Let us generate code for:

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

And for this program using arrays and a procedure:

```latex
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
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:

  \[
  \text{Command} \rightarrow \text{Identifier} := \text{Expression} \mid \text{Command} ; \text{Command} \mid \ldots
  \]

- **Code Function**: maps a source phrase to an instruction sequence. For example:
  - execute : Command $\rightarrow$ Instruction*
  - evaluate : Expression $\rightarrow$ Instruction*
  - elaborate : Declaration $\rightarrow$ Instruction*

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*:

  \[
  \text{execute} \left[ \left[ C_1 \right] ; C_2 \right] = \text{execute} C_1 \text{execute} C_2
  \]

  - 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 [ ] 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:

\[
\text{execute } [ I := E ] = \\
\text{evaluate } E \\
\text{STORE } addr(I)
\]

where

addr : Identifier → 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.

Not Quite that Simple . . .

However, something is clearly missing! Recall:

\[
\text{execute} : \text{Command} \rightarrow \text{Instruction}^* \\
\text{evaluate} : \text{Expression} \rightarrow \text{Instruction}^* \\
\text{elaborate} : \text{Declaration} \rightarrow \text{Instruction}^* \\
\text{addr} : \text{Identifier} \rightarrow \text{Address}
\]

and consider again:

\[
\text{execute } [ I := E ] = \\
\text{evaluate } E \\
\text{STORE } addr(I)
\]

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

Exercise: Code Templates

Generate code for the fragment

\[
f := f \times n; \\
n := n - 1
\]

using the following two templates:

\[
\text{execute } [C_1 ; C_2] = \\
\text{execute } [I := E] = \\
\text{execute } C_1 \\
\text{execute } C_2 \\
\text{evaluate } E \\
\text{STORE } addr(I)
\]

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

Expand as far as the above templates allow.

Not Quite that Simple . . . (2)

In more detail:

- \( \text{elaborate} \) is responsible for \( \text{assigning} \) addresses to variables
- a function like \( \text{addr} \) needs \text{access} to the addresses assigned by \( \text{elaborate} \)
- but the given type signatures for the code functions do \text{not permit} this communication!
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.

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.

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.

Moreover, need to generate `fresh names` for jump targets (recall the demo).
Code Functions for MiniTriangle

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

- `run : Program → Instruction*`
- `execute : Command → Instruction*`
- `execute* : Command* → Instruction*`
- `evaluate : Expression → Instruction*`
- `evaluate* : Expression* → Instruction*`
- `fetch : Identifier → Instruction*`
- `assign : Identifier → Instruction*`
- `elaborate : Declaration → Instruction*`
- `elaborate* : Declaration* → Instruction*`

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.)

A Code Generation Monad

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

```haskell```
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`

MiniTriangle Abstract Syntax Part I

(Simplified: no procedures, functions, arrays)

- `Program → Command`
- `Command → Identifier := Expression`
- `Identifier ( Expression* )`
- `begin Command* end`
- `if Expression then Command`
- `else Command`
- `while Expression do Command`
- `let Declaration* in Command`

Program
CmdAssign
CmdCall
CmdSeq
CmdIf
CmdWhile
CmdLet
Meta Variable Conventions

\[ C \in \text{Command} \]
\[ Cs \in \text{Command}^* \]
\[ E \in \text{Expression} \]
\[ Es \in \text{Expression}^* \]
\[ D \in \text{Declaration} \]
\[ Ds \in \text{Declaration}^* \]
\[ I \in \text{Identifier} \]
\[ O \in \text{Operator} \]
\[ IL \in \text{IntegerLiteral} \]
\[ TD \in \text{TypeDenoter} \]

Code Function \textit{execute} (1)

\textbf{run} : \text{Program} \rightarrow \text{Instruction}^*
\textbf{execute} : \text{Command} \rightarrow \text{Instruction}^*

\[ \text{run} \ [C] = \]
\[ \text{execute} \ C \text{HALT} \]
\[ \text{execute} \ [I := E] = \]
\[ \text{evaluate} \ E \text{assign} \ I \]

Code Function \textit{execute} (2)

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

\[ \text{execute} \ l \ env \ n \ [E_v := E] = \ do \]
\[ \text{evaluate} \ l \ env \ E \]
\[ \text{evaluate} \ l \ env \ E_v \]
\[ \text{case} \ \text{sizeof}(E) \ of \]
\[ 1 \rightarrow \text{emit} \ (\text{STOREI} 0) \]
\[ s \rightarrow \text{emit} \ (\text{STOREIB} \ s) \]

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

Code Function \textit{execute} (3)

\[ \text{execute} \ [I \ (Es)] = \]
\[ \text{evaluate}^* \ Es \]
\[ \text{CALL addr}(I) \]
\[ \text{execute} \ [\text{begin} \ Cs \text{ end}] = \]
\[ \text{execute}^* \ Cs \]
**Code Function** execute (4)

execute \[ \text{if } E \text{ then } C_1 \text{ else } C_2 \] = execute \[ E \]
JUMPIFZ \( g \)
execute \( C_1 \)
JUMP \( h \)
g : execute \( C_2 \)
h :

where \( g \) and \( h \) are fresh names.

**Exercise: Code Function** execute

Given

\[
\begin{align*}
\text{evaluate } [I] & = \\
\text{addr}(a) & = [SB + 11] \\
\text{LOAD } addr(I) & \\
\text{addr}(b) & = [SB + 12] \\
\text{execute } [I := IL] & = \\
\text{addr}(c) & = [SB + 13] \\
\text{LOADL } IL & \\
\text{STORE } addr(I) & \\
\end{align*}
\]

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):

execute \( l \ env \ n \ [\text{if } E \text{ then } C_1 \text{ else } C_2] = \text{do} \)
g ← newName
h ← newName
\text{evaluate } l \ env \ E
\text{emit } (\text{JUMPIFZ } g)
execute \( l \ env \ n \ C_1 \)
\text{emit } (\text{JUMP } h)
\text{emit } (\text{Label } g)
execute \( l \ env \ n \ C_2 \)
\text{emit } (\text{Label } h)

**Code Function** execute (6)

execute \[ \text{while } E \text{ do } C \] =
JUMP \( h \)
g : execute \( C \)
h : \text{evaluate } E
\text{JUMPIFZNZ } \( g \)

where \( g \) and \( h \) are fresh names.
**Code Function** execute (7)

```plaintext
execute \{\text{let } Ds \text{ in } C\} =
\text{elaborate}^* Ds
execute C
POP 0 s
```

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

**Code Function** execute* (8)

In detail (pseudo Haskell, code generation monad):

```plaintext
execute l env n \{\text{let } Ds \text{ in } C\} = \textbf{do}
(env', n') \leftarrow \text{elaborate}^* l env n Ds
execute l env' n' C
\text{emit } (POP 0 (n' - n))
```

where:

\[
\text{elaborate}^* : \text{Level} \rightarrow \text{CGEnv} \rightarrow \text{Depth} \\
\rightarrow \text{Declaration}^* \\
\rightarrow \text{CG TAMInst} (\text{Env, Depth})
\]

**MiniTriangle Abstract Syntax Part II**

- **Expression**
  - `IntegerLiteral`  \(\rightarrow\) `ExpLitInt`
  - `Identifier`  \(\rightarrow\) `ExpVar`
  - `Operator Expression`  \(\rightarrow\) `ExpUnOpApp`
  - `Expression Operator Expression`  \(\rightarrow\) `ExpBinOpApp`

- **Declaration**
  - `const Identifier : TypeDenoter = Expression`  \(\rightarrow\) `DeclConst`
  - `var Identifier : TypeDenoter ( : = Expression | \epsilon)`  \(\rightarrow\) `DeclVar`

- **TypeDenoter**
  - `Identifier`  \(\rightarrow\) `TDBaseType`
**Code Function evaluate (1)**

\[ \text{evaluate} : \text{Expression} \rightarrow \text{Instruction}^* \]

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

**Code Function evaluate (2)**

Consider evaluating \(2 + 4 \times 3 - 5\). Plausible instruction sequence:

- LOADL 2 \quad \text{Stack}: 2
- LOADL 4 \quad \text{Stack}: 4, 2
- LOADL 3 \quad \text{Stack}: 3, 4, 2
- CALL mul \quad \text{Stack}: 12, 2
- CALL add \quad \text{Stack}: 14
- LOADL 5 \quad \text{Stack}: 5, 14
- CALL sub \quad \text{Stack}: 9

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

**Code Function evaluate (3)**

\[ \text{evaluate} \left[ IL \right] = \]

LOADL \(c\)

where \(c\) is the value of \(IL\).

\[ \text{evaluate} \left[ I \right] = \]

fetch \(I\)

**Code Function evaluate (4)**

\[ \text{evaluate} \left[ \ominus E \right] = \]

\[ \text{evaluate} \left[ E_1 \otimes E_2 \right] = \]

\[ \text{evaluate} E_1 \]

\[ \text{evaluate} E_2 \]

CALL addr(\(\ominus\))

CALL addr(\(\otimes\))

(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.

\[
fetch [I] = LOAD [SB + d]
\]

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

\[
assign [I] = STORE [SB + d]
\]

where \(d\) is offset of \(I\) relative to \(SB\).

**Exercise: Code Function evaluate**

Given

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

generate code for:

\[
a + (b \ast 2)
\]

**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.

**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*: Declaration \(\rightarrow\) Instruction*

\[
\text{elaborate } [\text{const } I : TD = E] = \text{evaluate } E
\]

(Additionally, the offset (w.r.t. SB) has to be recorded for the identifier denoted by \(I\).)
**Code Function elaborate (2)**

\[
\text{elaborate \ [\text{var } I : TD ] = LOADL 0}
\]

\[
\text{elaborate \ [\text{var } I : TD := E ] = evaluate } E
\]

(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).

**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.)

**Code Function elaborate (3)**

For procedures and functions:

- Generate a fresh name for the entry point.
- Elaborate the formal arguments to extend the environment.
- Generate code for the body at a scope level incremented by 1 and in the extended environment.

**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 \texttt{let}).

\[
\text{type TermSym} = \text{Either ExtTermSym IntTermSym}
\]

\[
data \text{ExtTermSym} = \text{ExtTermSym} \{ \ldots \}
\]

\[
data \text{IntTermSym} = \text{IntTermSym} \{ \ldots \}
\]
### 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.

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

```haskell
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.

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