7 Translation to Intermediate Code

(**7. Translation to Intermediate Code, p. 150**)

- This chpater marks the transition from the source program analysis phase to the **target program synthesis** phase.

- All **static aspects** of the semantics have already been checked (scopes, types, levels). The compiler is now ready to emit **intermediate representation (IR)** of code which precisely defines the **dynamic behaviour** of the compiled program.

- Why do we use IR code, not real *machine* (assembly) code here?
  - Keep machine specifics from invading this compiler module, make compiler **portable**.
  - With IR, compiling m languages for n machines requires $m + n$ compiler modules, without IR we require $m \times n$ modules (e.g., $\rightarrow$ `gcc`, **GNU Compiler Collection**, `gcc`'s IR = RTL).

- IR code is designed such that

  ① it is reasonably *easy for the semantic analysis phase to produce IR* (certain machine specific restrictions are lifted, e.g., unlimited availability of *temporaries* [registers]),

  ② IR code instructions *can be mapped to real machine code* for various types of machines,

  ③ each IR code instruction has a clear and simple meaning, thus enabling optimizing **IR code rewrites** in later phases.

- Typical IR code instructions describe extremely simple operations:
  - **fetch** a word (MIPS: 4 bytes) from given memory, address (label) into temporary
  - **jump** to given address (label),
  - **move** value from temporary to temporary,
  - ...

- Later: subsequent **instruction selection** phase maps (a sequence of) of IR instructions to (a sequence of) real assembly code instructions.

- **Tiger Compiler IR Instruction Set**

① IR instructions yielding a value (*expressions*, `T_exp`):

IR Instruction	**Semantics**
`CONST(`i`)`	integer constant i of size W (machine word size)
`NAME(`n`)`	a symbolic name, used as a label in assembly code (e.g., jump target address used in `CALL` below)
`TEMP(`t`)`	the value currently stored in temporary (register) t
`BINOP(`o, e_1, e_2`)`	result of applying binary operator o to operands e_1, e_2. e_1 is evaluated before e_2; available operators o: arithmetic: `PLUS`, `MINUS`, `MUL`, `DIV` bitwise logic: `AND`, `OR`, `XOR` bitwise shifting: `LSHIFT`, `RSHIFT`, `ARSHIFT`
`MEM(`e`)`	contents of memory word (size W) at address e
`CALL(`f, l`)`	result of applying function f to argument list l, f is evaluated before arguments in l are evaluated left to right
`ESEQ(`s, e`)`	execute statement s (for side effects), then return result of expression e

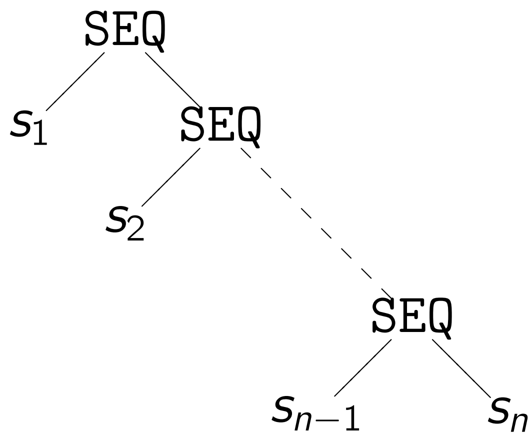
② IR instructions to perform side effects and for flow control (*statements*, `T_stm`):

IR Instruction	**Semantics**
`MOVE(TEMP(`t`),`e`)`	evaluate expression e and move result into temporary t
`MOVE(MEM(`e_1`),`e_2`)`	evaluate e_1 to yield memory address, then evaluate e_2 store resulting word at address e_1 temporary t
`EXP(`e`)`	evaluate e, discard the result
`JUMP(`e, l`)`	jump to address (or label) returned by expression e, e must evaluate to one of the possible jump targets listed in l[26]; simple jump to known label *lab*: `JUMP(`*lab*, `[`*lab*`])`
`CJUMP(`o, e_1, e_2, t, f`)`	evaluate e_1 then e_2, compare results using binary relational operator o, if *true* jump to t, otherwise jump to f; available operators o: signed comparisons: `LT`, `GT`, `LE`, `GE` unsigned comparisons: `ULT`, `UGT`, `ULE`, `UGE`
`SEQ(`s_1, s_2`)`	execute statement s_1, then execute s_2
`LABEL(`n`)`	define label n to be the current machine instruction address (see `NAME(`n`)`)

[26] l is used for *dataflow analysis* purposes later on.

- Much like the source program representation used *abstract syntax* **trees**, we will use **IR instruction trees** to represent intermediate code.

  Example: the sequence of statements $s_1, s_2, \ldots s_n$ will be represented in IR form via



  or, equivalently, $\mathtt{SEQ}(s_1, \mathtt{SEQ}(s_2, \ldots, \mathtt{SEQ}(s_{n-1}, s_n) \cdots))$.

7.1 Translation Into IR Trees

- Translating abstract syntax trees, i.e., trees of type `A_exp`, into IR trees is not too difficult. However, we need to take care of quite a number of cases.

- First off, not all `A_exp` trees map directly into `T_exp` trees.
  - Some Tiger "expressions" do not yield a value (`while` loops, procedure calls, assignments, ...). These should be translated into IR statements (`T_stm`).
  - Boolean Tiger expressions (e.g., `a < b`) might best be translated into a pair of two statements, one to be executed if the expression yields *true*, the other in case of *false*.

- A translated expression is either a **statement**, an **expression**, or a **conditional**:

Translation module

```
typedef struct Tr_exp_ *Tr_exp;
struct Tr_exp_ { enum  { Tr_nx, Tr_ex, Tr_cx } kind;
                 union { T_stm     nx,
                         T_exp     ex,
                         struct Cx cx }        u; }
```

- How shall we represent a conditional (i.e., `struct Cx`) internally?

  Example: the boolean Tiger expression $a > b \mid c < d$ could be compiled into the following IR statement s:

IR statement s

```
    SEQ (CJUMP (GT, a, b, □_t, NAME (z)),
         SEQ (LABEL (z),
              CJUMP (LT, c, d, □_t, □_f)))
```

  Control is transferred (the CPU jumps) to label $\square_t$ whenever the conditional evaluates to *true*, otherwise control is transferred to label $\square_f$.

- When we compile the conditional, however, the **destination labels** $\square_t$ and $\square_f$ are yet **unknown**.

  > **Idea:** represent conditional as statement `T_stm` and two **lists of holes (patches)** that need to be filled with actual destination labels later.

Translation module

```
struct Cx { patchList trues;
            patchList falses;
            T_stm     stm };
```

- During generation of IR instructions we will face situations in which we need to convert on type of IR statement (expression, conditional) into a different type.

Example: convert a conditional (`struct Cx`) into an IR expression `T_exp`). Suppose we need to compiler the Tiger assignment statement (assigning `0` or `1` to `flag`):

$$\texttt{flag := } a > b \mid c < d$$

Plan:

① Translate the right-hand side as a conditional, as shown before.
② Invent a new temporary r, initialize it with `1`.
③ Patch the holes of the conditional with two new labels t and f. At label f place a statement that moves `0` into r.
④ At label t simply return the value of temporary r as the result of the overall expression.

For the example conditional above, we obtain the following IR expression:

IR expression

```
    ESEQ (MOVE (TEMP (r), CONST (1)),
          ESEQ (SEQ (CJUMP (GT, a, b, NAME (t), NAME (z)),
                     SEQ (LABEL (z),
                          CJUMP (LT, c, d, NAME (t), NAME (f)))),
                ESEQ (LABEL (f),
                      ESEQ (MOVE (TEMP (r), CONST (0)),
                            ESEQ (LABEL (t),
                                  TEMP (r))))))
```

- In the translation module (`translate.c`), this conversion strategy is encapsulated in a function `unEx`

`translate.c`

```
T_exp unEx (Tr_exp);
```

that can convert any IR statement (expression, conditional) into an IR expression.

- Similarly, the translation module needs `unCx` and `unNx`.

7.1.1 Simple Variables

- Whenever the translation to IR code comes across the **use** of a simple variable (e.g., `a` in `a + 42`), it depends on the *access* for the variable (see previous chapter) which IR code we need to produce:

① If the access is of the form `InReg(`t`)`, i.e., the variable has been placed in a temporary by the frame module, we translate into the IR expression

$$\texttt{TEMP (}t\texttt{)}$$

② If the variable is frame-resident, i.e., its access is `InFrame(`o`)`, the IR code needs to access the correct memory location (offset o) in the current frame:

$$\texttt{MEM (BINOP (PLUS (TEMP (fp), CONST (}o\texttt{))))}$$

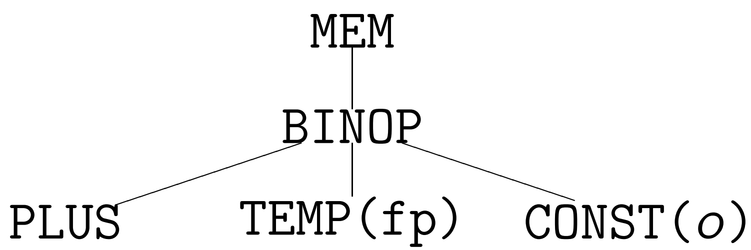
This assumes that all variables are of the same size (word size W of the machine).

- Note that to generate the IR code, we need to know details of the machine's frame layout.
  - Translating accesses into IR code is thus done by a service routine imnplemented in the **frame module**:

frame.h
```
T_exp F_Exp (F_access acc, T_exp frameptr);
```

  - To translate a **local variable** *v*, extract its access information *a* from the environment entry associated with *v*, then call `F_Exp (`*a*`, TEMP (fp))` which constructs the correct IR tree

`TEMP (`*t*`)` *or*



- Are we done with variables?

7.1.2 Following Static Links

- Routine `F_exp ()` receives an expression that computes the current frame pointer `frameptr`. Only for **local** variables, `frameptr` = `TEMP (fp)`.

- For **non-local** (escaped) variables *v*, we need to follow the static link chain to compute `frameptr`.

  - Let *d* be the difference of the static scope depths of *v*'s use and declaration (see previous chapter), then follow the chain *d* times:[27]

frameptr

```
     MEM (BINOP (PLUS,
                   ⋮
                 MEM (BINOP (PLUS,
                             MEM (BINOP (PLUS,
                                         TEMP (fp),
                                         CONST (sl))),
                             CONST (sl)))···))
```

[27] *sl* denotes the offset of the *static link* in the frame's layout, often $sl = 0$.

7.1.3 Array and Record Variables

- In the "*Tiger Language Reference Manual*" the semantics of array (and record) assignment is given as

  When an array or record variable *a* is assigned a value *b*, then *a* **references** the same array or record as *b*. Future updates of *a* will affect *b*, and vice versa, until *a* is reassigned. ...

  – **Example:** Array *a* **aliases** array *b* after assignment:

  Tiger: array reference semantics

```
let
   type vector = array of int
   var a := vector[12] of 0
   var b := vector[12] of 42
in
   (a := b; a[3] := 1)
```

  – Similarly, arrays and records are passed to functions/procedures by reference.

- Tiger's reference semantics allows us to handle **array** or **record variables** just like simple variables (see previous subsection). An assignment like `a := b` merely copies the reference (pointer), i.e., a machine word of size W.
  - Compare the situation in Tiger with C's arrays and records (`struct`).

7.1.4 Array Subscripting, Record Field Selection

- Due to the fact, that *all* Tiger values have an internal representation that occupies W bytes, translating **array subscripts** like

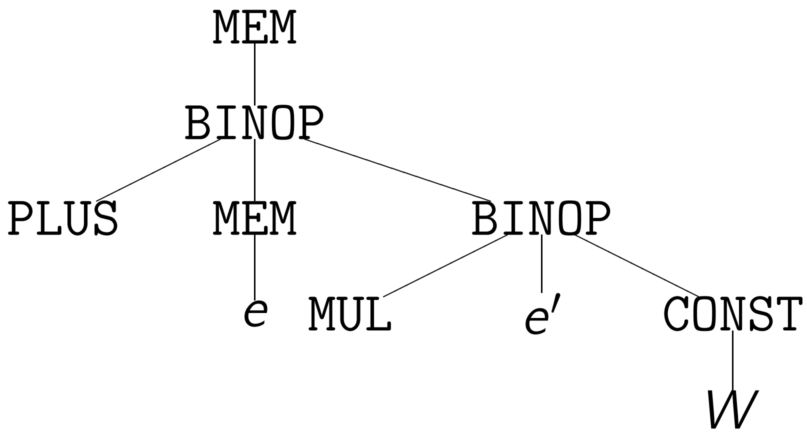
$$a[i]$$

is straightforward:

① translate the array variable, yielding an IR code expression of the form `MEM(`e`)`,

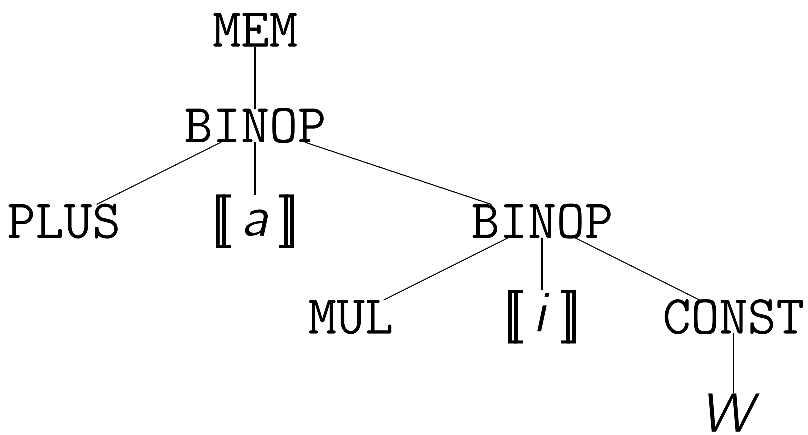
② translate the **subscript expression** i, yielding an IR code expression e'.

③ Emit the IR code



– **Remark:** In textbooks/articles on the semantics and translation of languages, you will often find a notation similar to[28]

$$[\![\, a[i] \,]\!] \quad = $$



which characterizes the translation process to proceed *bottom-up*.

[28] The $[\![\ ]\!]$ are called the *semantic brackets*.

- How does all of this carry over to **record field selection**, i.e., expressions of the form

$$r\,.\,f \ ?$$

  – **Reality check:** give the IR code for the Tiger expression

```
a[i + 1].first := a[i].second + 2
```

  given that `int` variable `i` is a *local* variable with *access* `InFrame(24)` and *a* is a variable of the *immediate outer scope*, located at *access* `InFrame(40)` in the caller's frame.

Declaration of variable `a` in caller

```
let
   type pair  = { first: int, second: int }
   type pairs = array of pair

   var a := pairs[42] of pair { first = 0, second = 0 }
in
   ...
```

7.1.5 Assignment

- Quite obviously, **assignment** in Tiger is mapped into IR code as follows

$$[\![\, a := b \,]\!] \quad = \quad \texttt{MOVE}\ ([\![\, a \,]\!], [\![\, b \,]\!]) \ .$$

  - The occurrence of a variable on the *lhs* of `:=` denotes an **address**, the use of a variable on the *rhs* denotes a **value**. However, the translation scheme above does not make that distinction.
  - If both *a* and *b* above are variables, the resulting IR code will be

$$\texttt{MOVE (MEM (}e\texttt{), MEM (}e'\texttt{))} \ .$$

    (See the description of the IR instruction `MOVE` to see why this works.)

7.1.6 Arithmetic

- Translating arithmetic expressions to IR code s straightforward, e.g.:

$$[\![a \oplus b]\!] \quad = \quad \texttt{BINOP} \; ([\![\oplus]\!], [\![a]\!], [\![b]\!])$$

  with $[\![\texttt{+}]\!] = \texttt{PLUS}, \ldots$

- Unary operators (negation `-`, logical `NOT`) are

  ① **rewritten** into an **equivalent abstract syntax tree** using binary operators, and
  ② translated using the above scheme:

$$[\![\texttt{-}a]\!] \quad =$$

$$[\![\texttt{NOT (}a\texttt{)}]\!] \quad =$$

7.1.7 Conditionals

- Depending on the type (statement/expression/conditional) of the `then` and `else` branches of a conditional

$$\texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3$$

  it might be beneficial to apply distinct translation schemes.

  - The condition e_1 is translated as a conditional (`struct Cx`) with the `trues` and `falses` left to patch.

① If both e_2, e_3 are **expressions**, the whole conditional yields a value:

⟦if e_1 then e_2 else e_3⟧

```
ESEQ (CJUMP (NE, ⟦ e1 ⟧, CONST (0), NAME (true), NAME (false)),
      ESEQ (LABEL (true),
            ESEQ (MOVE (TEMP (r), ⟦ e2 ⟧),
                  ESEQ (JUMP (NAME (join), [ NAME (join) ]),
                        ESEQ (LABEL (false),
                              ESEQ (MOVE (TEMP (r), ⟦ e3 ⟧),
                                    ESEQ (LABEL (join),
                                          TEMP (r))))))))
```

② We could apply the above translation scheme also in case e_2 or e_3 are **statements**: use `unEx` (e_2) and `unEx` (e_3) to turn the statements into expressions[29], then apply case ①.

A better translation scheme for such conditionals would use the fact that both e_1 and e_2 are statements (the conditional yields no value at all in this case):

$[\![$ `if` e_1 `then` e_2 `else` e_3 $]\!]$

```
SEQ (CJUMP (NE, ⟦ e1 ⟧, CONST (0), NAME (true), NAME (false)),
     SEQ (LABEL (true),
          SEQ (⟦ e2 ⟧),
               SEQ (JUMP (NAME (join), [ NAME (join) ]),
                    SEQ (LABEL (false),
                         SEQ (⟦ e3 ⟧), LABEL (join)))))
```

[29] For a statement s, `unEx` (s) = `ESEQ` $(s,$ `CONST (0)`$)$.

③ In case e_2, e_3 are **conditionals** themselves, again, the application of `unEx` followed by case ① gives a valid translation scheme.

Example:

⟦ if x < 5 then a < b else 0 ⟧

```
ESEQ (CJUMP (LT, ⟦ x ⟧, CONST (5), NAME (t1), NAME (f1)),
   ESEQ (LABEL (t1),
      ESEQ (MOVE (TEMP (r1),
            ESEQ (MOVE (TEMP (r2), CONST (1)),
               ESEQ (CJUMP (LT, ⟦ a ⟧, ⟦ b ⟧, NAME (t2), NAME (f2)),
                  ESEQ (LABEL (f2),
                     ESEQ (MOVE (TEMP (r2), CONST (0)),
                        ESEQ (LABEL (t2),
                           TEMP (r2))))))),
         ESEQ (JUMP (NAME (join), [ NAME (join) ]),
            ESEQ (LABEL (f1),
               ESEQ (MOVE (TEMP (r1), CONST (0)),
                  ESEQ (LABEL (join),
                     TEMP (r1))))))))
```

We can generate better code if we

① translate the conditional e_1 into IR code cx_1 and patch its holes with labels `NAME (true)` and `NAME (false)`, respectively,

② apply `unCx` to both e_2 and e_3, giving us IR code fragments cx_2 and cx_3,

③ join the `trues` and `falses` patch list of both conditionals cx_2 and cx_3,

④ and then simply translate into a conditional as follows:

$[\![$ `if` e_1 `then` e_2 `else` e_3 $]\!]$

```
SEQ (cx1,
     SEQ (LABEL (true),
          SEQ (cx2,
               SEQ (LABEL (false),
                    cx3))))
```

Example:

$[\![\,\texttt{if}\ x < 5\ \texttt{then}\ a < b\ \texttt{else}\ 0\,]\!]$

```
SEQ (CJUMP (LT, ⟦ x ⟧, CONST (5), NAME (true), NAME (false)),
     SEQ (LABEL (true),
          SEQ (CJUMP (LT, ⟦ a ⟧, ⟦ b ⟧, □_t, □_f),
               SEQ (LABEL (false),
                    CJUMP (NE, CONST (0), CONST (0), □_t, □_f)))))
```

which can be obviously simplified[30] to

$[\![\,\texttt{if}\ x < 5\ \texttt{then}\ a < b\ \texttt{else}\ 0\,]\!]$

```
SEQ (CJUMP (LT, ⟦ x ⟧, CONST (5), NAME (true), NAME (false)),
     SEQ (LABEL (true),
          SEQ (CJUMP (LT, ⟦ a ⟧, ⟦ b ⟧, □_t, □_f),
               SEQ (LABEL (false),
                    JUMP (□_f, [ □_f ])))))
```

which could be optimized to yield the final translation

$[\![\,\texttt{if}\ x < 5\ \texttt{then}\ a < b\ \texttt{else}\ 0\,]\!]$

```
SEQ (CJUMP (LT, ⟦ x ⟧, CONST (5), NAME (true), □_f),
     SEQ (LABEL (true),
          CJUMP (LT, ⟦ a ⟧, ⟦ b ⟧, □_t, □_f)))
```

[30] This simplifcation should be built into `unCx` already.

7.1.8 Strings

- In Tiger, a **string constant** is a constant **pointer to a segment of memory** initialized with

  ① a machine word storing the length (in characters) of the string,

  ② a sequence of appropriate characters.

  Example: representation of string constant `"foobar"` in MIPS assembly:[31]

  MIPS assembly

```
        .data             # place the following in the data segment
L42:    .word  6          # length of string constant
        .ascii "foobar"   # character sequence
```

  In IR code, this string expression would simply be represented as `NAME (L42)`.

- All other operations on values of type `string` (<, =, `substring`, ...) are *not* implemented as IR code but are provided as routines in the Tiger **runtime system** (e.g., `CALL (NAME (substring), [...])`).

[31] The translation module does the necessary bookkeeping to "remember" that this assembly fragment needs to be output in the final assembly code.

7.1.9 Record and Array Creation

- Just like with strings, records and arrays obey reference semantics. A record created in a function may *outlive* this function and may be referenced even when the creating function has already returned.
  - Records (and arrays) are thus *not* allocated on the stack but on the **heap**. The Tiger runtime system provides a `malloc` routine such that the IR expression

    `CALL (NAME (malloc), [ CONST (`n`) ])`

    returns a pointer to heap memory of size n bytes.
  - Given this heap allocation mechanism, develop IR code that implements the Tiger record creation expression (returning a pointer to the initialized record)

$$t\, \{ f_1 = e_1 \,, f_2 = e_2 \,, \ldots \,, f_n = e_n \}$$

  - The record constant `nil` is translated into `CONST (0)`.

7.1.10 `while` Loops

- The IR equivalent of the Tiger `while` loop

  `while` e_1 `do` e_2

  is not too difficult to construct:

  ① translate the conditional e_1 into the `struct Cx` cx_1 and patch its holes with labels `NAME (loop)` and `NAME (done)`, respectively.

  ② Then translate as shown here:

⟦`while` e_1 `do` e_2⟧

```
SEQ (JUMP (NAME (test), [ NAME (test) ]),
     SEQ (LABEL (loop),
          SEQ (⟦ e2 ⟧,
               SEQ (LABEL (test),
                    SEQ (cx1,
                         LABEL (done))))))
```

- Given this translation scheme, how would you translate `break`?

7.1.11 `for` Loops

- Given a translation scheme for `while` and a translation strategy for variable declarations (see below), we rather **rewrite** the `for` into an equivalent `while` loop:

⟦`for` i `:=` e_1 `to` e_2 `do` e_3⟧

```
⟦
      let var i := e1
          var k := e2
       in
          while i <= k do
             (e3; i := i + 1)
⟧
```

7.1.12 Function Calls

- Tiger function calls $f(e_1, \ldots, e_n)$ carry over to IR code almost one-to-one. We only need to remember to add the **static link** sl as the zero'th pseudo argument:

$$[\![f(e_1, \ldots, e_n)]\!] \quad = \quad \texttt{CALL (NAME (}f\texttt{), [} sl\texttt{, } [\![e_1]\!], \ldots, [\![e_n]\!] \texttt{])}$$

7.1.13 Function Definitions

- Translating a function **body** into IR code poses no extra challenge. In the final compiled program, however, the body needs to be embraced by a function **prologue** and **epilogue**:

  - **Prologue:**

    ① Label definition for the function name,
    ② instructions to adjust the stack pointer (frame allocation),
    ③ instructions to save callee-save registers (including return address),
    ④ instructions to save escaping arguments in the frame and to save non-escaping arguments into registers.

  - **Epilogue:**

    ⑤ Instructions to move return value into appropriate register(s) (MIPS: `$v0`, `$v1`),
    ⑥ instructions to restore callee-save registers from frame,
    ⑦ instructions to reset the stack pointer (discard frame),
    ⑧ return instruction to resume in caller (MIPS: `j $ra`).

- Note that items ②, ③, and ⑥ depend on the actual assignment of IR temporaries to CPU registers or frame slots.
  - Prologue/epilogue generation is thus delayed until after **register allocation**.

- Since prologue/epilogue generation is target machine dependent anyway, the IR translation phase relies on the frame module for this task.
  - **Example:**

    Next slide: MIPS prologue–body–epilogue for Tiger function `foo`:

Tiger function `foo`

```
let
    function foo (x: int) : int =
        x + 42
in
    foo (0)
end
```

MIPS assembly code for Tiger function `foo`

```
foo:
                                # prologue
        addi    $sp,$sp,-32     # addi: add constant
        sw      $v0,32($sp)     # sw: store word
        or      $t8,$0,$a0
        or      $t9,$0,$ra

                                # body
        addi    $t8,$t8,42

                                # epilogue
        or      $v0,$0,$t8
        or      $ra,$0,$t9
        addi    $sp,$sp,32
        j       $ra             # j: jump
```