Let us generate code for:

\[
\begin{aligned}
\text{let} \\
\quad \text{var } f : \text{Integer} := 1; \\
\quad \text{var } i : \text{Integer} := 1 \\
\text{in} \\
\quad \text{while } i \leq 10 \text{ do} \\
\quad \quad f := f \times i; \\
\quad \quad \text{putint}(f); \\
\quad \quad i := i + 1 \\
\text{end}
\end{aligned}
\]

And for this program using arrays and a procedure:

\[
\begin{aligned}
\text{let} \\
\quad \text{proc } swap(\text{var } x : \text{Integer}, \text{var } y : \text{Integer}) \\
\quad \text{let} \\
\quad \quad \text{var } t : \text{Integer} \\
\quad \text{in} \\
\quad \quad t := x; x := y; y := t \\
\quad \text{end}; \\
\quad \text{var } a : \text{Integer}[5] := [7,3,1,9,2]; \\
\quad \text{var } i : \text{Integer}; \\
\quad \text{var } j : \text{Integer}
\text{in} \\
\quad \text{begin} \\
\quad \quad i := 0; \\
\quad \quad \text{while } i < 4 \text{ do} \\
\quad \quad \quad j := i + 1; \\
\quad \quad \quad \text{while } j < 5 \text{ do} \\
\quad \quad \quad \quad \text{if } a[i] > a[j] \text{ then} \\
\quad \quad \quad \quad \quad \text{swap}(a[i], a[j]) \\
\quad \quad \quad \quad \quad \text{else } \text{skip}(); \\
\quad \quad \quad j := j + 1 \\
\quad \quad \text{end}; \\
\quad \quad i := i + 1 \\
\quad \text{end}; \\
\quad \text{end}
\end{aligned}
\]

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.

Note: meta variables range over abstract syntax.
Exercise: Code Templates

Generate code for the fragment
\[ f := f \times n; \]
\[ n := n - 1 \]
using the following two templates:
\[ \text{execute} \left[ C_1 ; C_2 \right] = \text{execute} \left[ I := E \right] = \]
\[ \text{execute} C_1 \]
\[ \text{evaluate} E \]
\[ \text{execute} C_2 \]
\[ \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 . . .

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

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

Not Quite that Simple . . . (3)

Consequently:

- The code functions need an additional stack environment argument, associating variables with addresses.
- The code function \( \text{elaborate} \) must return an updated stack environment.
- 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 . . . (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).

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:
\[ \text{run} : \text{Program} \rightarrow \text{Instruction}^* \]
\[ \text{execute} : \text{Command} \rightarrow \text{Instruction}^* \]
\[ \text{execute}^* : \text{Command}^* \rightarrow \text{Instruction}^* \]
\[ \text{evaluate} : \text{Expression} \rightarrow \text{Instruction}^* \]
\[ \text{evaluate}^* : \text{Expression}^* \rightarrow \text{Instruction}^* \]
\[ \text{fetch} : \text{Identifier} \rightarrow \text{Instruction}^* \]
\[ \text{assign} : \text{Identifier} \rightarrow \text{Instruction}^* \]
\[ \text{elaborate} : \text{Declaration} \rightarrow \text{Instruction}^* \]
\[ \text{elaborate}^* : \text{Declaration}^* \rightarrow \text{Instruction}^* \]

HMT uses a Code Generation monad to facilitate some of the bookkeeping:
\[
\text{instance Monad (CG instr)}
\]

Takes care of:
- Collation of generated instructions
- Generation of fresh names

Typical operations:
- \( \text{emit} :: \text{instr} \rightarrow \text{CG instr} () \)
- \( \text{newName} :: \text{CG instr Name} \)
**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 → Command

Program → 

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

**Code Function execute (1)**

run : Program → Instruction*
execute : Command → Instruction*

run [C] =
  execute C
HALT

execute [I := E] =
  evaluate E
assign I

**Code Function 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 expression that gets evaluated to an address:

execute l env n [E := E] = do
  evaluate l env E
  evaluate l env E_s
  case sizeof (E) of
    1 → emit (STOREI 0)
    s → emit (STOREIB s)

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

**Code Function execute (3)**

execute [I := E] =
evaluate* E
CALL addr(I)
execute [begin Cs end] = execute* Cs

**Code Function execute (4)**

execute [if E then C1 else C2] =
  evaluate E
  JUMPIFZ g
  execute C1
  JUMP h
  g : execute C2
  h :

where g and h are fresh names.

**Exercise: Code Function execute**

Given

execute [I] =
  addr(a) = [SB + 11]
  addr(b) = [SB + 12]
  addr(c) = [SB + 13]

LOAD addr(I)

execute [I := IL] =
  addr(b) = [SB + 12]
  addr(b) = [SB + 13]

LOADIL
STORE 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):

execute l env n [if E then C1 else C2] = do
  g ← newName
  h ← newName
  evaluate l env E
  emit (JUMPIFZ g)
execute l env n C1
  emit (JUMP h)
execute l env n C2
  emit (Label h)

**Meta Variable Conventions**

C ∈ Command
Cs ∈ Command*
E ∈ Expression
Es ∈ Expression*
D ∈ Declaration
Ds ∈ Declaration*
I ∈ Identifier
O ∈ Operator
IL ∈ IntegerLiteral
TD ∈ TypeDenoter
execute \[ \text{while } E \text{ do } C \] =
\begin{align*}
\text{JUMP } h \\
g : & \text{ execute } C \\
h : & \text{ evaluate } E \\
\text{JUMP } \text{FIZZ } g
\end{align*}
where \( g \) and \( h \) are fresh names.

The code function execute* has the obvious definition:

\[
\text{execute}^* [\epsilon] = \epsilon
\]
\[
\text{execute}^* [C; Cs] = \\
\text{execute } C
\]
\[
\text{execute}^* Cs
\]

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

\[
\begin{aligned}
\text{LOADL } 2 & \quad \text{Stack: } 2 \\
\text{LOADL } 4 & \quad \text{Stack: } 4, 2 \\
\text{LOADL } 3 & \quad \text{Stack: } 3, 4, 2 \\
\text{CALL } \text{mul} & \quad \text{Stack: } 12, 2 \\
\text{CALL } \text{add} & \quad \text{Stack: } 14 \\
\text{LOADL } 5 & \quad \text{Stack: } 5, 14 \\
\text{CALL } \text{sub} & \quad \text{Stack: } 9
\end{aligned}
\]
where \( c \) is the value of \( IL \).

MiniTriangle Abstract Syntax Part II

Expression → IntegerLiteral | Identifier | Operator Expression | Expression Operator Expression

Declaration → const Identifier : TypeDenoter = Expression |
\text{vax Identifier : TypeDenoter} \quad \text{DecVar} |
\text{=} Expression | \epsilon

TypeDenoter → Identifier \quad \text{TDBaseType}

In detail (pseudo Haskell, code generation monad):

execute l env n [let Ds in C] =
\begin{align*}
\text{elaborate}^* Ds \\
\text{execute } C \\
\text{POP } 0 \ s
\end{align*}
where \( s \) is the amount of storage allocated by elaborates\(^* \) Ds.

evaluate : Expression → Instruction*

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

(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 \text{add} \Rightarrow \text{ADD}.)
In simplified MiniTriangle, all constants and variables are **global**. Hence addressing relative to \( SB \).

\[
\text{fetch } [I] = \text{LOAD } [SB + d]
\]

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

\[
\text{assign } [I] = \text{STORE } [SB + d]
\]

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

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.

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**).

\[
\text{elaborate } : \text{Declaration } \rightarrow \text{Instruction}^*
\]

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

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

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

- 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`).

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.)

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.

Given

\[
\begin{align*}
\text{addr}(a) &= [SB + 11] \\
\text{addr}(b) &= [SB + 12] \\
\text{addr}(+) &= \text{add} \\
\text{addr}(\ast) &= \text{mult}
\end{align*}
\]

generate code for:

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