G53CMP: Lecture 12 & 13

Code Generation II

Henrik Nilsson

University of Nottingham, UK

Code Generation: Demo I

Let us generate code for:

```
let
    var f: Integer := 1;
    var i: Integer := 1
in
    while i <= 10 do begin
        f := f * i;
        putint(f);
        i := i + 1
    end
```

Code Generation: Demo II (1)

And for this program using arrays and a procedure:

```
let
    proc swap(var x: Integer, var y: Integer)
        let
            var t: Integer
        in begin
            t := x; x := y; y := t
        end;
    var a: Integer[5] := [7,3,1,9,2];
    var i: Integer;
    var j: Integer
```

Code Generation: Demo II (2)

```
in begin
    i := 0;
    while i < 4 do begin
        j := i + 1;
        while j < 5 do begin
            if a[i] > a[j] then
                swap(a[i], a[j])
            else skip();
            j := j + 1
        end;
        i := i + 1
    end;
```

Code Generation: Demo II (3)

```
    i := 0;
    while i <= 4 do begin
        putint(a[i]);
        i := i + 1
    end
end
```

Specifying Code Selection (1)

- Code selection is specified ***inductively*** over the phrases of the source language:

$$
\begin{array}{rcl}
\mathit{Command} & \rightarrow & \underline{\mathit{Identifier}}\ \mathbf{:=}\ \mathit{Expression} \\
& | & \mathit{Command}\ \mathbf{;}\ \mathit{Command} \\
& \ldots &
\end{array}
$$

Specifying Code Selection (1)

- Code selection is specified ***inductively*** over the phrases of the source language:

$$
\begin{array}{lcl}
\mathit{Command} & \rightarrow & \underline{\mathit{Identifier}}\ \mathbf{:=}\ \mathit{Expression} \\
 & | & \mathit{Command}\ \mathbf{;}\ \mathit{Command} \\
 & \ldots &
\end{array}
$$

- ***Code Function***: maps a source phrase to an instruction sequence. For example:

$$
\begin{array}{rcl}
\mathit{execute} & : & \mathit{Command} \rightarrow \mathit{Instruction}^* \\
\mathit{evaluate} & : & \mathit{Expression} \rightarrow \mathit{Instruction}^* \\
\mathit{elaborate} & : & \mathit{Declaration} \rightarrow \mathit{Instruction}^*
\end{array}
$$

Specifying Code Selection (2)

Note:

Specifying Code Selection (2)

Note:

- $execute$ ***generates code*** for executing a command (it does not execute a command directly);

Specifying Code Selection (2)

Note:

- *execute* **generates code** for executing a command (it does not execute a command directly);
- *evaluate* **generates code** for evaluating an expression, leaving the result on the top of the stack.

Specifying Code Selection (2)

Note:

- *execute* ***generates code*** for executing a command (it does not execute a command directly);
- *evaluate* ***generates code*** for evaluating an expression, leaving the result on the top of the stack.
- *elaborate* ***generates code*** for reserving storage for declared constants and variables, evaluating any initialisation expressions, and for declared procedures and functions.

Specifying Code Selection (3)

- Code functions are specified by means of ***code templates***:

$$
\begin{aligned}
&\mathit{execute}\ [\![\ C_1 \quad \textbf{;} \quad C_2\ ]\!] = \\
&\qquad \mathit{execute}\ C_1 \\
&\qquad \mathit{execute}\ C_2
\end{aligned}
$$

Specifying Code Selection (3)

- Code functions are specified by means of ***code templates***:

$$
\begin{aligned}
execute \, [\![C_1 \; ; \; C_2]\!] = \\
execute \; C_1 \\
execute \; C_2
\end{aligned}
$$

  - The brackets $[\![$ and $]\!]$ enclose pieces of ***concrete syntax*** and ***meta variables***.

Specifying Code Selection (3)

- Code functions are specified by means of ***code templates***:

$$\begin{array}{l} execute\ [\![C_1 \ ;\ C_2]\!] = \\ \quad execute\ C_1 \\ \quad execute\ C_2 \end{array}$$

  - The brackets $[\![$ and $]\!]$ enclose pieces of ***concrete syntax*** and ***meta variables***.
  - Note the ***recursion***; i.e. inductive definition over the underlying phrase structure.

Specifying Code Selection (3)

- Code functions are specified by means of ***code templates***:

$$\begin{aligned} \mathit{execute}\ [\![C_1\ \mathbf{;}\ C_2]\!] = \\ \mathit{execute}\ C_1 \\ \mathit{execute}\ C_2 \end{aligned}$$

  - The brackets $[\![$ and $]\!]$ enclose pieces of ***concrete syntax*** and ***meta variables***.
  - Note the ***recursion***; i.e. inductive definition over the underlying phrase structure.

(Think of $[\![\cdot]\!]$ as a map from concrete to abstract syntax as specified by the abstract syntax grammars.)

Specifying Code Selection (4)

In a simple language, the code template for assignment might be:

$$\begin{aligned} execute \, [\![I := E]\!] &= \\ & evaluate \; E \\ & \mathtt{STORE} \; addr(I) \end{aligned}$$

where

$$addr : Identifier \rightarrow Address$$

Specifying Code Selection (4)

In a simple language, the code template for assignment might be:

$$\begin{aligned} execute\ [\![I\ := E]\!] &= \\ &evaluate\ E \\ &\mathtt{STORE}\ addr(I) \end{aligned}$$

where

$$addr \;:\; Identifier \rightarrow Address$$

Note that the instruction sequences and individual instructions in the RHS of the defining equation are implicitly ***concatenated***.

Specifying Code Selection (4)

In a simple language, the code template for assignment might be:

$$\begin{aligned} execute\ [\![I\ := E\,]\!] &= \\ & evaluate\ E \\ & \mathtt{STORE}\ addr(I) \end{aligned}$$

where

$$addr\ :\ Identifier \to Address$$

Note that the instruction sequences and individual instructions in the RHS of the defining equation are implicitly ***concatenated***.

Note: meta variables range over ***abstract syntax***.

Exercise: Code Templates

Generate code for the fragment

```
f := f * n;
n := n - 1
```

using the following two templates:

$$execute\ [\![C_1 \ \mathtt{;}\ C_2]\!] = \\ \quad execute\ C_1 \\ \quad execute\ C_2$$

$$execute\ [\![I\ \mathtt{:=}\ E]\!] = \\ \quad evaluate\ E \\ \quad \mathtt{STORE}\ addr(I)$$

and $addr(\mathtt{f}) = \mathtt{[SB\ +\ 11]}$, $addr(\mathtt{n}) = \mathtt{[SB\ +\ 17]}$.

Expand as far as the above templates allow.

Not Quite that Simple . . .

However, something is clearly missing! Recall:

$$\begin{aligned} \mathit{execute} &: \mathit{Command} \rightarrow \mathit{Instruction}^* \\ \mathit{evaluate} &: \mathit{Expression} \rightarrow \mathit{Instruction}^* \\ \mathit{elaborate} &: \mathit{Declaration} \rightarrow \mathit{Instruction}^* \\ \mathit{addr} &: \mathit{Identifier} \rightarrow \mathit{Address} \end{aligned}$$

and consider again:

$$\mathit{execute}\ [\![I \ := E]\!] =$$
$$\mathit{evaluate}\ E$$
$$\mathtt{STORE}\ \mathit{addr}(I)$$

Not Quite that Simple ...

However, something is clearly missing! Recall:

$$\begin{aligned} execute &: Command \to Instruction^* \\ evaluate &: Expression \to Instruction^* \\ elaborate &: Declaration \to Instruction^* \\ addr &: Identifier \to Address \end{aligned}$$

and consider again:

$$\begin{aligned} & execute \, [\![I \; := E]\!] = \\ & \quad evaluate \; E \\ & \quad \mathtt{STORE} \; addr(I) \end{aligned}$$

How can the function $addr$ possibly map an identifier (a name) to an address?

Not Quite that Simple . . . (2)

In more detail:

Not Quite that Simple . . . (2)

In more detail:

- $elaborate$ is responsible for ***assigning*** addresses to variables

Not Quite that Simple . . . (2)

In more detail:

- $elaborate$ is responsible for ***assigning*** addresses to variables
- a function like $addr$ needs ***access*** to the addresses assigned by $elaborate$

Not Quite that Simple . . . (2)

In more detail:

- *elaborate* is responsible for ***assigning*** addresses to variables
- a function like *addr* needs ***access*** to the addresses assigned by *elaborate*
- but the given type signatures for the code functions do ***not permit*** this communication!

Not Quite that Simple . . . (3)

Consequently:

Not Quite that Simple . . . (3)

Consequently:

- The code functions need an additional ***stack environment argument***, associating variables with addresses.

Not Quite that Simple . . . (3)

Consequently:

- The code functions need an additional ***stack environment argument***, associating variables with addresses.
- The code function $elaborate$ must ***return an updated stack environment***.

Not Quite that Simple . . . (3)

Consequently:

- The code functions need an additional ***stack environment argument***, associating variables with addresses.
- The code function *elaborate* must ***return an updated stack environment***.
- Need to keep track of the ***current stack depth*** (with respect to `LB`) to allow *elaborate* to determine the address (within activation record) for a new variable.

Not Quite that Simple . . . (5)

- Need to keep track of the current scope level as the difference of current scope level and the scope level of a variable is needed in addition to its address to access it (see later lecture on run-time organisation and ***static links***).

Not Quite that Simple . . . (5)

- Need to keep track of the current scope level as the difference of current scope level and the scope level of a variable is needed in addition to its address to access it (see later lecture on run-time organisation and ***static links***).

Moreover, need to generate ***fresh names*** for jump targets (recall the demo).

Not Quite that Simple . . . (6)

To clearly convey the basic ideas first, we will:

- Use simplified MiniTriangle as main example:
  - No user-defined procedures or functions (only predefined, global ones).
  - Consequently, all variables are global (addressed with respect to `SB`).
  - No arrays (only simple variables, all of size 1 word) .
- Gloss over the bookkeeping details for the most part.

Not Quite that Simple . . . (7)

However:

- Additional details will be given occasionally.
- Will revisit at appropriate points in lectures on run-time organisation.
- Refer to the HMTC (coursework compiler) source code for full details.

Code Functions for MiniTriangle

In the simplified exposition, we can consider the code functions to have the following types:

$$
\begin{aligned}
\mathit{run} &: \mathit{Program} \rightarrow \mathit{Instruction}^* \\
\mathit{execute} &: \mathit{Command} \rightarrow \mathit{Instruction}^* \\
\mathit{execute}^* &: \mathit{Command}^* \rightarrow \mathit{Instruction}^* \\
\mathit{evaluate} &: \mathit{Expression} \rightarrow \mathit{Instruction}^* \\
\mathit{evaluate}^* &: \mathit{Expression}^* \rightarrow \mathit{Instruction}^* \\
\mathit{fetch} &: \mathit{Identifier} \rightarrow \mathit{Instruction}^* \\
\mathit{assign} &: \mathit{Identifier} \rightarrow \mathit{Instruction}^* \\
\mathit{elaborate} &: \mathit{Declaration} \rightarrow \mathit{Instruction}^* \\
\mathit{elaborate}^* &: \mathit{Declaration}^* \rightarrow \mathit{Instruction}^*
\end{aligned}
$$

A Code Generation Monad

HMTC uses a ***Code Generation monad*** to facilitate some of the bookkeeping:

```
instance Monad (CG instr)
```

A Code Generation Monad

HMTC uses a ***Code Generation monad*** to facilitate some of the bookkeeping:

```
instance Monad (CG instr)
```

Takes care of:

- Collation of generated instructions
- Generation of fresh names

A Code Generation Monad

HMTC uses a ***Code Generation monad*** to facilitate some of the bookkeeping:

```
instance Monad (CG instr)
```

Takes care of:

- Collation of generated instructions
- Generation of fresh names

Typical operations:

- `emit :: instr -> CG instr ()`
- `newName :: CG instr Name`

Some HMTC Code Functions

```
execute :: Level -> CGEnv -> Depth -> Command
           -> CG TAMInst ()

evaluate :: Level -> CGEnv -> Expression
           -> CG TAMInst ()

elaborateDecls :: Level -> CGEnv -> Depth
                  -> [Declaration]
                  -> CG TAMInst (CGEnv, Depth)
```

(In essence: actual signatures differ in minor ways.)

MiniTriangle Abstract Syntax Part I

(Simplified: no procedures, functions, arrays)

Program	$\rightarrow$	*Command*	Program
Command	$\rightarrow$	*Identifier* **:=** *Expression*	CmdAssign
	$\vert$	*Identifier* **(** *Expression** **)**	CmdCall
	$\vert$	**begin** *Command** **end**	CmdSeq
	$\vert$	**if** *Expression* **then** *Command* **else** *Command*	CmdIf
	$\vert$	**while** *Expression* **do** *Command*	CmdWhile
	$\vert$	**let** *Declaration** **in** *Command*	CmdLet

Meta Variable Conventions

$$
\begin{array}{rcl}
C & \in & \mathit{Command} \\
Cs & \in & \mathit{Command}^* \\
E & \in & \mathit{Expression} \\
Es & \in & \mathit{Expression}^* \\
D & \in & \mathit{Declaration} \\
Ds & \in & \mathit{Declaration}^* \\
I & \in & \mathit{Identifier} \\
O & \in & \mathit{Operator} \\
IL & \in & \mathit{IntegerLiteral} \\
TD & \in & \mathit{TypeDenoter}
\end{array}
$$

Code Function *execute* (1)

$$\begin{aligned} run &: \text{Program} \rightarrow \text{Instruction}^* \\ execute &: \text{Command} \rightarrow \text{Instruction}^* \end{aligned}$$

$$\begin{aligned} &run\ [\![\, C \,]\!] = \\ &\qquad execute\ C \\ &\qquad \texttt{HALT} \end{aligned}$$

$$\begin{aligned} &execute\ [\![\, I\ := E \,]\!] = \\ &\qquad evaluate\ E \\ &\qquad assign\ I \end{aligned}$$

Code Function *execute* (2)

$$execute\ [\![\ I\ \texttt{(}\ Es\ \texttt{)}\ ]\!] =$$
$$\quad evaluate^{*}\ Es$$
$$\quad \texttt{CALL}\ addr(I)$$

$$execute\ [\![\ \texttt{begin}\ Cs\ \texttt{end}\ ]\!] =$$
$$\quad execute^{*}\ Cs$$

Code Function *execute* (3)

$$
\begin{array}{rl}
\multicolumn{2}{l}{execute\ [\![\texttt{if}\ E\ \texttt{then}\ C_1\ \texttt{else}\ C_2]\!] =} \\
 & evaluate\ E \\
 & \texttt{JUMPIFZ}\ g \\
 & execute\ C_1 \\
 & \texttt{JUMP}\ h \\
g: & execute\ C_2 \\
h: &
\end{array}
$$

where g and h are ***fresh*** names.

Exercise: Code Function *execute*

Given

$$\mathit{evaluate}\ [\![I]\!] = \qquad \mathit{addr}(\mathtt{a}) = \mathtt{[SB + 11]}$$
$$\quad \mathtt{LOAD}\ \mathit{addr}(I) \qquad \mathit{addr}(\mathtt{b}) = \mathtt{[SB + 12]}$$
$$\mathit{execute}\ [\![I := IL]\!] = \qquad \mathit{addr}(\mathtt{c}) = \mathtt{[SB + 13]}$$
$$\quad \mathtt{LOADL}\ IL$$
$$\quad \mathtt{STORE}\ \mathit{addr}(I)$$

generate code for:

```
if b then
    if c then a := 1 else a := 2
else
    a := 3
```

Code Function *execute* (5)

In detail (pseudo Haskell, code generation monad):

$$
\begin{aligned}
&\mathit{execute}\ l\ \mathit{env}\ n\ [\![\, \texttt{if}\ E\ \texttt{then}\ C_1\ \texttt{else}\ C_2 \,]\!] = \textbf{do} \\
&\qquad g \leftarrow \mathit{newName} \\
&\qquad h \leftarrow \mathit{newName} \\
&\qquad \mathit{evaluate}\ l\ \mathit{env}\ E \\
&\qquad \mathit{emit}\ (\texttt{JUMPIFZ}\ g) \\
&\qquad \mathit{execute}\ l\ \mathit{env}\ n\ C_1 \\
&\qquad \mathit{emit}\ (\texttt{JUMP}\ h) \\
&\qquad \mathit{emit}\ (\texttt{Label}\ g) \\
&\qquad \mathit{execute}\ l\ \mathit{env}\ n\ C_2 \\
&\qquad \mathit{emit}\ (\texttt{Label}\ h)
\end{aligned}
$$

Code Function *execute* (6)

$$
\begin{aligned}
execute\ [\![\, \texttt{while}\ E\ \texttt{do}\ C\,]\!] = &\\
& \texttt{JUMP}\ h\\
g: \quad & execute\ C\\
h: \quad & evaluate\ E\\
& \texttt{JUMPIFNZ}\ g
\end{aligned}
$$

where g and h are ***fresh*** names.

Code Function *execute* (7)

$$
\begin{aligned}
execute\,[\![\,\texttt{let}\ Ds\ \texttt{in}\ C\,]\!] = \;& elaborate^{*}\ Ds \\
& execute\ C \\
& \texttt{POP}\ 0\ s
\end{aligned}
$$

where s is the amount of storage allocated by $elaborate^{*}\ Ds$.

Code Function *execute* (8)

In detail (pseudo Haskell, code generation monad):

$$
\begin{aligned}
execute\ l\ env\ n\ [\![\, \texttt{let}\ Ds\ \texttt{in}\ C\,]\!] = \textbf{do}\\
(env', n') &\leftarrow elaborate^{*}\ l\ env\ n\ Ds\\
&execute\ l\ env'\ n'\ C\\
&emit\ (\texttt{POP}\ 0\ (n' - n))
\end{aligned}
$$

where:

$$
\begin{aligned}
elaborate^{*} : \ & Level \rightarrow CGEnv \rightarrow Depth\\
& \rightarrow Declaration^{*}\\
& \rightarrow CG\ TAMInst\ (Env, Depth)
\end{aligned}
$$

Code Function $execute^*$

The code function $execute^*$ has the obvious definition:

$$execute^* \, [\![\, \epsilon \,]\!] = \epsilon$$

$$\begin{aligned} execute^* \, [\![\, C \; ; \; Cs \,]\!] = \\ & execute \;\; C \\ & execute^* \;\; Cs \end{aligned}$$

MiniTriangle Abstract Syntax Part II

Expression	→	*IntegerLiteral*	ExpLitInt
	|	*Identifier*	ExpVar
	|	*Operator Expression*	ExpUnOpApp
	|	*Expression Operator Expression*	ExpBinOpApp
Declaration	→	**const** *Identifier* **:** *TypeDenoter* **=** *Expression*	DeclConst
	|	**var** *Identifier* **:** *TypeDenoter* (**:=** *Expression* | ϵ)	DeclVar
TypeDenoter	→	*Identifier*	TDBaseType

Code Function *evaluate* (1)

$$evaluate \quad : \quad Expression \rightarrow Instruction^{*}$$

Fundamental invariant: all operations take arguments from the stack and writes result back onto the stack.

Code Function *evaluate* (2)

Consider evaluating $2 + 4 * 3 - 5$. Plausible instruction sequence:

`LOADL 2`	Stack: 2
`LOADL 4`	Stack: 4, 2
`LOADL 3`	Stack: 3, 4, 2
`CALL mul`	Stack: 12, 2
`CALL add`	Stack: 14
`LOADL 5`	Stack: 5, 14
`CALL sub`	Stack: 9

(`mul`, `add`, `sub` are routines in the MiniTriangle standard library.)

Code Function *evaluate* (3)

$$evaluate \llbracket IL \rrbracket = \\ \texttt{LOADL}\ c$$

where c is the value of IL.

$$evaluate \llbracket I \rrbracket = \\ fetch\ I$$

Code Function *evaluate* (4)

$$
\begin{aligned}
&\mathit{evaluate}\ [\![\ominus\ E]\!] = \\
&\qquad \mathit{evaluate}\ E \\
&\qquad \texttt{CALL}\ \mathit{addr}(\ominus) \\
&\mathit{evaluate}\ [\![E_1\ \otimes\ E_2]\!] = \\
&\qquad \mathit{evaluate}\ E_1 \\
&\qquad \mathit{evaluate}\ E_2 \\
&\qquad \texttt{CALL}\ \mathit{addr}(\otimes)
\end{aligned}
$$

(A call to a known function that can be replaced by a short code sequence can be optimised away at a later stage; e.g. `CALL add` $\Rightarrow$ `ADD`.)

Code Functions *fetch* and *assign* (1)

In simplified MiniTriangle, all constants and variables are ***global***. Hence addressing relative to `SB`.

$$fetch\ [\![I]\!] =$$

$$\texttt{LOAD [SB + } d\texttt{]}$$

where d is offset (or *displacement*) of I relative to `SB`.

$$assign\ [\![I]\!] =$$

$$\texttt{STORE [SB + } d\texttt{]}$$

where d is offset of I relative to `SB`.

Code Functions *fetch* and *assign* (2)

In a more realistic language, *fetch* and *assign* would take the current scope level and the scope level of the variable into account:

Code Functions *fetch* and *assign* (2)

In a more realistic language, *fetch* and *assign* would take the current scope level and the scope level of the variable into account:

- Global variables addressed relative to `SB`.

Code Functions $fetch$ and $assign$ (2)

In a more realistic language, $fetch$ and $assign$ would take the current scope level and the scope level of the variable into account:

- Global variables addressed relative to `SB`.
- Local variables addressed relative to `LB`.

Code Functions *fetch* and *assign* (2)

In a more realistic language, *fetch* and *assign* would take the current scope level and the scope level of the variable into account:

- Global variables addressed relative to `SB`.
- Local variables addressed relative to `LB`.
- Non-global variables in enclosing scopes would be reached by following the ***static links*** (see later lecture) in one or more steps, and *fetch* and *assign* would have to generate the appropriate code.

Assignment revisited

In detail (pseudo Haskell, code generation monad) the code for assignment looks more like this. Note that the variable actually is represented by an ***expression*** that gets evaluated to an ***address***:

$$
\begin{array}{l}
\mathit{execute}\ l\ \mathit{env}\ n\ [\![E_{\mathrm{v}}\ \mathbf{:=}\ E]\!] = \textbf{do} \\
\qquad \mathit{evaluate}\ l\ \mathit{env}\ E \\
\qquad \mathit{evaluate}\ l\ \mathit{env}\ E_{\mathrm{v}} \\
\qquad \textbf{case}\ \mathit{sizeof}(E)\ \textbf{of} \\
\qquad\qquad 1 \rightarrow \mathit{emit}\ (\texttt{STOREI}\ 0) \\
\qquad\qquad s \rightarrow \mathit{emit}\ (\texttt{STOREIB}\ s)
\end{array}
$$

(Reasons include: array references (`a[i]`), call by reference parameters.)

Exercise: Code Function *evaluate*

Given

$$addr(\texttt{a}) = \texttt{[SB + 11]}$$
$$addr(\texttt{b}) = \texttt{[SB + 12]}$$
$$addr(+) = \texttt{add}$$
$$addr(*) = \texttt{mult}$$

generate code for:

```
a + (b * 2)
```

Code Function *elaborate* (1)

Elaboration must deposit value/reserve space for value on stack. Also, address (offset) of elaborated entity must be recorded (to be used by *fetch* and *assign*).

Code Function *elaborate* (1)

Elaboration must deposit value/reserve space for value on stack. Also, address (offset) of elaborated entity must be recorded (to be used by *fetch* and *assign*).

$$elaborate : \text{Declaration} \rightarrow \text{Instruction}^*$$

$$elaborate\,[\![\, \texttt{const}\ I : TD = E \,]\!] = evaluate\ E$$

Code Function *elaborate* (1)

Elaboration must deposit value/reserve space for value on stack. Also, address (offset) of elaborated entity must be recorded (to be used by *fetch* and *assign*).

$$elaborate \; : \; \text{Declaration} \rightarrow \text{Instruction}^*$$

$$elaborate \, [\![\, \texttt{const}\ I\ \texttt{:}\ TD = E \,]\!] = evaluate\ E$$

(Additionally, the offset (w.r.t. SB) has to be recorded for the identifier denoted by I.)

Code Function *elaborate* (2)

$$elaborate\ [\![\, \texttt{var}\ I\ \texttt{:}\ TD\,]\!] =$$
$$\texttt{LOADL 0}$$
$$elaborate\ [\![\, \texttt{var}\ I\ \texttt{:}\ TD\ \texttt{:=}\ E\,]\!] =$$
$$evaluate\ E$$

(Additionally, the offset (w.r.t. SB) has to be recorded for the identifier denoted by I.)

`LOADL 0` is just used to reserve space on the stack; the value of the literal does not matter. More space must be reserved if the values of the type are big (e.g. record, array).

Code Function *elaborate* (3)

For procedures and functions:

- Generate a fresh name for the entry point.
- Extend the environment according to formal argument declarations (the caller will push actual arguments onto stack prior to call).
- Generate code for the body at a scope level incremented by 1 and in the extended environment.

Identifiers vs. Symbols (1)

- The coursework compiler HMTC uses ***symbols*** instead of identifiers in the latter stages.
- Symbols are introduced in the type checker (responsible for identification in HMTC) in place of identifiers (rep. changed from AST to MTIR).
- Symbols carry ***semantic information*** (e.g., type, scope level) to make that information readily available to e.g. the code generator.

  (Cf. the lectures on identification where applied identifier occurrences were annotated with semantic information.)

Identifiers vs. Symbols (2)

- Two kinds of (term-level) symbols:
  - External: defined outside the current compilation unit (e.g., in a library).
  - Internal: defined in the current compilation unit (in a `let`).

```
type TermSym = Either ExtTermSym IntTermSym

data ExtTermSym = ExtTermSym { ... }

data IntTermSym = IntTermSym { ... }
```

External Symbols

- External symbols are known entities.
- Can thus be looked up once and for all (during identification).
- Have a value, such as a (symbolic) address.

```
data ExtTermSym = ExtTermSym {
                      etmsName :: Name,
                      etmsType :: Type,
                      etmsVal  :: ExtSymVal
                  }

data ExtSymVal = ESVLbl Name | ESVInt MTInt | ...
```

Internal Symbols

- Internal symbols do ***not*** carry any value such as stack displacement because this is not computed until the time of code generation.
- Such "late" information about an entity referred to by an internal symbol thus has to be looked up in the code generation environment.

```
data IntTermSym = IntTermSym {
                      itmsLvl    :: ScopeLvl,
                      itmsName   :: Name,
                      itmsType   :: Type,
                      itmsSrcPos :: SrcPos
                  }
```