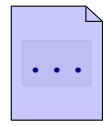
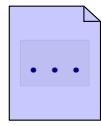


# Part I

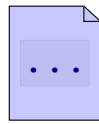
# From Shplait to Machine Code



# From Shplait to Machine Code

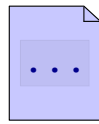


# From Shplait to Machine Code



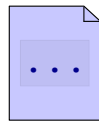
- Everything must be a number

## From Shplait to Machine Code



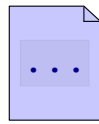
- Everything must be a number
- No **type** or **match**

## From Shplait to Machine Code



- Everything must be a number
- No **type** or **match**
- No implicit continuations

## From Shplait to Machine Code



- Everything must be a number
- No **type** or **match**
- No implicit continuations
- No implicit allocation

## Part 2



## Variable Names at Run Time

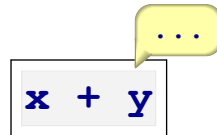
```
let x = 1:  
  let y = 2:  
    x + y
```

## Identifier Address

Suppose that

```
let x = 88:  
  x + y
```

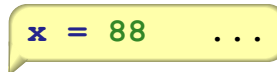
appears in a program; the body is eventually evaluated:



x + y

where will **x** be in the environment?

**Answer:** always at the beginning:



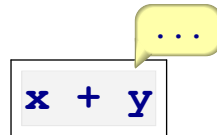
x = 88 ...

## Identifier Address

Suppose that

```
let y = 1:  
  x + y
```

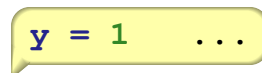
appears in a program; the body is eventually evaluated:



x + y

where will **y** be in the environment?

**Answer:** always at the beginning:



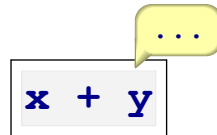
y = 1 ...

## Identifier Address

Suppose that

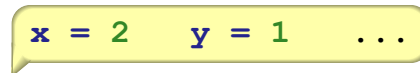
```
let y = 1:  
  let x = 2:  
    x + y
```

appears in a program; the body is eventually evaluated:

  
x + y

where will **y** be in the environment?

**Answer:** always second:

  
x = 2 y = 1 ...

## Identifier Address

Suppose that

```
let y = 1:  
  let x = 88:  
    (x + y) * 17
```

appears in a program; the body is eventually evaluated:

...

x + y

where will **x** and **y** be in the environment?

**Answer:** always first and second:

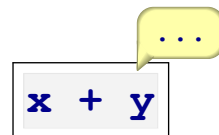
x = 88    y = 1    ...

## Identifier Address

Suppose that

```
let y = 1:  
  let w = 10:  
    let z = 9:  
      let x = 0:  
        x + y
```

appears in a program; the body is eventually evaluated:

A rectangular box containing the text `x + y`. Above the box is a yellow speech bubble containing three dots `...`.

where will **x** and **y** be in the environment?

**Answer:** always first and fourth:

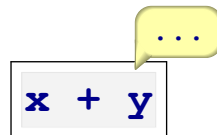
`x = 0   z = 9   w = 10   y = 1   ...`

## Identifier Address

Suppose that

```
let y = (let r = 9: r * 8):  
  let w = 10:  
    let z = (let q = 9: q):  
      let x = 0:  
        x + y
```

appears in a program; the body is eventually evaluated:

  
x + y

where will **x** and **y** be in the environment?

**Answer:** always first and fourth:

x = 0   z = 9   w = 10   y = 1   ...

## Lexical Scope

- For any expression, we can tell which identifiers will be in the environment at run time
- The order of the environment is predictable



## Part 3

## Compilation of Variables

A compiler can transform an **Exp** expression to an expression without identifiers — only lexical addresses

```
compile :: (Exp, ...) -> ExpD
```

```
type Exp
| intE(n :: Int)
| addE(l :: Exp,
      r :: Exp)
| multE(l :: Exp,
        r :: Exp)
| idE(n :: Symbol)
| funE(n :: Symbol,
      body :: Exp)
| appE(fn :: Exp,
      arg :: Exp)

type ExpD
| intD(n :: Int)
| addD(l :: ExpD,
      r :: ExpD)
| multD(l :: ExpD,
        r :: ExpD)
| atD(pos :: Int)
| funD(body :: ExpD)
| appD(fn :: ExpD,
      arg :: ExpD)
```

## Compile Examples

`compile(1, ...)` ⇒ `1`

`compile(1 + 2, ...)` ⇒ `1 + 2`

`compile(x, ...)` ⇒ *compile: free identifier*

`compile(fun (x): 1 + x, ...)`  
⇒ `fun: 1 + at(0)`

`compile(fun (y): fun (x): x + y, ...)`  
⇒ `fun: fun: at(0) + at(1)`

## Implementing the Compiler

```
fun compile(a :: Exp, env :: EnvC) :
  match a
  | intE(n) : intD(n)
  | plusE(l, r) : plusD(compile(l, env),
                        compile(r, env))
  | multE(l, r) : multD(compile(l, env),
                        compile(r, env))
  | idE(n) : atD(locate(n, env))
  | funE(n, body_expr) :
      funD(compile(body_expr,
                  extend_env(bindC(n),
                              env)))
  | appE(fun_expr, arg_expr) :
      appD(compile(fun_expr, env),
           compile(arg_expr, env))
```

## Compile-Time Environment

Mimics the run-time environment, but without values:

```
type BindingC
| bindC(name :: Symbol)

type EnvC = Listof(BindingC)

fun locate(name, env):
  match env
  | []: error('#'locate, "free variable")
  | cons(b, rst_env): cond
      | name == bindC.name(b):
          0
      | ~else:
          1 + locate(name, rst_env)
```

## interp for Compiled

Almost the same as `interp` for `Exp`:

```
fun interp(a :: ExpD, env :: Listof(Value)) :
  match a
  | intD(n) : intV(n)
  | plusD(l, r) : num_plus(interp(l, env),
                           interp(r, env))
  | multD(l, r) : num_mult(interp(l, env),
                           interp(r, env))
  | atD(pos) : list_get(env, pos)
  | funD(body_expr) :
    closV(body_expr, env)
  | appD(fun_expr, arg_expr) :
    def fun_val = interp(fun_expr, env)
    def arg_val = interp(arg_expr, env)
    interp(closV.body(fun_val),
          cons(arg_val,
              closV.env(fun_val)))
```

## Timing Effect of Compilation

Given

```
def c = (fun (x) :  
  fun (y) :  
    fun (z) : x + x + x + x) (1) (2) (3)  
def d = compile(c, mt_env)
```

then

```
interp(d, [])
```

is significantly faster than

```
interp(c, mt_env)
```

Using the built-in `list_get` simulates machine array indexing, but don't take timings too seriously

## Part 4



## From Shplait to Machine Code

Step 1:

**Exp** → **ExpD**

```
fun (x) :  
  1 + x
```

```
fun :  
  1 + at(0)
```

Eliminates all run-time names

## From Shplait to Machine Code

Step 2:

`interp` → `interp` + `continue`

Eliminates implicit continuations

## From Shplait to Machine Code

Step 3:

function calls → registers and `goto`

## From Shplait to Machine Code

Step 3:

function calls → registers and goto

```
interp(l,                               exp_reg := l
      env,                               k_reg := plusSecondK(r,
      plusSecondK(r,                     env_reg,
                    env,                 k_reg)
                    k))                 interp()
```

Makes argument passing explicit

## Part 5

## From Shplait to Machine Code

Step 4:

```
match exp_reg                                → match ref(exp_reg, 0)
| .....                                     | .....
| plusD(1, r):                               | 10:
    .....                                   .....
    plusSecondK(r,                           malloc3(1,
        env_reg,                               ref(exp_reg, 2),
        k_reg)                                env_reg,
                                              k_reg)
```

# Registers and Memory

Registers:      **expr\_reg**                      **k\_reg**                      **v\_reg**  
                                                               

Memory:

|  |  |  |  |  |  |  |  |
|--|--|--|--|--|--|--|--|
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |

# Registers and Memory

Registers:      **expr\_reg**                      **k\_reg**                      **v\_reg**  
                    10                      28                      6

Memory:

|   |    |    |   |   |   |    |   |
|---|----|----|---|---|---|----|---|
|   |    |    |   |   |   | 15 | 1 |
|   |    | 8  | 5 |   |   |    |   |
| 9 | 10 | 32 |   |   |   |    |   |
|   |    |    |   | 0 | 0 |    |   |
| 8 | 2  |    |   |   |   |    |   |
|   |    |    |   |   |   |    |   |



# Registers and Memory

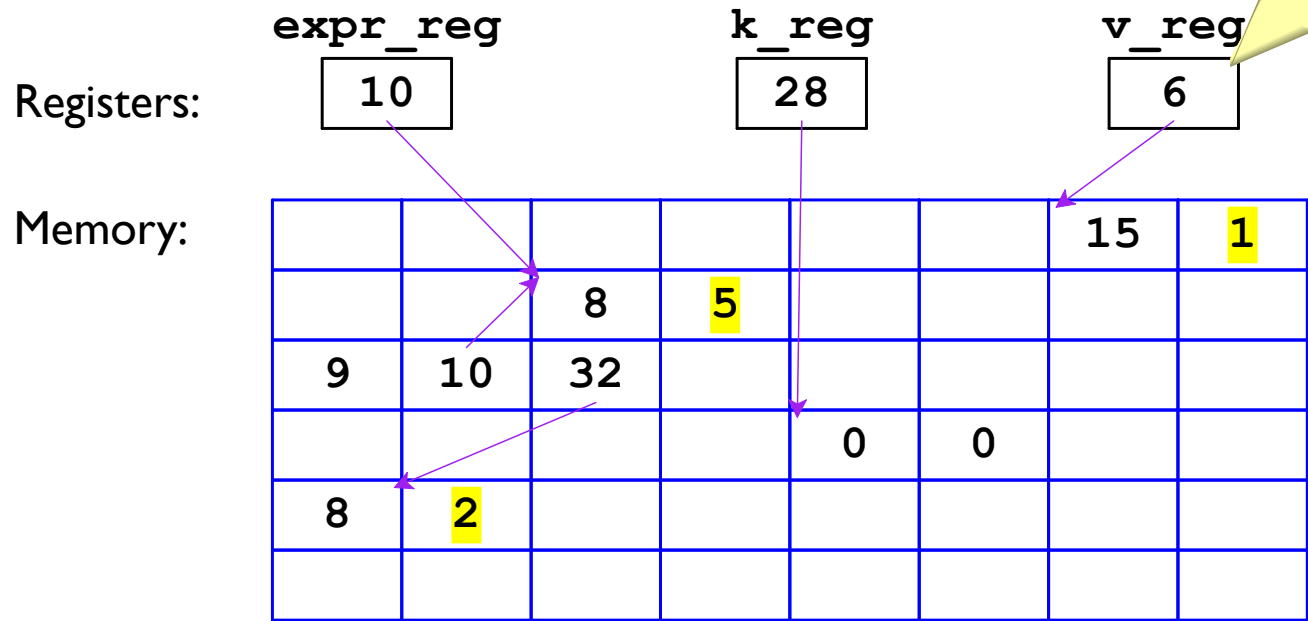
Registers:      **expr\_reg**                      **k\_reg**                      **v\_reg**  
                    10                      28                      6

Memory:

|   |    |    |   |   |   |    |   |
|---|----|----|---|---|---|----|---|
|   |    |    |   |   |   | 15 | 1 |
|   |    | 8  | 5 |   |   |    |   |
| 9 | 10 | 32 |   |   |   |    |   |
|   |    |    |   | 0 | 0 |    |   |
| 8 | 2  |    |   |   |   |    |   |
|   |    |    |   |   |   |    |   |

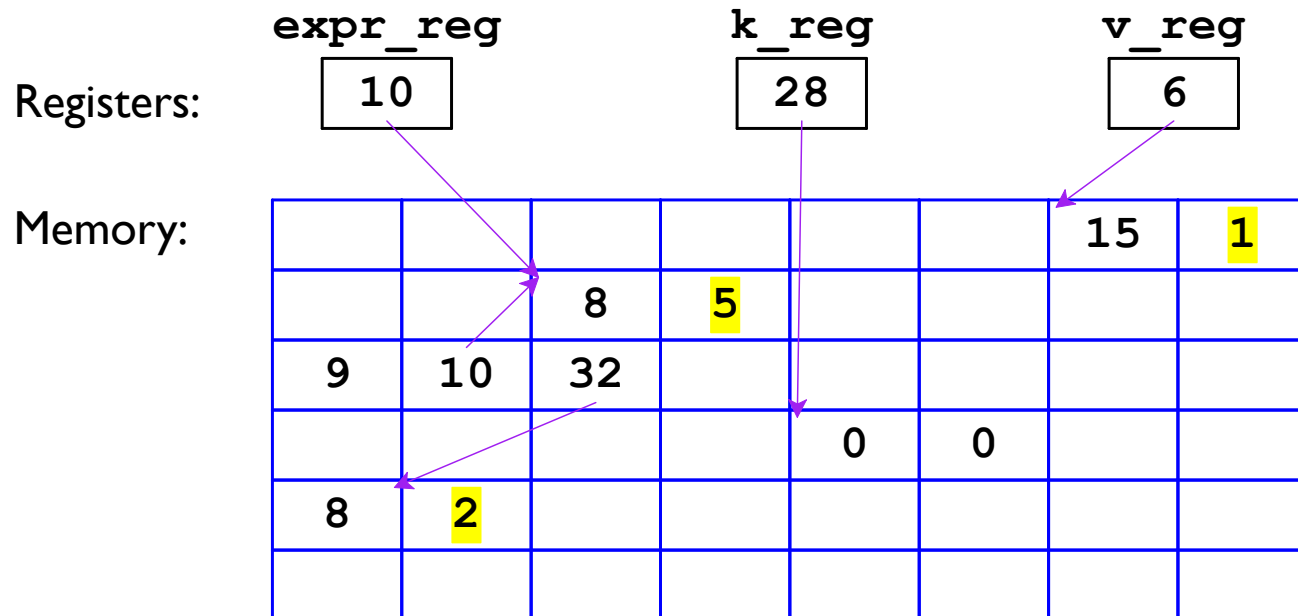
Some numbers are literal values

# Registers and Memory



Some numbers are addresses

# Registers and Memory



`Int` → literal

`ExprD` → address

`Env` → address

`Value` → address

`Cont` → address

# Registers and Memory

Registers:

`expr_reg`

10

`k_reg`

28

`v_reg`

6

Memory:

|   |    |    |   |   |   |    |   |
|---|----|----|---|---|---|----|---|
|   |    |    |   |   |   | 15 | 1 |
|   |    | 8  | 5 |   |   |    |   |
| 9 | 10 | 32 |   |   |   |    |   |
|   |    |    |   | 0 | 0 |    |   |
| 8 | 2  |    |   |   |   |    |   |
|   |    |    |   |   |   |    |   |

Interpreter registers always have addresses

`Int` → literal

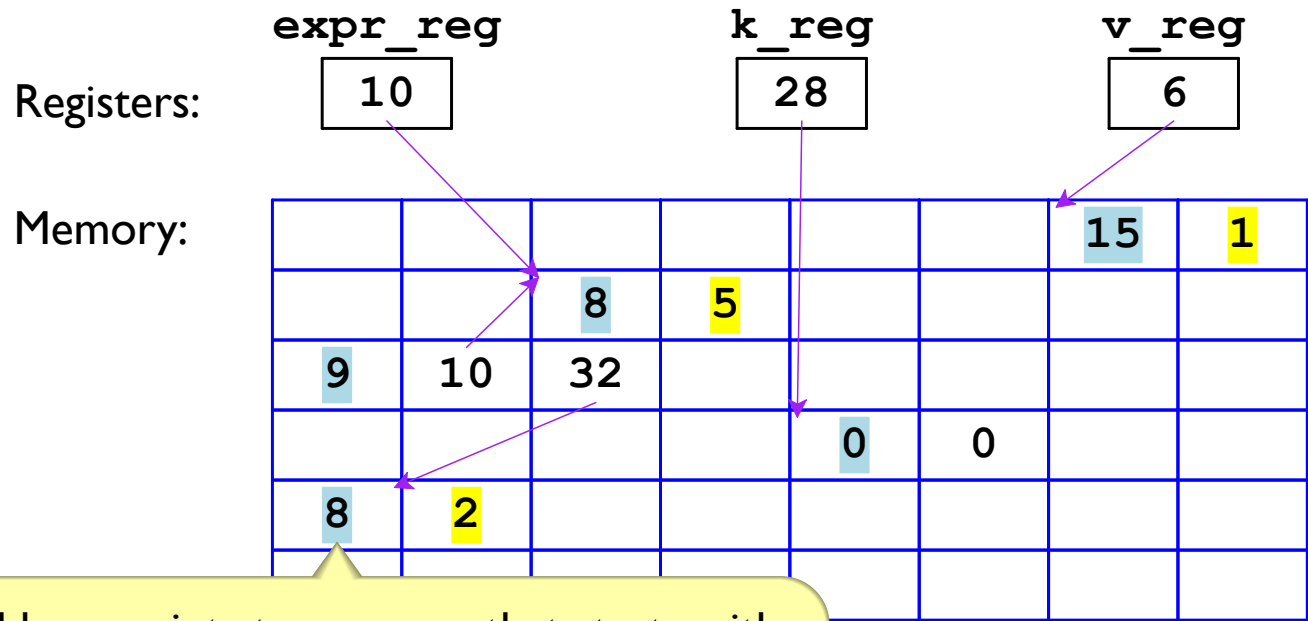
`ExprD` → address

`Env` → address

`Value` → address

`Cont` → address

# Registers and Memory



**Int** → literal

**ExprD** → address

**Env** → address

**Value** → address

**Cont** → address

An address points to memory that starts with a tag to indicate the variant allocated there

## From Shplait to Machine Code

Step 4:

```
match exp_reg                                → match ref(exp_reg, 0)
| .....                                     | .....
| plusD(1, r):                               | 10:
    .....                                   .....
    plusSecondK(r,                          malloc3(1,
        env_reg,                             ref(exp_reg, 2),
        k_reg)                               env_reg,
                                             k_reg)
```

## From Shplait to Machine Code

Step 4:

|                          |                |                 |
|--------------------------|----------------|-----------------|
| <code>doneK</code>       | <code>→</code> | <code>0</code>  |
| <code>plusSecondK</code> | <code>→</code> | <code>1</code>  |
| <code>...</code>         |                |                 |
| <code>intD</code>        | <code>→</code> | <code>8</code>  |
| <code>plusD</code>       | <code>→</code> | <code>9</code>  |
| <code>...</code>         |                |                 |
| <code>intV</code>        | <code>→</code> | <code>15</code> |
| <code>closV</code>       | <code>→</code> | <code>16</code> |

## From Shplait to Machine Code

Step 4:

```
match k_reg                                → match ref(k_reg, 0)
| ....                                     | ....
| multSecondK(r, env, k):                 | 3:
    .... r                                 .... ref(k_reg, 1)
    .... env                               .... ref(k_reg, 2)
    .... k .....                           .... ref(k_reg, 3) .....
| ....                                     | ....
```



## From Shplait to Machine Code

Step 4:

```
def memory = make_array(1500, 0)
def ptr_reg = 0

fun malloc3(tag, a, b, c) :
  memory[ptr_reg] := tag
  memory[ptr_reg + 1] := a
  memory[ptr_reg + 2] := b
  memory[ptr_reg + 3] := c
  ptr_reg := ptr_reg + 4
  ptr_reg - 4
```

Makes all allocation explicit

Makes everything a number