KAZM

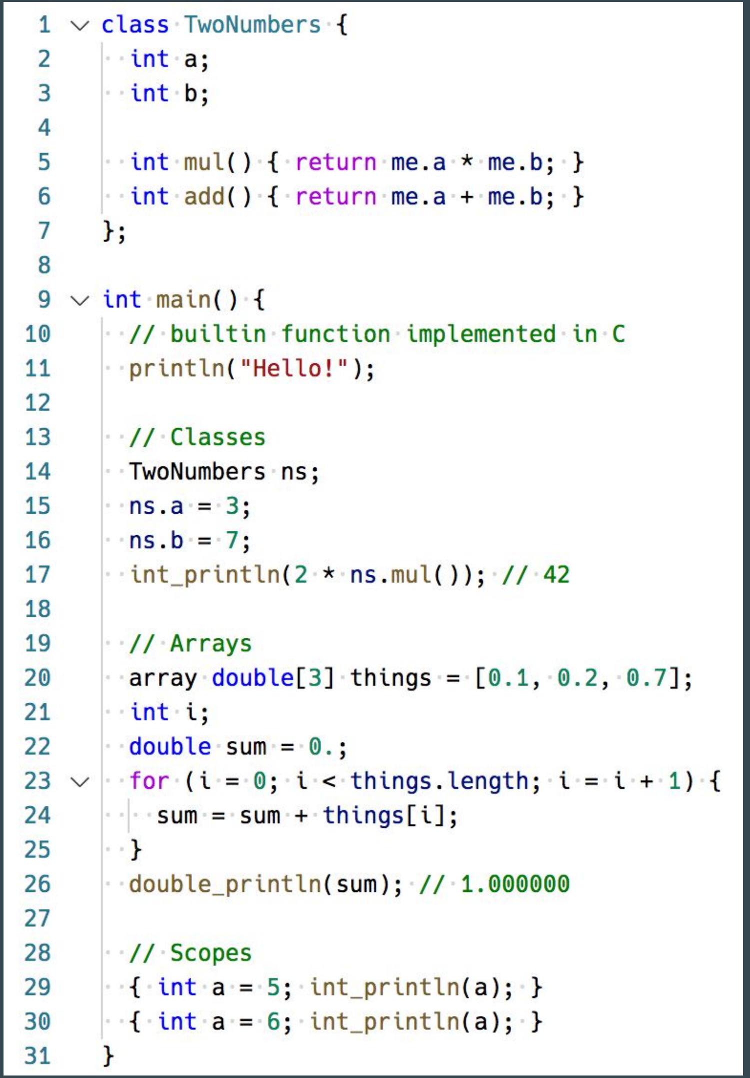
Katie Kim
Aapeli Vuorinen
Zhonglin Yang
Molly McNutt

Authors listed in an order that makes their first names spell out `Kazm'.

Overview — Language

- Kazm is a C-like language
- Subset of basic C functionality
- Written in OCaml{,Yacc,Lex}, outputs LLVM IR

- Interesting functionality:
  - Classes: on heap, member variables, class methods (implemented using pointer to self, “me”)
  - Arrays: heap-allocated, static length
  - Scopes: curly braces, tracking of variable lifetime

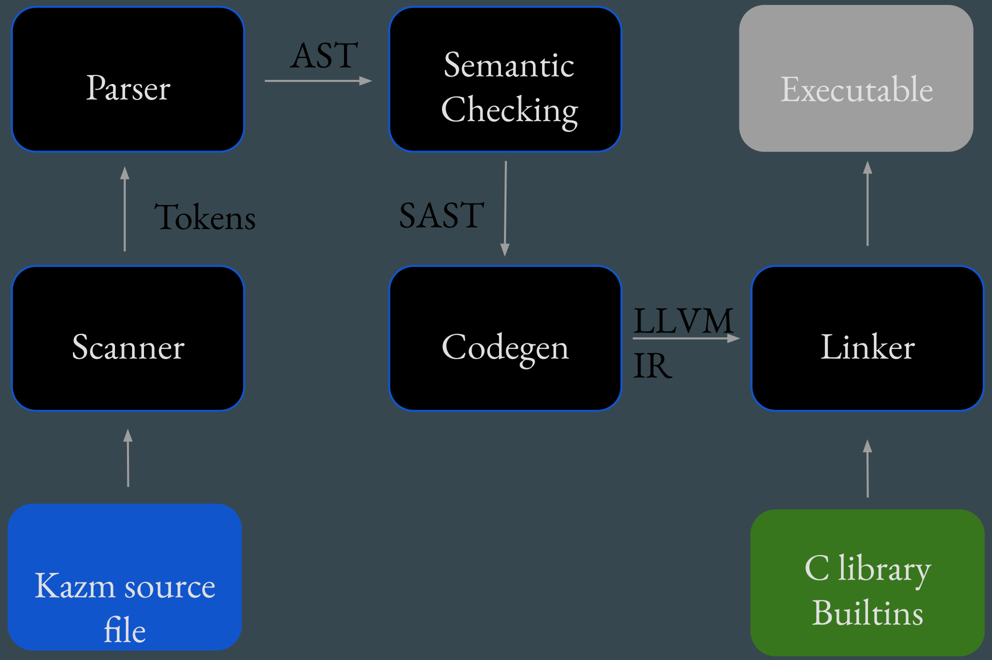


```
class TwoNumbers {
  int a;
  int b;

  int mul() { return me.a * me.b; }
  int add() { return me.a + me.b; }
};

int main() {
  // builtin function implemented in C
  println("Hello!");

  // Classes
  TwoNumbers ns;
  ns.a = 3;
  ns.b = 7;
  int_println(2 * ns.mul()); // 42

  // Arrays
  array double[3] things = [0.1, 0.2, 0.7];
  int i;
  double sum = 0.;
  for (i = 0; i < things.length; i = i + 1) {
    sum = sum + things[i];
  }
  double_println(sum); // 1.000000

  // Scopes
  { int a = 5; int_println(a); }
  { int a = 6; int_println(a); }
}
```

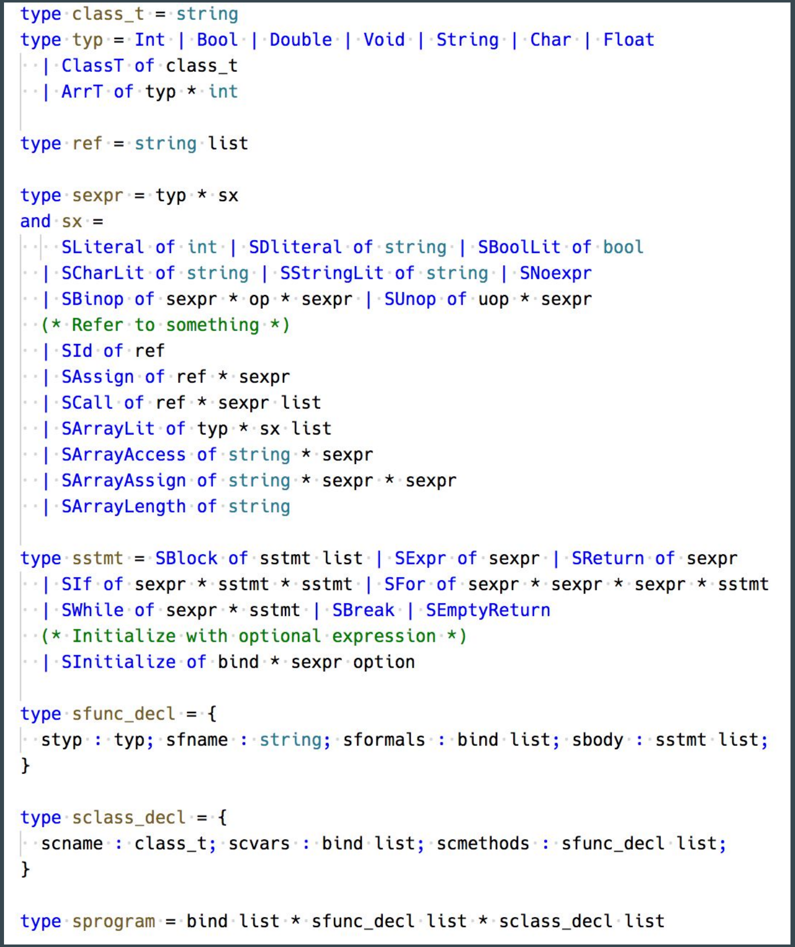
Overview — Architecture

- We chose to have a separate SAST that contains the types and other info of all nodes in the program

- Minimal copy-pasting from MicroC!



(S)AST



```
type class_t = string
type typ = Int | Bool | Double | Void | String | Char | Float
  | ClassT of class_t
  | ArrT of typ * int

type ref = string list

type sexpr = typ * sx
and sx =
    SLiteral of int | SDliteral of string | SBoolLit of bool
  | SCharLit of string | SStringLit of string | SNoexpr
  | SBinop of sexpr * op * sexpr | SUnop of uop * sexpr
  (* Refer to something *)
  | SId of ref
  | SAssign of ref * sexpr
  | SCall of ref * sexpr list
  | SArrayLit of typ * sx list
  | SArrayAccess of string * sexpr
  | SArrayAssign of string * sexpr * sexpr
  | SArrayLength of string

type sstmt = SBlock of sstmt list | SExpr of sexpr | SReturn of sexpr
  | SIf of sexpr * sstmt * sstmt | SFor of sexpr * sexpr * sexpr * sstmt
  | SWhile of sexpr * sstmt | SBreak | SEmptyReturn
  (* Initialize with optional expression *)
  | SInitialize of bind * sexpr option

type sfunc_decl = {
  styp : typ; sfname : string; sformals : bind list; sbody : sstmt list;
}

type sclass_decl = {
  scname : class_t; scvars : bind list; scmethods : sfunc_decl list;
}

type sprogram = bind list * sfunc_decl list * sclass_decl list
```

Semantic checker

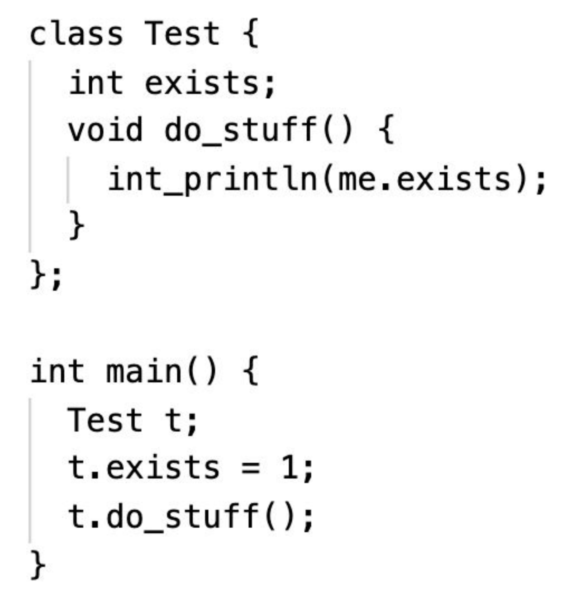
- Checks functions

  With a StringMap storing all functions’ info

  - No duplicate function name
  - Correct variable binding list
  - Correct return type
- Checks classes

  With a StringMap storing all class name and variables and StringMaps storing all class methods’ info

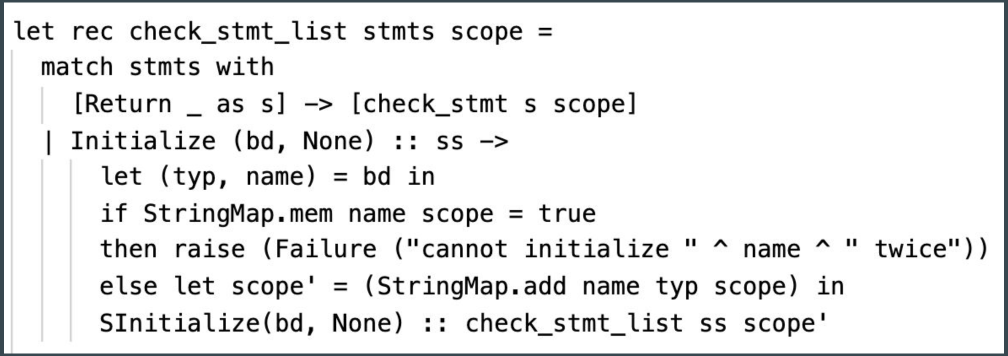
  - Defined class type
  - Defined class instance variables
  - Defined class methods
  - Correct constructors and destructors
- Checks array types and length (partially e.g. my_arr[i])
- Checks operators type
- Checks variables’ scopes



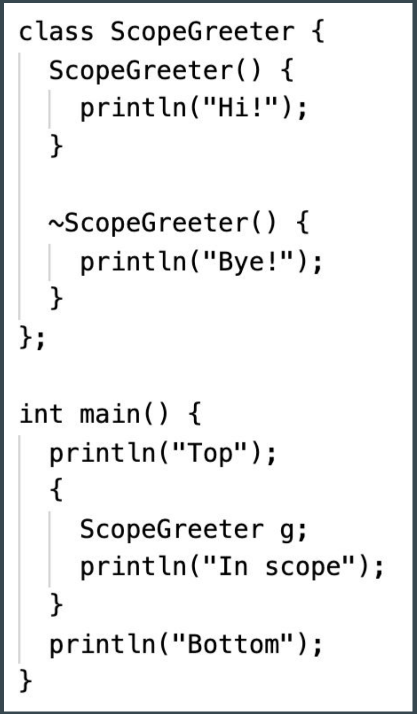
```
class Test {
  int exists;
  void do_stuff() {
    int_println(me.exists);
  }
};

int main() {
  Test t;
  t.exists = 1;
  t.do_stuff();
}
```

Scopes

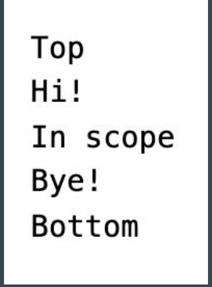
- Variables are allowed to be initialized anywhere inside a function
  - int i; -> Initialize((Int, “i”), None)
  - int i = 1; -> Initialize((Int, “i”), Literal 1)
- Variables accessible inside the scope (block { }) where they are initialized



```
let rec check_stmt_list stmts scope =
  match stmts with
    [Return _ as s] -> [check_stmt s scope]
  | Initialize (bd, None) :: ss ->
      let (typ, name) = bd in
      if StringMap.mem name scope = true
      then raise (Failure ("cannot initialize " ^ name ^ " twice"))
      else let scope' = (StringMap.add name typ scope) in
      SInitialize(bd, None) :: check_stmt_list ss scope'
```



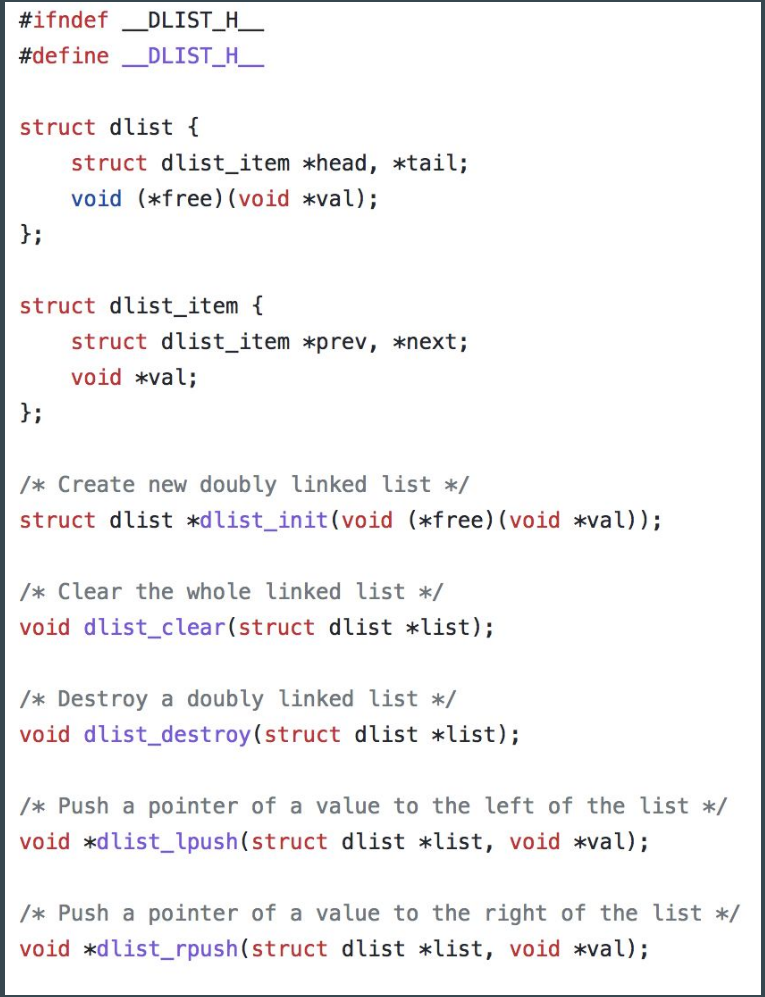
```
class ScopeGreeter {
  ScopeGreeter() {
    println("Hi!");
  }

  ~ScopeGreeter() {
    println("Bye!");
  }
};

int main() {
  println("Top");
  {
    ScopeGreeter g;
    println("In scope");
  }
  println("Bottom");
}
```



```
Top
Hi!
In scope
Bye!
Bottom
```

Classes: motivation

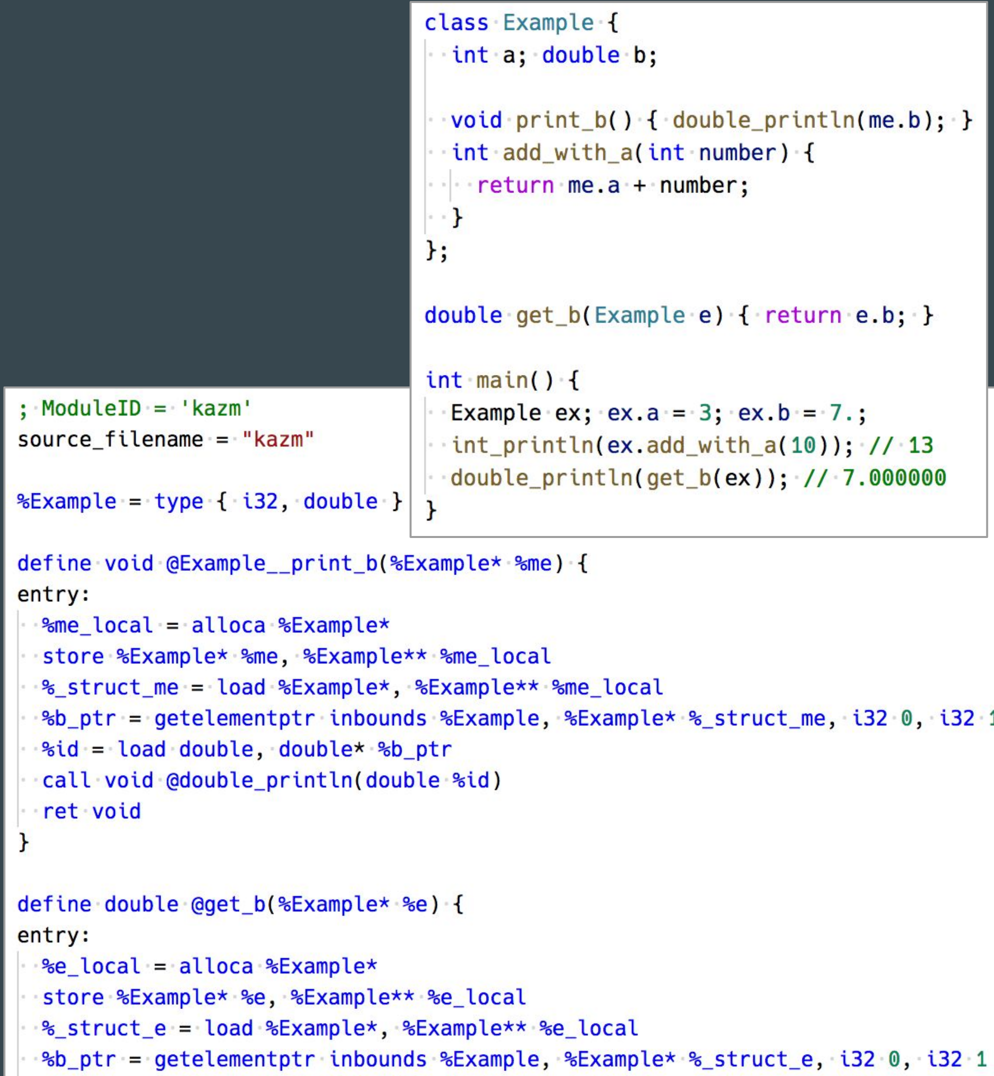
- You see this a lot when writing C:
  - Struct with the members
  - Bunch of functions prefix with “structname_”
  - First param is always a pointer to the struct



```c
#ifndef __DLIST_H__
#define __DLIST_H__

struct dlist {
    struct dlist_item *head, *tail;
    void (*free)(void *val);
};

struct dlist_item {
    struct dlist_item *prev, *next;
    void *val;
};

/* Create new doubly linked list */
struct dlist *dlist_init(void (*free)(void *val));

/* Clear the whole linked list */
void dlist_clear(struct dlist *list);

/* Destroy a doubly linked list */
void dlist_destroy(struct dlist *list);

/* Push a pointer of a value to the left of the list */
void *dlist_lpush(struct dlist *list, void *val);

/* Push a pointer of a value to the right of the list */
void *dlist_rpush(struct dlist *list, void *val);
```

Classes: implementation

- Heap allocated named structs
- Methods are normal functions with:
  - Name mangling
  - First param is always a pointer to the struct
  - Refer to self as “me”
- Classes can be passed as parameters to other functions (passed as ptr), including other classes’ methods
- Optional constructor and destructor

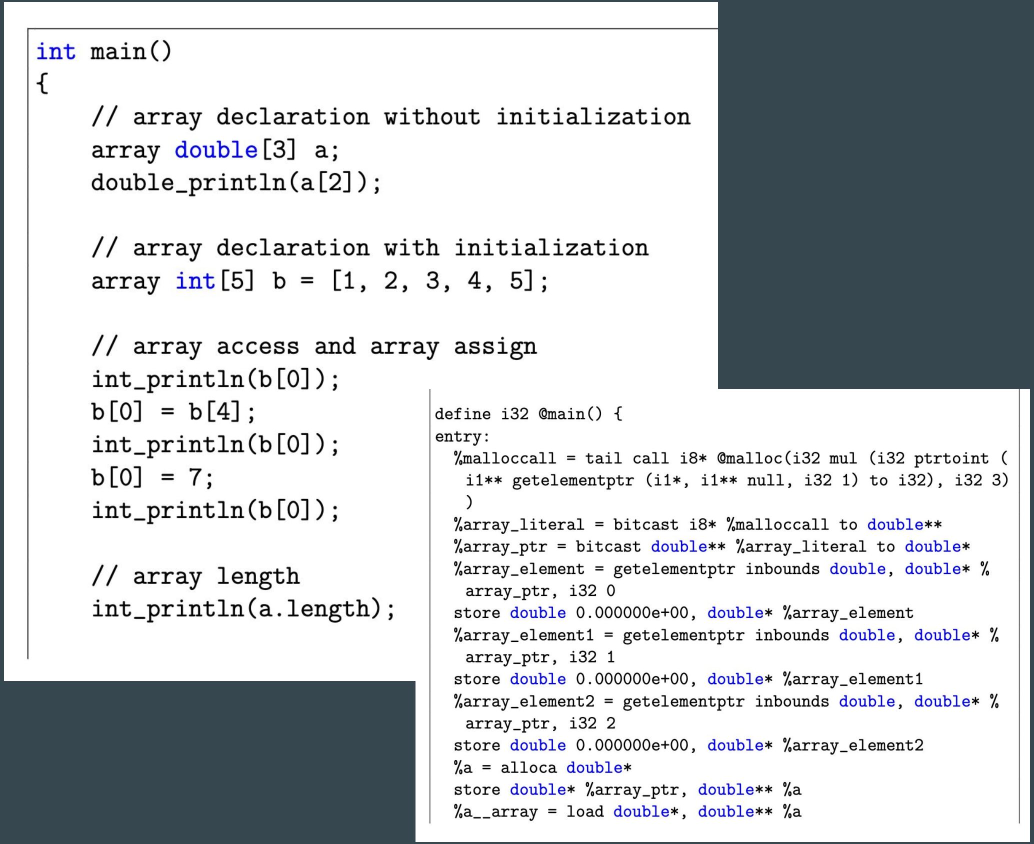


```
class Example {
  int a; double b;

  void print_b() { double_println(me.b); }
  int add_with_a(int number) {
    return me.a + number;
  }
};

double get_b(Example e) { return e.b; }

int main() {
  Example ex; ex.a = 3; ex.b = 7.;
  int_println(ex.add_with_a(10)); // 13
  double_println(get_b(ex)); // 7.000000
}
```

```
; ModuleID = 'kazm'
source_filename = "kazm"

%Example = type { i32, double }

define void @Example__print_b(%Example* %me) {
entry:
  %me_local = alloca %Example*
  store %Example* %me, %Example** %me_local
  %_struct_me = load %Example*, %Example** %me_local
  %b_ptr = getelementptr inbounds %Example, %Example* %_struct_me, i32 0, i32 1
  %id = load double, double* %b_ptr
  call void @double_println(double %id)
  ret void
}

define double @get_b(%Example* %e) {
entry:
  %e_local = alloca %Example*
  store %Example* %e, %Example** %e_local
  %_struct_e = load %Example*, %Example** %e_local
  %b_ptr = getelementptr inbounds %Example, %Example* %_struct_e, i32 0, i32 1
```

Arrays — Katie

- Allocated on the heap
  - Design choice with future improvements in mind
- Fixed length
  - Access with .length
- Declaration with and without initialization supported
  - Without initialization: initialized to default value – not left empty
- Cannot be returned by functions
- Cannot have arrays of classes
- Leak memory

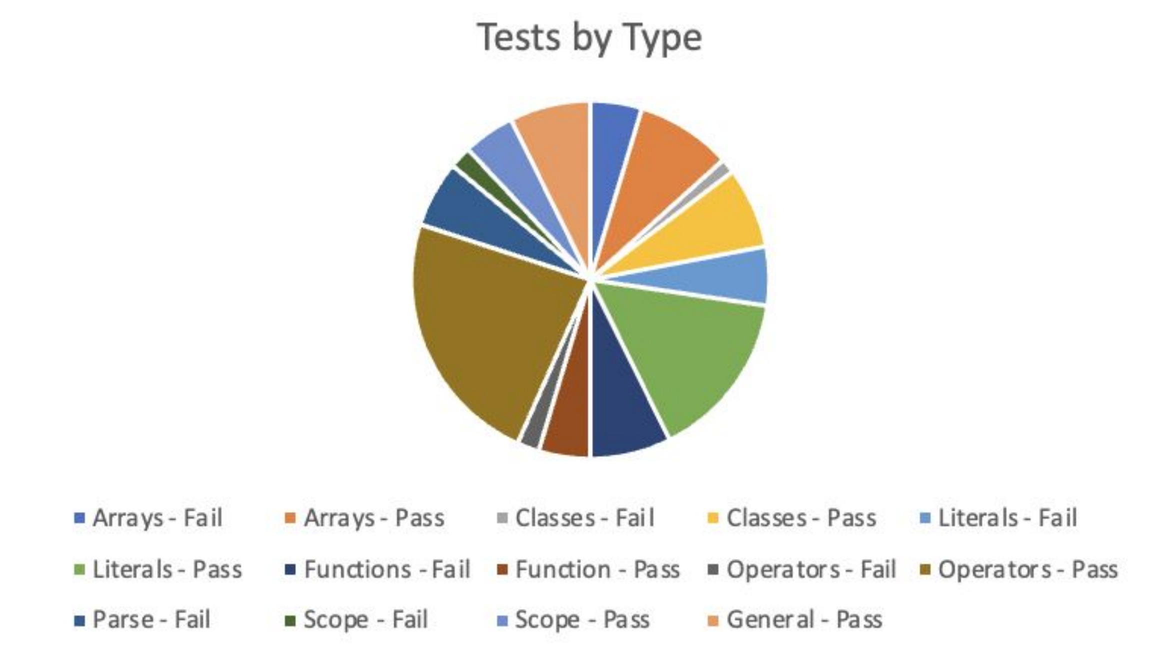


```
int main()
{
    // array declaration without initialization
    array double[3] a;
    double_println(a[2]);

    // array declaration with initialization
    array int[5] b = [1, 2, 3, 4, 5];

    // array access and array assign
    int_println(b[0]);
    b[0] = b[4];
    int_println(b[0]);
    b[0] = 7;
    int_println(b[0]);

    // array length
    int_println(a.length);
```

```
define i32 @main() {
entry:
  %malloccall = tail call i8* @malloc(i32 mul (i32 ptrtoint (
   i1** getelementptr (i1*, i1** null, i32 1) to i32), i32 3)
   )
  %array_literal = bitcast i8* %malloccall to double**
  %array_ptr = bitcast double** %array_literal to double*
  %array_element = getelementptr inbounds double, double* %
   array_ptr, i32 0
  store double 0.000000e+00, double* %array_element
  %array_element1 = getelementptr inbounds double, double* %
   array_ptr, i32 1
  store double 0.000000e+00, double* %array_element1
  %array_element2 = getelementptr inbounds double, double* %
   array_ptr, i32 2
  store double 0.000000e+00, double* %array_element2
  %a = alloca double*
  store double* %array_ptr, double** %a
  %a__array = load double*, double** %a
```

Testing

- Comprehensive test suite of 145 tests
- Test Runner Output
- Loggy.txt
- Tests.md
- Makefile

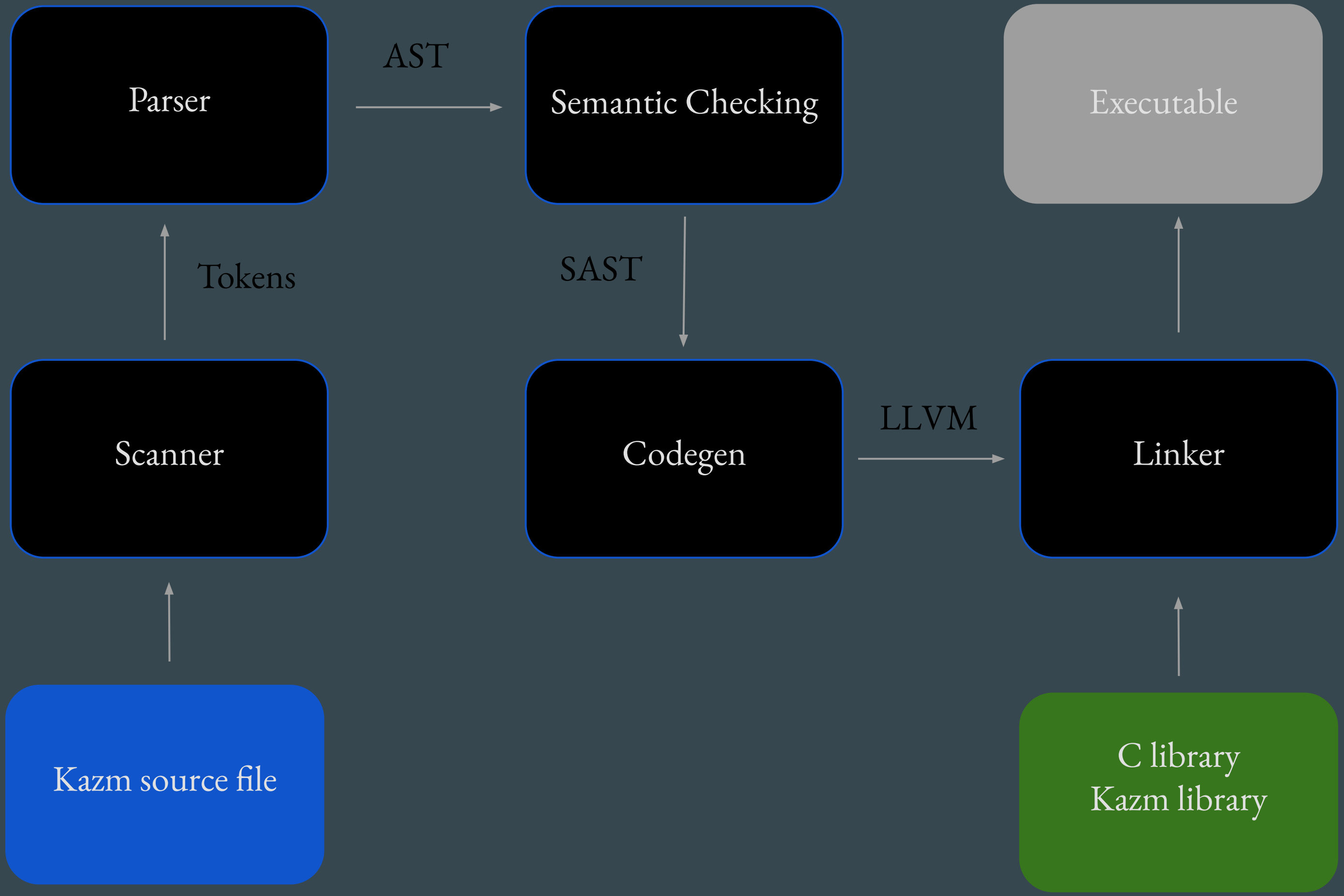


Enhancements: aka what’s broken

- Dynamic length arrays (currently fixed length)
- Array and class interop (can’t do arrays of classes)
- Classes don’t have well defined semantics for assignment, move, copy, etc
- Arrays leak memory! No free, only malloc!!

Who did what

- Aapeli: Codegen, Classes, Scopes, Test Runner, Docker, GitHub actions
- Zhonglin: Semantic analysis and SAST, Literals, Strings
- Molly: Testing & Test Suite, Arrays (mostly support), Final Report
- Katie: Arrays, Final Report


Parser
AST
Semantic Checking
Executable
Tokens
SAST
Scanner
Codegen
LLVM
Linker
Kazm source file
C library
Kazm library