plangCompiler

Chris Lee
Luke Metz

Functional to LLVM

TO SUPERFAST BINARIES

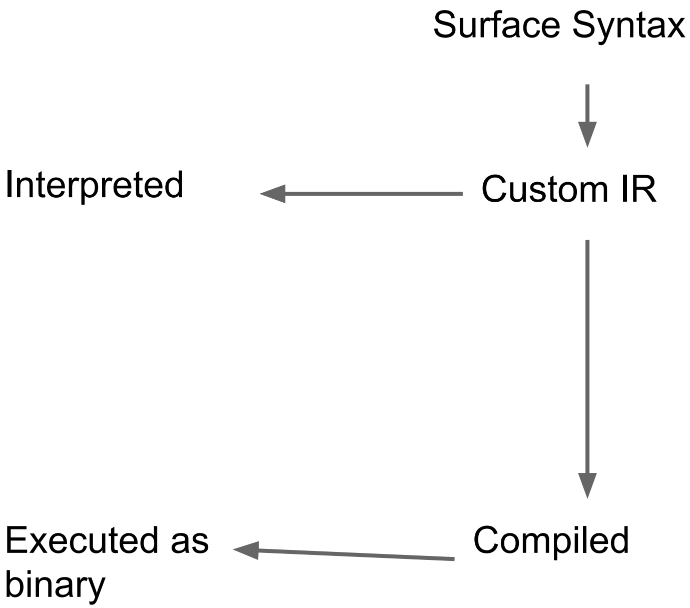
with

Standard ML New Jersey

What is LLVM?

- Tools developed at University of Illinois in 2000
- Term used to describe a number of projects related to compiling and low level code
- Open source
- Used a lot!
    - Apple

Before



Surface Syntax

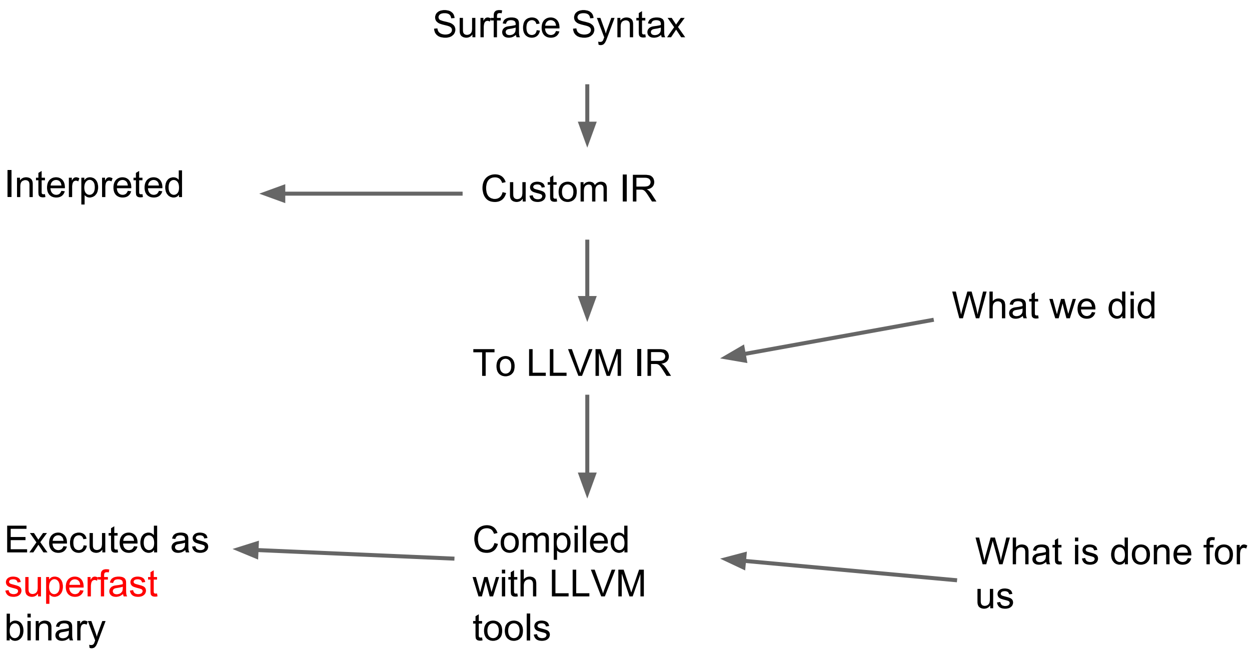
Custom IR

Interpreted

Compiled

Executed as binary

After



Surface Syntax

Interpreted

Custom IR

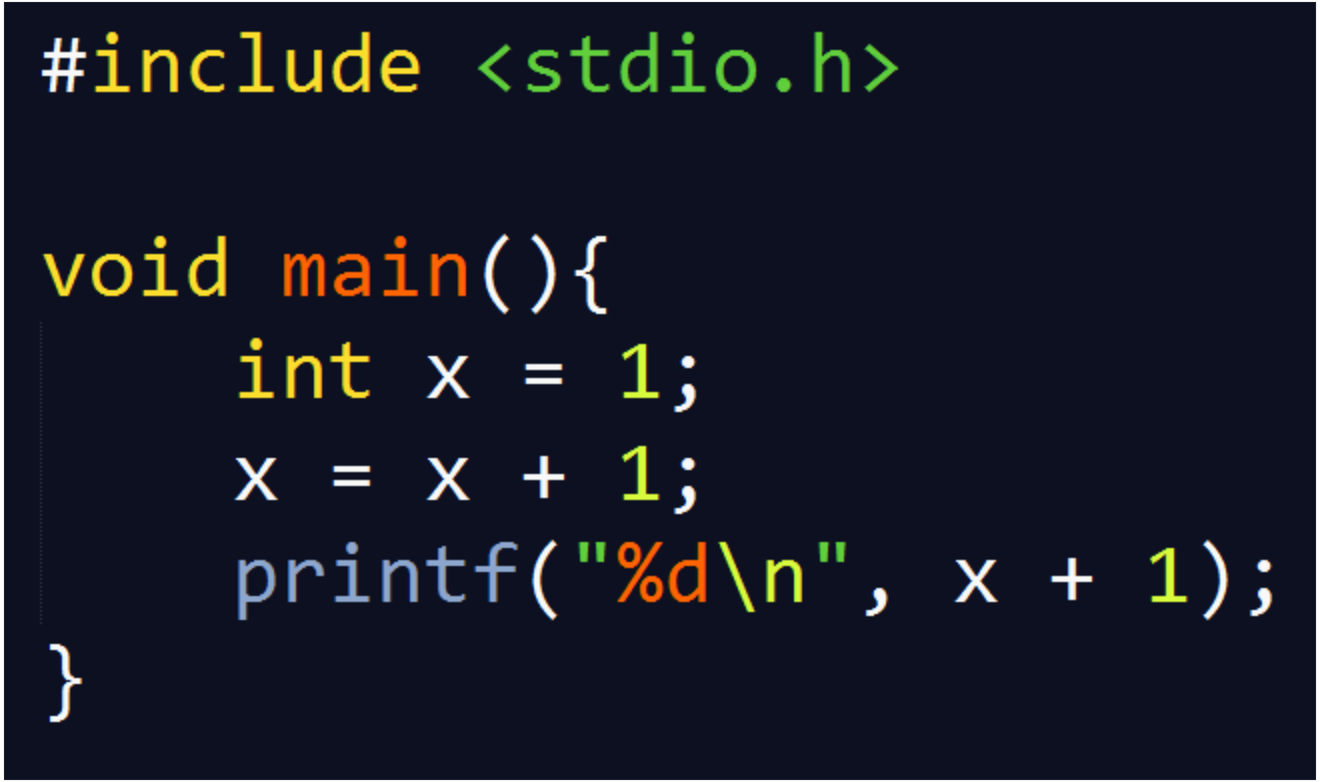
What we did

To LLVM IR

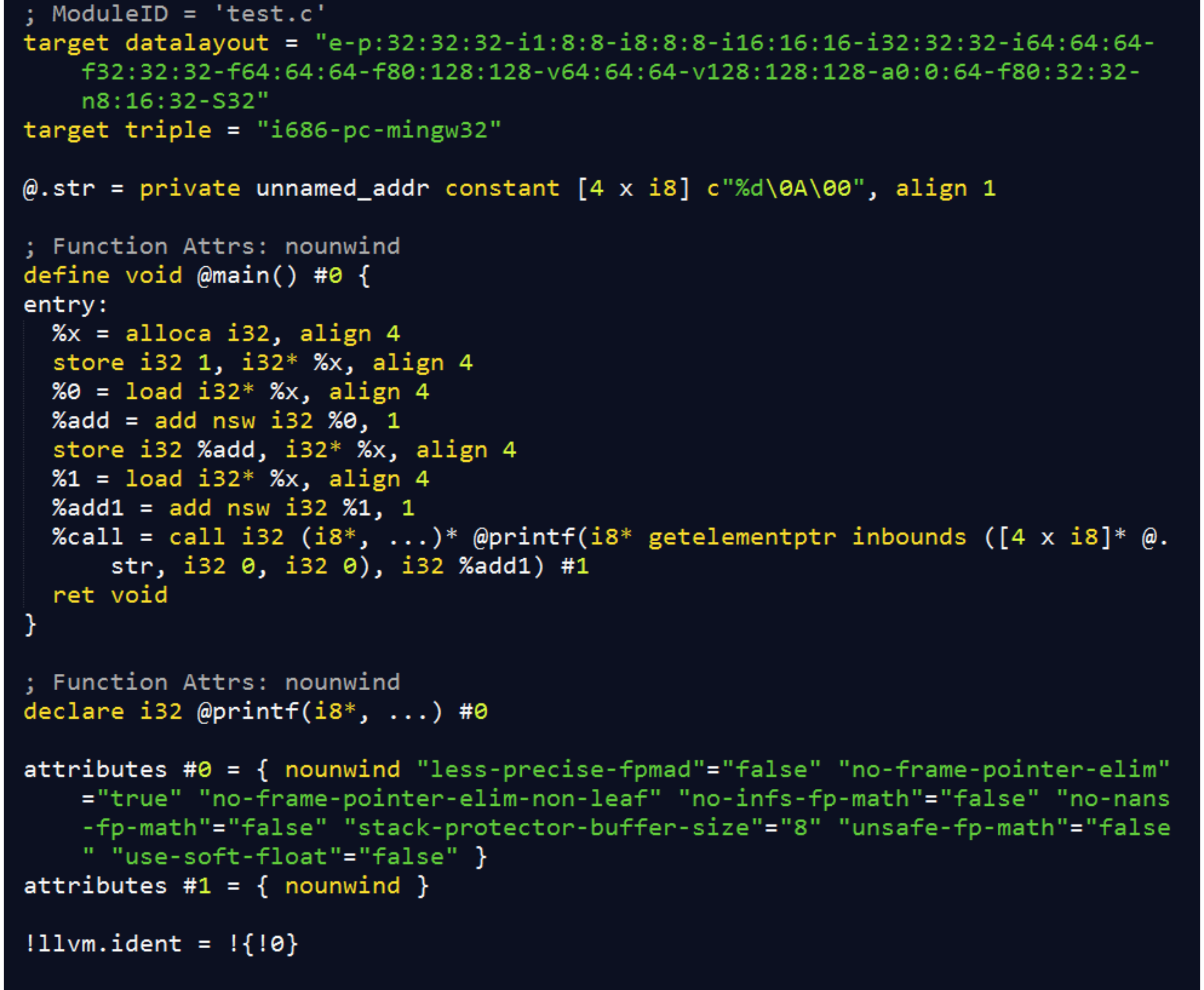
Executed as superfast binary

Compiled with LLVM tools

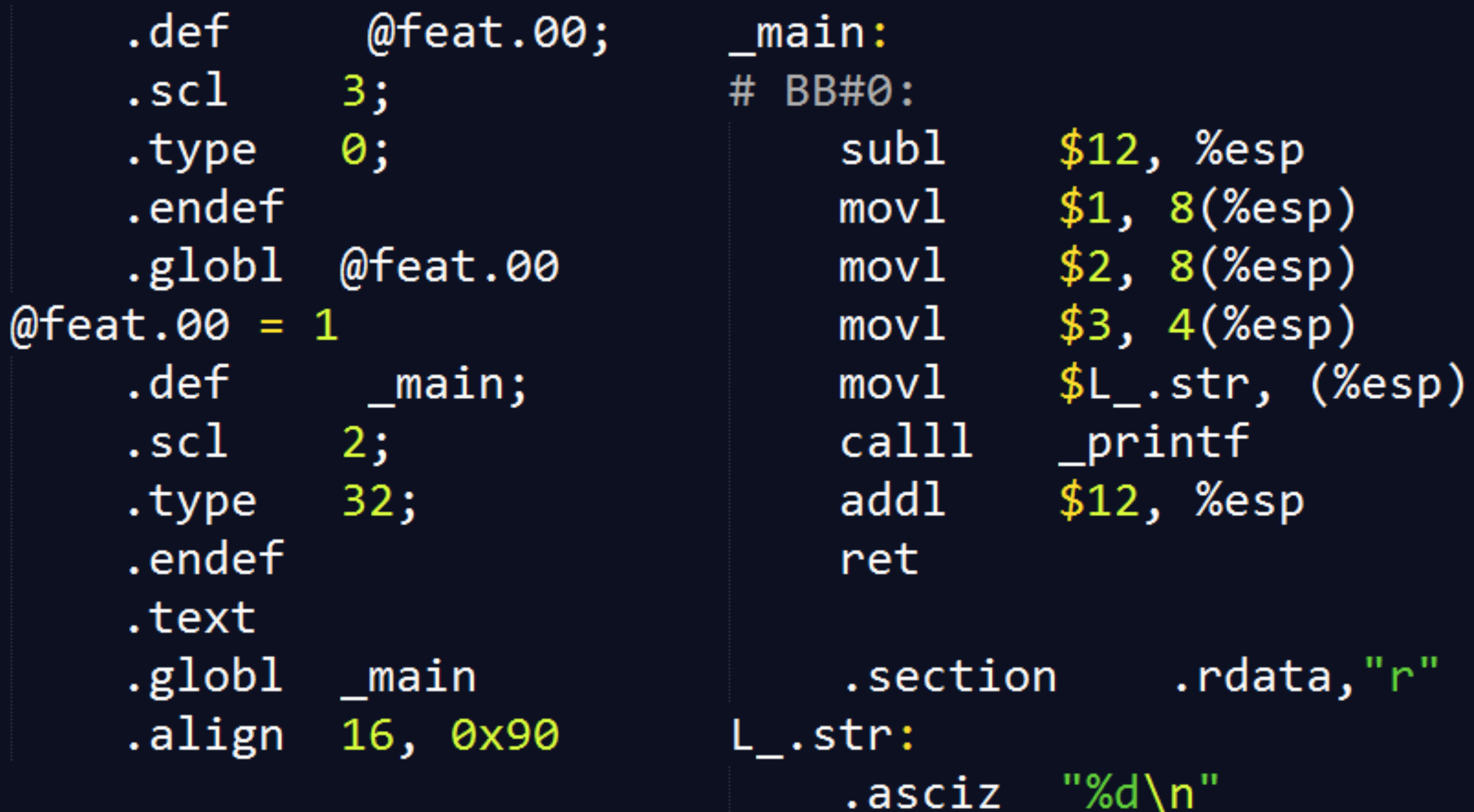
What is done for us



```
#include <stdio.h>

void main(){
    int x = 1;
    x = x + 1;
    printf("%d\n", x + 1);
}
```



```
; ModuleID = 'test.c'
target datalayout = "e-p:32:32:32-i1:8:8-i8:8:8-i16:16:16-i32:32:32-i64:64:64-
    f32:32:32-f64:64:64-f80:128:128-v64:64:64-v128:128:128-a0:0:64-f80:32:32-
    n8:16:32-S32"
target triple = "i686-pc-mingw32"

@.str = private unnamed_addr constant [4 x i8] c"%d\0A\00", align 1

; Function Attrs: nounwind
define void @main() #0 {
entry:
  %x = alloca i32, align 4
  store i32 1, i32* %x, align 4
  %0 = load i32* %x, align 4
  %add = add nsw i32 %0, 1
  store i32 %add, i32* %x, align 4
  %1 = load i32* %x, align 4
  %add1 = add nsw i32 %1, 1
  %call = call i32 (i8*, ...)* @printf(i8* getelementptr inbounds ([4 x i8]* @.
      str, i32 0, i32 0), i32 %add1) #1
  ret void
}

; Function Attrs: nounwind
declare i32 @printf(i8*, ...) #0

attributes #0 = { nounwind "less-precise-fpmad"="false" "no-frame-pointer-elim"
    ="true" "no-frame-pointer-elim-non-leaf" "no-infs-fp-math"="false" "no-nans
    -fp-math"="false" "stack-protector-buffer-size"="8" "unsafe-fp-math"="false
    " "use-soft-float"="false" }
attributes #1 = { nounwind }

!llvm.ident = !{!0}
```



```
	.def	 @feat.00;
	.scl	3;
	.type	0;
	.endef
	.globl	@feat.00
@feat.00 = 1
	.def	 _main;
	.scl	2;
	.type	32;
	.endef
	.text
	.globl	_main
	.align	16, 0x90
```

```
_main:
# BB#0:
	subl	$12, %esp
	movl	$1, 8(%esp)
	movl	$2, 8(%esp)
	movl	$3, 4(%esp)
	movl	$L_.str, (%esp)
	calll	_printf
	addl	$12, %esp
	ret

	.section	.rdata,"r"
L_.str:
	.asciz	"%d\n"
```

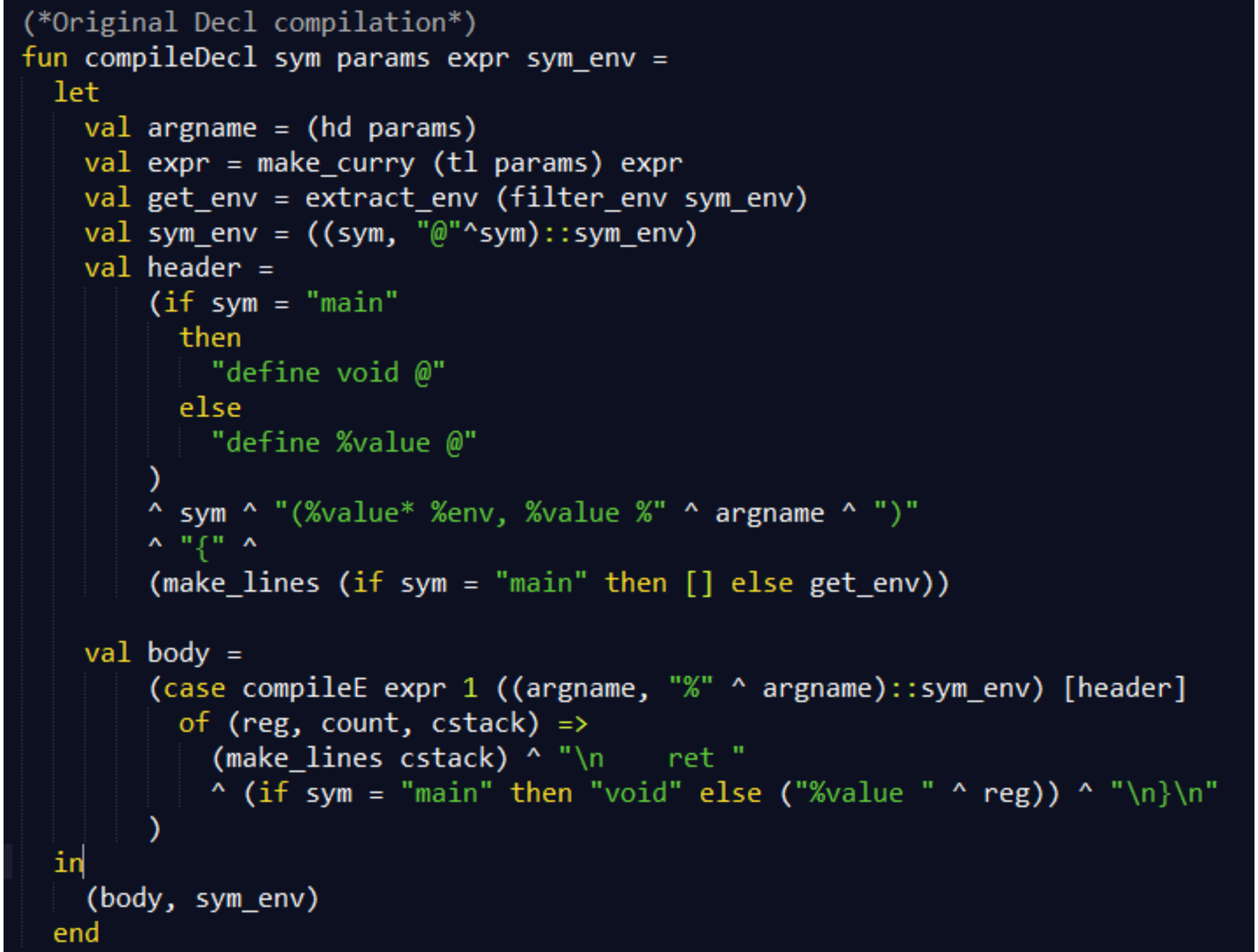
Now on to what we are doing

- Compiling to LLVM
- Dynamic typing
- First class functions

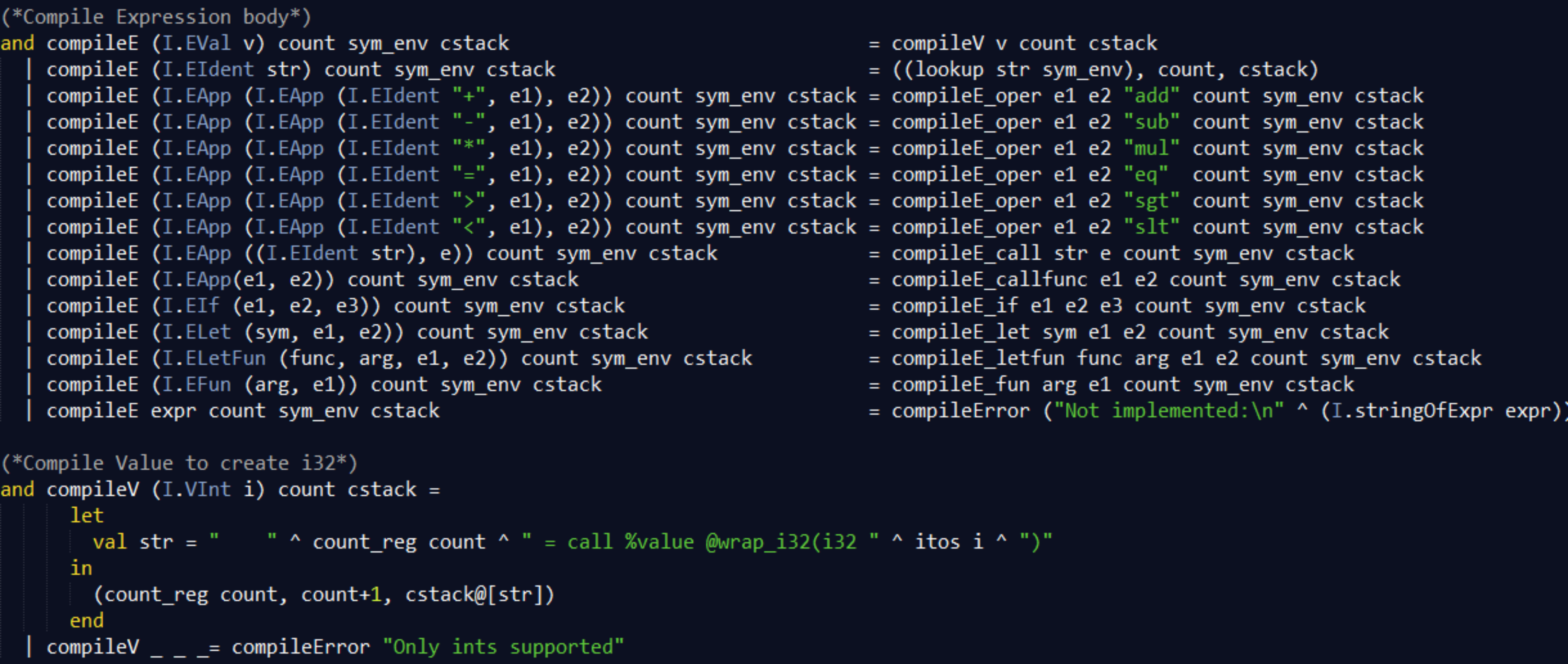
Grammar

3 main pieces to our language to compile

- Declarations,
  - def x = x+1
- Expressions
  - x+1
- Values
  - 1

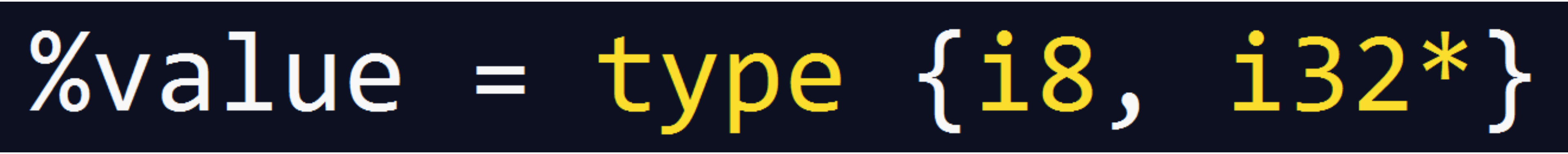


```
(*Original Decl compilation*)
fun compileDecl sym params expr sym_env =
  let
    val argname = (hd params)
    val expr = make_curry (tl params) expr
    val get_env = extract_env (filter_env sym_env)
    val sym_env = ((sym, "@"^sym)::sym_env)
    val header =
        (if sym = "main"
          then
            "define void @"
          else
            "define %value @"
        )
        ^ sym ^ "(%value* %env, %value %" ^ argname ^ ")"
        ^ "{" ^
        (make_lines (if sym = "main" then [] else get_env))

    val body =
        (case compileE expr 1 ((argname, "%" ^ argname)::sym_env) [header]
          of (reg, count, cstack) =>
            (make_lines cstack) ^ "\n    ret "
            ^ (if sym = "main" then "void" else ("%value " ^ reg)) ^ "\n}\n"
        )
  in
    (body, sym_env)
  end
```



```
(*Compile Expression body*)
and compileE (I.EVal v) count sym_env cstack                               = compileV v count cstack
  | compileE (I.EIdent str) count sym_env cstack                           = ((lookup str sym_env), count, cstack)
  | compileE (I.EApp (I.EApp (I.EIdent "+", e1), e2)) count sym_env cstack = compileE_oper e1 e2 "add" count sym_env cstack
  | compileE (I.EApp (I.EApp (I.EIdent "-", e1), e2)) count sym_env cstack = compileE_oper e1 e2 "sub" count sym_env cstack
  | compileE (I.EApp (I.EApp (I.EIdent "*", e1), e2)) count sym_env cstack = compileE_oper e1 e2 "mul" count sym_env cstack
  | compileE (I.EApp (I.EApp (I.EIdent "=", e1), e2)) count sym_env cstack = compileE_oper e1 e2 "eq"  count sym_env cstack
  | compileE (I.EApp (I.EApp (I.EIdent ">", e1), e2)) count sym_env cstack = compileE_oper e1 e2 "sgt" count sym_env cstack
  | compileE (I.EApp (I.EApp (I.EIdent "<", e1), e2)) count sym_env cstack = compileE_oper e1 e2 "slt" count sym_env cstack
  | compileE (I.EApp ((I.EIdent str), e)) count sym_env cstack             = compileE_call str e count sym_env cstack
  | compileE (I.EApp(e1, e2)) count sym_env cstack                         = compileE_callfunc e1 e2 count sym_env cstack
  | compileE (I.EIf (e1, e2, e3)) count sym_env cstack                     = compileE_if e1 e2 e3 count sym_env cstack
  | compileE (I.ELet (sym, e1, e2)) count sym_env cstack                   = compileE_let sym e1 e2 count sym_env cstack
  | compileE (I.ELetFun (func, arg, e1, e2)) count sym_env cstack          = compileE_letfun func arg e1 e2 count sym_env cstack
  | compileE (I.EFun (arg, e1)) count sym_env cstack                       = compileE_fun arg e1 count sym_env cstack
  | compileE expr count sym_env cstack                                     = compileError ("Not implemented:\n" ^ (I.stringOfExpr expr)

(*Compile Value to create i32*)
and compileV (I.VInt i) count cstack =
      let
        val str = "    " ^ count_reg count ^ " = call %value @wrap_i32(i32 " ^ itos i ^ ")"
      in
        (count_reg count, count+1, cstack@[str])
      end
  | compileV _ _ _= compileError "Only ints supported"
```

Dynamic Typing

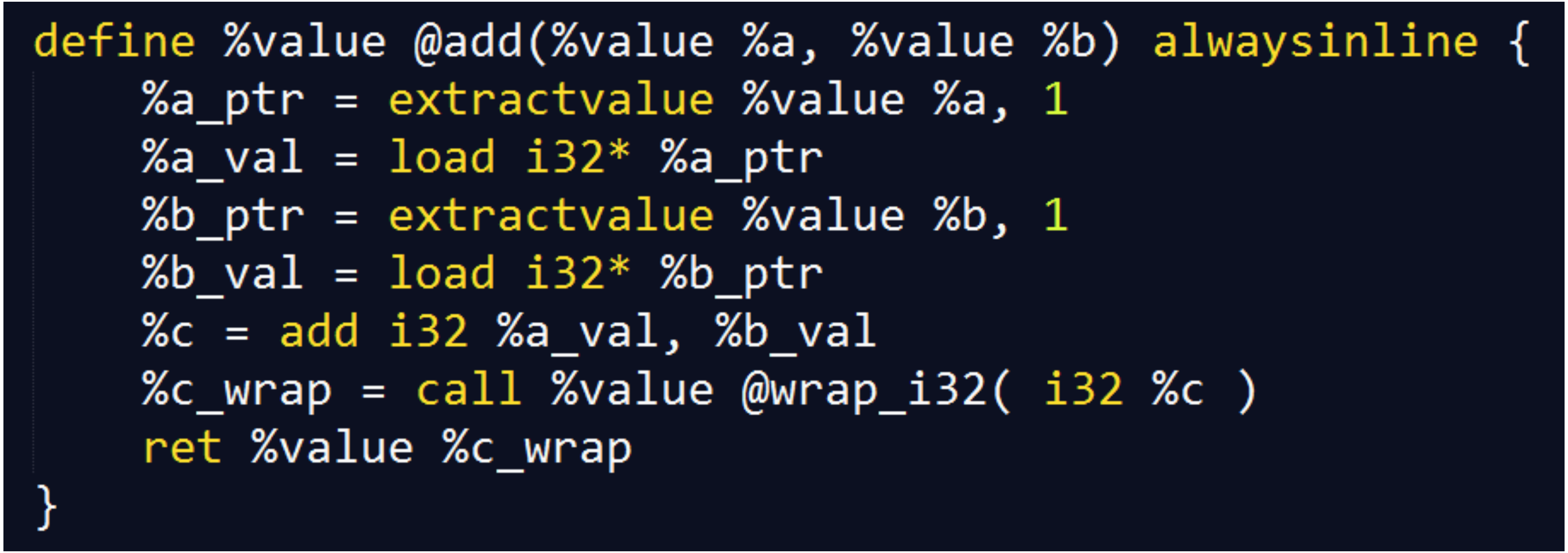


```
%value = type {i8, i32*}
```

- Type
- Pointer to value
  - Think of second arg as void*

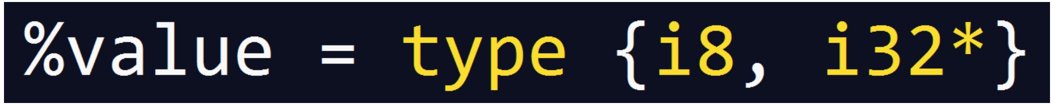
Boilerplate

Basic math operations

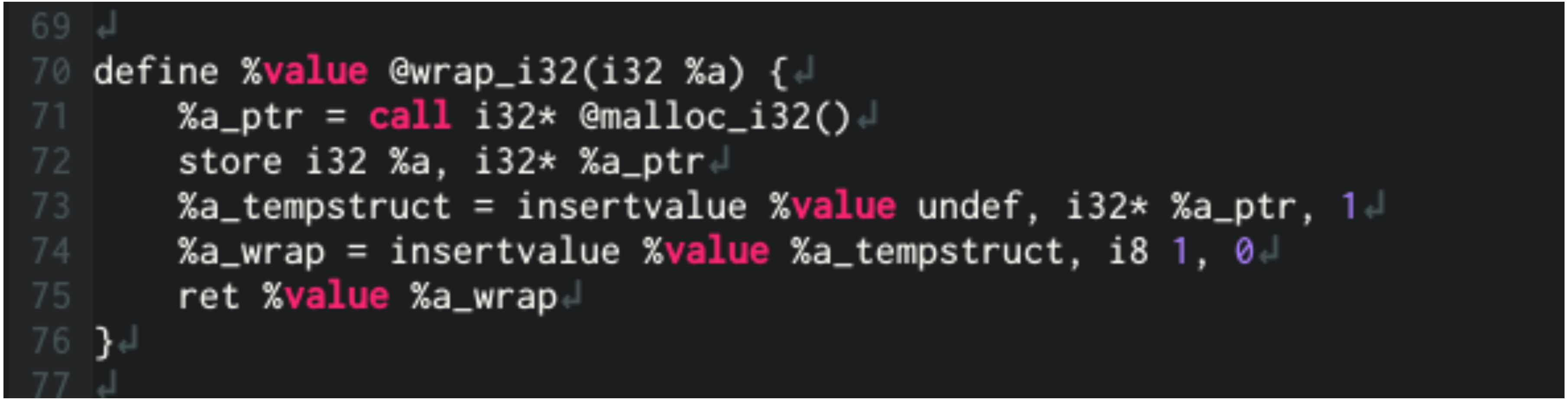


```
define %value @add(%value %a, %value %b) alwaysinline {
    %a_ptr = extractvalue %value %a, 1
    %a_val = load i32* %a_ptr
    %b_ptr = extractvalue %value %b, 1
    %b_val = load i32* %b_ptr
    %c = add i32 %a_val, %b_val
    %c_wrap = call %value @wrap_i32( i32 %c )
    ret %value %c_wrap
}
```

@wrap_i32

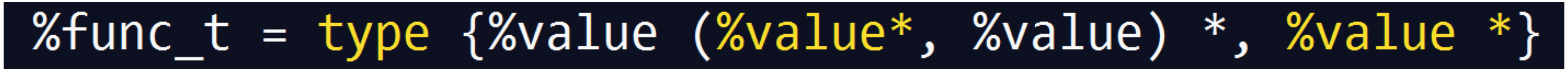


```
%value = type {i8, i32*}
```

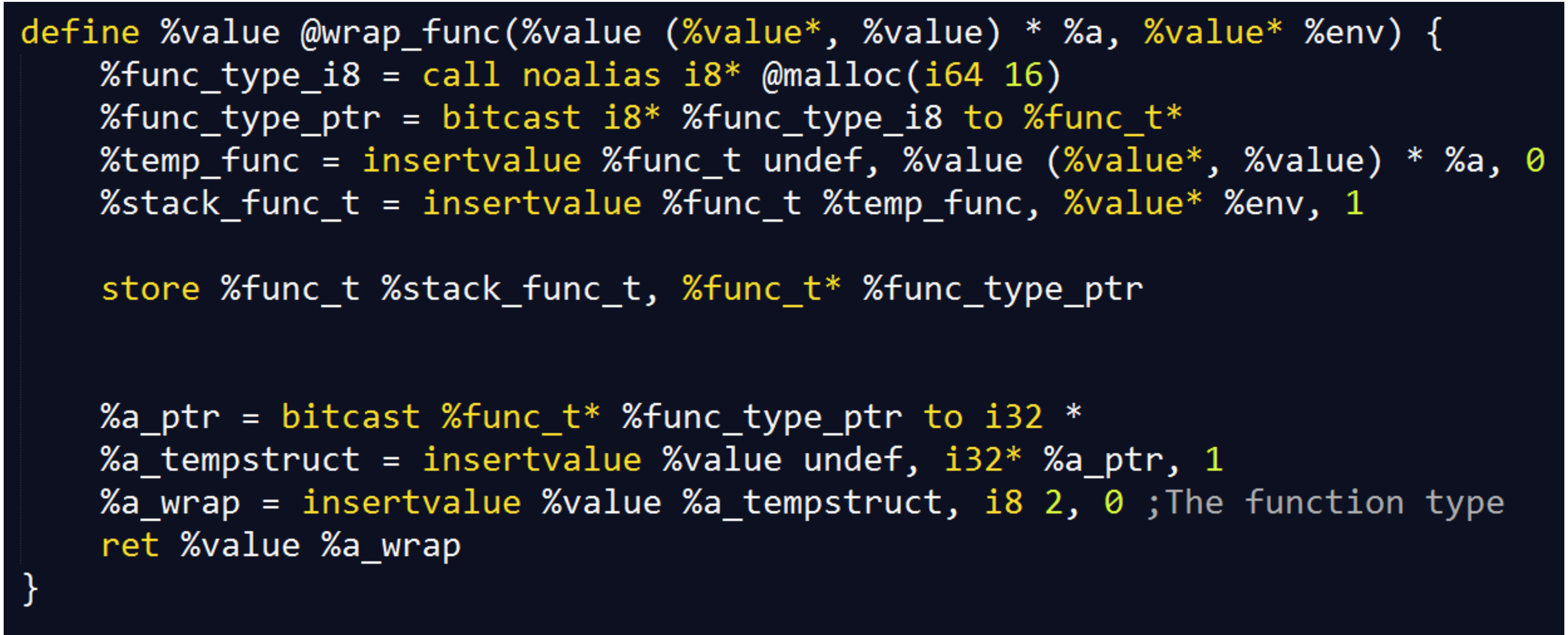


```

define %value @wrap_i32(i32 %a) {
    %a_ptr = call i32* @malloc_i32()
    store i32 %a, i32* %a_ptr
    %a_tempstruct = insertvalue %value undef, i32* %a_ptr, 1
    %a_wrap = insertvalue %value %a_tempstruct, i8 1, 0
    ret %value %a_wrap
}

```

@wrap_func

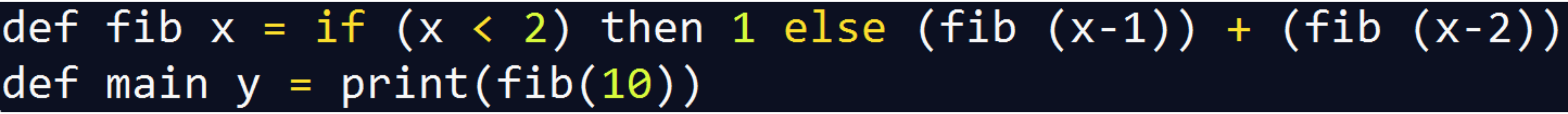


```
%func_t = type {%value (%value*, %value) *, %value *}
```

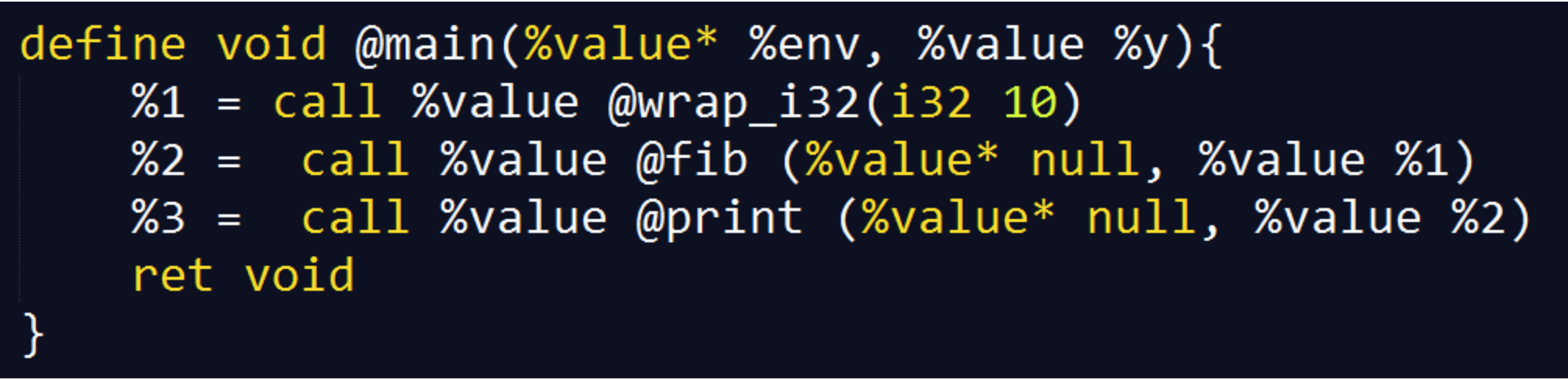


```
define %value @wrap_func(%value (%value*, %value) * %a, %value* %env) {
    %func_type_i8 = call noalias i8* @malloc(i64 16)
    %func_type_ptr = bitcast i8* %func_type_i8 to %func_t*
    %temp_func = insertvalue %func_t undef, %value (%value*, %value) * %a, 0
    %stack_func_t = insertvalue %func_t %temp_func, %value* %env, 1

    store %func_t %stack_func_t, %func_t* %func_type_ptr


    %a_ptr = bitcast %func_t* %func_type_ptr to i32 *
    %a_tempstruct = insertvalue %value undef, i32* %a_ptr, 1
    %a_wrap = insertvalue %value %a_tempstruct, i8 2, 0 ;The function type
    ret %value %a_wrap
}
```

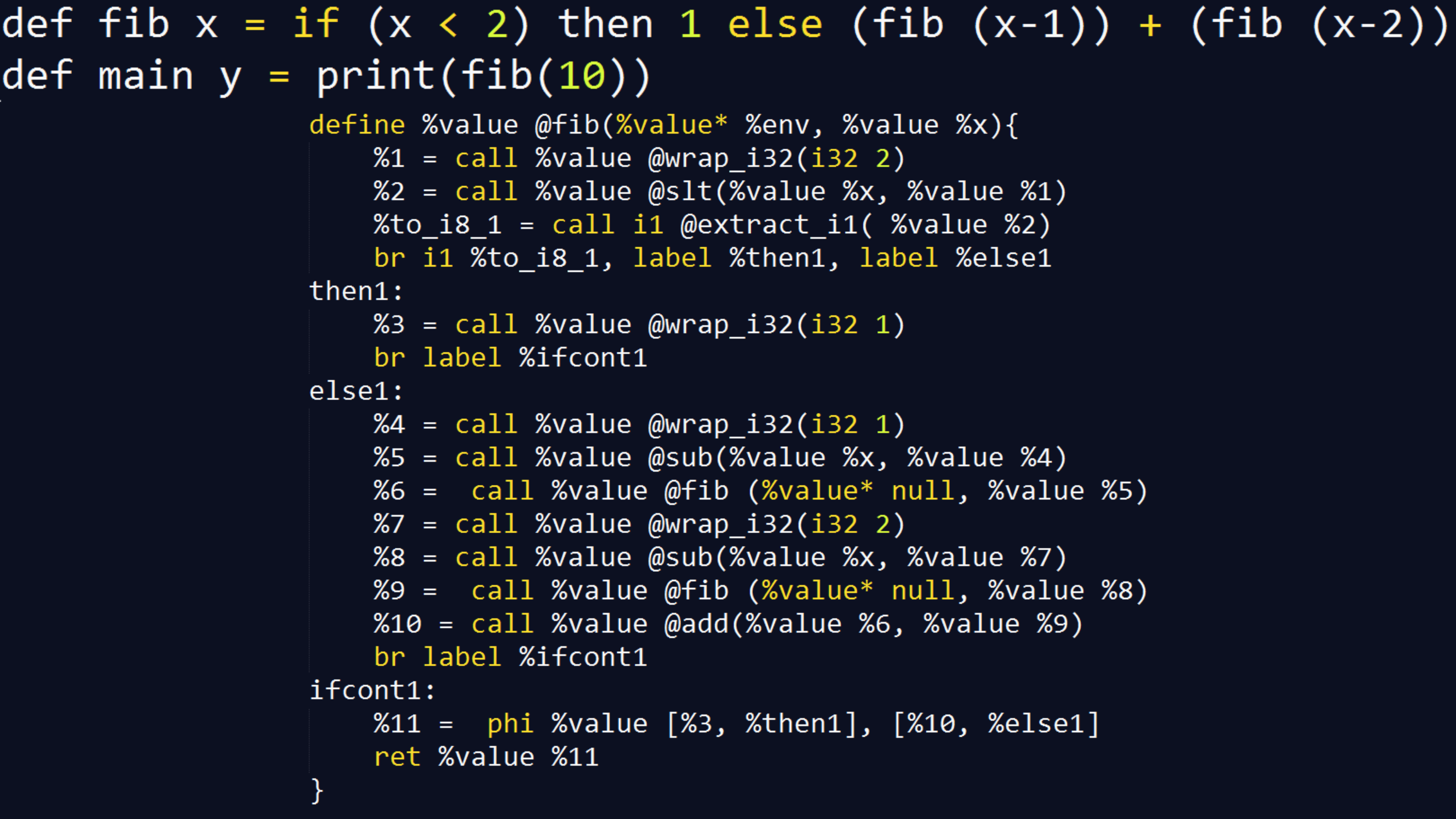
Sample input and output



```
def fib x = if (x < 2) then 1 else (fib (x-1)) + (fib (x-2))
def main y = print(fib(10))
```



```
define void @main(%value* %env, %value %y){
    %1 = call %value @wrap_i32(i32 10)
    %2 =  call %value @fib (%value* null, %value %1)
    %3 =  call %value @print (%value* null, %value %2)
    ret void
}
```



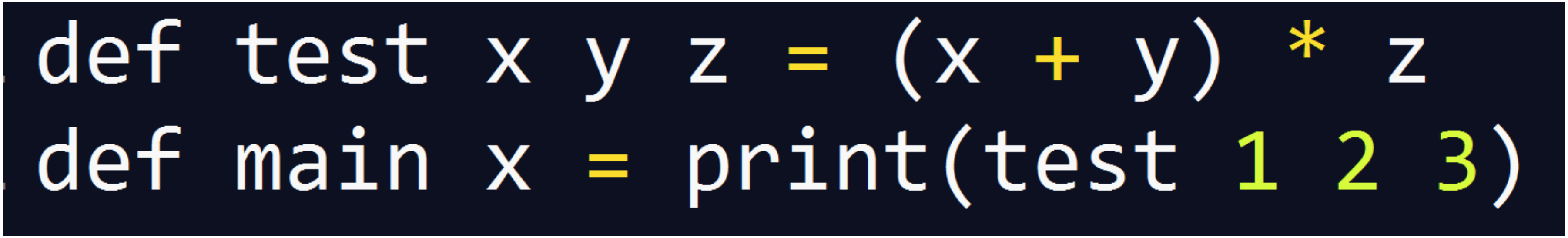
```
def fib x = if (x < 2) then 1 else (fib (x-1)) + (fib (x-2))
def main y = print(fib(10))
```

```
define %value @fib(%value* %env, %value %x){
    %1 = call %value @wrap_i32(i32 2)
    %2 = call %value @slt(%value %x, %value %1)
    %to_i8_1 = call i1 @extract_i1( %value %2)
    br i1 %to_i8_1, label %then1, label %else1
then1:
    %3 = call %value @wrap_i32(i32 1)
    br label %ifcont1
else1:
    %4 = call %value @wrap_i32(i32 1)
    %5 = call %value @sub(%value %x, %value %4)
    %6 =  call %value @fib (%value* null, %value %5)
    %7 = call %value @wrap_i32(i32 2)
    %8 = call %value @sub(%value %x, %value %7)
    %9 =  call %value @fib (%value* null, %value %8)
    %10 = call %value @add(%value %6, %value %9)
    br label %ifcont1
ifcont1:
    %11 =  phi %value [%3, %then1], [%10, %else1]
    ret %value %11
}
```

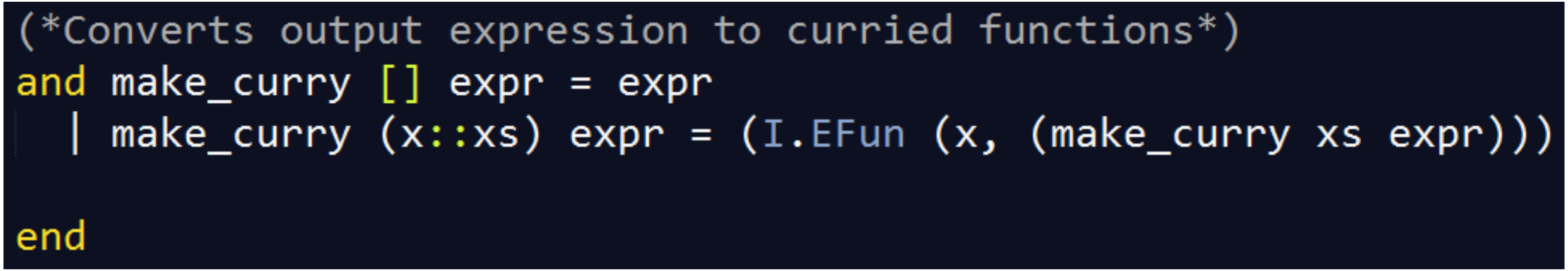
Challenge: Closures

- Higher order functions, without host language support
- Jump out of functions to make new functions
- Environment
  - Keep track of it in SML
  - compiler has no notion of it

Currying

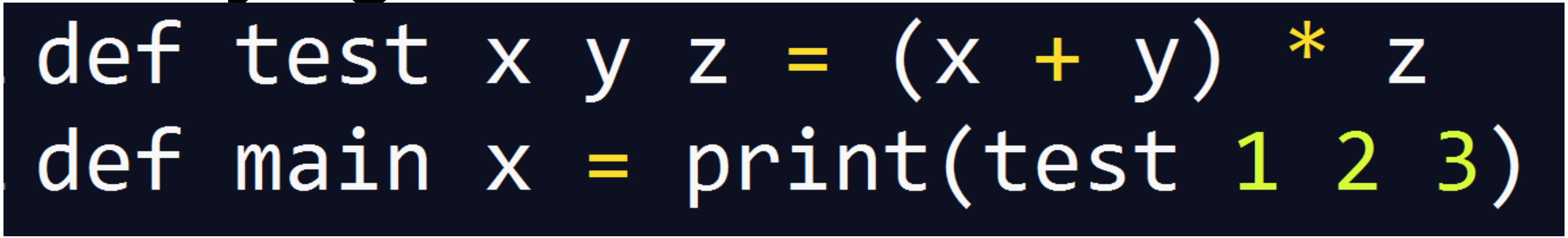


```
def test x y z = (x + y) * z
def main x = print(test 1 2 3)
```

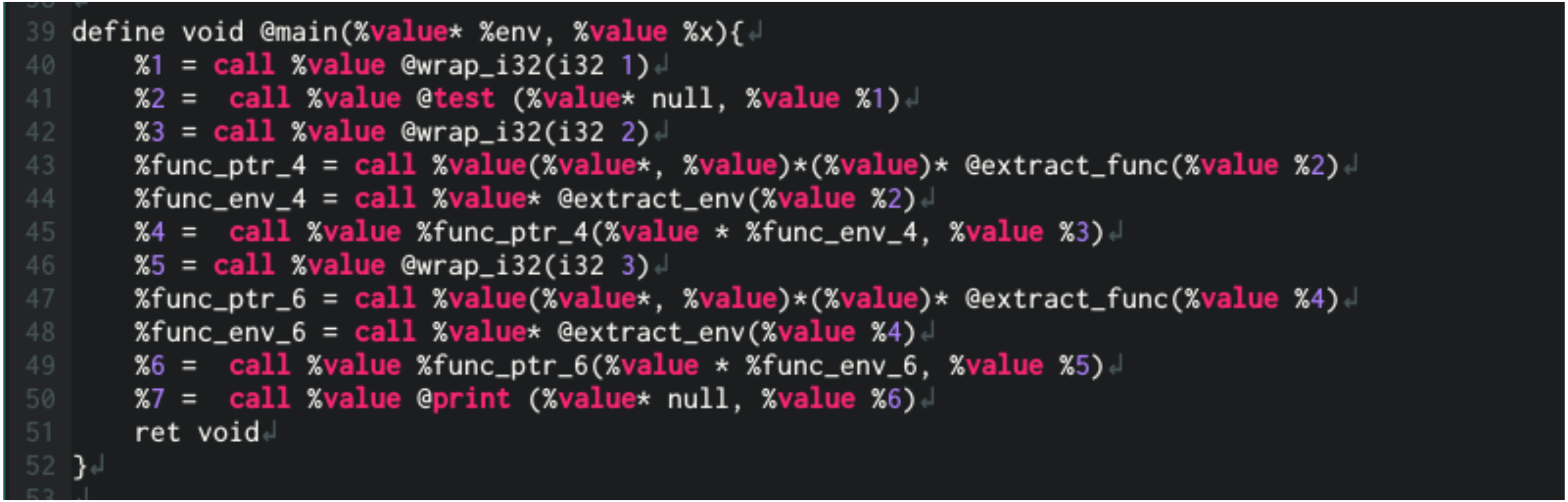


```
(*Converts output expression to curried functions*)
and make_curry [] expr = expr
  | make_curry (x::xs) expr = (I.EFun (x, (make_curry xs expr)))

end
```

Currying

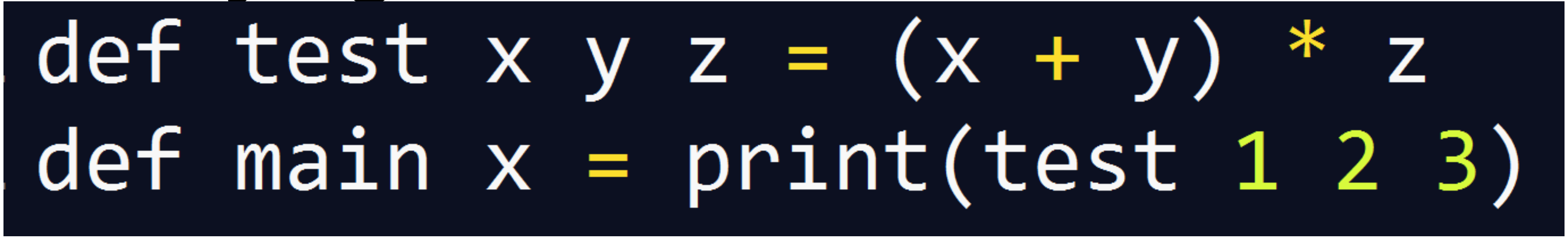


```
def test x y z = (x + y) * z
def main x = print(test 1 2 3)
```

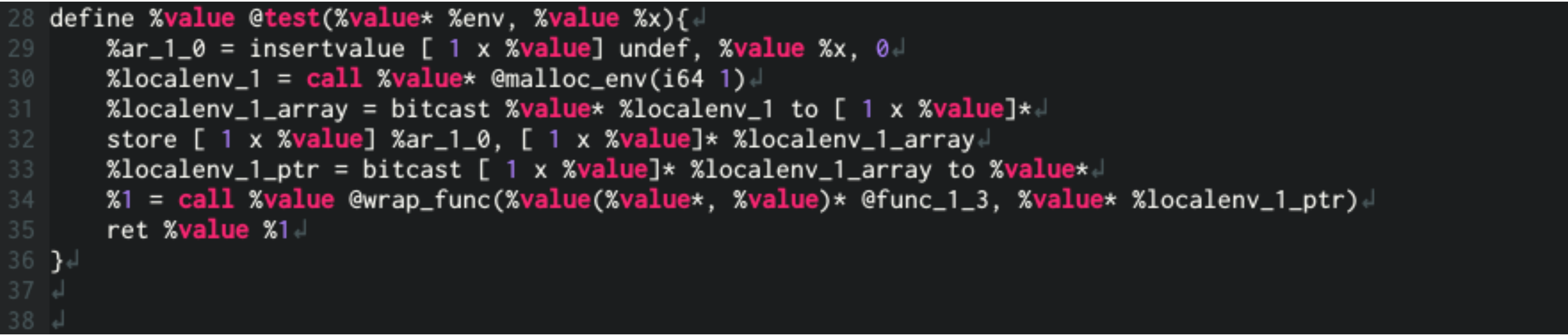


```
define void @main(%value* %env, %value %x){
    %1 = call %value @wrap_i32(i32 1)
    %2 =  call %value @test (%value* null, %value %1)
    %3 = call %value @wrap_i32(i32 2)
    %func_ptr_4 = call %value(%value*, %value)*(%value)* @extract_func(%value %2)
    %func_env_4 = call %value* @extract_env(%value %2)
    %4 =  call %value %func_ptr_4(%value * %func_env_4, %value %3)
    %5 = call %value @wrap_i32(i32 3)
    %func_ptr_6 = call %value(%value*, %value)*(%value)* @extract_func(%value %4)
    %func_env_6 = call %value* @extract_env(%value %4)
    %6 =  call %value %func_ptr_6(%value * %func_env_6, %value %5)
    %7 =  call %value @print (%value* null, %value %6)
    ret void
}
```

Currying: Test

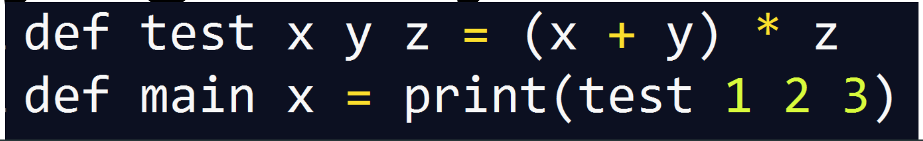


```
def test x y z = (x + y) * z
def main x = print(test 1 2 3)
```

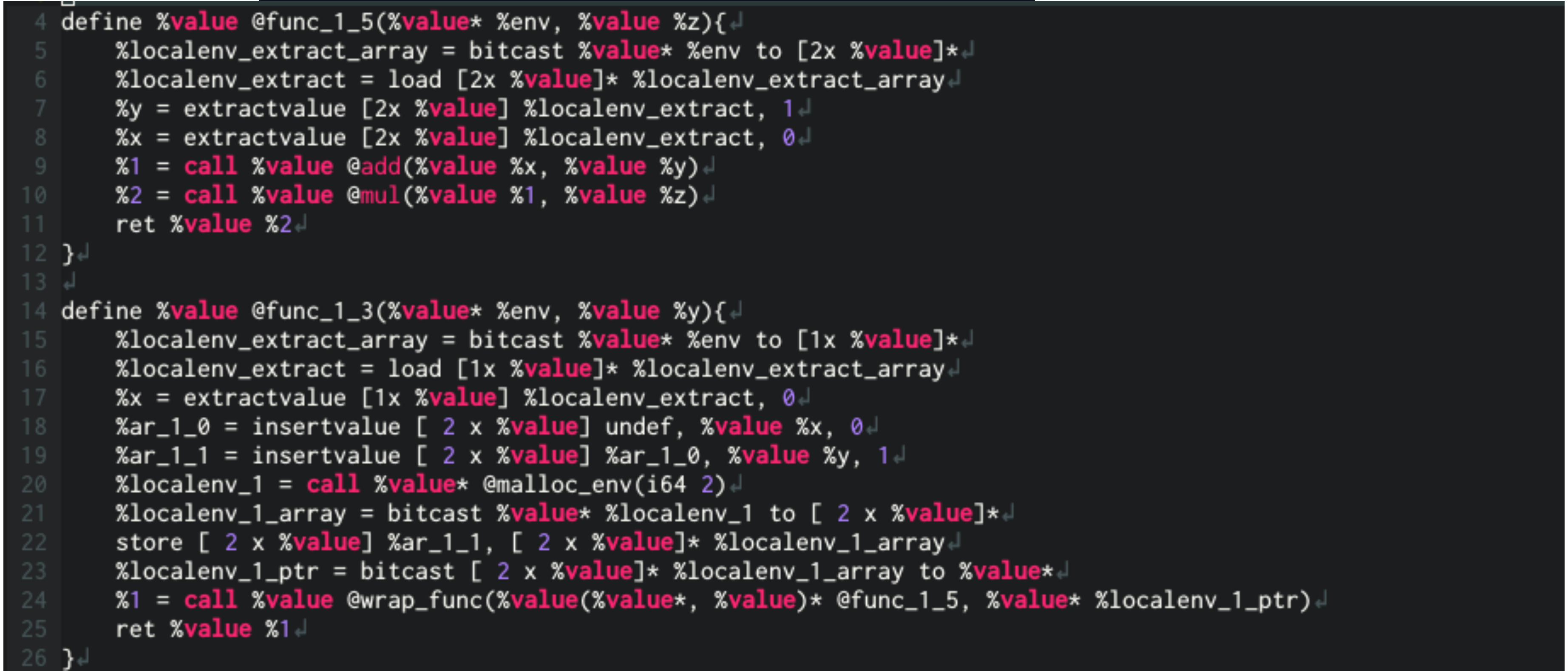


```
define %value @test(%value* %env, %value %x){
    %ar_1_0 = insertvalue [ 1 x %value] undef, %value %x, 0
    %localenv_1 = call %value* @malloc_env(i64 1)
    %localenv_1_array = bitcast %value* %localenv_1 to [ 1 x %value]*
    store [ 1 x %value] %ar_1_0, [ 1 x %value]* %localenv_1_array
    %localenv_1_ptr = bitcast [ 1 x %value]* %localenv_1_array to %value*
    %1 = call %value @wrap_func(%value(%value*, %value)* @func_1_3, %value* %localenv_1_ptr)
    ret %value %1
}


```

Curying anonymous functions



```
def test x y z = (x + y) * z
def main x = print(test 1 2 3)
```



```
define %value @func_1_5(%value* %env, %value %z){
    %localenv_extract_array = bitcast %value* %env to [2x %value]*
    %localenv_extract = load [2x %value]* %localenv_extract_array
    %y = extractvalue [2x %value] %localenv_extract, 1
    %x = extractvalue [2x %value] %localenv_extract, 0
    %1 = call %value @add(%value %x, %value %y)
    %2 = call %value @mul(%value %1, %value %z)
    ret %value %2
}

define %value @func_1_3(%value* %env, %value %y){
    %localenv_extract_array = bitcast %value* %env to [1x %value]*
    %localenv_extract = load [1x %value]* %localenv_extract_array
    %x = extractvalue [1x %value] %localenv_extract, 0
    %ar_1_0 = insertvalue [ 2 x %value] undef, %value %x, 0
    %ar_1_1 = insertvalue [ 2 x %value] %ar_1_0, %value %y, 1
    %localenv_1 = call %value* @malloc_env(i64 2)
    %localenv_1_array = bitcast %value* %localenv_1 to [ 2 x %value]*
    store [ 2 x %value] %ar_1_1, [ 2 x %value]* %localenv_1_array
    %localenv_1_ptr = bitcast [ 2 x %value]* %localenv_1_array to %value*
    %1 = call %value @wrap_func(%value(%value*, %value)* @func_1_5, %value* %localenv_1_ptr)
    ret %value %1
}
```

Challenge: Memory (Malloc all the things)

Leaking

How do we keep track of allocated %value?

Slow

Mallocing small bits of memory is SLOW

Solutions:

- Free up all of them at the end of function calls, or just stack allocate everything.
  - How would closures work then?
- LLVM has basic support for garbage collection
- Use a more complicated structure to hold what needs to be freed (sml)
- Stack allocate more
- Reuse memory

Questions / Comments?