Loops & Nested Functions

Eelco Visser

TU Delft

CS4200 | Compiler Construction | December 9, 2021

This Lecture

Loops

- while

Intermezzo

- dynamic rewrite rules in Stratego

Functions

- activation records
- nested functions
- static links

While Loops

Loops in ChocoPy

$$
\begin{aligned}
\textit{stmt} ::= \;& \textit{simple\_stmt}\ \text{NEWLINE} \\
\mid\; & \textbf{if}\ \textit{expr}\ \textbf{:}\ \textit{block}\ [\![\textbf{elif}\ \textit{expr}\ \textbf{:}\ \textit{block}\ ]\!]^{*}\ [\![\textbf{else}\ \textbf{:}\ \textit{block}]\!]^{?} \\
\mid\; & \textbf{while}\ \textit{expr}\ \textbf{:}\ \textit{block} \\
\mid\; & \textbf{for}\ \text{ID}\ \textbf{in}\ \textit{expr}\ \textbf{:}\ \textit{block}
\end{aligned}
$$

Semantics of While Loops

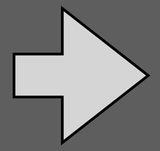
$$\frac{G, E, S \vdash e : bool(false), S_1, \_}{G, E, S \vdash \texttt{while}\ e\texttt{:}\ b : \_, S_1, \_} \quad [\text{WHILE-FALSE}]$$

$$\frac{\begin{array}{l} G, E, S \vdash e : bool(true), S_1, \_ \\ G, E, S_1 \vdash b : \_, S_2, \_ \\ G, E, S_2 \vdash \texttt{while}\ e\texttt{:}\ b : \_, S_3, R \end{array}}{G, E, S \vdash \texttt{while}\ e\texttt{:}\ b : \_, S_3, R} \quad [\text{WHILE-TRUE-LOOP}]$$

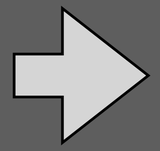
$$\frac{\begin{array}{l} G, E, S \vdash e : bool(true), S_1, \_ \\ G, E, S_1 \vdash b : \_, S_2, R \\ R \text{ is not } \_ \end{array}}{G, E, S \vdash \texttt{while}\ e\texttt{:}\ b : \_, S_2, R} \quad [\text{WHILE-TRUE-RETURN}]$$

Translating While Loops

```
x : int = 1
while x < 10:
  x = x + 3
x
```



```
CProgram(
  [ CBlock(
      CLabel("Main")
    , CSeq(
        CVarDec(CVar("x3"), CIntT(), CInt("1"))
      , CSeq(
          CVarDec(CVar("tmp0"), CIntT(), CInt("0"))
        , CGoto(CLabel("While1"))
        )
      )
    )
  , CBlock(
      CLabel("While1")
    , CIf(
        CLt(CVar("x3"), CInt("10"))
      , CLabel("Do1")
      , CLabel("Join1")
      )
    )
  , CBlock(
      CLabel("Join1")
    , CSeq(CAssign(CVar("tmp0"), CVar("x3")), CReturnNone())
    )
  , CBlock(
      CLabel("Do1")
    , CSeq(
        CAssign(CVar("x3"), CAdd(CVar("x3"), CInt("3")))
      , CGoto(CLabel("While1"))
      )
    )
  ]
)
```



```
.data

.text

Main:
  nop # x3: int ⇒ 0
  li      t0, 1
  nop # tmp1: int ⇒ 1
  li      t1, 0
  j       While0

While0:
  li      t1, 10
  blt     t0, t1, Do0
  j       Join0

Do0:
  addi    t0, t0, 3
  j       While0

Join0:
  mv      t1, t0
```

Notes on Loops

Liveness analysis

- should join information from branches
- should compute a fixpoint

For loops

- iterate over lists

Context-Sensitive Transformation with Scoped Dynamic Rewrite Rules

Building Control-Flow Graph

```
rules // control-flow graph

  add-cfg-node  :: CBlock → CBlock
  all-cfg-nodes :: List(CBlock) → List(CBlock)

  add-cfg-node =
    ?block
    ; rules( CFGNode :+ _ → block )

  all-cfg-nodes =
    <bagof-CFGNode <+ ![]>()
    ; is(List(CBlock))
```

Counting Stack

```
rules

  stack-set(|n) =
    rules(Stack : () → n); !n

  stack-get =
    <Stack>() <+ !0

  stack-inc(|n) =
    stack-set(|<add>(<stack-get>, n))
```

define rewrite rule dynamically

invoke dynamic rewrite rule

```
{| Stack
 : stack-set(|0)
 ; <some-transformation> t ⇒ instrs
 ; size := <stack-get>
 |}
```

dynamic rule scope

forget dynamic rules added within scope

Keeping Track of Local Variables

```
rules

  var-offset-set(|x, n) =
    rules(VarOffset : x → n)

  var-offset-get :
    x → n
    with <VarOffset> x ⇒ n
```

define rewrite rule dynamically

invoke dynamic rewrite rule

```
rules

  fun-arg :
    TypedVar(x, t) → offset
    with var-offset-set(|x, <stack-get ⇒ offset>)
    with stack-inc(|4)

  exp-to-instrs-(|r, regs) :
    Var(x) → [Lw(r, <int-to-string>offset, "fp")]
    with <var-offset-get>x ⇒ offset
```

bind offset of formal parameter

lookup offset of formal parameter

Functions

Functions in ChocoPy

function name

local variables

return to caller

call function

```
def callee(x : int, y : int, z: int) → int:
  a : int = 1
  b : int = 2
  return x + y + z + a + b

def caller():
  d : int = 0
  d = callee(345, 4357, 235)
```

formal parameters

actual parameters

Operational Semantics: Invoke Function

$$
\frac{\begin{array}{l}
S_0(E(f)) = (x_1, \ldots, x_n, y_1 = e'_1, \ldots, y_k = e'_k, b_{body}, E_f) \\
n, k \geq 0 \\
G, E, S_0 \vdash e_1 : v_1, S_1, \_ \\
\vdots \\
G, E, S_{n-1} \vdash e_n : v_n, S_n, \_ \\
l_{x1}, \ldots, l_{xn}, l_{y1}, \ldots, l_{yk} = newloc(S_n, n+k) \\
E' = E_f[l_{x1}/x_1] \ldots [l_{xn}/x_n][l_{y1}/y_1] \ldots [l_{yk}/y_k] \\
G, E', S_n \vdash e'_1 : v'_1, S_n, \_ \\
\vdots \\
G, E', S_n \vdash e'_k : v'_k, S_n, \_ \\
S_{n+1} = S_n[v_1/l_{x1}] \ldots [v_n/l_{xn}][v'_1/l_{y1}] \ldots [v'_k/l_{yk}] \\
G, E', S_{n+1} \vdash b_{body} : \_, S_{n+2}, R \\
R' = \begin{cases} None, \text{ if } R \text{ is } \_ \\ R, \text{otherwise} \end{cases}
\end{array}}{G, E, S_0 \vdash f(e_1, \ldots, e_n) : R', S_{n+2}, \_} \quad [\textsc{invoke}]
$$

Operational Semantics: Define Function

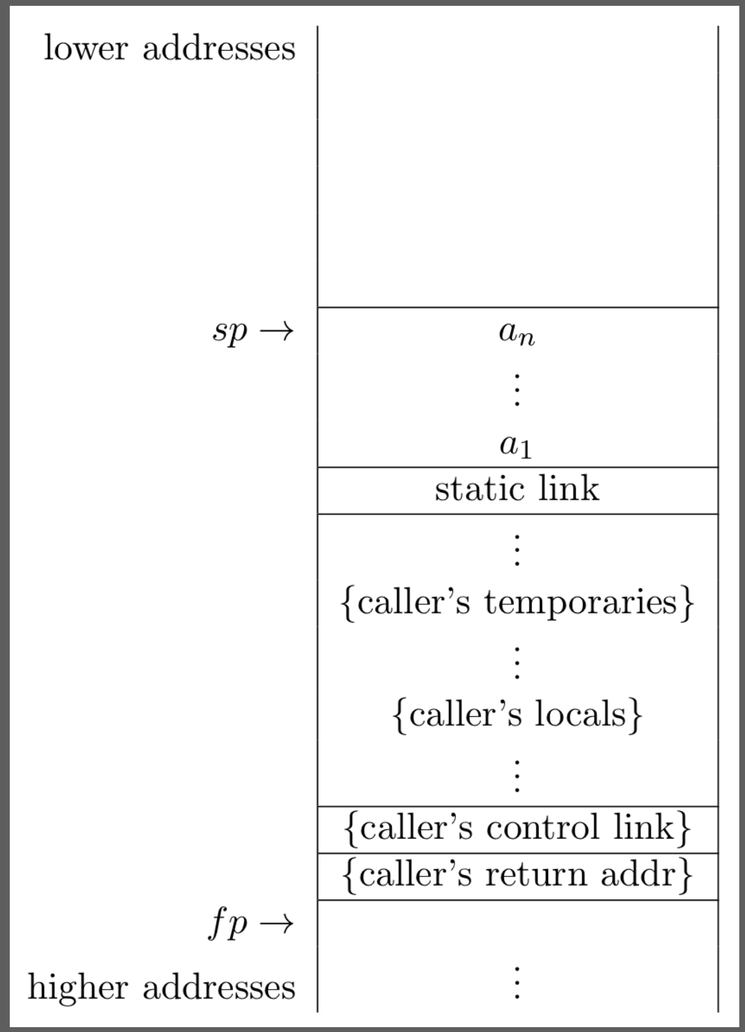
$g_1, \ldots, g_L$ are the variables explicitly declared as global in f
$y_1 = e_1, \ldots, y_k = e_k$ are the local variables and nested functions defined in f
$E_f = E[G(g_1)/g_1] \ldots [G(g_L)/g_L]$
$v = (x_1, \ldots, x_n, y_1 = e_1, \ldots, y_k = e_k, b_{body}, E_f)$

$$\frac{}{G, E, S \vdash \texttt{def}\ f(x_1 : T_1, \ldots, x_n : T_n)\ [\![\texttt{->}\ T_0]\!]^? : b : v, S, \_}$$

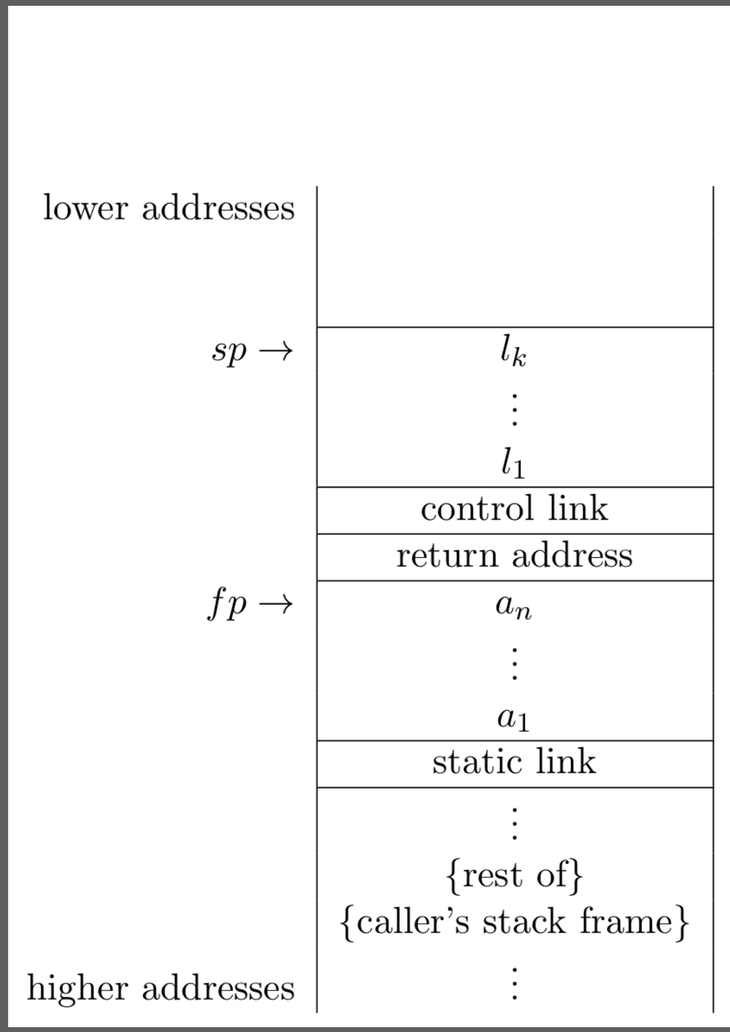
[FUNC-METHOD-DEF]

Activation Records

```
def callee(x : int, y : int, z: int) → int:
  a : int = 1
  b : int = 2
  return x + y + z + a + b

def caller():
  d : int = 0
  d = callee(345, 4357, 235)
```



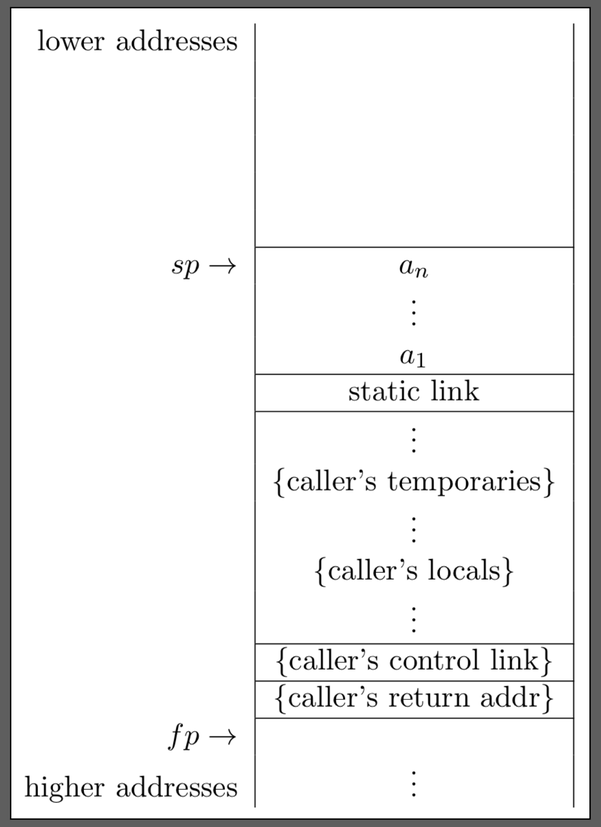
before/after invocation



during callee's execution

Calling Convention: Caller

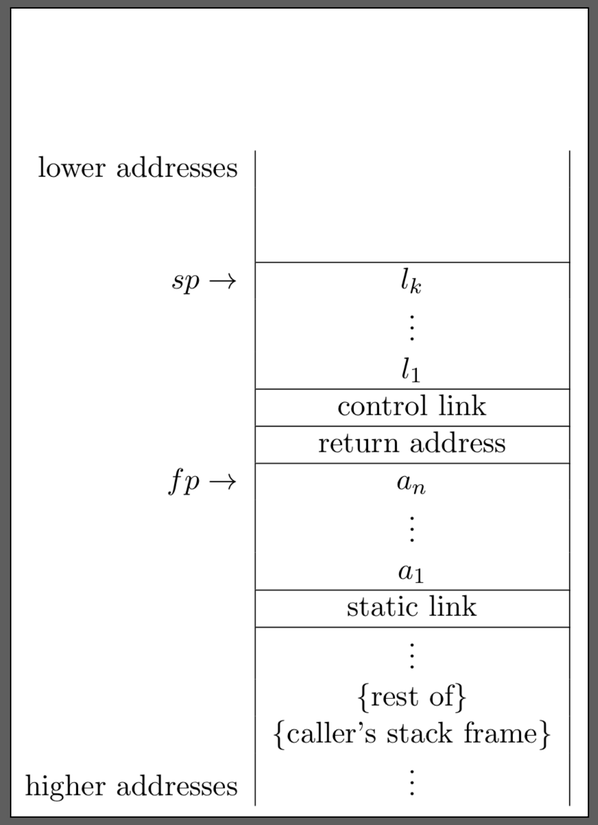
```
def callee(x : int, y : int, z: int) → int:
  a : int = 1
  b : int = 2
  return x + y + z + a + b

def caller():
  d : int = 0
  d = callee(345, 4357, 235)
```




```
.globl $caller

$caller:
  addi    sp, sp, -@$caller.size    # Reserve space for stack frame
  sw      ra, @$caller.size-4(sp)   # Save return address
  sw      fp, @$caller.size-8(sp)   # Save control link (fp)
  addi    fp, sp, @$caller.size     # New fp is at old SP.
  li      a0, 0                     # Load integer constant 0
  sw      a0, -12(fp)               # init local variable $caller.d
  addi    sp, sp, -12               # allocate space for actual arguments
  li      a0, 235                   # Load integer constant 235
  sw      a0, 0(sp)                 # push argument on stack
  li      a0, 4357                  # Load integer constant 4357
  sw      a0, 4(sp)                 # push argument on stack
  li      a0, 345                   # Load integer constant 345
  sw      a0, 8(sp)                 # push argument on stack
  jal     $callee                   # call function $callee
  addi    sp, fp, -@$caller.size    # restore stack pointer
  sw      a0, -12(fp)               # write local variable $caller.d
label_97:
.equiv @$caller.size, 12            # Epilogue of $caller
  lw      ra, -4(fp)                # Restore return address
  lw      fp, -8(fp)                # Restore caller's fp
  jr      ra                        # Return to caller
```

Calling Convention: Callee

```
def callee(x : int, y : int, z: int) → int:
  a : int = 1
  b : int = 2
  return x + y + z + a + b

def caller():
  d : int = 0
  d = callee(345, 4357, 235)
```




```
$callee:
  addi    sp, sp, -@$callee.size    # Reserve space for stack frame
  sw      ra, @$callee.size-4(sp)   # Save return address
  sw      fp, @$callee.size-8(sp)   # Save control link (fp)
  addi    fp, sp, @$callee.size     # New fp is at old SP.
  li      a0, 1                     # Load integer constant 1
  sw      a0, -12(fp)               # init local variable $callee.a
  li      a0, 2                     # Load integer constant 2
  sw      a0, -16(fp)               # init local variable $callee.b
  lw      a0, 8(fp)                 # read formal parameter $callee.x
  lw      t1, 4(fp)                 # read formal parameter $callee.y
  add     a0, a0, t1                # Addition
  lw      t1, 0(fp)                 # read formal parameter $callee.z
  add     a0, a0, t1                # Addition
  lw      t1, -12(fp)               # read local variable $callee.a
  add     a0, a0, t1                # Addition
  lw      t1, -16(fp)               # read local variable $callee.b
  add     a0, a0, t1                # Addition
  j       label_96
label_96:
.equiv @$callee.size, 16            # Epilogue of $callee
  lw      ra, -4(fp)                # Restore return address
  lw      fp, -8(fp)                # Restore caller's fp
  jr      ra                        # Return to caller
```

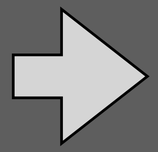
Calling a Function in Function Call Argument

```
def callee(x : int, y : int, z: int) → int:
  a : int = 1
  b : int = 2
  return x + y + z + a + b

def inc(i : int) → int:
  return i + 1

def caller():
  d : int = 0
  d = callee(345 + 81 + inc(13), 4357, 235)
```

problem: callee overwrites registers for temporaries

Calling a Function in Function Call Argument

```
def callee(x : int, y : int, z: int) → int:
  a : int = 1
  b : int = 2
  return x + y + z + a + b

def inc(i : int) → int:
  return i + 1

def caller():
  d : int = 0
  d = callee(345 + 81 + inc(13), 4357, 235)
```



```
def callee(x : int, y : int, z : int) → int:
  a : int = 1
  b : int = 2
  return x + y + z + a + b

def inc(i : int) → int:
  return i + 1

def caller( ) :
  d : int = 0
  temp_2 : int = 0
  temp_2 = inc(13)
  d = callee(345 + 81 + temp_2, 4357, 235)
```

problem: callee overwrites registers for temporaries

solution: lift calls from call expressions
store result in local variable

Calling a Function in Function Call Argument

```
def callee(x : int, y : int, z: int) -> int:
  a : int = 1
  b : int = 2
  return x + y + z + a + b

def inc(i : int) -> int:
  return i + 1

def caller():
  d : int = 0
  d = callee(345 + 81 + inc(13), 4357, 235)
```

problem: callee overwrites registers for temporaries

```
def callee(x : int, y : int, z : int) -> int:
  a : int = 1
  b : int = 2
  return x + y + z + a + b

def inc(i : int) -> int:
  return i + 1

def caller( ) :
  d : int = 0
  temp_2 : int = 0
  temp_2 = inc(13)
  d = callee(345 + 81 + temp_2, 4357, 235)
```

solution: lift calls from call expressions
store result in local variable

```
$caller:
  addi    sp, sp, -@$caller.size    # Reserve space for stack frame
  sw      ra, @$caller.size-4(sp)   # Save return address
  sw      fp, @$caller.size-8(sp)   # Save control link (fp)
  addi    fp, sp, @$caller.size     # New fp is at old SP.
  li      a0, 0                     # Load integer constant 0
  sw      a0, -12(fp)               # init local variable $caller.d
  li      a0, 0                     # Load integer constant 0
  sw      a0, -16(fp)               # init local variable $caller.temp_2
  addi    sp, sp, -4                # allocate space for actual arguments
  li      a0, 13                    # Load integer constant 13
  sw      a0, 0(sp)                 # push argument on stack
  jal     $inc                      # call function $inc
  addi    sp, fp, -@$caller.size    # restore stack pointer
  sw      a0, -16(fp)               # write local variable $caller.temp_2
  addi    sp, sp, -12               # allocate space for actual arguments
  li      a0, 235                   # Load integer constant 235
  sw      a0, 0(sp)                 # push argument on stack
  li      a0, 4357                  # Load integer constant 4357
  sw      a0, 4(sp)                 # push argument on stack
  li      a0, 345                   # Load integer constant 345
  addi    a0, a0, 81                # Add with constant 81
  lw      t1, -16(fp)               # read local variable $caller.temp_2
  add     a0, a0, t1                # Addition
  sw      a0, 8(sp)                 # push argument on stack
  jal     $callee                   # call function $callee
  addi    sp, fp, -@$caller.size    # restore stack pointer
  sw      a0, -12(fp)               # write local variable $caller.d

label_98:
.equiv @$caller.size, 16            # Epilogue of $caller
  lw      ra, -4(fp)                # Restore return address
  lw      fp, -8(fp)                # Use control link to restore caller's fp
  jr      ra                        # Return to caller
```

Shadowing

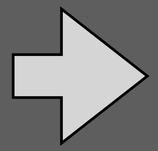
Shadowing

```
a : int = 10

def foo(a: int) → int:
  def foo(b : int) → int:
    a : int = 20
    return a + b
  return foo(a + 10)

print(foo(a))
```

problem: identifier can be used for multiple declarations

Shadowing

```
a : int = 10

def foo(a: int) → int:
  def foo(b : int) → int:
    a : int = 20
    return a + b
  return foo(a + 10)

print(foo(a))
```



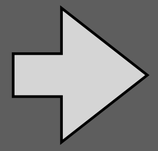
```
$a : int = 10

def $foo($foo.a: int) → int:
  def $foo.foo($foo.foo.b : int) → int:
    $foo.foo.a : int = 20
    return $foo.foo.a + $foo.foo.b
  return $foo($foo.a + 10)

print($foo($a))
```

problem: identifier can be used for multiple declarations

solution: rename identifiers so that declarations have unique names

Shadowing

```
a : int = 10

def foo(a: int) → int:
  def foo(b : int) → int:
    a : int = 20
    return a + b
  return foo(a + 10)

print(foo(a))
```



```
$a : int = 10

def $foo($foo.a: int) → int:
  def $foo.foo($foo.foo.b : int) → int:
    $foo.foo.a : int = 20
    return $foo.foo.a + $foo.foo.b
  return $foo.foo($foo.a + 10)

print($foo($a))
```

problem: identifier can be used for multiple declarations

solution: rename identifiers so that declarations have unique names

implementation: dynamic rule to rename function and variable names (sketch)

```
f2 := $[[<Parent>].[f1]];
rules(
  FunctionName : f1 → f2
)
```

Nested Functions

Closed Nested Functions are Just Functions

global variable

nested function definition

```
x : int = 10

def foo(y : int) → int:
  def bar(z : int) → int:
    return z + 10
  return bar(y + 10)

print(foo(x))
```

reference to local variable

reference to local variable

reference to global variable

Closed Nested Functions are Just Functions

```
x : int = 10

def foo(y : int) → int:
  def bar(z : int) → int:
    return z + 10
  return bar(y + 10)

print(foo(x))
```

nested function name is hidden from context

but otherwise it is a normal function

```
.globl $foo
$foo:
  addi   sp, sp, -@$foo.size       # Reserve space for stack frame
  sw     ra, @$foo.size-4(sp)      # Save return address
  sw     fp, @$foo.size-8(sp)      # Save control link (fp)
  addi   fp, sp, @$foo.size        # New fp is at old SP.
  addi   sp, sp, -4                # allocate space for actual arguments
  lw     a0, 0(fp)                 # read formal parameter $foo.y
  addi   a0, a0, 10                # Add with constant 10
  sw     a0, 0(sp)                 # push argument on stack
  jal    $foo.bar                  # call function $foo.bar
  addi   sp, fp, -@$foo.size       # restore stack pointer
  …
  jr     ra                        # Return to caller

.globl $foo.bar
$foo.bar:
  addi   sp, sp, -@$foo.bar.size   # Reserve space for stack frame
  sw     ra, @$foo.bar.size-4(sp)  # Save return address
  sw     fp, @$foo.bar.size-8(sp)  # Save control link (fp)
  addi   fp, sp, @$foo.bar.size    # New fp is at old SP.
  lw     a0, 0(fp)                 # read formal parameter $foo.bar.z
  addi   a0, a0, 10                # Add with constant 10
  …
  jr     ra                        # Return to caller
```

Nested Functions with ‘Free’ Variables

global variable

nested function definition

```
x : int = 10

def foo(y : int) → int:
  def bar(z : int) → int:
    return y + z
  return bar(y + 10)

print(foo(x))
```

reference to variable in enclosing function

reference to local variable

reference to global variable

Accessing Lexically Enclosing Frame via Static Link

```
x : int = 10

def foo(y : int) → int:
  def bar(z : int) → int:
    return y + z
  return bar(y + 10)

print(foo(x))
```

```
.globl $foo
$foo:
  addi   sp, sp, -@$foo.size  # Reserve space for stack frame
  sw     ra, @$foo.size-4(sp) # Save return address
  sw     fp, @$foo.size-8(sp) # Save control link (fp)
  addi   fp, sp, @$foo.size   # New fp is at old SP.
  addi   sp, sp, -8           # allocate space for actual arguments
  mv     t0, fp               # load static link
  sw     t0, 0(sp)            # pass static link as parameter
  lw     a0, 0(fp)            # read formal parameter $foo.y
  addi   a0, a0, 10           # Add with constant 10
  sw     a0, 4(sp)            # push argument on stack
  jal    $foo.bar             # call function $foo.bar
  addi   sp, fp, -@$foo.size  # restore stack pointer
  j      label_105
label_105:
.equiv @$foo.size, 8          # Epilogue of $foo
  lw     ra, -4(fp)           # Restore return address
  lw     fp, -8(fp)           # Use control link to restore caller's fp
  jr     ra                   # Return to caller
```

```
.globl $foo.bar
$foo.bar:
  addi   sp, sp, -@$foo.bar.size  # Reserve space for stack frame
  sw     ra, @$foo.bar.size-4(sp) # Save return address
  sw     fp, @$foo.bar.size-8(sp) # Save control link (fp)
  addi   fp, sp, @$foo.bar.size   # New fp is at old SP.
  lw     t0, 0(fp)                # load static link 1
  lw     a0, 0(t0)                # read variable $foo.y
  lw     t1, 4(fp)                # read formal parameter $foo.bar.z
  add    a0, a0, t1               # Addition
  j      label_106
label_106:
.equiv @$foo.bar.size, 8          # Epilogue of $foo.bar
  lw     ra, -4(fp)               # Restore return address
  lw     fp, -8(fp)               # Use control link to restore caller's fp
  jr     ra                       # Return to caller
```

Accessing Lexically Enclosing Frame via Static Link

```
x : int = 10

def foo(y : int) → int:
  def bar(z : int) → int:
    return y + z
  return bar(y + 10)

print(foo(x))
```

```
.globl $foo
$foo:
  addi   sp, sp, -@$foo.size  # Reserve space for stack frame
  sw     ra, @$foo.size-4(sp) # Save return address
  sw     fp, @$foo.size-8(sp) # Save control link (fp)
  addi   fp, sp, @$foo.size   # New fp is at old SP.
  addi   sp, sp, -8           # allocate space for actual arguments
  mv     t0, fp               # load static link
  sw     t0, 0(sp)            # pass static link as parameter
  lw     a0, 0(fp)            # read formal parameter $foo.y
  addi   a0, a0, 10           # Add with constant 10
  sw     a0, 4(sp)            # push argument on stack
  jal    $foo.bar             # call function $foo.bar
  addi   sp, fp, -@$foo.size  # restore stack pointer
  …
  jr     ra                   # Return to caller
```

```
.globl $foo.bar
$foo.bar:
  addi   sp, sp, -@$foo.bar.size  # Reserve space for stack frame
  sw     ra, @$foo.bar.size-4(sp) # Save return address
  sw     fp, @$foo.bar.size-8(sp) # Save control link (fp)
  addi   fp, sp, @$foo.bar.size   # New fp is at old SP.
  lw     t0, 0(fp)                # load static link 1
  lw     a0, 0(t0)                # read variable $foo.y
  lw     t1, 4(fp)                # read formal parameter $foo.bar.z
  add    a0, a0, t1               # Addition
  …
  jr     ra                       # Return to caller
```

Offset in Activation Record

```
x : int = 10

def foo(y : int) → int:
  a : int = 0

  def bar(z : int) → int:
    b : int = 0
    b = z
    return a + b + x

  a = y + 1
  return bar(y + 10)

print(foo(x))
```

Offset in Activation Record

```
x : int = 10

def foo(y : int) → int:
  a : int = 0

  def bar(z : int) → int:
    b : int = 0
    b = z
    return a + b + x

  a = y + 1
  return bar(y + 10)

print(foo(x))
```

```
.globl $foo
$foo:
  addi   sp, sp, -@$foo.size  # Reserve space for stack frame
  sw     ra, @$foo.size-4(sp) # Save return address
  sw     fp, @$foo.size-8(sp) # Save control link (fp)
  addi   fp, sp, @$foo.size   # New fp is at old SP.
  li     a0, 0                # Load integer constant 0
  sw     a0, -12(fp)          # init local variable $foo.a
  lw     a0, 0(fp)            # read formal parameter $foo.y
  addi   a0, a0, 1            # Add with constant 1
  sw     a0, -12(fp)          # write local variable $foo.a
  addi   sp, sp, -8           # allocate space for actual arguments
  mv     t0, fp               # load static link
  sw     t0, 0(sp)            # pass static link as parameter
  lw     a0, 0(fp)            # read formal parameter $foo.y
  addi   a0, a0, 10           # Add with constant 10
  sw     a0, 4(sp)            # push argument on stack
  jal    $foo.bar             # call function $foo
  addi   sp, fp, -@$foo.size  # restore stack poin
  …
  jr     ra                   # Return to caller
```

offset from frame pointer

same offset from static link

```
.globl $foo.bar
$foo.bar:
  addi   sp, sp, -@$foo.bar.size  # Reserve space for stack frame
  sw     ra, @$foo.bar.size-4(sp) # Save return address
  sw     fp, @$foo.bar.size-8(sp) # Save control link (fp)
  addi   fp, sp, @$foo.bar.size   # New fp is at old SP.
  li     a0, 0                    # Load integer constant 0
  sw     a0, -12(fp)              # init local variable $foo.bar.b
  lw     a0, 4(fp)                # read formal parameter $foo.bar.z
  sw     a0, -12(fp)              # write local variable $foo.bar.b
  lw     t0, 0(fp)                # load static link 1
  lw     a0, -12(t0)              # read variable $foo.a
  lw     t1, -12(fp)              # read local variable $foo.bar.b
  add    a0, a0, t1               # Addition
  lw     t1, $x                   # read global variable $x
  add    a0, a0, t1               # Addition
  …
  jr     ra                       # Return to caller
```

Recursive Nested Functions

nested function definition

```
def exp(base: int, n: int) → int:
  def aux(x: int) → int:
    if x == 0:
      return 1
    else:
      return base * aux(x - 1)
  return aux(n)

print(exp(2, 4))
```

reference to variable in lexically enclosing function

Recursive Nested Functions

```
def exp(base: int, n: int) → int:
  def aux(x: int) → int:
    if x == 0:
      return 1
    else:
      return base * aux(x - 1)
  return aux(n)

print(exp(2, 4))
```

nested function definition

```
.globl $exp.aux
$exp.aux:
  addi    sp, sp, -@$exp.aux.size  # Reserve space for stack frame
  sw      ra, @$exp.aux.size-4(sp) # Save return address
  sw      fp, @$exp.aux.size-8(sp) # Save control link (fp)
  addi    fp, sp, @$exp.aux.size   # New fp is at old SP.
  li      a0, 0                    # Load integer constant 0
  sw      a0, -12(fp)              # init local variable temp_29
  lw      a0, 4(fp)                # read formal parameter $exp.aux.x
  li      t1, 0                    # Load integer constant 0
  xor     a0, a0, t1               # Test integer equality
  seqz    a0, a0
  beqz    a0, false_3
  li      a0, 1                    # Load integer constant 1
  j       label_110
  j       end_3
false_3:
  addi    sp, sp, -8               # allocate space for actual arguments
  lw      t0, 0(fp)                # load static link 1
  sw      t0, 0(sp)                # pass static link as parameter
  lw      a0, 4(fp)                # read formal parameter $exp.aux.x
  li      t1, 1                    # Load integer constant 1
  sub     a0, a0, t1               # Subtraction
  sw      a0, 4(sp)                # push argument on stack
  jal     $exp.aux                 # call function $exp.aux
  addi    sp, fp, -@$exp.aux.size  # restore stack pointer
  sw      a0, -12(fp)              # write local variable temp_29
  lw      t0, 0(fp)                # load static link 1
  lw      a0, 4(t0)                # read variable $exp.base
  lw      t1, -12(fp)              # read local variable temp_29
  mul     a0, a0, t1
  …
  jr      ra                       # Return to caller
```

Nested Functions: Calling Up

```
def f(a: int) → int:
  z : int = 17
  def g(b: int) → int:
    def h(c: int) → int:
      def i(d: int) → int:
        print(d)
        if d == 1:
          return g(d - 1)
        else:
         return d
      print(c)
      return i(c - 1)
    print(b)
    if b == 0:
      return z
    else:
      return h(b - 1)
  print(a)
  return g(a - 1)

print(f(4))
```

Nested Functions: Calling Up

```
def f(a: int) → int:
  z : int = 17
  def g(b: int) → int:
    def h(c: int) → int:
      def i(d: int) → int:
        print(d)
        if d == 1:
          return g(d - 1)
        else:
         return d
      print(c)
      return i(c - 1)
    print(b)
    if b == 0:
      return z
    else:
      return h(b - 1)
  print(a)
  return g(a - 1)

print(f(4))
```

```
.globl $f.g
$f.g:
  addi   sp, sp, -@$f.g.size  # Reserve space for stack f
  sw     ra, @$f.g.size-4(sp) # Save return address
  sw     fp, @$f.g.size-8(sp) # Save control link (fp)
  addi   fp, sp, @$f.g.size   # New fp is at old SP.
  addi   sp, sp, -4           # allocate space for actual
  lw     a0, 4(fp)            # read formal parameter $f.
  sw     a0, 0(sp)            # push argument on stack
  jal    $printInt            # call function $printInt
  addi   sp, fp, -@$f.g.size  # restore stack pointer
  lw     a0, 4(fp)            # read formal parameter $f.
  li     t1, 0                # Load integer constant 0
  xor    a0, a0, t1           # Test integer equality
  seqz   a0, a0
  beqz   a0, false_28
  lw     t0, 0(fp)            # load static link 1
  lw     a0, -12(t0)          # read variable $f.z
  j      label_153
  j      end_28
false_28:
  addi   sp, sp, -8           # allocate space for actual
  mv     t0, fp               # load static link
  sw     t0, 0(sp)            # pass static link as param
  lw     a0, 4(fp)            # read formal parameter $f.
  li     t1, 1                # Load integer constant 1
  sub    a0, a0, t1           # Subtraction
  sw     a0, 4(sp)            # push argument on stack
  jal    $f.g.h               # call function $f.g.h
  addi   sp, fp, -@$f.g.size  # restore stack pointer
  …
  jr     ra                   # Return to caller
```

```
.globl $f.g.h.i
$f.g.h.i:
  addi   sp, sp, -@$f.g.h.i.size  # Reserve space for stack frame
  sw     ra, @$f.g.h.i.size-4(sp) # Save return address
  sw     fp, @$f.g.h.i.size-8(sp) # Save control link (fp)
  addi   fp, sp, @$f.g.h.i.size   # New fp is at old SP.
  addi   sp, sp, -4               # allocate space for actual argu
  lw     a0, 4(fp)                # read formal parameter $f.g.h.i
  sw     a0, 0(sp)                # push argument on stack
  jal    $printInt                # call function $printInt
  addi   sp, fp, -@$f.g.h.i.size  # restore stack pointer
  lw     a0, 4(fp)                # read formal parameter $f.g.h.i
  li     t1, 1                    # Load integer constant 1
  xor    a0, a0, t1               # Test integer equality
  seqz   a0, a0
  beqz   a0, false_29
  addi   sp, sp, -8               # allocate space for actual argu
  lw     t0, 0(fp)                # load static link 1
  lw     t0, 0(t0)                # load static link 2
  lw     t0, 0(t0)                # load static link 3
  sw     t0, 0(sp)                # pass static link as parameter
  lw     a0, 4(fp)                # read formal parameter $f.g.h.i
  li     t1, 1                    # Load integer constant 1
  sub    a0, a0, t1               # Subtraction
  sw     a0, 4(sp)                # push argument on stack
  jal    $f.g                     # call function $f.g
  addi   sp, fp, -@$f.g.h.i.size  # restore stack pointer
  j      label_155
  j      end_29
false_29:
  lw     a0, 4(fp)                # read formal parameter $f.g.h.i
  …
  jr     ra                       # Return to caller
```

identify call frame of function g

Nested Functions: Mutual Recursion

```
def pred(x: int) → bool:
  true : bool = True
  false : bool = False

  def even(a : int) → bool:
    if a == 0:
      return true
    else:
      return odd(a - 1)

  def odd(b : int) → bool:
    if b == 0:
      return false
    else:
      return even(b - 1)

  return even(x)

print(pred(2))
```

what is the static link?

what is the static link?

Making Nesting Explicit

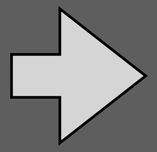
Nesting: How Many Frames Up?

```
a : int = 10

def foo(x : int) → int:
  b : int = 0

  def aux(i : int) → int:
    return b + i

  def bar(y : int) → int:
    c : int = 0

    def baz(z : int) → int:
      d : int = 0
      d = aux(c + 1)
      return a + x + y + z

    return baz(a + b + x)

  b = aux(x)
  return bar(b + 10)

print(foo(a))
```

how many static links should we follow to find a variable or (static link of) a function?

Nesting: How Many Frames Up?

```
a : int = 10

def foo(x : int) → int:
  b : int = 0

  def aux(i : int) → int:
    return b + i

  def bar(y : int) → int:
    c : int = 0

    def baz(z : int) → int:
      d : int = 0
      d = aux(c + 1)
      return a + x + y + z

    return baz(a + b + x)

  b = aux(x)
  return bar(b + 10)

print(foo(a))
```



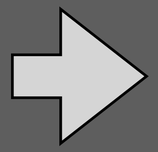
```
a : int = 10

def foo(x : int) → int:
  b : int = 0

  def aux(i : int) → int:
    return b/1 + i/0

  def bar(y : int) → int:
    c : int = 0

    def baz(z : int) → int:
      d : int = 0
      d = aux/2(c/1 + 1)
      return a/0 + x/2 + y/1 + z/0

    return baz/0(a/0 + b/1 + x/1)

  b = aux/0(x/0)
  return bar/0(b/0 + 10)

print(foo/0(a/0))
```

how many static links should we follow to find a variable or (static link of) a function?

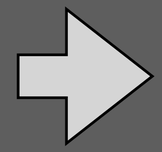
difference between nesting level of occurrence and nesting level of definition

Nesting: How Many Frames Up?

```
a : int = 10

def foo(x : int) → int:
  b : int = 0

  def aux(i : int) → int:
    return b + i

  def bar(y : int) → int:
    c : int = 0

    def baz(z : int) → int:
      d : int = 0
      d = aux(c + 1)
      return a + x + y + z

    return baz(a + b + x)

  b = aux(x)
  return bar(b + 10)

print(foo(a))
```



```
a : int = 10

def foo(x : int) → int:
  b : int = 0

  def aux(i : int) → int:
    return b/1 + i/0

  def bar(y : int) → int:
    c : int = 0

    def baz(z : int) → int:
      d : int = 0
      d = aux/2(c/1 + 1)
      return a/0 + x/2 + y/1 + z/0

    return baz/0(a/0 + b/1 + x/1)

  b = aux/0(x/0)
  return bar/0(b/0 + 10)

print(foo/0(a/0))
```



```
Return(
  AddInt(
    AddInt(
      AddInt(
        Var(("$a", 0))
      , Var(("$foo.x", 2))
      )
    , Var(("$foo.bar.y", 1))
    )
  , Var(("$foo.bar.baz.z", 0))
  )
)
```

how many static links should we follow to find a variable or (static link of) a function?

difference between nesting level of occurrence and nesting level of definition

transformation pairs levels with variables

Functions as First-Class Citizens

Challenge: Closures

Static link only works with nested functions

- the environment is still on the stack

Functions as first-class citizens

- `map((x: int) ⇒ x + 1, [1, 2, 3])`
- anonymous functions (lambdas)

Function values

- function value may escape the call frame in which it is created
- formal parameters + function body + values of free variables
- encoding in OO languages as objects with apply function

Challenge

- Extend ChocoPy with first-class functions