is the sum of the sizes of the preceding variables \(n_0, n_1, \ldots, n_{i-1}\))
\[ v : \text{array}[1 : 10] \text{ of int} \quad \alpha(v) = 22 \]

\[
\begin{array}{cccccc}
  \uparrow & \uparrow & \uparrow & & \uparrow \\
  22 & & & & 31 \\
\end{array}
\]

“translation” with STO and LDV

\[
\tau_{\text{STAT}}(v[e_i] := e) \alpha = \tau_{\text{EXPR}}(e) \alpha \\
\quad \text{STO}(\alpha(v) + \text{val}(e_i) - 1)
\]

\[
\tau_{\text{EXPR}}(v[e_i]) \alpha = \text{LDV}(\alpha(v) + \text{val}(e_i) - 1)
\]

\textbf{problem:} value of \( e_i \) is not known at compile time

\textbf{solution:}

- runtime computation of addresses on the expression stack
- indirect load and store instructions
  (extension instruction set of TL-interpreter)
**indirect addressing:** indirect load LDI / store STI

**indirect load** LDI

specification

\[ \{ es = \langle iv(a) \rangle \oplus ES \} \text{ LDI } \{ es = \langle M[a] \rangle \oplus ES \} \]

(address on stack represented by integer value)

TL-interpreter fragment

LDI : \( v, es : = \text{Top}(es), \text{Pop}(es) ; \)

\textbf{if} \( v :: iv(a) \rightarrow es : = \langle M[a] \rangle \oplus es \) \textbf{fi}

**indirect store** STI

specification

\[ \{ es = \langle v \rangle \oplus \langle iv(a) \rangle \oplus ES \} \]

STI

\[ \{ es = ES \land M[a] = v \} \]

TL-interpreter fragment

STI : \( v, v_a, es : = \text{Top}(es), \text{Top}(\text{Pop}(es)), \text{Pop}(\text{Pop}(es)) ; \)

\textbf{if} \( v_a :: iv(a) \rightarrow M[a] : = v \) \textbf{fi}
\( \tau_{\text{VAR}}(v) \alpha = \text{TL-program that computes the address of variable } v \text{ and puts it on top of the expression stack} \)

\[ \text{VAR} \langle -d, +t \rangle \rightarrow \text{ID} \langle +n \rangle \{ [\text{EXPR} \langle -d, +te_i \rangle] \}_{i=0}^k \]

\[\begin{align*}
&\text{let } s_i = \text{size}(t_i) \quad (0 \leq i \leq k+1)
\end{align*}\]
\[ \tau_{VAR}(n[e_0][e_1] \cdots [e_k]) \alpha = \]

\[ \text{LDC}(iv(\alpha(n))) \quad \{ es = \langle\text{address n}\rangle \oplus ES \} \]

\[ \tau_{EXPR}(e_0) \alpha \quad \{ es = \langle\text{val}(e_0)\rangle \oplus \langle a \rangle \oplus ES \} \]

\[ \text{LDC}(iv(l_0)) \]

\[ \text{SUB} \quad \{ es = \langle\text{val}(e_0) - l_0\rangle \oplus \langle a \rangle \oplus ES \} \]

\[ \text{LDC}(iv(s_1)) \]

\[ \text{MUL} \quad \{ es = \langle\text{val}(e_0) - l_0\rangle \ast s_1 \rangle \oplus \langle a \rangle \oplus ES \} \]

\[ \text{ADD} \quad \{ es = \langle\text{address n[e_0]}\rangle \oplus ES \} \]

: \[
\tau_{EXPR}(e_k) \alpha \]

\[ \text{LDC}(iv(l_k)) \]

\[ \text{SUB} \]

\[ \text{LDC}(iv(s_{k+1})) \]

\[ \text{MUL} \]

\[ \text{ADD} \quad \{ es = \langle\text{addr. n[e_0] \cdots [e_k]}\rangle \oplus ES \} \]
computing address of variable (alternative description) (1)

\[
VAR\langle -d, +t_v \rangle \rightarrow ID\langle +n \rangle RVAR\langle -d, -t, +t_v \rangle
\]

\[
t : (n, t) \in d
\]

\[
RVAR\langle -d, -t, +t \rangle \rightarrow \varepsilon
\]

\[
RVAR\langle -d, -t, +t' \rangle \rightarrow [EXPR\langle -d, +t_e \rangle] RVAR\langle -d, -t_1, +t' \rangle
\]

\[
t_e = \text{inttype}
\]

\[
t_1 : (\exists l, u : l, u \in \mathbb{Z} : t = \text{array}(t_1, l, u))
\]

translation functions

\[
\tau_{VAR}(n[e_0] \cdots [e_k])\alpha =
\]

\[
\begin{align*}
\ & LDC(iv(\alpha(n))) \quad \{\text{addr. } n \text{ on top of stack}\} \\
\ & \tau_{RVAR}([e_0] \cdots [e_k], t)\alpha \quad \{\text{addr. } n[e_0] \cdots [e_k] \text{ on top of stack}\}
\end{align*}
\]

where \( t \) is the type of variable \( n \) and

\[
\tau_{RVAR} : Term_{RVAR} \times Type \times Adrf \rightarrow TL
\]

specification

\[
\begin{align*}
\ & \{\text{start addr. variable } n \text{ of type } t \text{ on top of stack}\} \\
\ & \tau_{RVAR}([e_0] \cdots [e_k], t)\alpha \\
\ & \{\text{start addr. of } n[e_0] \cdots [e_k] \text{ on top of stack}\}
\end{align*}
\]
recursive definition

\[ \tau_{RVAR}(\varepsilon, t)\alpha = \varepsilon \]

\[ \tau_{RVAR}([e_0] \cdots [e_k], t_0)\alpha = \]

\[
\begin{align*}
\tau_{EXPR}(e_0)\alpha & \quad \{ es = \ll a \rr \oplus ES \} \\
LDC(iv(l_0)) & \quad \{ es = \ll \text{val}(e_0) \rr \oplus \ll a \rr \oplus ES \} \\
\text{SUB} & \quad \{ es = \ll \text{val}(e_0) - l_0 \rr \oplus \ll a \rr \oplus ES \} \\
LDC(iv(s_1)) & \quad \{ es = \ll \text{val}(e_0) - l_0 \rr \ast s_1 \rr \oplus \ll a \rr \oplus ES \} \\
\text{MUL} & \quad \{ es = \ll a + (\text{val}(e_0) - l_0) \ast s_1 \rr \oplus \ll a \rr \oplus ES \} \\
\text{ADD} & \quad \{ es = \ll a + (\text{val}(e_0) - l_0) \ast s_1 \rr \oplus \ll a \rr \oplus ES \}
\end{align*}
\]

\[ \tau_{RVAR}([e_1] \cdots [e_k], t_1)\alpha \]

where \( t_0 = \text{array}[l_0 : u_0] \text{ of } t_1 \) and \( s_1 = \text{size}(t_1) \)
translations using indirect addressing (1)

\[ EXPR \rightarrow_{vare} VAR \]

\[ \tau_{EXPR}(v)\alpha = \]

\[ \tau_{VAR}(v)\alpha \quad \text{(address of } v \text{ on top of stack)} \]
\[ \text{LDI} \quad \text{(value of } v \text{ on top of stack)} \]

(provided \( v \) is of a base type)

\[ STAT \rightarrow_{assign} VAR := EXPR \]

\[ \tau_{STAT}(v := e)\alpha = \]

\[ \tau_{VAR}(v)\alpha \quad \text{(address of } v \text{ on top of stack)} \]
\[ \tau_{EXPR}(e)\alpha \quad \text{(value of } e \text{ on top of stack)} \]
\[ \text{STI} \quad (M[\text{address } v] = \text{value of } e) \]

(provided \( v \) is of a base type)
- \( STAT \rightarrow_{\text{read}} \text{read}(\text{VAR } \{, \text{ VAR}\}) \)

\[
\tau_{\text{STAT}}(\text{read}(v_0, v_1, \ldots, v_k))\alpha = \\
\tau_{\text{VAR}}(v_0)\alpha \\
\text{GET} \\
\text{STI} \\
\vdots \\
\tau_{\text{VAR}}(v_k)\alpha \\
\text{GET} \\
\text{STI}
\]

(provided \( v_0, v_1, \ldots, \) and \( v_k \) are of base types)
\[
\begin{align*}
\begin{align*}
\text{var } & \ldots v_1 : \text{int}, v_2 : \text{int} \ldots | \ldots v_1 := v_2 \ldots \]
\tau_{\text{STAT}}(v_1 := v_2)\alpha & = \text{LDC}(iv(\alpha(v_1))) \\
& \quad \text{LDC}(iv(\alpha(v_2))) \\
& \quad \text{LDI} \\
& \quad \text{STI}
\end{align*}
\end{align*}
\]
solution: generate direct addressing code for variable references of which the address is known at compile time

\[
\begin{align*}
\tau_{STAT}(v := a[e])\alpha &= \tau_{EXPR}(e)\alpha \\
\text{LDC}(iv(\alpha(a))) &\quad \text{LDC}(iv(\alpha(a))) \\
\tau_{EXPR}(e)\alpha &\quad \tau_{EXPR}(e)\alpha \\
\text{LDC}(iv(1)) &\quad \text{LDC}(iv(1)) \\
\text{SUB} &\quad \text{SUB} \\
\text{LDC}(iv(1)) &\quad \text{LDC}(iv(1)) \\
\text{MUL} &\quad \text{MUL} \\
\text{ADD} &\quad \text{ADD} \\
\text{LDI} &\quad \text{LDV}(\alpha(v)) \\
\text{STO}(\alpha(v)) &\quad \text{STI}
\end{align*}
\]
\{es = \langle \text{iv}(a) \rangle \oplus ES\}\}

\text{CHK}(\text{iv}(l), \text{iv}(u)) \{es = \langle \text{bv}(l \leq a \leq u) \rangle \oplus \langle \text{iv}(a) \rangle \oplus ES\}\}

\tau_{VAR}(n[e])\alpha = \begin{align*}
\tau_{LDC}(\text{iv}(\alpha(n))) \\
\tau_{EXPR}(e)\alpha \\
\text{CHK}(\text{iv}(l), \text{iv}(u)) \\
\text{FJP}(l_{oob}) \\
\text{LDC}(\text{iv}(l)); \text{SUB} \\
\text{LDC}(\text{iv}(s)); \text{MUL} \\
\text{ADD}
\end{align*}

\begin{aligned}
\text{(range check)}
\end{aligned}

where \( n \) is of type \( t = \text{array}[l : u] \text{ of } t_{elt} \), \( s = \text{size}(t_{elt}) \) and

\begin{align*}
\tau_{PROG}(\cdots)\alpha = \begin{align*}
\text{HLT} \\
l_{oob} : \begin{aligned}
\text{generate index out of bounds error message}
\end{aligned}
\end{align*}
\end{align*}

\begin{align*}
\text{HLT}
\end{align*}
numcase statement

```
STAT →_{numcase} numcase EXPR in 
    STATS
    { [ ] STATS }
out STATS
esac
```

statement

```
numcase e in ss₁ || ⋯ || ssₙ out ss_{out} esac
```

context condition

- case expression e has type integer

semantics

- if the value of e equals j with j ∈ {1, …, n} then execute statements ssₗ otherwise execute statements ss_{out}
\[ \tau_{STAT}(\text{numcase } e \text{ in } ss_1 \mid \cdots \mid ss_n \text{ out } ss_{out} \text{ esac})\alpha = \]
\[ \tau_{EXPR}(e)\alpha \]
\[ \text{CHK}(iv(1), iv(n)) \]
\[ \text{FJP}(l_{out}) \]
\[ \text{XJP} \{ \text{jump val}(e) \text{ instr. forward} \} \]
\[ \text{UJP}(l_1) \]
\[ \vdots \]
\[ \text{UJP}(l_n) \]
\[ l_1 : \quad \tau_{STAT}(ss_1)\alpha \]
\[ \quad \text{UJP}(l_{end}) \]
\[ \vdots \]
\[ l_n : \quad \tau_{STAT}(ss_n)\alpha \]
\[ \quad \text{UJP}(l_{end}) \]
\[ l_{out} : \quad \text{POP} \{ \text{remove val}(e) \text{ from stack} \} \]
\[ \quad \tau_{STAT}(ss_{out})\alpha \]
\[ l_{end} : \]

where \( l_1, \ldots, l_n, l_{out}, \) and \( l_{end} \) are fresh labels
code generation: translation functions on AST

SDTS (syntax directed translation scheme)

**signature** \((S, O)\)

\[ \tau_s : \text{Term}_s \rightarrow Z_s \quad (s \in S) \]

**operator** \(o : s_1 \times s_2 \times \cdots \times s_n \rightarrow s\)

\[ \sigma_o : Z_{s_1} \times Z_{s_2} \times \cdots \times Z_{s_n} \rightarrow Z_s \]

**terms** \(t_1 : s_1, t_2 : s_2, \ldots, t_n : s_n\)

\[ \tau_s(o(t_1, t_2, \ldots, t_n)) = \sigma_o(\tau_{s_1}(t_1), \tau_{s_2}(t_2), \ldots, \tau_{s_n}(t_n)) \]

- order in which the subtrees (subterms) are translated is not fixed, but there may be dependencies
- translation of a subtree may be used more than once
\[ N_s \rightarrow_o N_{s_1} N_{s_2} \ldots N_{s_n} \]

extra synthesized attribute nonterminals: **code attribute**
\[
N_s \langle -i_0, +sy_0, +z_0 \rangle \rightarrow_o N_{s_1} \langle -i_1, +sy_1, +z_1 \rangle \ldots N_{s_n} \langle -i_n, +sy_n, +z_n \rangle
\]
\[
i_{\pi(1)} : C_1(i_0, i_{\pi(1)})
\]
\[
i_{\pi(2)} : C_2(i_0, sy_{\pi(1)}, z_{\pi(1)}, i_{\pi(2)})
\]
\[
i_{\pi(3)} : C_3(i_0, sy_{\pi(1)}, z_{\pi(1)}, sy_{\pi(2)}, z_{\pi(2)}, i_{\pi(3)})
\]
\[\vdots\]
\[
i_{\pi(n)} : C_n(i_0, sy_{\pi(1)}, z_{\pi(1)}, \ldots, sy_{\pi(n-1)}, z_{\pi(n-1)}, i_{\pi(n)})
\]
\[
sy_0 : C_0(i_0, sy_1, z_1, \ldots, sy_n, z_n, sy_0)
\]
\[
z_0 : z_0 = \sigma'_o(z_1, z_2, \ldots, z_n, i_0, sy_1, sy_2, \ldots, sy_n)
\]

\(\pi\) is a permutation of \(\{1, 2, \ldots, n\}\); **general one-pass condition**
\[
\text{PROG} \langle +z_0 \rangle \rightarrow |\text{[DECS} \langle +d, +z_1 \rangle|\text{STATS} \langle -d, -z_1, +z_2 \rangle]|\]
\[
z_0 : z_0 = z_2; \text{HLT}
\]

(attribute \(z_1\) of \text{DECS}: address function \(\alpha\))
storing code attributes in abstract syntax tree nodes

uniform target language

<table>
<thead>
<tr>
<th>TNode</th>
</tr>
</thead>
<tbody>
<tr>
<td>:</td>
</tr>
<tr>
<td>FCode :</td>
</tr>
<tr>
<td>TCode</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>TNode</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ts</td>
</tr>
<tr>
<td>:</td>
</tr>
<tr>
<td>FCode :</td>
</tr>
<tr>
<td>TsCode</td>
</tr>
</tbody>
</table>

- conforms precisely to code attribute scheme
- code generator computes code attributes on traversing the abstract syntax trees
- duplication of information
(uniform target language $TL$)

accumulation of code during traversal

\[
\begin{align*}
z & \quad z; \tau_{EXPR}(e_1); \tau_{EXPR}(e_2); ADD \\
+ & \quad z; \tau_{EXPR}(e_1); \tau_{EXPR}(e_2) \\
e_1 & \quad z; \tau_{EXPR}(e_1) \\
e_2 & \quad z
\end{align*}
\]

no storage of code in node attributes but code generator has a field containing the accumulated code generated for the part of the abstract syntax tree traversed thus far

\[Traverse(e_1); Traverse(e_2); addcode(ADD)\]

if the translation of a subtree appears twice in the translation of a node then that subtree must be traversed twice (at the appropriate times)
\( T\text{CodeGenerator} = \text{class} \)

\begin{verbatim}
private . . .
    \textit{FCode} : String; \{ accumulated code for traversed part \}
procedure addlab(\textit{l} : String); \{ add label \textit{l} to \textit{FCode} \}
procedure addcode(\textit{c} : String); \{ add code line \textit{c} to \textit{FCode} \}
    \{ using addlab and addcode ensures \}
    \{ a readable layout of the generated code \}
\textit{ffl} : integer; \{ first free label \}
    \{ invariant \textit{llab} : set of reserved labels = \{100, 101, \ldots, ffl - 1\} \}
function freshlabel : String;
    \{ pre : ffl = L \land \textit{llab} \}
        \textit{ret} : $L$
    \{ post : ffl = L + 1 \land \textit{llab} \}
function Size(\textit{t} : TType) : integer;
public . . .
procedure Traverse(\textit{t} : TNode);
property Code : String read \textit{FCode};
end; \{ T\text{CodeGenerator} \}
\end{verbatim}
procedure TCodeGen. Traverse(t : TNode);
...

case t.NodeClass of
  ...
  ncTDecs : with t as TDecs do begin
    i := 0; adr := 0;
    { inv. : adr = (∑j : 0 ≤ j < i : Size(Decs[j].Typ))
      set of allocated addresses = {0, 1, ..., adr − 1} }
  while i <> DecsCount do begin
    dec := Decs[i];
    { store address adr in declaration node }
    dec.storeinfo(‘address’, TInteger.Create(adr));
    { addresses adr, adr + 1, ..., adr + Size(dec.Typ) − 1 are allocated }
    adr := adr + Size(dec.Typ);
    i := i + 1
  end{ while }
end; { ncTDecs }
\[ STAT \rightarrow_{if1} if \ EXPR \ then \ STATS \ fi \]

\texttt{ncTlf1Stat} :

\begin{verbatim}
with t as Tlf1Stat do
begin
    Traverse(Expr);
    { code for evaluation of guard generated }
    l1 := freshlabel;
    addcode('FJP ' + l1);
    Traverse(Stats);
    { code for then-part generated }
    addlab(l1)
end { ncTlf1Stat }
\end{verbatim}
\[
STAT \rightarrow_{assign} VAR := EXPR
\]

\(ncTAssignStat\) :

\[
\text{with } t \text{ as } TAssignStat \text{ do begin}
\]

\[
\text{Traverse(Variable); }
\]

\{ if possible, address computed and stored

\hspace{1cm} \text{else address computation code generated } \}

\[
\text{Traverse(Expr); } \{ \text{code for evaluation of } Expr \text{ generated} \}
\]

\[
\text{if } Variable.getinfo(’address’) <> \text{ nil}
\]

\{ direct addressing \} \text{ then begin}

\[
vadr := TInteger(Variable.getinfo(’address’)).Value;
\]

\[
\text{addcode(’STO ’ + InttoStr(vadr))}
\]

\text{end}

\{ indirect addressing \} \text{ else begin}

\[
\text{addcode(’STI’)}
\]

\text{end; } \{ ncTAssignStat \}
\[
EXPR \rightarrow_{\text{var}} \text{VAR}
\]

\[\text{ncTVarExpr :} \]

\[\text{with } t \text{ as } T\text{VarExpr do} \]

\[\begin{align*}
\text{begin} \\
\quad \text{Traverse(Variable);} \\
\quad \{ \text{if possible, address computed and stored} \} \\
\quad \text{else address computation code generated } \}
\end{align*}\]

\[\begin{align*}
\text{if } \text{Variable.getinfo(’address’) }<> \text{ nil} & \\
\text{then } \{ \text{direct addressing} \} & \\
\text{begin} & \\
\quad vadr : = T\text{Integer(Variable.getinfo(’address’)).Value;} & \\
\quad \text{addcode(’LDV ’ } + \text{InttoStr(vadr)}) & \\
\end{align*}\]

\[\begin{align*}
\text{end} & \\
\text{else } \{ \text{indirect addressing} \} & \\
\text{addcode(’LDI’) & \\
\text{end; } \{ \text{ncTVarExpr } \}
\end{align*}\]
VAR \rightarrow ID \{[EXPR]\}

\tau_{VAR}(n[e_0] \cdots [e_k], t)\alpha

\text{ncTVar :}
\begin{verbatim}
with t as TVar do
begin
  dec : = TDec(getinfo('declaration'));
  vadr : = TInteger(dec.getinfo('address')).Value;
  vtype : = dec.Type;
  case vtype.NodeClass of
    ncTIntType, ncTRealType,
    ncTCharType, ncTBoolType : \{ store address \}
      storeinfo('address', TInteger.Create(vadr));
    ncTArrayType : \{ generate addr. comp. code \}
      ...
  end \{ vtype.NodeClass \}
end; \{ ncTVar \}
\end{verbatim}
generating address computation code (arrays)

addcode('LDC iv ' + InttoStr(vadr));
for i := 0 to ExprsCount – 1 do
{ vtype is type of t.Name[e0]...[ei-1] }
begin
  Traverse(Exprs[i]);
  with vtype as TArrayType do
  begin
    addcode('LDC iv ' + InttoStr(lb));
    addcode('SUB');
    addcode('LDC iv ' + InttoStr(Size(ElementType)));
    addcode('MUL');
    addcode('ADD');
    vtype := ElementType
  end { with vtype }
end { for i }
extra field in code generator:

\[ ffa : \text{integer} \]

invariants

\[ ffa = \text{sum of the sizes of all variables that are allocated addresses} \]

set of allocated addresses = \{0, 1, \ldots, ffa - 1\}

temporary need of \( k \) additional memory cells

\[
\begin{align*}
\{ & ffa = FFA \\ ffa : &= ffa + k; \} \\
\text{“generate code using additional memory addresses } FFA, FFA + 1, \ldots, FFA + k - 1\”; \\
ffa : &= ffa - k
\end{align*}
\]
- machine independent
  (abstract syntax tree transformations, intermediate code transformations)
- machine dependent
  (machine code transformations, e.g. peephole optimization

\[
\begin{align*}
\text{LDC}(iv(1)) & \quad \text{can be omitted} \\
\text{MUL} & \\
\text{LDC}(iv(0)) & \quad \text{can be omitted} \\
\text{ADD} & \\
\end{align*}
\]

(these sequences may be a result of code generation for arrays, other optimizations, ... )
transformations increasing efficiency (time/space)

- extraction of loop-invariant computations from loops

```latex
\textbf{while } e \textbf{ do }
\begin{align*}
S_0; \\
i : = 5; \\
S_1 \\
\textbf{od}
\end{align*}
```
e and $S_0$ do no inspect $i$
$S_0$ and $S_1$ contain no assignments to $i$

transformed into

```latex
i := 5; \\
\textbf{while } e \textbf{ do }
\begin{align*}
S_0; \\
S_1 \\
\textbf{od}
\end{align*}
```
elimination of redundant computations, e.g. common subexpressions

\[ i := (k + l) \times (k + l) \]

LDV(\(\alpha(k)\))
LDV(\(\alpha(l)\))
ADD
DUP \{storing on stack\}
MUL
STO(\(\alpha(i)\))

\[ i := z \times (k + l); \ j := w \times (k + l) \]

LDV(\(\alpha(z)\))
LDV(\(\alpha(k)\))
LDV(\(\alpha(l)\))
ADD
DUP
MUL
STO(\(\alpha(i)\))
LDV(\(\alpha(w)\))
LDV(\(a_{temp}\))
MUL
STO(\(\alpha(j)\))
“dead” code elimination
e.g. unreachable program fragments

```java
while true do S_loop od; S_rest
```

statement list $S_{rest}$ is unreachable (provided there are no goto-construct and break statement)

```java
if false then S_t else S_e fi
```

statement list $S_t$ is unreachable

(constant guards may arise as a result of other optimizations, e.g. constant propagation)
constant folding

\[
\begin{align*}
& + \quad 3 \quad 4 \quad \rightarrow \quad 7 \\
& \Rightarrow \quad \text{false} \quad b \quad \rightarrow \quad \text{true} \\
& \Rightarrow \quad b \quad \text{true} \quad \rightarrow \quad \text{true} \\
& * \quad 0 \quad x \quad \rightarrow \quad 0 \\
& * \quad 3 \quad 4 \quad \rightarrow \quad 12 \\
& + \quad 0 \quad x \quad \rightarrow \quad x \\
& \Rightarrow \quad \text{true} \quad b \quad \rightarrow \quad b \\
& \Rightarrow \quad b \quad \text{false} \quad \rightarrow \quad \neg \ b \\
& * \quad 1 \quad x \quad \rightarrow \quad x
\end{align*}
\]
constant propagation

\[
a := 4; \{a = 4\} \quad x := 7 \ast a; \quad y := x + 4; \quad x := k + 5
\]

\[
a := 4; \quad x := 28; \{a = 4 \land x = 28\} \quad y := x + 4; \quad x := k + 5
\]

\[
a := 4; \quad x := 28; \quad y := 32; \{a = 4 \land x = 28 \land y = 32\} \quad x := k + 5
\]

\[
a := 4; \quad x := 28; \quad y := 32; \quad x := k + 5\{a = 4 \land y = 32\}
\]
\[
PROG \quad \rightarrow \quad BLOCK \\
BLOCK \quad \rightarrow \quad [ [ \text{DECS} \mid \text{STATS} ] ] \\
STAT \quad \rightarrow \quad BLOCK \\
\]

\[
[ \text{var } x : \text{int}, y : \text{int}, z : \text{bool} \\
| \quad ss_1; \\
| \quad [ \quad \text{var } z : \text{int}, w : \text{char} \mid ss_2 \quad ]]; \\
\quad ss_3 \\
\] \\

\[
ss_1, ss_3 : \quad [(x, \text{int}), (y, \text{int}), (z, \text{bool})] \\
ss_2 : \quad [(x, \text{int}), (y, \text{int}), (z, \text{int}), (w, \text{char})]
\]
\[ \tau_{\text{BLOCK}} : \text{Term}_{\text{BLOCK}} \times \text{Adrf} \rightarrow TL \]

\[ \tau_{\text{BLOCK}}(\| \text{var} \ n_0 : t_0, \ldots, n_k : t_k \mid ss \|)\alpha = \tau_{\text{STATS}}(ss)\alpha' \]

\[ \alpha' = [m/n_0, (m + 1)/n_1, \ldots, (m + k)/n_k]\alpha \]

\[ m = (\text{MAX} \ v : v \in \text{dom}(\alpha) : \alpha(v)) + 1 \]

\[ \alpha'(n) = \begin{cases} \alpha(n) & \text{if } n \in \text{dom}(\alpha) \setminus \{n_0, \ldots, n_k\} \\ m + i & \text{if } n = n_i \text{ for some } i : 0 \leq i \leq k \end{cases} \]

\[ \tau_{\text{STAT}}(b)\alpha = \tau_{\text{BLOCK}}(b)\alpha \]

\[ \tau_{\text{PROG}}(b) = \tau_{\text{BLOCK}}(b)\emptyset \]

HLT

disadvantage: not usable if recursive procedures with parameters and/or local variables are allowed
change of addressing: (block height, position within block)

\[
x : (0, 0) \\
y : (0, 1) \\
z : (0, 2) \quad \text{outer block} \\
z : (1, 0) \quad \text{inner block} \\
w : (1, 1)
\]

\[Adrf' = Name \rightarrow \IN \times \IN\]

\[\alpha(x) = (b, d)\]

where \( b = \) block height
\( d = \) displacement within block

location \( x \) in memory: \( M[display[b] + d]\)

extensions TL interpreter \( \text{display : array}[0..B] \text{of Memaddress} \)
\( ffa : \text{Memaddress} \) (first free address)

memory addresses 0, \ldots, \( ffa - 1 \) are in use

used memory behaves like a stack
\{\text{block height } b - 1\} \\
|| [ \text{var } n_0 : t_0, \ldots, n_k : t_k \mid ss ] || \\
\vdots \\
(b, 0) \ldots (b, k) \\

local block variables are assigned the unused memory addresses 
ffa, \ldots, ffa + k \\

block entry: \textit{display}[b] := ffa; \\
ffa := ffa + k + 1 \\

block exit: ffa := ffa - k - 1 \\
or \\
ffa := \textit{display}[b]
extra TL instructions

\[ \text{ENTER}'(b, d) : \ display[b], ffa := ffa, ffa + d \]

\[ \text{EXIT}'(b) : \ ffa := display[b] \]

\[ \text{LDV}'(b, d) : \ es := \langle\langle M[display[b] + d]\rangle\rangle \oplus es \]

\[ \text{STO}'(b, d) : \ M[display[b] + d], es := Top(es), Pop(es) \]

add current block height as extra parameter to all translation functions e.g.

\[ \tau_{STAT} : Term_{STAT} \times Adrf' \times IN \rightarrow TL \]

\[ \tau_{STAT}(\text{repeat } ss \text{ until } e)\alpha, \gamma = \]

\[ l : \ \tau_{STATS}(ss)\alpha, \gamma \]

\[ \tau_{EXPR}(e)\alpha, \gamma \]

\[ \text{FJP}(l) \]
\[ \tau_{\text{BLOCK}}(\langle \textbf{var} \ n_0 : t_0, \ldots, n_k : t_k | ss \rangle) \alpha, \gamma = \]
\[ \text{ENTER}'(\gamma', k + 1) \]
\[ \tau_{\text{STATS}}(ss) \alpha', \gamma' \]
\[ \text{EXIT}'(\gamma') \]

where
\[ \gamma' = \gamma + 1 \]
\[ \alpha' = \left[ (\gamma', 0)/n_0, \ldots, (\gamma', k)/n_k \right] \alpha \]

translation of a program

\[ \tau_{\text{PROG}}(b) = \tau_{\text{BLOCK}}(b) \emptyset, -1 \]
\[ \text{HLT} \]
procedures without parameters

\[\text{DECS} \rightarrow \text{var DEC}\{, \text{DEC}\}\{; \text{PDEC}\}\]

\[\text{PDEC} \rightarrow \text{proc ID} = (\text{STATS}) \quad \text{(no parameters)}\]

\[\text{STAT} \rightarrow \text{ID} \quad \text{(procedure call)}\]

\[
\begin{array}{l}
\text{[var } n_0 : t_0, \ldots, n_k : t_k; \\
\text{proc } p_0 = (ss_0); \ldots; \text{proc } p_m = (ss_m) | ss ]
\end{array}
\]

translation body of declared procedure preceded by a label
translation of procedure call contains a.o. a jump to this label

\[P_{\text{adr}} = \text{Name} \rightarrow \text{Plabel}\]

\[
\begin{array}{l}
\tau_{\text{STATS}} : \text{Term}_{\text{STATS}} \times \text{Adrf}' \times P_{\text{adr}} \times \text{IN} \rightarrow \text{TL} \\
\tau_{\text{BLOCK}} : \text{Term}_{\text{BLOCK}} \times \text{Adrf}' \times P_{\text{adr}} \times \text{IN} \rightarrow \text{TL} \\
\tau_{\text{STATS}}(ss)\alpha, \beta, \gamma
\end{array}
\]
extension TL interpreter

return stack $rs : \text{Progaddress}^\ast$

stack of return addresses of not yet completed procedures calls

extra TL instructions

CALL($l$) : $ip, rs := Lmap(l), \langle ip \rangle \oplus rs$

RET : $ip, rs := \text{Top}(rs), \text{Pop}(rs)$

($l \in \text{Plabel}$, $ip$ in right hand side assignment is the address of the instruction after the procedure call)
procedures without parameters (cross recursion allowed)

\[ \tau_{\text{BLOCK}} \left( \left[ \begin{array}{l}
\text{var} \ n_0 : t_0, \ldots, n_k : t_k; \\
\text{proc} \ p_0 = (ss_0); \ldots; \text{proc} \ p_m = (ss_m) \mid ss \end{array} \right] \right) \alpha, \beta, \gamma \]

= 

\left\{ \begin{array}{l}
\text{UJP}(l_1) \\
\tau_{\text{STATS}}(ss_0) \alpha', \beta', \gamma' \\
\text{RET} \\
\vdots \\
\tau_{\text{STATS}}(ss_m) \alpha', \beta', \gamma' \\
\text{RET} \\
\text{ENTER}'(\gamma', k + 1) \\
\tau_{\text{STATS}}(ss) \alpha', \beta', \gamma' \\
\text{EXIT}'(\gamma') \\
\end{array} \right\} 

\text{translation } p_0

\text{translation } p_m

\alpha' = \left[ (\gamma', 0)/n_0, \ldots, (\gamma', k)/n_k \right] \alpha

\beta' = \left[ lb_0/p_0, \ldots, lb_m/p_m \right] \beta

\gamma' = \gamma + 1

l_1, lb_0, \ldots, lb_m \text{ fresh labels}
translation procedure call

\[ \tau_{STAT}(p)\alpha, \beta, \gamma = \text{CALL}(\beta(p)) \]

translation program

\[ \tau_{PROG}(p) = \tau_{BLOCK}(p)\emptyset, \emptyset, -1 \]

HLT

more general translation of program block

\[ \tau_{BLOCK}(p)\alpha_{sys}, \beta_{sys}, 0 \]

where \( \alpha_{sys} \) and \( \beta_{sys} \) contain information on system variables and procedures
the invariant

*a block with (static) block height* $b$ *can only be entered from a position within a block with (static) block height* $b - 1$

holds in case only (inner) blocks are allowed, but does not hold in case procedures are introduced

```
[[ \texttt{var} \ q_1 : \texttt{int}; \\
    \texttt{proc} \ p = ([[ 3 \texttt{var} \ r_1 : \texttt{bool}, r_2 : \texttt{int} \ | \ ss_0 ]]) \\
    \ldots s_1 \ldots; 1 \ p \ldots \\
    [[ \texttt{var} \ q_2 : \texttt{bool} \ | \ \ldots s_2 \ldots; 2 \ p \ldots ]]]
```

the block within the procedure body and the block declaring $q_2$ both have static block height 1, so variables within the one block are not available within the other

- call $p$ at 1: block of $p$ entered from block with height 0
- call $p$ at 2: block of $p$ entered from block with height 1
- recursive call of $p$ in $ss_0$: entering from block with height 1
- calls of $p$ from blocks in $ss_0$ or $s_2$: entering from block with height $> 1$
- on entry of block 3 in all but the first case 1 or more blocks with static height 1 are in use (only the last entered is visible at the moment)
extension TL interpreter (storing display values)
display stack $ds : \text{Memaddress}^*$
changed TL instructions

ENTER($b, d$) : $ds, display[b], ffa :=$
\langle\langle display[b]\rangle\rangle \oplus ds, ffa, ffa + d$

EXIT($b$) : $ds, display[b], ffa :=$
$\text{Pop}(ds), \text{Top}(ds), display[b]$

(should be used in translations instead of ENTER′ and EXIT′)
translation of procedures (7) 86/89

procedures with parameters

\[
\begin{align*}
PDEC & \rightarrow \ proc \ ID = (PPARS \mid STATS) \\
PPARS & \rightarrow \ PPAR\{,PPAR\} \\
PPAR & \rightarrow \ ?ID : TYPE \\
PPAR & \rightarrow \ !ID : TYPE \\
STAT & \rightarrow \ ID(PARGS) \\
PARGS & \rightarrow \ PARG\{,PARG\} \\
PARG & \rightarrow \ ?EXPR \\
PARG & \rightarrow \ !VAR
\end{align*}
\]

before execution procedure body memory allocation for (formal) parameters (new block with parameters as local variables)

before execution procedure body values of actual input parameters (expressions) are assigned to formal input parameters (using expression stack)

after execution procedure body values of formal output parameters are assigned to actual output parameters (variable references) (using expression stack)
\[
\tau_{\text{STAT}}(p_0(\ldots, ?e_p, !v_0, \ldots, !v_q))\alpha, \beta, \gamma = \\
\tau_{\text{EXPR}}(e_0)\alpha, \beta, \gamma \\
\vdots \\
\tau_{\text{EXPR}}(e_p)\alpha, \beta, \gamma \\
\text{CALL}(\beta(p_0)) \\
\text{STO}(\alpha(v_0)) \\
\vdots \\
\text{STO}(\alpha(v_q)) \\
\text{placing values actual input parameters on expression stack} \\
\text{removing values of formal output parameters from expression stack and assigning to actual output parameters}
\]
\[ \tau_{\text{BLOCK}}( \left[ \text{var } n_0 : t_0, \ldots, n_k : t_k; \right. \]
\[ \quad \text{proc } p_0 = (?x_0 : r_0, \ldots, ?x_p : r_p, !y_0 : r'_0, \ldots, !y_q : r'_q \mid ss) \]
\[ \quad \mid \text{ss} ] ] ) \alpha, \beta, \gamma \]
\[ = \]
\[ \text{UJP}(l_1) \]
\[ lb_0 : \text{translation of } p_0 \]
\[ l_1 : \text{ENTER}(\gamma', k + 1) \]
\[ \tau_{\text{STATS}(\text{ss})} \alpha', \beta', \gamma' \]
\[ \text{EXIT}(\gamma') \]
\[ \alpha' = [(\gamma', 0)/n_0, \ldots, (\gamma', k)/n_k] \alpha \]
\[ \beta' = [lb_0/p_0] \beta \]
\[ \gamma' = \gamma + 1 \]
translation of procedure $p_0$

ENTER($\tilde{\gamma}, p + q + 2$)

\[
\begin{align*}
&\text{STO}(\tilde{\alpha}(x_p)) \\
&\quad \vdots \\
&\text{STO}(\tilde{\alpha}(x_0))
\end{align*}
\]

\[
\tau_{STATS}(ss_0)\tilde{\alpha}, \tilde{\beta}, \tilde{\gamma}
\]

\[
\begin{align*}
&\text{LDV}(\tilde{\alpha}(y_q)) \\
&\quad \vdots \\
&\text{LDV}(\tilde{\alpha}(y_0))
\end{align*}
\]

EXIT($\tilde{\gamma}$)

RET

\[
\begin{align*}
\alpha' &= [(\gamma', 0)/n_0, \ldots, (\gamma', k)/n_k]\alpha \\
\beta' &= [lb_0/p_0]\beta \quad \gamma' = \gamma + 1 \\
\tilde{\alpha} &= [(\tilde{\gamma}, 0)/x_0, \ldots, (\tilde{\gamma}, p + q + 1)/y_q]\alpha' \\
\tilde{\beta} &= \beta' \quad \tilde{\gamma} = \gamma' + 1
\end{align*}
\]