function \( f(x \colon \text{int}) : \text{int} = \)

\[
\begin{align*}
&\text{let var } y := x + x \\
&\text{in if } y < 10 \\
&\quad \text{then } f(y) \\
&\quad \text{else } y - 1 \\
&\text{end}
\end{align*}
\]

- we need multiple copies of \( x \) and \( y \) — one per call of \( f \)

- storage for \( x \) and \( y \) can be freed when \( f \) returns, i.e., in stack fashion

- Actually, this function is end-recursive. Only one copy is needed for \( x \) and \( y \).
Stack Frames
a.k.a. Activation Records

↑ high mem

incoming args

arg n
...arg 1
static link

previous frame

local vars

ret addr
temp.
saved regs.

current frame

outgoing args

arg m
...arg 1
static link

next frame

↓ low mem

sp

fp
Stack Frames

- push-pop of individual variables not good enough
- push-pop entire frames
- for historical reasons, slack grows down-like icicles

- language-specific
  - e.g., C doesn't need a static link
  - C++ needs pointer to current object (this)

- machine specific
- layout prescribed by manufacturer
Frame Pointer

- incoming arguments at positive offset from fp
- local vars at negative offset from fp

- fp distinct from sp since the size of the frame varies
- may be implemented as one or two registers
- fp needs to be saved on function call, sp becomes new fp
Registers

- lots of them (e.g., 32)
  for RISC processors

- caller-save

- callee-save
  (on MIPS: 16-23)
Parameter Passing

- 1960: statically allocated
- 1970: on stack
- now: in registers

- typically 4-6 arguments in registers, rest on stack
- need to save old contents of registers
- space reserved on stack for all arguments, e.g., for functions with variable number of args in C
Registers

- Leaf procedures, procedures that don't call other procedures, don't need to save registers
- Interprocedural register alloc., saves registers (not used in Tijer compiler)
- For end-recursive procedures, the frame can be reused
- Sparc has register windows

\[10-17 \quad 16-17 \quad 80-07\]
Escaping Variables

- used as non-local variables in inner scope
- address of variable taken (in C: &x)
- passed by reference to other function
- need to be allocated in frame
- non-escaping variables that fit into registers can be kept in registers
var x := 1

function foo () =
  let
    var y := 2
  in
    function bar () =
      let
        var z := 3
      in
        x := y + z
      end
    in
      x := x + y
    end
  end

- static link needed for accessing y in bar
- not needed for C
Lab 4

Absyn
Find Escape

Frame

Access
AccessList
Frame

Mips
InFrame
In Reg
MipsFrame

Semantic
Entry — contains Translate, Access

Translate — contains Frame, Access

Access
AccessList
Exp

Level — contains Frame, Frame

Util
Temp

abstract

machine-specific

language-specific
Accesses

Semant.
VarEntry

Translate.
Access

Translate.
Level

Frame.
Access

Mips.
InFrame

Mips.
InReg

Int offset

Temp. Temp.
Levels / Frames

Semant. FunEntry

Translate. Level

Translate. Frame

Temp. Label

Mips. MipsFrame
public abstract class Frame {
    abstract public Frame
    newFrame (Label, Util.BoolList),
    public Label name;
    public AccessList formals;
    abstract public Access
    allocLocal (bodeaun escape);
}
public class Level {
    Frame frame;
    public AccessList formulas;
    public Level (Level parent, Symbol name, BoolList funs);
    public Level (Frame frame);
    public Access allocLocal (boolean escape);
};
View Shift

- caller passes args in registers
- callee might need to save them (if they escape) or move them elsewhere

Example:

Formals 1
2
3
View Shift

In Frame (0)
In Reg (t157)
In Reg (t158)

sp ← sp + \(-k\)
M[sp+K+0] ← r2
\(t\ 154 \leftarrow r\ 4\)
\(t\ 158 \leftarrow r\ 5\)
Fortran I

- no recursion
- no static scopes
- call-by-reference

- activation records in static run
  - params are pointers to args
  - locals vars
  - return address
  - temps
  - saved registers
C

```c
int x;
int foo () { .. x .. }
int bar (int (*f)()) {
  int y;
  ..
  x + y + f(); ... ..
  ..
}

-no static scoping
-no static link
-no problem
```
C++

```cpp
int x;

class C {
    int y;

 public:
    int foo () { return ++y; }
};

C* p = new C(5);

-C::foo needs access to y
-no static link
-needs pointer to receiver obj.
-generated code:
    int C_foo_Cptr (C* this) {
        return ++this->y;
    }
```
Higher-order functions

\[ f(1): \]
\[ x = 1 \]
\[ \text{static} \]

\[ h(5): \]
\[ y = 5 \]
\[ \text{static} \]

\[ f(3): \]
\[ x = 3 \]
\[ \text{static} \]

\[ i(5): \]
\[ y = 5 \]
\[ \text{static} \]

\[ \text{main:} \]
\[ h \quad \text{code} \]
\[ \text{static} \]
\[ j \quad \text{code} \]
\[ \text{static} \]
\[ z = 6 \]
\[ w = 8 \]
Higher-order functions

ML

\[
\text{fun } f(x: \text{int}) = \\
\text{let } \\
\text{fun } g(y) = x + y \\
\text{in } \\
g \\
\text{end}
\]

\[
\text{val } h = f(1); \\
\text{val } j = f(3); \\
\text{val } z = h(5); \\
\text{val } w = j(5);
\]

C

\[
\text{int } (\ast)(\text{int}) \ f(\text{int } x) \{ \\
\text{int } g(\text{int } y) \{ \\
\text{return } x + y; \\
\} \\
\} \\
\}
\]

\[
\text{int } (\ast h)(\text{int}) = f(1); \\
\text{int } (\ast j)(\text{int}) = f(3); \\
\text{int } z = h(5); \\
\text{int } w = j(5);
\]

- \( h \) must remember \( x = 1 \)
- \( j \) must remember \( x = 3 \)
- activation record of \( f \) must be allocated on heap
ML

- special syntax for higher-order functions

\[ \text{fun } f \times y = x + y : \text{int} \]

is equivalent to

\[ \text{fun } f (x) = \]
\[ (\text{fun } y \Rightarrow x + y : \text{int}) \]