6 Activation Records

(6. Activation Records , p. 125)

- Tiger allows the use of **recursive** function calls and the use of **local variables** within a function.
  - **Example:** compute the greatest common divisor of *m* and *n* (we assume the presence of a function `mod`):

Recursive Tiger function `gcd`

```
function gcd (m : int, n : int) : int =
   let var x := mod (m, n)
    in
      if n = 0 then m
               else gcd (n, x)
   end
```

  - We expect each invocation of `gcd` to be **independent** of all others: each invocation has its own copy of variables `m`, `n`, and `x`.

- Since many such invocations may be active concurrently, we can *not* use some global memory location to hold these variables.

  – **Example: call graph** for invocation `gcd (21,12)`:

```
gcd (21,12)
     |
 gcd (12,9)
     |
  gcd (9,3)
     |
  gcd (3,0)
```

- Each **invocation needs to be assigned its own local variable storage** that is released in a **LIFO** manner when the function returns.

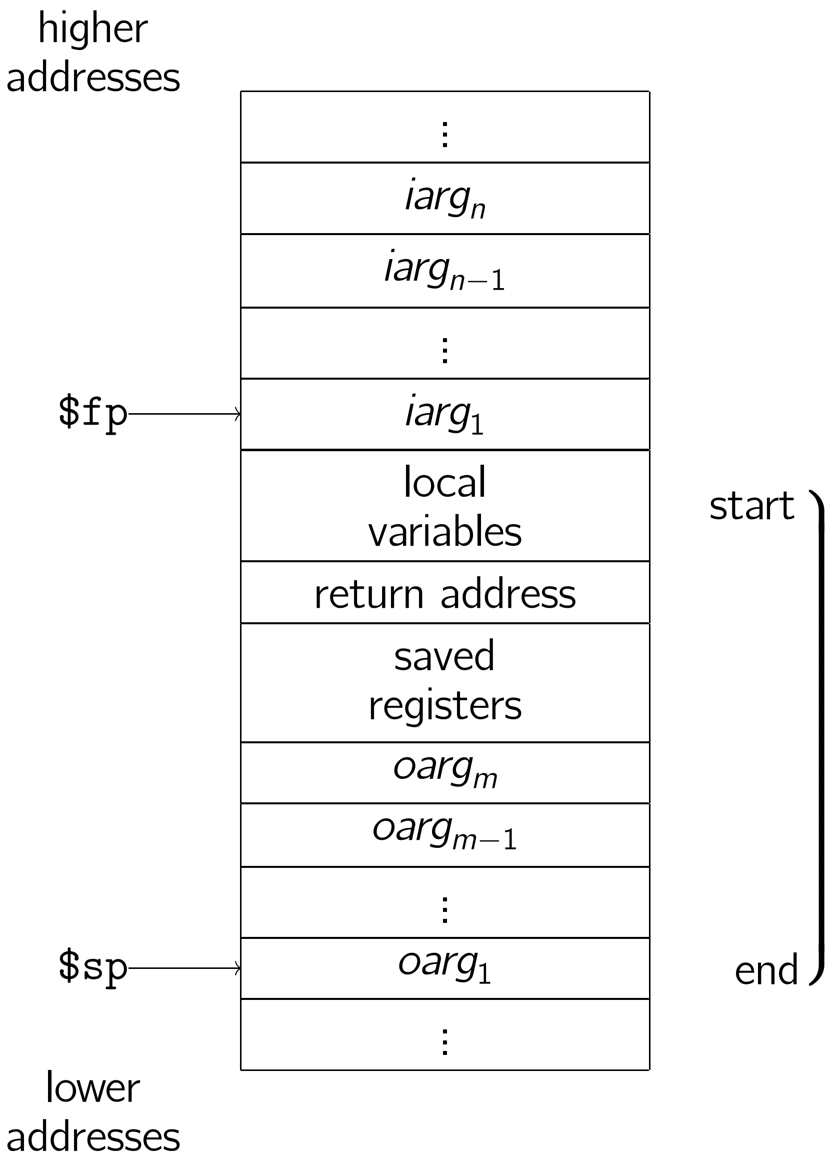
- In Tiger (as is most programming languages), this local variable storage is placed in the CPU **stack**. The depth of the call graph is thus bounded by the stack size (typically, the stack may grow until it fills the entire available heap memory).

6.1 Stack Frames

- Consider an invocation of function `gcd`:
  - Whenever `gcd` is called, we need to grow the stack by 3 entries (*push* arguments `m`, `n`, and local variable `x`, although `x`'s value is yet unknown – what should we *push* there?).
  - Similarly, when `gcd` returns we need to dispose the 3 topmost stack entries in one batch ($3 \times$ *pop*).

- So, instead of the "classical" *push*/*pop* stack model, we rather need a technique to grow/shrink the stack in batches:
  - We use the stack like a dynamic array and adjust the **stack pointer** (MIPS: CPU register `$sp`) for each invocation only once.
  - Stack entries *beyond* `$sp` are garbage, entries *before* `$sp` are considered allocated.
  - On function invocation, we decrease `$sp` so that function arguments and local variables will fit in the newly allocated area (the **stack frame**/**activation record**).

- A typical stack frame layout:[22]



[22] CPU vendors often prescribe a standard frame layout to ensure interoperability of different programming languages.

- In the following, we now turn to and discuss each of the depicted frame slots.

6.1.1 The Frame Pointer

- When a function g() calls out to a function $\mathtt{f}(a_1, \ldots, a_n)$, we refer to g as the **caller** and `f` as the **callee**.

- What happens during a call?

  ① On entry to `f`, `$sp` points to the first **incoming argument** a_1.
  ② `f` allocates a frame on the stack by subtracting the needed **frame size** s from `$sp`.
  ③ The current value of the **frame pointer** (MIPS: `$fp`) is stored in the frame (slot *saved registers*). The old value of `$sp` is remembered in register `$fp`.

- What happens when function `f` exits?

  ① Copy `$fp` to `$sp` (discards the frame).
  ② Restore the old `$fp` value from the just discarded frame.

6.1.2 Registers

- Modern CPUs, especially RISC CPUs, come with a large number (MIPS: 32) of CPU internal **registers**.
  - Register read/write access is much faster than any memory read/write using *load*/*store* instructions (also faster than CPU cache memory).
  - An optimizing compiler will thus try to keep local variables, intermediate evaluation results, etc. in CPU registers.
  - Given that many function invocations can be active in parallel (depth of the call graph), registers are a scarce resource.
  - **Example:**
    - `g` uses register *r* and calls out to `f`.
    - `f` needs to store *r* in its stack frame before it may write to *r* and later restores *r* from the frame before `f` returns (*r* is **callee-saved**)

      OR

      `g` stores *r* in its frame, then calls `f` which clobbers *r* at will until return to `g`, then `g` restores *r* from its own frame (*r* is **caller-saved**).

- CPU vendors usually introduce conventions about CPU registers usage.

  Example: MIPS:

MIPS Register	**Usage**
`$zero`	constant 0
`$at`	reserved for assembler
`$v0`	expression evaluation and function results
`$v1`	expression evaluation and function results
`$a0`	argument 1
⋮	⋮
`$a3`	argument 4
`$t0`	**temporary (caller-save)**
⋮	⋮
`$t7`	**temporary (caller-save)**
`$s0`	**saved temporary (callee-save)**
⋮	⋮
`$s7`	**saved temporary (callee-save)**
`$t8`	**temporary (caller-save)**
`$t9`	**temporary (caller-save)**
`$k0`	reserved for OS kernel
`$k1`	reserved for OS kernel
`$gp`	pointer to global area
`$sp`	stack pointer
`$fp`	frame pointer
`$ra`	return address (used by function call)

- Sometimes, however, caller and or callee can do without register saving.

  Examples:

  - If `g` (caller) knows it will not **need** variable x after calling `f`, hold x in a caller-save register and *do not save* r.
  - If `g` knows it will **need** variable x after several function calls have been made, hold x in a caller-save register r and save/restore r just once.

- The "wise" assignment of variables to caller/callee-save registers is an important compiler optimization (backed up by **data-flow analysis** techniques).

6.1.3 Parameter Passing

- In the frame layout shown before, a function can address the different entries in its frame **relative** to the **frame pointer** `$fp` and the **stack pointer** `$sp`:
  - The function may access its **incoming arguments** $iarg_1$, ... $iarg_k$ (if the function is of arity k) at or just above `$fp`.
  - Likewise, if the function calls out, it places **outgoing arguments** in frame slots $oarg_1 \ldots oarg_m$ at or just above `$sp`.[23]

- Stack-based parameter passing causes significant memory traffic.

- Modern CPU architectures thus reserve a fixed set of k, say, **registers for parameter passing** (MIPS: $k = 4$, $\texttt{\$a0} \ldots \texttt{\$a3}$). If a function takes $n > k$ parameters, only parameters $k + 1 \ldots n$ are passed via the stack.

[23] Note that $oarg_i$ will be $iarg_i$ for the called function.

- By definition, argument passing registers need to be operated in caller-save mode:
  - Suppose $\texttt{g}(a_1, \ldots, a_n)$ receives a_1 in `$a0`. By convention, if `g` calls $\texttt{f}(x)$, it needs to pass x in `$a0`, so `g` must save `$a0` (in its stack frame).
    So, how is any memory traffic saved?

① Some functions are **leaf functions**, i.e., they are leaves in the program's call graph:

- $\texttt{g}(a_1, \ldots, a_n)$
  - $\texttt{g}(a_1, \ldots, a_n)$
    - $\texttt{g}(a_1, \ldots, a_n)$
      - $\texttt{f}(x)$
    - $\texttt{g}(a_1, \ldots, a_n)$
      - $\texttt{f}(x)$
  - $\texttt{g}(a_1, \ldots, a_n)$
    - $\texttt{g}(a_1, \ldots, a_n)$
      - $\texttt{f}(x)$
    - $\texttt{g}(a_1, \ldots, a_n)$
      - $\texttt{f}(x)$
      - $\texttt{f}(y)$

Leaf functions *never* need to caller-save arguments. More importantly, leaf functions often *need no stack frame at all*.

② Some compilers use **interprocedural register allocation**, e.g., g might receive a_1 in `$a0` but pass x to `f`(x) in `$a3` (and the code of `f` is prepared accordingly).

③ g might not be a leaf procedure but may be done with its use of argument register `$a0` before it calls out to `f`.

- Parameter passing via argument registers introduces problems, when the callee needs to access the **address** of an argument.

Examples:

- The `varargs` feature of C, e.g., used in `printf`.
- Taking the address of an argument a_i, e.g. in C:

C code fragment

```
int *f (int x)
{
  return &x;
}
```

6.1.4 Return Address

- When the callee f has completed its work, its last job is to ensure that **execution is resumed** just after the associated $f(\cdots)$ call located in f's caller g.
  - Resuming is implemented by loading the CPU's instruction pointer with the **return address** *ra*:

Return address

```
function g (···) =
    (s_1; ···; f(···); s_k; ···; s_n)
                       ↑
                       ra
```

- Modern CPUs save *ra* in a register rather than on the stack.
  - Non-leaf functions may then save *ra* in their stack frame only if actually needed. (MIPS: *ra* is automatically saved in register `$ra` by instruction `jal` (*jump and link*) , the callee resumes by executing the *jump* instruction

`j $ra` .

MIPS function calling and return

```
g: ⋮
   jal f
   ⋮

f: ⋮
   j $ra
```

6.1.5 Frame-Resident Variables

- Function calling conventions try to ensure to generate as few memory traffic as possible.

- There are some circumstances where a function cannot avoid writing variable values, temporary expression evaluation results, etc., to the stack frame, however:

  ① The value is **too big** to fit into a single CPU register.
  ② The register currently holding the value is needed for a different purpose (**caller/callee saving**, see previous subsections).
  ③ There are too many variables and temporary results that will not all fit into CPU registers (some of these thus need to be **spilled** into the stack frame).
  ④ The variable holding the value is accessed by a function **nested inside** the current one ($\rightarrow$ *static links*, see below). The variable **escapes**.

- One strategy to compile a program would be to **assign each variable a location** (either a register or a slot in the stack frame) **as soon as it is declared**.

  - This is *not* feasible, however, since the compiler will not yet know which registers will be available, how many local variables and temporary expression results will be needed, if a variable will be accessed from inside a nested function etc.

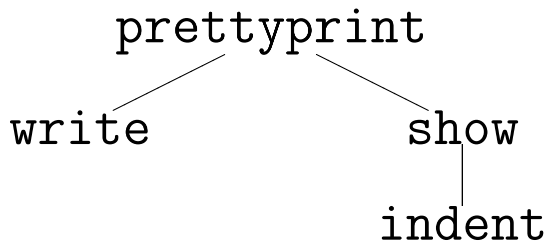
    > **Play safe:** assign (provisional) locations for *all* parameters, local variables, and temporary results *in the stack frame*. An optimization phase may later "move" in-frame locations to registers.

    This optimization phase is commonly referred to as **register allocation**.

6.1.6 Static Links

- Tiger (like Pascal) allows for the declaraction of functions **nested inside** other functions. Consequently, a source program imposes a **static hierarchical structure** on the function declarations contained in it.

  Example: for the Tiger program given on the next slide, the static hierarchical structure is:



- The **scoping rules** of Tiger permit that a function in this hierarchy may **access the parameters and variables of any of its ancestors**.

Tiger program with nested functions

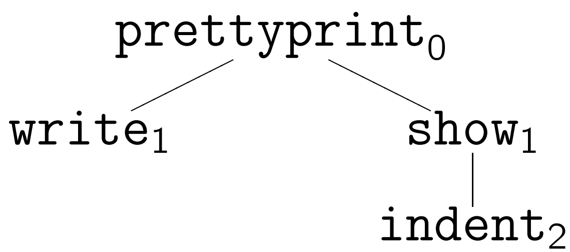
```
type tree = { key: string, left: tree, right: tree }

function prettyprint (tree: tree) : string =
    let
        var output := ""

        function write (s: string) =
            output := concat (output, s)

        function show (n: int, t: tree) =
            let function indent (s: string) =
                (for i := 1 to n do
                     write (" ");
                 output := concat (output, s); write ("\n"))
             in if t = nil then indent (".")
                else (indent (t.key);
                      show (n + 1, t.left);
                      show (n + 1, t.right ))
            end

     in show (0, tree); output
    end
```

- This program contains several references to non-local variables of functions higher up in the static hierarchy:

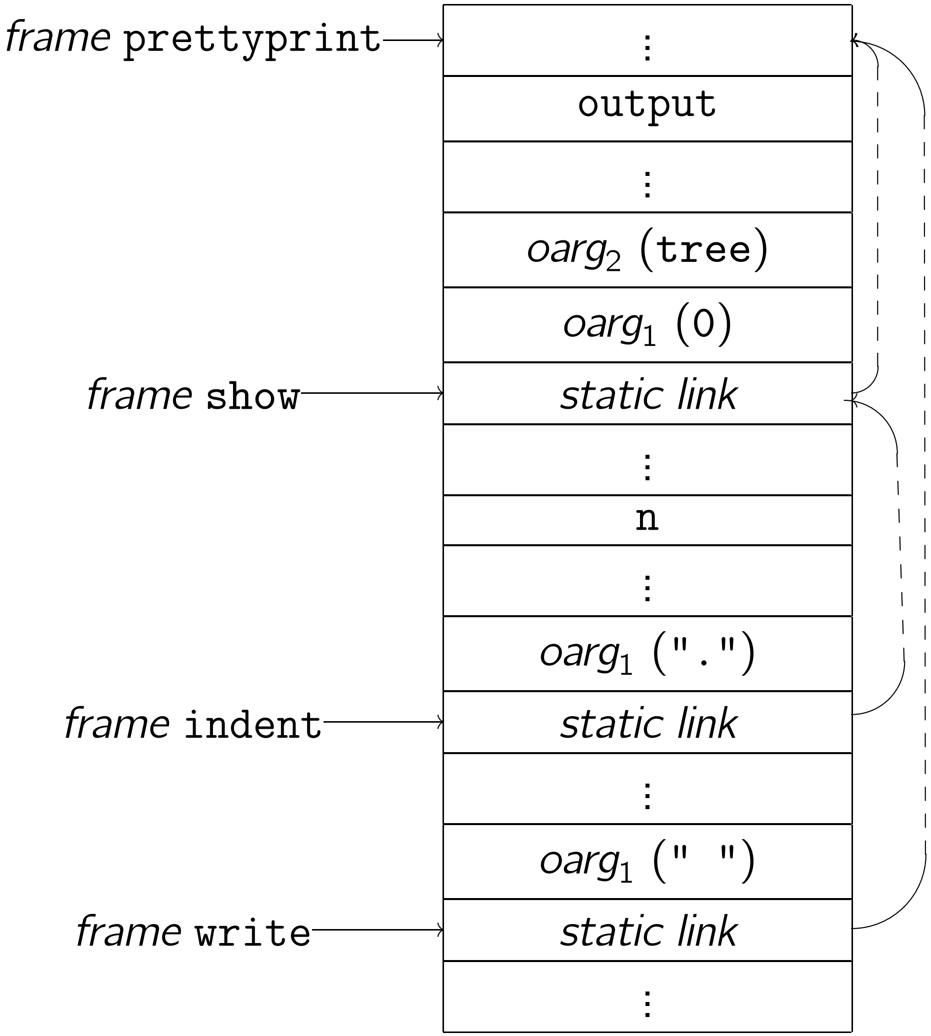
  ① line 8: `write` references `output` (`output` **escapes** from `prettyprint`).
  ② line 12: `indent` references `n` (`n` **escapes** from `show`).
  ③ line 14: `indent` references `output`.

- To implement references to escaped variables, we make sure that each called function f finds the **frame pointer of its parent in the static hierarchy** in a predefined slot in its own frame. This is called the **static link**.

  – If f needs to access a non-local variable of any ancestor g higher up in the hierarchy, it simply follows the chain of static link pointers to find g's frame pointer.
  – *length of chain = difference of static nesting depths of g and f*:



- State of the CPU stack after calling sequence `prettyprint`–`show`–`indent`–`write`:



  – The static link of `write`'s activation record points to the frame of its static parent `prettyprint`, *not* the frame of its "dynamic parent" (= caller) `indent`.

Examples:

- `indent` uses `show`'s escaped parameter `n`:

$$\mathit{depth}_{\texttt{indent}} - \mathit{depth}_{\texttt{show}} \quad = \quad 2 - 1 \quad = \quad 1 \ .$$

  The compiler thus needs to insert code to follow the static link chain *once* to obtain `show`'s frame pointer which is then used to access `n`.

- `write` uses `prettyprint`'s escaped variable `output`:

$$\mathit{depth}_{\texttt{write}} - \mathit{depth}_{\texttt{prettyprint}} \quad = \quad 1 - 0 \quad = \quad 1 \ .$$

  Insert code to follow the static link chain *once* to obtain `prettyprint`'s frame pointer which is then used to access `output`.

- `indent` uses `prettyprint`'s escaped variable `output`:

$$\mathit{depth}_{\texttt{indent}} - \mathit{depth}_{\texttt{prettyprint}} \quad = \quad 2 - 0 \quad = \quad 2 \ .$$

  Insert code to follow the static link chain *twice* to obtain `prettyprint`'s frame pointer which is then used to access `output`.

6.2 Frames in the Tiger Compiler

- The actual frame layout our compiler needs to use depends on a number of issues, among these notably

  ① the target CPU type,
  ② and programming language interoperability conventions (suppose we want to interface Tiger programs with a C library).

- To prevent such details from invading the semantic analysis phase of the compiler, we instead rely on an **abstract frame interface** (specified in C header file `frame.h`).

  – Each target-specific backend then only needs to implement this well-defined frame interface (e.g., we can have implementations like `mipsframe.c`, `pentiumframe.c`, `sparcframe.c`).

- The C data structure representing a frame remains abstract (i.e., the actual C `struct F_frame_ { ... }` is only found in the target-specific `mipsframe.c`, ...):

`frame.h` **(to be continued)**

```
typedef struct F_frame_ *F_frame;
```

- The following function `F_newFrame` creates a new frame:

`frame.h` **(continued)**

```
F_frame F_newFrame (Temp_label name, U_boolList formals);
```

  – `name` denotes the function name[24],
  – `formals` is a list of k Booleans (if the function is k-ary). If the n-th entry in this list is `TRUE`, the n-th parameter of the function escapes, otherwise it does not.
    Example: the frame for function `show (n: int, t: tree)` with `n` escaping from `show`, could be created via

```
f = F_newFrame (Temp_namedlabel ("show"),
                U_BoolList (TRUE /* n */, U_BoolList (FALSE /* t */, NULL)));
```

[24]For now, think of `Temp_label` as of other symbols used in the compiler.

- Whenever a local variable needs to be allocated in the frame:

frame.h **(continued)**

```
F_access F_allocLocal (F_frame f, bool escape);
```

  - `f` denotes the frame we need to allocate the variable in,
  - `escape` indicates if this variable needs to be allocated in frame or may live in a temporary register (`escape` = `false`).
  - `F_allocLocal` returns the **access** information needed to reference the new variable later on. An access is either
    1. `InFrame(`o`)`, where o is an offset relative to the frame pointer,
    2. `InReg(`t`)`, with t denoting a **temporary location**[25]. In this compilation phase we simply assume an infinite pool of temporaries.

Example: on the MIPS, successive `F_allocLocal (`f`,TRUE)` calls yield accesses of the form

`InFrame(-4)`, `InFrame(-8)`, `InFrame(-12)`, ...

because the **word size** W for the MIPS architecture is 4.

[25] Register allocation will later replace t by a specific CPU register.

- Likewise, the access information for all formal parameters of a function is returned by

frame.h **(continued)**

```
F_accessList F_formals (F_frame f);
```

Example: for the frame `f` of function `show (n: int, t: tree)` with `n` escaping, `F_formals (f)` might return the list

$$[\texttt{InFrame (0)}, \texttt{InReg (}t_2\texttt{)}]$$

or, if we conservatively assume that all formals escape:

$$[\texttt{InFrame (0)}, \texttt{InFrame (4)}]$$

- Lastly, the implementation of the frame module (e.g., `mipsframe.c`) needs to encapsulate details of the **calling conventions** used on the target machine.

  Example:

  – MIPS conventions demand that caller passes the first four parameters in `$a0` ... `$a3`.

  – In a simple compiler we might however choose to use frame-resident variables only (i.e., all accesses are of the form `InFrame(`o`)`).

  – The frame module provides a *sequence of instructions*, the **view shift**, ensuring that, on function entry, parameters are moved into the callee's frame where they are expected by the callee's body.

  Example: MIPS view shift for a two-argument function (parameters passed by caller in `$a0`, `$a1`) with two frame-resident local variables (frame size s, $M[a]$: memory access at address a):

MIPS view shift (pseudo code)

```
$fp ← $sp
$sp ← $sp - s
M[fp-12] ← $a0
M[fp-16] ← $a1
```