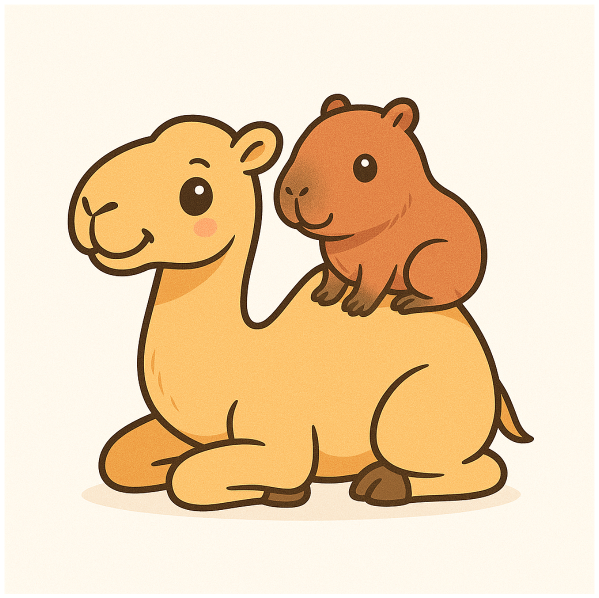

CAPybara

A C-Style Array Programming Language

by
Claudia Cortell
Chelsea Soemitro
Albert Jan
Faustina Cheng
Jake Torres
Mara Dominguez

Contents

1 Introduction **3**

2 Language Tutorial **3**
- 2.1 Language Installation . . . 3
- 2.2 Running the Program . . . 3

3 Language Reference Manual **3**
- 3.1 Tokens . . . 3
  - 3.1.1 Identifiers . . . 3
  - 3.1.2 Literals . . . 3
  - 3.1.3 Operators . . . 4
  - 3.1.4 Keywords . . . 4
  - 3.1.5 Delimiters . . . 4
- 3.2 Types . . . 4
  - 3.2.1 Primitive Types . . . 4
  - 3.2.2 Data Structures . . . 5
- 3.3 Operators . . . 5
  - 3.3.1 Arithmetic Operators . . . 5
  - 3.3.2 Assignment Operators . . . 5
  - 3.3.3 Relational Operators . . . 5
  - 3.3.4 Logical Operators . . . 5
  - 3.3.5 Array Operators . . . 6
    - 3.3.5.1 Mathematical Operators . . . 6
    - 3.3.5.2 Access and Slicing Operators . . . 6
    - 3.3.5.3 Length . . . 6
    - 3.3.5.4 Tranpose . . . 6
    - 3.3.5.5 Map . . . 6
    - 3.3.5.6 Reduce . . . 7
    - 3.3.5.7 Examples of Array Operators . . . 7
- 3.4 Declarations . . . 8
  - 3.4.1 Declarations of primitive types . . . 8
  - 3.4.2 Declarations of Arrays . . . 9
- 3.5 Statements and Expressions . . . 9
  - 3.5.1 If and Else . . . 10
  - 3.5.2 While . . . 10
  - 3.5.3 For . . . 10
  - 3.5.4 Break . . . 10
- 3.6 Functions . . . 11
  - 3.6.1 Function Declaration . . . 11
  - 3.6.2 Library Functions . . . 11
- 3.7 Scoping . . . 12
- 3.8 Example Programs . . . 12

4 Architectural Diagram **14**
- 4.1 CAPybara Source File (.cap) . . . 14
- 4.2 Scanner . . . 14
- 4.3 Parser . . . 14
- 4.4 Semantic Checking . . . 14
- 4.5 IR Generation . . . 14
- 4.6 Linking & Machine Code Generation . . . 14

5 Test Plan **15**
- 5.1 Sample Programs . . . 15
- 5.2 Automated Testing . . . 15

6 Project Summary **15**
6.1 Division of Work . . . 15
6.2 Takeaways . . . 15
6.2.1 Chelsea . . . 15
6.2.2 Claudia . . . 16
6.2.3 Faustina . . . 16
6.2.4 Albert . . . 16
6.2.5 Jake . . . 16
6.2.6 Mara . . . 17
6.3 Advice for Future Teams . . . 17

7 Appendix **17**
7.1 Generated Code Examples . . . 17
7.1.1 Example 1 . . . 17
7.1.2 Example 2 . . . 19
7.2 GitHub Link . . . 22
7.3 scanner.mll . . . 22
7.4 capyparse.mly . . . 24
7.5 ast.ml . . . 28
7.6 sast.ml . . . 32
7.7 semant.ml . . . 35
7.8 codegen.ml . . . 47
7.9 capybara.ml . . . 85
7.10 Makefile . . . 85
7.11 test_runner.sh . . . 86

1 Introduction

CAPybara is a programming language which shares many features with C, including static typing, lexical scoping, and eager evaluation, but provides support for advanced array and matrix interaction. It includes basic operations such as addition, subtraction, element-wise multiplication, and matrix multiplication, while also supporting advanced operations such as slicing, length, transpose, map, and reduce. CAPybara does not include all features in the C standard library, such as multithreading, pointers, and preprocessing, but will provide operators needed for data science applications to arrays. A link to our GitHub repository containing our full code can be found in the Appendix under section 7.2.

2 Language Tutorial

2.1 Language Installation

In order to compile and run CAPybara programs, use OCaml version 4.14.2 and LLVM version 14.0.6.

2.2 Running the Program

To build the CAPybara compiler, run make:

```
$ make
```

To compile the CAPybara source program, run the following command with flag -c:

```
$ ./capybara.native [-a|-s|-l|-c] <program_name>.cap
```

This will generate an executable of the same program name as the source file.

The other flag options, -a, -s, -l, will print the abstract syntax tree, semantically-checked abstract syntax tree, and the generated LLVM IR code to the console, respectively.

To run the executable, run the following command:

```
$ lli <program_name>
```

3 Language Reference Manual

3.1 Tokens

The tokens in CAPybara are identifiers, literals, operators, keywords, and delimiters.

3.1.1 Identifiers

CAPybara identifiers are a sequence of ASCII letters, digits, or underscores. An identifier must start with a letter, but can be followed by any amount of letters, digits, or underscores.

3.1.2 Literals

CAPybara literals are a value of a primitive type, which include **`int`**, **`double`**, **`bool`**, **`char`**, **`string`**.

- **`int`** literals are 4-byte values. They are a sequence of digits from 0 to 9 and may include a negative sign at the beginning.
- **`double`** literals are 8-byte values. They are a sequence of digits from 0 to 9 and must contain a decimal point with a sequence of digits that follow it.
- **`bool`** literals are 1-byte values. 0 corresponds to false; 1 corresponds to true.

- **`char`** literals are 1-byte values. They are a single printable ASCII character inside of single quotation marks.
- **`string`** literals are sequences of chars. They are a sequence of ASCII chars inside of double quotation marks.

3.1.3 Operators

CAPybara supports arithmetic, assignment, logical, relational, and array operators.

- Arithmetic: `+`, `-`, `*`, `/`, `%`
- Assignment: `=`
- Relational: `==`, `!=`, `<`, `<=`, `>`, `>=`
- Logical: `&&`, `||`, `!`
- Array: `+`, `-`, `*`,`/`, `@`, `[]`, `:`, **`length`**, **`transpose`**, **`map`**, **`reduce`**

3.1.4 Keywords

The following are keywords in the language and cannot be used in other contexts:
`int, double, void, bool, char, string, if, else, for, while, break, return, true, false, length, transpose, map, reduce`.

3.1.5 Delimiters

CAPybara relies on delimiters to provide structure to its code. Whitespace is ignored. It will include semicolons (`;`), commas (`,`), curly braces (`{}`), and parenthesis (`()`).

- Semicolons mark the ends of statements, and also delineate 1-dimensional sub-arrays within a 2-dimensional array.
- Commas separate function arguments, loop control expressions, and elements in arrays.
- Curly braces delineate blocks of code, including function bodies, conditional blocks, and loop blocks.
- Parentheses are used in function calls and definitions, conditional control structures, and for operator precedence grouping.
- Comments are both single-line (`//`) and multi-line (`/* */`).

3.2 Types

3.2.1 Primitive Types

`int, double, bool, char, string`

`int`: Primitive type that uses 4 bytes to store a numerical signed integer value.

- Examples: `7, `**`int`**` j = 5`

`double`: Primitive type that uses 8 bytes to store a signed floating point value.

- Examples: `7.0, `**`double`**` j = 5.7`

`bool`: Primitive type that uses 1 byte to store true or false as a value.

- Examples: **`true`**`, `**`bool`**` x = `**`false`**

`char`: Primitive type that uses 1 byte to store a printable ASCII character value from 32 - 126. The characters backslash (\), newline (`n`), tab (`t`), carriage return (`r`), single quote (`'`), and double quote(`"`) need to be escaped with a \ prefix.

- Examples: `'a'`, `char x = 'a'`

`string`: Primitive type that is a sequence of chars.

- Examples: `"capybara"`, `string s = "capybara"`

3.2.2 Data Structures

CAPybara arrays are fixed sized and all elements must be the same primitive type. The declaration of a 1-dimensional array follows this syntax:
`data_type[size] array_name;`

The declaration and initialization of a 1-dimensional array follows this syntax:
`data_type[size] array_name = [elem_1, elem_2, elem_3];`

Arrays can be 1-dimensional or 2-dimensional. A 2-dimensional array, or a matrix, contains elements that are 1-dimensional arrays. The declaration of a matrix follows this syntax:
`data_type[row_size][column_size] array_name;`

The declaration and initialization of a 2-dimensional array follows this syntax:
`data_type[row_size][column_size] array_name = [elem1, elem2; elem3, elem4;];`

The explicit sizes in the declaration must be an integer literal for the size to be known at compile-time.

Arrays have multiple operators defined, which are addition, subtraction, multiplication, division, access, slicing, length, transpose, map, and reduce. Arrays also support element-wise scaling by integers or doubles.

3.3 Operators

3.3.1 Arithmetic Operators

`+, -, *, /, %`
Arithmetic operators are binary operators that are left-to-right associative. Arithmetic operators follow order of operations with multiplication, division and modulo having higher precedence than addition and subtraction.

3.3.2 Assignment Operators

`=`
The assignment operator is a binary operator that is left-to-right associative. = assigns the expression on the right to the variable on the left and evaluates to the value assigned. The compiler will throw an error if there are different types on each side of the assignment operator.
Example: **int x = 5;**

3.3.3 Relational Operators

`==, !=, <, <=, >, >=`
Relational Operators are left-to-right associative. Relational operators have lower precedence than arithmetic operators. Equality checking operators (`==` and `!=`) have lower precedence than relational operators (`<`, `<=`, `>`, `>=`), which all have the same precedence.

3.3.4 Logical Operators

`&&, ||, !`
Logical operators are binary or unary operators that result in a boolean value. Logical and (`&&`) and logical or (`||`) have lower precedence than relational operators, but logical not (`!`) is higher than any logical or relational operator.

3.3.5 Array Operators

`+, -, *, /, @, [], :, length, transpose, map, reduce`

3.3.5.1 Mathematical Operators

The mathematical array operators are:

- `+`: element-wise addition
- `-`: element-wise subtraction
- `*`: element-wise multiplication by constants or other arrays
- `/`: element-wise division of arrays
- `@`: matrix multiplication

These mathematical array operators can only be used on arrays of ints and doubles. They are not defined for arrays of chars, bools, and strings.

These operators require the arrays they are used on to have the proper shapes; they will throw an error for invalid shapes. `+`, `-`, `*`, and `/` are defined for both 1 and 2-dimensional arrays, but the matrix-multiplication operator, `@`, is only defined for 2-dimensional arrays with compatible shapes. Arrays can only be element-wise scaled by integers or doubles.

3.3.5.2 Access and Slicing Operators

The operator `[]` is array-indexing, which retrieves the element at the specified index in the brackets.

The operator `:` is array slicing. It is used inside of brackets to indicate which indices to slice the array at. The indices used for slicing must be integer literals in order to know the resulting shape at compile-time.

```
m[0:2, 0:2] // OK
m[0:x, y:1+3] // NOT OK, expressions used in place of int literals
```

The access and slice operators must be done only on array variables, not all array expressions. Array access and slicing operators can be used on arrays of all types and dimensions.

3.3.5.3 Length

The length operator, using keyword **`length`**, will return the integer length of the array in its parentheses. For 2D arrays, it will return the number of rows. The array passed into length can be any array expression, including array literals and results of other operators.

3.3.5.4 Tranpose

The transpose operator, using keyword **`transpose`**, will return an array that is a transposed version of the array passed in. The array passed into transpose can be any array expression, including array literals and results of other operators.

The transpose operator can be used on arrays of all types and is only defined for 2-dimensional arrays.

3.3.5.5 Map

The map operator, using the keyword **`map`**, applies a user-provided function to all elements in an array. For 2D arrays, it applies the function to all primitive elements in the matrix. The array passed into map can be any array expression, including array literals and results of other operators.

The user should pass a function that takes in one primitive element of the array and returns an element of the same type.

- For example, when applying **`map`** to an **`int[]`**, the user must pass a function that takes in an **`int`** and returns an **`int`**.
- When applying **`map`** to an **`int[][]`**, the user must also pass a function that both takes in an **`int`** and returns an **`int`**, since it will be applied to all primitive elements in the 2D array.

The map operator can be used on arrays of all types and dimensions.

3.3.5.6 Reduce

The reduce operator, using the keyword **`reduce`**, applies a user-provided aggregation function to elements in an array. For 2D arrays, it aggregates the primitive elements in each row into a single value. The array passed into reduce can be any array expression, including array literals and results of other operators.

This means that the result of the reduce operator will be one dimension less than the passed array.

- For example, passing a **`bool[]`** into the **`reduce`** operator will return a **`bool`**.
- When passing a **`double[][]`** into **`reduce`**, it will return a **`double[]`**, where each element represents an aggregated row in the original array.

The user should pass a function that takes in two primitive elements of the array and returns a single element of the same type.

- For example, when applying **`reduce`** to an **`int[]`**, the user must pass a function that takes two **`int`**s and returns an **`int`**.
- When applying **`reduce`** to an **`int[][]`**, the user must also pass a function that both takes in two **`int`**s and returns an **`int`**, since each row will be aggregated.

The reduce operator can be used on arrays of all types and dimensions.

3.3.5.7 Examples of Array Operators

Example 1: mathematical operators, access/slicing, transpose and length

```
int[4] x = [1,2,3,4];
int[4] y = [5,6,7,8];

// Element-wise addition
int[4] a = x + y; // [6, 8, 10, 12]

// Element-wise subtraction
int[4] b = y - x; // [4, 4, 4, 4]

// Element-wise multiplication
int[4] c = x * y; // [5, 12, 21, 32]

// Element-wise division
int[4] d = y / x; // [5, 3, 2, 2]

int[2][3] m = [1, 2, 3; 4, 5, 6;];
int[3][2] n = [1, 2; 3, 4; 5, 6;];

// Matrix multiplication
int[2][2] o = m @ n; // [22, 28; 49, 64;]
```

```
// Slicing
int[2][2] p = m[0:2, 0:2]; // [1, 2; 4, 5;]

// Transpose
int[3][2] t = transpose(m); // [1, 4; 2, 5; 3, 6;]

// Length
int n_rows = length(t); // 3
int n_cols = length(t[0]); // 2
```

Example 2: **`map`** and **`reduce`** operators

```
int add(int x, int y) {
    return x + y;
}

int add_ten(int x) {
    return x + 10;
}

void main() {
    int[3] a = [1, 2, 3];
    int[2][3] b = [1, 2, 3; 4, 5, 6;];

    // Map on 1D array
    int[3] a_plus_ten = map(a, add_ten); // [11, 12, 13]
    // Map on 2D array
    int[2][3] b_plus_ten = map(b, add_ten) // [11, 12, 13; 14, 15,
       16;]

    // Reduce on 1D array
    int a_sum = reduce(a, add); // 6
    // Reduce on 2D array
    int[2] b_row_sum = reduce(b, add); // Reduces each row -> [6, 15]
}
```

3.4 Declarations

All variables in CAPybara must be declared before use. All variable declarations need a type associated with the declaration; CAPybara is strongly typed. Arrays include their dimensions in their type, which must be known at compile-time. While the type is required, initialization of the variables — i.e. assigning a value to a variable — is not required.

The value assigned to the variable has to be of the given type. In CAPybara, only one variable can be declared in an expression. Variables can be declared and then initialized in any order, ie. declarations and statements can be interleaved in your program.

3.4.1 Declarations of primitive types

Example declarations without initialization:

```
    int x;
    double y;
    char c;
    bool b;
    string s;
```

Example declarations with initialization:

```
int x = 5;
double y = 4.5;
char c = 'a';
bool b = true;
string s = "hello";
```

Global primitive variables must be initialized with literal values, unlike local variables which can be initialized with expressions.

3.4.2 Declarations of Arrays

Example declarations without initialization:

```
int[4] ax;
double[2] ay;
char[3] ac;
bool[2] ab;
string[2] as;
int[2][4] mx;
```

Example declarations with initialization:

```
int[4] ax = [1, 2, 3, 4];
double[2] ay = [1.2, 3.4];
char[3] ac = ['c', 'a', 'p'];
bool[2] ab = [true, false];
string[2] as = ["hello", "world"];
int[2][4] mx = [1, 2, 3, 4; 5, 6, 7, 8;];
```

Array size must be declared explicitly, regardless if the array is initialized or not. Arrays cannot have a size of 0, meaning there are no empty arrays.

If global arrays are declared with an initialization, all elements must be primitive literals, unlike local array initializations which can be expressions.

When arrays are assigned to an expression that returns another array, which could be the result of an array operation, it must have the correct and expected shape. If not, an error will be thrown.

```
int[2][4] mx = [1, 2, 3, 4; 5, 6, 7, 8;];
int[3][2] ax = [3, 3; 4, 4; 5, 5;];
int[4][2] bx = [3, 3; 4, 4; 5, 5; 6, 6;];
ax = transpose(mx); // Error, because shape of transpose(mx) is [4][2]
bx = transpose(mx); // Valid, because bx and transpose(mx) have same
    shape
```

3.5 Statements and Expressions

Statements are formed by terminating an expression with a semicolon (;). They can be grouped into a statement block using curly braces ({), following the following statements:

- **if**
- **else**
- **while**
- **for**

3.5.1 If and Else

The `if` and `else` statements are conditional statements used to perform different actions based on different conditions. For any given `if` statement block, the `else` statement is optional. The statements within any `if` or `else` block must be enclosed with curly braces.

```
if (expression1) {
    statement1
} else {
    statement2
}
```

This example would be valid even if the `else` block is omitted.

3.5.2 While

The `while` loop statement executes all of the statements within its block repeatedly until a specified expression evaluates to false. This expression is enclosed within parentheses immediately following the while keyword, and the statement block is defined within curly braces.

```
while (expr)
{
    statement block
}
```

3.5.3 For

`for` loops execute a statement block until the second boolean expression evaluates to false. The syntax is as follows:

```
for (stmt1; expr2; expr3)
{
    statement block
}
```

The statement and expressions within the parentheses are defined as:

- `stmt1` is either an expression or a variable initialization, but it cannot be just a variable declaration.
- `expr2` is a boolean expression that will break the loop when evaluated to false.
- `expr3` is evaluated at the end of each iteration.

Example of typical usage of a for loop:

```
for (int i = 0; i < 5; i = i + 1)
{
    // statement block
}
```

3.5.4 Break

During looping statements, you may need to escape the execution flow earlier than the loop will complete otherwise. The `break` statement allows you to immediately escape the flow of execution in a loop at any point. Breaks cause the innermost loop to be escaped.

```
int i = 0;
while (true) {
   i = i + 1;
```

```
    if (i == 5) { // Program stops after 5 iterations
        break;
    }
}
```

3.6 Functions

The execution of CAPybara requires a main function which takes no arguments and has a void return type. A CAPybara program starts from the first line contained within the main function.

```
void main () {
    // program goes here
}
```

3.6.1 Function Declaration

A CAPybara function requires a return type, a name, and argument types and names. To declare a function, the return type is followed by the name of the function and parentheses containing the arguments types and names separated by commas. The body of the function is enclosed by curly braces — similar to C.

Functions can have `void` return type, meaning that the function does not return a value.

```
type func_name(type arg1, type arg2) {
    // function body goes here
}
```

To invoke a function, provide the name of the functions and values for each of the required arguments. A variable can be assigned to the return value of a function.

```
int sum_ints(int a, int b) {
    return a + b;
}

void main() {
    int x = 1;
    int y = 2;
    int z = sum_ints(x, y); // 3 will be assigned to z
}
```

Every type is passed and returned by value. Therefore, users should be aware that passing or returning an array induces a potentially expensive deep-copy of the array. Since arrays, like all types, are stack allocated, there is no notion of returning an array by reference (as the reference would be dangling).

Because arrays include their size in their type, there is no way to take an array of an unspecified size as an argument or return an array of an unspecified size (unlike in Java).

3.6.2 Library Functions

We provide a suite of built-in functions to print primitive types to standard output:

- `print_int()`
- `print_double()`
- `print_char()`
- `print_bool()`

- print_str()

Example program:

```
void main() {
    // Primitive Types
    int x = 1;
    double y = 100;
    bool b = False;
    char c = 'a';
    string s = "capybara";

    print_int(x); // prints 1
    print_double(y); // prints 100
    print_char(c); // prints a
    print_bool(b); // prints false
    print_str(s); // prints capybara
}
```

3.7 Scoping

CAPybara is a statically scoped language. The scope of a variable is defined using curly braces ({}), meaning that they exist only within the statement block in which they are declared.

Global variables are defined outside of the constraints of any curly braces, and are scoped to the entire program – they are accessible anywhere in the program after their definition. All functions in CAPybara are accessible globally – we do not support nested functions.

Variables and functions of the same name are forbidden from being declared in the same scope, regardless of type.

In global scope, global variable initializations must be with literals.

```
int y = 10;
int x = 9;

void print_y() {
    print_int(y); // y is accessible from this function
}

void main() {
    int x = 0;
    print_int(x); // 0
    print_y(); // 10
}
```

3.8 Example Programs

Example 1: Filling a square matrix with the identity matrix, returning 0 on success and 1 on failure.

```
void main() {
    int[5][5] m;

    int dimension = length(m);
    if (dimension != length(m[0])) {
   print_str("Error: Cannot form identity with non-square matrix");
    }
    for (int i = 0; i < dimension; i = i + 1) {
        for (int j = 0; j < dimension; j = j + 1) {
      if (i == j) {
         m[i][j] = 1;
      } else {
         m[i][j] = 0;
      }
        }
    }

    print_str("Success!");
}
```

Example 2: Finding the nth number of the Fibonacci sequence where the first and second terms are 0 and 1

```
int fibonacci_seq(int n) {
    int t1 = 0;
    int t2 = 1;
    if (n == 1) {
        return t1;
    }

    if (n == 2) {
   return t2;
    }

    for (int i = 3; i <= n; i = i + 1) {
        int temp = t2;
        t2 = t1 + t2;
        t1 = temp;
    }

    return t2;
}

int fibonacci_rec(int n) {
    if (n == 1) {
        return 0;
    }

    if (n == 2) {
        return 1;
    }

    return fibonacci_rec(n - 2) + fibonacci_rec(n - 1);
}

void main() {
    print_int(fibonacci_seq(37));
    print_int(fibonacci_rec(37));
}
```

4 Architectural Diagram

4.1 CAPybara Source File (.cap)

Source raw text file of the program.

4.2 Scanner

The scanner turns the source code file into a list of tokens.

4.3 Parser

The parser takes in the list of tokens and produces an abstract syntax tree (AST).

4.4 Semantic Checking

The semantic checker takes in an AST and produces a semantically-checked AST (SAST).

4.5 IR Generation

The IR code generator takes in a SAST and produces LLVM IR code.

4.6 Linking & Machine Code Generation

The linker, lli, JIT-compiles the LLVM IR code into assembly and links the machine code to the C Standard Library before executing the program.

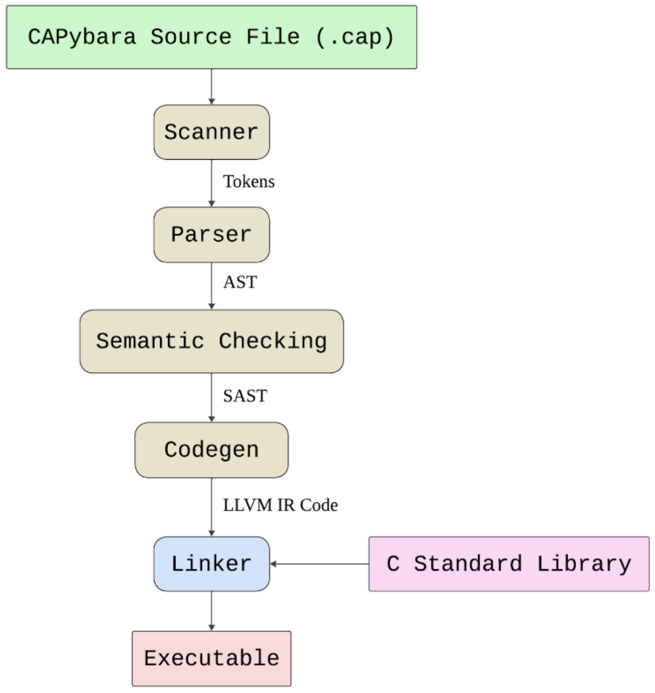

Figure 1: A descriptive caption for your image.

5 Test Plan

5.1 Sample Programs

We provide two examples of sample programs in section 3.8. The generated code for both programs can be found in the appendix under section 7.1.

5.2 Automated Testing

Our language features 47 unit test programs that validate all of the language's features through success and fail tests with expected results. Language features tested include:

Successful Tests Global variables Function calls Comments		
Printing & Variable Assignment Tests - `print_int()` - `print_double()` - `print_bool()` - `print_char()` - `print_str()` - Print 1D array - Print 2D array	**Basic Array Functions** - Entire array assignment - Single element assignment (`int[2][1] a = 3`) - Passing arrays as function parameters and returning arrays	**Array Operations** - Length - Transpose - Map - Reduce - Slice - Element-wise addition - Element-wise subtraction - Element-wise multiplication - Element-wise division - Scaling (array * scalar) - Matrix multiplication
Arithmetic & Conditional Operators - Mod - Comparisons (<, >, <=, >=, ==, !=) - And/Or - Not - Negation	**Operator Precedence**	**Conditionals and loops** - If-else statement (including no else) - While loop - For loop - Loop with a break, If statement nested in loop
Failing tests - Failing return - Failing loop — no boolean expression - Failing function call – wrong arguments - Failing function call – function DNE - Failing addition — mismatched types - Failing 1D array assignment – mismatched types - Failing 2D array assignment – mismatched lengths - Failing matrix mult — wrong dimensions - Failing array map — function w/ wrong argument type - Failing array element-wise add — wrong dimensions - Failing global var — has an expr on the RHS		

We provide `test_runner.sh`, a shell script that runs all tests and displays their status (PASS/FAIL).

6 Project Summary

6.1 Division of Work

The majority of the project was completed via live pair programming, with most/all group members meeting in person and programming together.

6.2 Takeaways

6.2.1 Chelsea

I learned a lot from this project, both on the technical side, as well as design and organization. At the start of our project, it was pretty difficult to envision how exactly we would implement some of the features we were discussing, given most of our limited experience with compilers. During the planning phase, we questioned everything, even if we were unsure of how this would actually translate to the code. I found that making these concrete decisions early on was extremely helpful in the long run, as we constantly referenced our LRM during the coding stage to remember what decisions we made regarding semantics, allowing us to focus on getting the code to work, instead of thinking about how we wanted to define this paradigm. At the same time, we didn't force ourselves to stick to these requirements, remaining flexible if the previously agreed upon semantics proved too complicated to implement.

As a group, I felt that we were very efficient during our meeting times – pair programming is extremely helpful. Since all components of the compiler lead to the next, there is at least some understanding of each component necessary to be able to work on a different component. By pair programming, we ensured that everyone understood all stages of the compiler, and this meant that there was less time needed to catch up on code that other people wrote. Additionally, debugging was much easier with more heads, especially since we all knew the code that was written.

6.2.2 Claudia

One of my main takeaways was the value of discussing our design decisions. At the start of the project, we spent a while trying to scope our project and discuss what features we should support or not. For example, we discussed the semantics of 2-dimensional map and reduce in depth, which led to discussing how arrays are stored in memory. Later on during the code generation phase, we had to be more clear about if arrays decay to a pointer to the underlying array, if they are always pointers, and passing/returning arrays. I am proud that we were able to figure out how to pass and return arrays by value at the end, because it was satisfying to figure out and also mirrors how C passes and returns structs.

In terms of timeline, as the group manager, I made sure that we met about twice a week throughout the semester which led to us finishing the project code with time to spare. The project overall went surprisingly smoothly, which I think was due to our frequent meetings, decision to pair-program pretty much the entire code, and also that we picked a language well-suited to our strengths. All of us are extremely familiar with C, so we had an easier time conceptualizing decisions such as pointers, stack allocation, and static-sized arrays. Designing a language that built upon concepts that we were familiar with allowed us to devote most of our energy to building the compiler phases.

6.2.3 Faustina

The main takeaway for me is the importance of consistent meetings. Because we met consistently from the beginning, our project was on track the whole time and there was minimal stress regarding meeting deadlines. I think knowing when to delay, such as waiting for the Micro C parser lectures before trying to implement our own parser was an efficient use of time, since having the background knowledge sped up our implementation by a lot.

Having clear example programs of our language also really helped because we could quickly refer to that to see how our syntax should look and what expected behaviors are. These could then be used as testing programs to verify correct behavior.

6.2.4 Albert

One takeaway is that pair-programming can be extremely productive, even if multiple people are watching someone else code. We quickly made progress by debugging together, debating small language design or implementation choices, and giving each other a refresher of the codebase when needed.

The biggest timesaver was "giving in" to the power of generative AI – cautiously of course. I learned how exceptional OpenAI's new reasoning models are at localizing obscure bugs (sometimes hidden in 1K+ LOC), generating small code snippets that don't require much editing if any, and explaining how we'd implement language features at a high level (and whether our design choices made sense or not). We made sure we fully understood any LLM-generated code we took, and discussed any LLM-generated advice before taking it. ChatGPT Plus is worth paying for!

6.2.5 Jake

One takeaway was experiencing the amount of time that goes into developing a programming language from top down. We spent weeks designing our language and making sure all the features would be suitable for implementation with OCaml and LLVM. Seeing our design come to fruition was really cool.

Additionally, like the others have mentioned we got to do pair programming which helped everyone know what was happening at all points during the project. Doing this helped us catch bugs faster, before we even tested the code and ensured that all the work was being done sequentially, as it seems would be most optimal when designing a programming language with OCaml. Consistent and frequent meetings also allowed for us to maintain knowledge over the codebase over the course of the semester, especially as it was getting larger during semantic analysis and code gen.

Seeing how the languages we use were compiled has given me a better understanding of the language's features and why things behave the way they do!

6.2.6 Mara

One takeaway is to get out of my comfort zone. While I've had some experience with multi-paradigm languages, I'd never ventured completely into the realm of functional programming. Namely, I had to significantly alter my narrower views on best practices for project structure, readability, and deduplication, since we were working in an entirely new paradigm. Partner Programming helped a lot with this, since for everyone to be on the same page, we have to be able to clearly express what we want to do.

I've had a bit of compiler experience before this class, so early on I helped with language design, and with making abstract problems concrete, and giving some rough predictions with regard to what would be more or less time consuming. Once we got into coding, I mainly helped debug logic and to come up with useful test cases.

6.3 Advice for Future Teams

We'd advise future groups to not overscope and be very intentional about what features to support and what not to support. The only parts of our language that we removed from our original scope were the pow operator and printing arrays. We found that aspects of the language that had a clear translation to Ocaml were easier to implement, such as operators. Remember that you can always add features later, but you'll waste time if you try implementing something and give up on it later. We found it useful to consult the TAs to better understand what features were more feasible for our language and what was difficult, considering we did not yet have a good idea of how to implement this when we were scoping.

When implementing the parsing phase, we ran into issues where we had two meanings for the same syntax. This was difficult to implement in context-free parsing, so we had to have slightly uglier syntax for matrices in order to keep the meanings unique when parsing. Some other topics that we spent time wrangling with were deciding the semantics for 2-dimensional map and reduce, allowing initialization in variable declarations, and definitely understanding how to allocate local and global arrays.

We had routine meetings twice a week throughout the semester which allowed us to finish the project before it was due and not need to cram into the last few days. We found that pair-programming for the whole project was helpful in order for everyone to fully understand the code at each point, because you need to understand the previous phase to write the next phase.

7 Appendix

7.1 Generated Code Examples

7.1.1 Example 1

```
; ModuleID = 'CAPybara'
source_filename = "CAPybara"

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

```
@fmt.1 = private unnamed_addr constant [4 x i8] c"%f\0A\00", align 1
@bool_fmt = private unnamed_addr constant [4 x i8] c"%s\0A\00", align
   1
@true_lit = private unnamed_addr constant [5 x i8] c"true\00", align 1
@false_lit = private unnamed_addr constant [6 x i8] c"false\00", align
    1
@fmt.2 = private unnamed_addr constant [4 x i8] c"%c\0A\00", align 1
@fmt.3 = private unnamed_addr constant [4 x i8] c"%s\0A\00", align 1
@.str = private unnamed_addr constant [51 x i8] c"Error: Cannot form
   identity with non-square matrix\00", align 1
@.str.4 = private unnamed_addr constant [9 x i8] c"Success!\00", align
    1

declare i32 @printf(i8*, ...)

define void @main() {
entry:
  %m = alloca [5 x [5 x i32]], align 4
  %dimension = alloca i32, align 4
  store i32 5, i32* %dimension, align 4
  %dimension1 = load i32, i32* %dimension, align 4
  %tmp = icmp ne i32 %dimension1, 5
  br i1 %tmp, label %then, label %else

then:                                              ; preds = %entry
  %printf = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([4
      x i8], [4 x i8]* @fmt.3, i32 0, i32 0), i8* getelementptr
     inbounds ([51 x i8], [51 x i8]* @.str, i32 0, i32 0))
  br label %if_end

else:                                              ; preds = %entry
  br label %if_end

if_end:                                            ; preds = %else, %
   then
  %i = alloca i32, align 4
  store i32 0, i32* %i, align 4
  br label %while

while:                                             ; preds = %
   while_end10, %if_end
  %i2 = load i32, i32* %i, align 4
  %dimension3 = load i32, i32* %dimension, align 4
  %tmp4 = icmp slt i32 %i2, %dimension3
  br i1 %tmp4, label %while_body, label %while_end

while_body:                                        ; preds = %while
  %j = alloca i32, align 4
  store i32 0, i32* %j, align 4
  br label %while5

while_end:                                         ; preds = %while
  %printf27 = call i32 (i8*, ...) @printf(i8* getelementptr inbounds
     ([4 x i8], [4 x i8]* @fmt.3, i32 0, i32 0), i8* getelementptr
     inbounds ([9 x i8], [9 x i8]* @.str.4, i32 0, i32 0))
  ret void
while5:                                           ; preds = %if_end22,
    %while_body
  %j6 = load i32, i32* %j, align 4
  %dimension7 = load i32, i32* %dimension, align 4
  %tmp8 = icmp slt i32 %j6, %dimension7
  br i1 %tmp8, label %while_body9, label %while_end10

while_body9:                                      ; preds = %while5
  %i11 = load i32, i32* %i, align 4
  %j12 = load i32, i32* %j, align 4
  %tmp13 = icmp eq i32 %i11, %j12
  br i1 %tmp13, label %then14, label %else17

while_end10:                                      ; preds = %while5
  %i25 = load i32, i32* %i, align 4
  %tmp26 = add i32 %i25, 1
  store i32 %tmp26, i32* %i, align 4
  br label %while

then14:                                           ; preds = %
   while_body9
  %i15 = load i32, i32* %i, align 4
  %j16 = load i32, i32* %j, align 4
  %arr_row = getelementptr inbounds [5 x [5 x i32]], [5 x [5 x i32]]*
     %m, i32 0, i32 %i15
  %arr_gep = getelementptr inbounds [5 x i32], [5 x i32]* %arr_row,
     i32 0, i32 %j16
  store i32 1, i32* %arr_gep, align 4
  br label %if_end22

else17:                                           ; preds = %
   while_body9
  %i18 = load i32, i32* %i, align 4
  %j19 = load i32, i32* %j, align 4
  %arr_row20 = getelementptr inbounds [5 x [5 x i32]], [5 x [5 x i32
     ]]* %m, i32 0, i32 %i18
  %arr_gep21 = getelementptr inbounds [5 x i32], [5 x i32]* %arr_row20
     , i32 0, i32 %j19
  store i32 0, i32* %arr_gep21, align 4
  br label %if_end22

if_end22:                                         ; preds = %else17, %
   then14
  %j23 = load i32, i32* %j, align 4
  %tmp24 = add i32 %j23, 1
  store i32 %tmp24, i32* %j, align 4
  br label %while5
}
```

7.1.2 Example 2

```
; ModuleID = 'CAPybara'
source_filename = "CAPybara"
```

```
@fmt = private unnamed_addr constant [4 x i8] c"%d\0A\00", align 1
@fmt.1 = private unnamed_addr constant [4 x i8] c"%f\0A\00", align 1
@bool_fmt = private unnamed_addr constant [4 x i8] c"%s\0A\00", align
   1
@true_lit = private unnamed_addr constant [5 x i8] c"true\00", align 1
@false_lit = private unnamed_addr constant [6 x i8] c"false\00", align
    1
@fmt.2 = private unnamed_addr constant [4 x i8] c"%c\0A\00", align 1
@fmt.3 = private unnamed_addr constant [4 x i8] c"%s\0A\00", align 1
@fmt.4 = private unnamed_addr constant [4 x i8] c"%d\0A\00", align 1
@fmt.5 = private unnamed_addr constant [4 x i8] c"%f\0A\00", align 1
@bool_fmt.6 = private unnamed_addr constant [4 x i8] c"%s\0A\00",
   align 1
@true_lit.7 = private unnamed_addr constant [5 x i8] c"true\00", align
    1
@false_lit.8 = private unnamed_addr constant [6 x i8] c"false\00",
   align 1
@fmt.9 = private unnamed_addr constant [4 x i8] c"%c\0A\00", align 1
@fmt.10 = private unnamed_addr constant [4 x i8] c"%s\0A\00", align 1
@fmt.11 = private unnamed_addr constant [4 x i8] c"%d\0A\00", align 1
@fmt.12 = private unnamed_addr constant [4 x i8] c"%f\0A\00", align 1
@bool_fmt.13 = private unnamed_addr constant [4 x i8] c"%s\0A\00",
   align 1
@true_lit.14 = private unnamed_addr constant [5 x i8] c"true\00",
   align 1
@false_lit.15 = private unnamed_addr constant [6 x i8] c"false\00",
   align 1
@fmt.16 = private unnamed_addr constant [4 x i8] c"%c\0A\00", align 1
@fmt.17 = private unnamed_addr constant [4 x i8] c"%s\0A\00", align 1

declare i32 @printf(i8*, ...)

define i32 @fibonacci_seq(i32 %n) {
entry:
  %n1 = alloca i32, align 4
  store i32 %n, i32* %n1, align 4
  %t1 = alloca i32, align 4
  store i32 0, i32* %t1, align 4
  %t2 = alloca i32, align 4
  store i32 1, i32* %t2, align 4
  %n2 = load i32, i32* %n1, align 4
  %tmp = icmp eq i32 %n2, 1
  br i1 %tmp, label %then, label %else

then:                                              ; preds = %entry
  %t13 = load i32, i32* %t1, align 4
  ret i32 %t13

else:                                              ; preds = %entry
  br label %if_end

if_end:                                            ; preds = %else
  %n4 = load i32, i32* %n1, align 4
  %tmp5 = icmp eq i32 %n4, 2
  br i1 %tmp5, label %then6, label %else8
```

```
then6:                                          ; preds = %if_end
  %t27 = load i32, i32* %t2, align 4
  ret i32 %t27

else8:                                          ; preds = %if_end
  br label %if_end9

if_end9:                                        ; preds = %else8
  %i = alloca i32, align 4
  store i32 3, i32* %i, align 4
  br label %while

while:                                          ; preds = %
   while_body, %if_end9
  %i10 = load i32, i32* %i, align 4
  %n11 = load i32, i32* %n1, align 4
  %tmp12 = icmp sle i32 %i10, %n11
  br i1 %tmp12, label %while_body, label %while_end

while_body:                                     ; preds = %while
  %temp = load i32, i32* %t2, align 4
  %temp14 = alloca i32, align 4
  store i32 %temp, i32* %temp14, align 4
  %t115 = load i32, i32* %t1, align 4
  %t216 = load i32, i32* %t2, align 4
  %tmp17 = add i32 %t115, %t216
  store i32 %tmp17, i32* %t2, align 4
  %temp18 = load i32, i32* %temp14, align 4
  store i32 %temp18, i32* %t1, align 4
  %i19 = load i32, i32* %i, align 4
  %tmp20 = add i32 %i19, 1
  store i32 %tmp20, i32* %i, align 4
  br label %while

while_end:                                      ; preds = %while
  %t221 = load i32, i32* %t2, align 4
  ret i32 %t221
}

define i32 @fibonacci_rec(i32 %n) {
entry:
  %n1 = alloca i32, align 4
  store i32 %n, i32* %n1, align 4
  %n2 = load i32, i32* %n1, align 4
  %tmp = icmp eq i32 %n2, 1
  br i1 %tmp, label %then, label %else

then:                                           ; preds = %entry
  ret i32 0

else:                                           ; preds = %entry
  br label %if_end

if_end:                                         ; preds = %else
  %n3 = load i32, i32* %n1, align 4
  %tmp4 = icmp eq i32 %n3, 2
  br i1 %tmp4, label %then5, label %else6

then5:                                            ; preds = %if_end
  ret i32 1

else6:                                            ; preds = %if_end
  br label %if_end7

if_end7:                                          ; preds = %else6
  %n8 = load i32, i32* %n1, align 4
  %tmp9 = sub i32 %n8, 2
  %fibonacci_rec_result = call i32 @fibonacci_rec(i32 %tmp9)
  %n10 = load i32, i32* %n1, align 4
  %tmp11 = sub i32 %n10, 1
  %fibonacci_rec_result12 = call i32 @fibonacci_rec(i32 %tmp11)
  %tmp13 = add i32 %fibonacci_rec_result, %fibonacci_rec_result12
  ret i32 %tmp13
}

define void @main() {
entry:
  %fibonacci_seq_result = call i32 @fibonacci_seq(i32 37)
  %printf = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([4
       x i8], [4 x i8]* @fmt.11, i32 0, i32 0), i32 %
      fibonacci_seq_result)
  %fibonacci_rec_result = call i32 @fibonacci_rec(i32 37)
  %printf1 = call i32 (i8*, ...) @printf(i8* getelementptr inbounds
      ([4 x i8], [4 x i8]* @fmt.11, i32 0, i32 0), i32 %
      fibonacci_rec_result)
  ret void
}
```

7.2 GitHub Link

Our GitHub repository can be found at this link: https://github.com/chelseasoemitro/CAPybara

7.3 scanner.mll

```
(* Ocamllex scanner for CAPybara *)

{ open Capyparse }

let digit = ['0'-'9']
let letter = ['a'-'z' 'A'-'Z']
let ascii_char = [' '-'!'] | ['#'-'['] | [']'-'~']  (* exclude
   backslash and double quotes *)
let escape_char = '\\' ['\\' 'n' 't' 'r' ''' '"' ]

rule token = parse
  [' ' '\t' '\r' '\n'] { token lexbuf }      (* Whitespace *)

(* Comments *)
| "//"     { singlecomment lexbuf}
| "/*"     { multicomment lexbuf }
```

```
(* Parentheses and brackets *)
| '('      { L_PAREN }
| ')'      { R_PAREN }
| '{'      { L_CBRACE }
| '}'      { R_CBRACE }
| '['      { L_SQBRACE }
| ']'      { R_SQBRACE }

(* Delimiters *)
| ';'      { SEMI }
| ','      { COMMA }

(* Arithmetic operators *)
| '+'      { PLUS }
| '-'      { MINUS }
| '*'      { MULT }
| '/'      { DIV }
| '%'      { MOD }

(* Assignment *)
| '='      { ASSIGN }

(* Relational operators *)
| "=="     { EQ }
| "!="     { NEQ }
| '<'      { LT }
| "<="     { LE }
| '>'      { GT }
| ">="     { GE }

(* Logical operators *)
| "!"      { NOT }
| "&&"     { AND }
| "||"     { OR }

(* Statements and function keywords *)
| "if"     { IF }
| "else"   { ELSE }
| "while"  { WHILE }
| "for"    { FOR }
| "break"  { BREAK }
| "return" { RETURN }
| "void"   { VOID }

(* Primitive types *)
| "int"    { INT }
| "double" { DOUBLE }
| "bool"   { BOOL }
| "char"   { CHAR }
| "string" { STRING }

(* Literals *)
| digit+ as lem                              { INT_LIT(
   int_of_string lem) }
| digit+ ('.' digit+)? as lem                { DOUBLE_LIT(
   float_of_string lem) }
```

```
| "true"                                          { BOOL_LIT(true) }
| "false"                                         { BOOL_LIT(false) }
| '\'' ((ascii_char | escape_char) as lem) '\'' { CHAR_LIT(String.get
   (Scanf.unescaped lem) 0)}
| '"' ((ascii_char | escape_char)* as lem) '"'  { STRING_LIT(Scanf.
   unescaped lem) }

(* Array operators *)
| ':'      { COLON }
| '@'      { MMULT }
| "length"    { LENGTH }
| "transpose" { TRANSPOSE }
| "map"       { MAP }
| "reduce"    { REDUCE }

| letter (digit | letter | '_')* as lem { ID(lem) }
| eof { EOF }
| _ as char { raise (Failure("illegal character " ^ Char.escaped char)
   ) }


and singlecomment = parse
'\n'    { token lexbuf }
| eof   { EOF }
| _     { singlecomment lexbuf }

and multicomment = parse
  "*/" { token lexbuf }
| _    { multicomment lexbuf }
```

7.4 capyparse.mly

```
/* Ocamlyacc parser for CAPybara */

%{
open Ast
%}

/* Parentheses and brackets */
%token L_PAREN R_PAREN L_CBRACE R_CBRACE L_SQBRACE R_SQBRACE

/* Delimiters */
%token SEMI COMMA

/* Arithmetic operators */
%token PLUS MINUS MULT DIV MOD

/* Assignment */
%token ASSIGN

/* Relational operators */
%token EQ NEQ LT LE GT GE

/* Logical operators */
%token NOT AND OR
```

```
/* Statements and function keywords */
%token IF ELSE WHILE FOR BREAK RETURN VOID

/* Primitive types */
%token INT DOUBLE BOOL CHAR STRING

/* Literals */
%token <int> INT_LIT
%token <float> DOUBLE_LIT
%token <bool> BOOL_LIT
%token <char> CHAR_LIT
%token <string> STRING_LIT

/* Array Operators */
%token COLON MMULT LENGTH TRANSPOSE MAP REDUCE

%token <string> ID
%token EOF

%start program
%type <Ast.program> program

/* lowest precedence */
%nonassoc NO_ELSE
%nonassoc ELSE
%right ASSIGN
%left OR
%left AND
%left EQ NEQ
%left LT GT LE GE
%left PLUS MINUS
%left MULT DIV MOD
%left MMULT
%left COLON
%left LENGTH TRANSPOSE MAP REDUCE
%right NOT NEG
%nonassoc L_PAREN R_PAREN L_SQBRACE R_SQBRACE
/* highest precedence */

%%

program:
  decls EOF { $1 }

decls:
   /* nothing */           { ([], [])                  }
 | vdecl_stmt SEMI decls  { (($1 :: fst $3), snd $3) }  /* vdecls here
     are global */
 | fdecl decls            { (fst $2, ($1 :: snd $2)) }

/* int x */
vdecl:
  typ ID { ($1, $2) }

base_typ:
    INT            { Int     }
  | DOUBLE         { Double  }
  | BOOL           { Bool    }
  | CHAR           { Char    }
  | STRING         { String  }

typ:
    base_typ       { $1 }
  | VOID           { Void    }
  | base_typ L_SQBRACE INT_LIT R_SQBRACE L_SQBRACE INT_LIT R_SQBRACE {
      Arr2D($1, $3, $6) }
  | base_typ L_SQBRACE INT_LIT R_SQBRACE           { Arr1D($1, $3)
         }

/* fdecl */
fdecl:
  typ ID L_PAREN formals_opt R_PAREN L_CBRACE stmt_list R_CBRACE
  {
    {
      rtyp=$1;
      fname=$2;
      formals=$4;
      body=$7
    }
  }

/* formals_opt */
formals_opt:
  /*nothing*/     { [] }
  | formals_list  { $1 }

formals_list:
  vdecl  { [$1] }
  | vdecl COMMA formals_list  { $1::$3 }

/* hi im coming - albert */
stmt_list:
  /* nothing */ { [] }
  | stmt stmt_list        { $1::$2 }

stmt:
    expr SEMI                                  { Expr $1         }
  | L_CBRACE stmt_list R_CBRACE                { Block $2        }
  /* if (condition) { block1} else {block2} */
  /* if (condition) stmt else stmt */
  | IF L_PAREN expr R_PAREN stmt %prec NO_ELSE          { If($3, $5,
     Block ([])) }
  | IF L_PAREN expr R_PAREN stmt ELSE stmt              { If($3, $5,
     $7)      }
  | FOR L_PAREN stmt expr SEMI expr R_PAREN stmt   { For($3, $4, $6,
     $8) }
  | WHILE L_PAREN expr R_PAREN stmt                     { While ($3,
     $5)      }
  /* return */
  | RETURN expr SEMI                         { Return $2 }
  | BREAK SEMI                               { Break     }
  | vdecl_stmt SEMI                        { VDecl($1) }

vdecl_stmt:
    typ ID              { BindDecl($1, $2)       }
  | typ ID ASSIGN expr  { BindInit(($1, $2), $4) }

expr_list:
  expr { [$1] }
  | expr COMMA expr_list { $1::$3 }

expr_list_2D:
  expr_list SEMI { [$1] }
  | expr_list SEMI expr_list_2D { $1::$3 }

expr:
   INT_LIT            { IntLit($1)    }
  | DOUBLE_LIT        { DoubleLit($1) }
  | BOOL_LIT          { BoolLit($1)   }
  | CHAR_LIT          { CharLit($1)   }
  | STRING_LIT        { StringLit($1) }
  | L_SQBRACE expr_list R_SQBRACE        { Arr1DLit($2)  }
  | L_SQBRACE expr_list_2D R_SQBRACE   { Arr2DLit($2)  }
  | ID               { Id($1)                 }
  | expr PLUS   expr { Binop($1, Add,   $3)   }
  | expr MINUS  expr { Binop($1, Sub,   $3)   }
  | expr MULT   expr { Binop($1, Mult,  $3)   }
  | expr DIV    expr { Binop ($1, Div, $3)    }
  | expr MOD    expr { Binop($1, Mod, $3)     }
  | expr EQ     expr { Binop($1, Equal, $3)   }
  | expr NEQ    expr { Binop($1, Neq, $3)     }
  | expr LT     expr { Binop($1, Lt,  $3)     }
  | expr GT     expr { Binop($1, Gt, $3)      }
  | expr LE     expr { Binop($1, Le, $3)      }
  | expr GE     expr { Binop($1, Ge, $3)      }
  | expr AND    expr { Binop($1, And,   $3)   }
  | expr OR     expr { Binop($1, Or,    $3)   }
  | expr MMULT  expr { Binop($1, Mmult, $3)   }
  | NOT         expr          { Uop(Not, $2)    }
  | MINUS expr %prec NEG      { Uop(Neg, $2)    }
  | ID ASSIGN expr            { Assign($1, $3)  }
  | L_PAREN expr R_PAREN      { $2              }
  /* call */
  | ID L_PAREN args_opt R_PAREN { Call($1, $3)  }
  /* array */
  | ID L_SQBRACE expr R_SQBRACE ASSIGN expr
                                        { Arr1DAssign($1, $3, $6)
          }
  | ID L_SQBRACE expr R_SQBRACE L_SQBRACE expr R_SQBRACE ASSIGN expr
                { Arr2DAssign($1, $3, $6, $9 )  }
  | ID L_SQBRACE expr R_SQBRACE
                                                        { Arr1DAccess($1,
      $3)           }
  | ID L_SQBRACE expr R_SQBRACE L_SQBRACE expr R_SQBRACE
                            { Arr2DAccess($1, $3, $6)     }
  | LENGTH L_PAREN expr R_PAREN
                                                        { ArrUop(Length,
    $3)            }
  | TRANSPOSE L_PAREN expr R_PAREN
                                                    { ArrUop(Transpose,
    $3)         }
  | MAP L_PAREN expr COMMA ID R_PAREN
                                                  { ArrOp(Map, $3, $5)
                }
  | REDUCE L_PAREN expr COMMA ID R_PAREN
                                                { ArrOp(Reduce, $3, $5)
            }
  | ID L_SQBRACE INT_LIT COLON INT_LIT R_SQBRACE
                                           { Arr1DSlice($1, $3, $5)        }
  | ID L_SQBRACE INT_LIT COLON INT_LIT COMMA INT_LIT COLON INT_LIT
    R_SQBRACE    { Arr2DSlice($1, $3, $5, $7, $9) }

/* args_opt*/
args_opt:
  /*nothing*/ { [] }
  | args { $1 }

args:
  expr  { [$1] }
  | expr COMMA args { $1::$3 }
```

7.5 ast.ml

```
type bop = Add | Sub | Mult | Div | Mod |
            Equal | Neq | Lt | Gt | Le | Ge |
            And | Or |
            Mmult

type uop = Not | Neg

type arrop = Map | Reduce

type arruop = Length | Transpose

type typ = Int | Double | Bool | Char | String | Arr1D of typ * int |
   Arr2D of typ * int * int | Void

type expr =
  | IntLit of int
  | DoubleLit of float
  | BoolLit of bool
  | CharLit of char
  | StringLit of string
  | Arr1DLit of expr list
  | Arr2DLit of expr list list
  | Id of string
  | Binop of expr * bop * expr
  | Uop of uop * expr
  | Assign of string * expr
  | Call of string * expr list
  | Arr1DAssign of string * expr * expr              (* string: array
     id, expr: index, expr: value*)
  | Arr2DAssign of string * expr * expr * expr        (* string: array
      id, expr: row index, expr: col index, expr: value *)
  | Arr1DAccess of string * expr                      (* string: array
      id, expr: index *)
  | Arr2DAccess of string * expr * expr               (* string: array
      id, expr: row index, expr: col index *)
  | ArrUop of arruop * expr                           (* arruop, expr:
      array *)
  | ArrOp of arrop * expr * string                    (* arrop, expr:
     array, string: function ptr id*)
  | Arr1DSlice of string * int * int                  (* string: array
      id, int: start index, int: end index *)
  | Arr2DSlice of string * int * int * int * int      (* string: array
      id, int: start row index, int: end row index, int: start col
     index, int: end col index *)
  | NoExpr                                            (* for empty
     else statements *)

type bind_decl = typ * string
type bind_init = bind_decl * expr

type bind =
  BindDecl of bind_decl
| BindInit of bind_init


type stmt =
  | Block of stmt list
  | Expr of expr
  | If of expr * stmt * stmt
  | For of stmt * expr * expr * stmt
  | While of expr * stmt
  | Return of expr
  | Break
  | VDecl of bind


(* func_def: ret_typ fname formals locals body *)
type func_def = {
  rtyp: typ;
  fname: string;
  formals: bind_decl list; (* parameters *)
  body: stmt list;
}

type program = bind list * func_def list

(* Pretty-printing functions *)
let string_of_op = function
    Add -> "+"
  | Sub -> "-"
  | Equal -> "=="
  | Neq -> "!="
  | Lt -> "<"
  | Gt -> ">"
  | Le -> "<="
  | Ge -> ">="
  | And -> "&&"
  | Or -> "||"
  | Mult -> "*"
  | Div -> "/"
  | Mod -> "%"
  | Mmult -> "@"

let string_of_uop = function
    Not -> "!"
  | Neg -> "-"

let string_of_arrop = function
    Map -> "map"
  | Reduce -> "reduce"

let string_of_arruop = function
    Length -> "length"
  | Transpose -> "transpose"

  let rec string_of_expr = function
  | IntLit(l) -> string_of_int l
  | DoubleLit(l) -> string_of_float l
  | BoolLit(true) -> "true"
  | BoolLit(false) -> "false"
  | CharLit(c) -> "\'" ^ String.make 1 c ^ "\'"
  | StringLit(s) -> "\"" ^ s ^ "\""
  | Arr1DLit(a) ->
      "[" ^ String.concat ", " (List.map string_of_expr a) ^ "]"
  | Arr2DLit(a) ->
      "[" ^ String.concat ", " (List.map (fun row ->
          String.concat ", " (List.map string_of_expr row) ^ ";"
      ) a) ^ "]"
  | Id(s) -> s
  | Binop(e1, o, e2) ->
      string_of_expr e1 ^ " " ^ string_of_op o ^ " " ^ string_of_expr
         e2
  | Uop(o, e) ->
      string_of_uop o ^ " " ^ string_of_expr e
  | Assign(v, e) -> v ^ " = " ^ string_of_expr e
  | Call(f, el) ->
      f ^ "(" ^ String.concat ", " (List.map string_of_expr el) ^ ")"
  | ArrOp(o, s1, s2) ->
      string_of_arrop o ^ "(" ^ string_of_expr s1 ^ ", " ^ s2 ^ ")"
  | ArrUop(o, a) ->
      string_of_arruop o ^ "(" ^ string_of_expr a ^ ")"
  | Arr1DAssign(a, idx, value) ->
      a ^ "[" ^ string_of_expr idx ^ "] = " ^ string_of_expr value
  | Arr2DAssign(a, idx1, idx2, value) ->
      a ^ "[" ^ string_of_expr idx1 ^ "][" ^ string_of_expr idx2 ^ "]
         = " ^ string_of_expr value
  | Arr1DAccess(a, idx) ->
      a ^ "[" ^ string_of_expr idx ^ "]"
  | Arr2DAccess(a, idx1, idx2) ->
      a ^ "[" ^ string_of_expr idx1 ^ "][" ^ string_of_expr idx2 ^ "]"
  | Arr1DSlice(a, start_idx, end_idx) ->
```

```
      a ^ "[" ^ string_of_int start_idx ^ ":" ^ string_of_int end_idx
          ^ "]"
  | Arr2DSlice(a, start_row, end_row, start_col, end_col) ->
    a ^ "[" ^ string_of_int start_row ^ ":" ^ string_of_int end_row ^
       ", "
           ^ string_of_int start_col ^ ":" ^ string_of_int end_col ^
              "]"
  | NoExpr -> ""


let rec string_of_typ = function
    Int -> "int"
  | Double -> "double"
  | Bool -> "bool"
  | Char -> "char"
  | String -> "string"
  | Arr1D (t, i) ->
    let t_str = string_of_typ t in
    t_str ^ "[" ^ string_of_int i ^ "]"
  | Arr2D (t, i1, i2) ->
    let t_str = string_of_typ t in
    t_str ^ "[" ^ string_of_int i1 ^ "][" ^ string_of_int i2 ^ "]"
  | Void -> "void"


let string_of_vdecl = function
BindDecl(decl) -> string_of_typ (fst decl) ^ " " ^ (snd decl) ^ ";\n"
| BindInit(decl, exp) -> string_of_typ (fst decl) ^ " " ^ (snd decl) ^
    " = " ^ string_of_expr exp ^ ";\n"


let rec string_of_stmt = function
    Block(stmts) ->
    "{\n" ^ String.concat "" (List.map string_of_stmt stmts) ^ "}\n"
  | Expr(expr) -> string_of_expr expr ^ ";\n"
  | If(e, s1, s2) ->  "if (" ^ string_of_expr e ^ ")\n" ^
                      string_of_stmt s1 ^ "else\n" ^ string_of_stmt s2
  | While(e, s) -> "while (" ^ string_of_expr e ^ ") " ^
     string_of_stmt s
  | For(s1, e1, e2, s2) -> "for (" ^ string_of_stmt s1 ^
     string_of_expr e1 ^ "; " ^ string_of_expr e2 ^ ") " ^
     string_of_stmt s2
  | Return(expr) -> "return " ^ string_of_expr expr ^ ";\n"
  | Break -> "break;\n"
  | VDecl(vdecl) -> string_of_vdecl vdecl

let string_of_fdecl fdecl =
  string_of_typ fdecl.rtyp ^ " " ^
  fdecl.fname ^ "(" ^ String.concat ", " (List.map snd fdecl.formals)
     ^
  ")\n{\n" ^
  (* String.concat "" (List.map string_of_vdecl fdecl.locals) ^ *)
  String.concat "" (List.map string_of_stmt fdecl.body) ^
  "}\n"
  let string_of_program (vars, funcs) =
    "\n\nParsed program: \n\n" ^
    String.concat "" (List.map string_of_vdecl vars) ^ "\n" ^
    String.concat "\n" (List.map string_of_fdecl funcs)
```

7.6 sast.ml

```
open Ast

type sexpr = typ * sx
and sx =
  | SIntLit of int
  | SDoubleLit of float
  | SBoolLit of bool
  | SCharLit of char
  | SStringLit of string
  | SArr1DLit of sexpr list
  | SArr2DLit of sexpr list list
  | SId of string
  | SBinop of sexpr * bop * sexpr
  | SUop of uop * sexpr
  | SAssign of string * sexpr
  | SCall of string * sexpr list
  | SArr1DAssign of string * sexpr * sexpr           (* string: array
    id, expr: index, expr: value*)
  | SArr2DAssign of string * sexpr * sexpr * sexpr   (* string: array
    id, expr: row index, expr: col index, expr: value *)
  | SArr1DAccess of string * sexpr                   (* string: array
    id, expr: index *)
  | SArr2DAccess of string * sexpr * sexpr           (* string: array
    id, expr: row index, expr: col index *)
  | SArrUop of arruop * sexpr                        (* arruop, expr:
    array *)
  | SArrOp of arrop * sexpr * string                 (* arrop, expr:
    array, string: function ptr id*)
  | SArr1DSlice of string * int * int                (* string: array
    id, expr: start index, expr: end index *)
  | SArr2DSlice of string * int * int * int * int    (* string: array
    id, expr: start row index, expr: end row index, expr: start col
    index, expr: end col index *)
  | SNoExpr                                          (* for empty else
     statements *)


type sbind_decl = typ * string
type sbind_init = sbind_decl * sexpr

type sbind =
  SBindDecl of sbind_decl
| SBindInit of sbind_init

type sstmt =
  | SBlock of sstmt list
  | SExpr of sexpr
  | SIf of sexpr * sstmt * sstmt
  | SFor of sstmt * sexpr * sexpr * sstmt
  | SWhile of sexpr * sstmt
  | SReturn of sexpr
  | SBreak
  | SVDecl of sbind


(* func_def: ret_typ fname formals locals body *)
type sfunc_def = {
  srtyp: typ;
  sfname: string;
  sformals: sbind_decl list; (* parameters *)
  sbody: sstmt list;
}

type sprogram = sbind list * sfunc_def list

(* Pretty-printing functions *)
let string_of_sop = function
    Add -> "+"
  | Sub -> "-"
  | Equal -> "=="
  | Neq -> "!="
  | Lt -> "<"
  | Gt -> ">"
  | Le -> "<="
  | Ge -> ">="
  | And -> "&&"
  | Or -> "||"
  | Mult -> "*"
  | Div -> "/"
  | Mod -> "%"
  | Mmult -> "@"

let string_of_suop = function
    Not -> "!"
  | Neg -> "-"

let string_of_sarrop = function
    Map -> "map"
  | Reduce -> "reduce"

let string_of_sarruop = function
    Length -> "length"
  | Transpose -> "transpose"

  let rec string_of_sexpr (t, e) =
  "(" ^ string_of_typ t ^ " : " ^ (
    match e with
   SIntLit(l) -> string_of_int l
  | SDoubleLit(l) -> string_of_float l
  | SBoolLit(true) -> "true"
  | SBoolLit(false) -> "false"
  | SCharLit(c) -> String.make 1 c
  | SStringLit(s) -> "\"" ^ s ^ "\""
  | SArr1DLit(a) ->
```

```
      "[" ^ String.concat ", " (List.map string_of_sexpr a) ^ "]"
  | SArr2DLit(a) ->
      "[" ^ String.concat " " (List.map (fun row ->
          String.concat ", " (List.map string_of_sexpr row) ^ ";"
      ) a) ^ "]"
  | SId(s) -> s
  | SBinop(e1, o, e2) ->
      string_of_sexpr e1 ^ " " ^ string_of_sop o ^ " " ^
          string_of_sexpr e2
  | SUop(o, e) ->
      string_of_suop o ^ " " ^ string_of_sexpr e
  | SAssign(v, e) -> v ^ " = " ^ string_of_sexpr e
  | SCall(f, el) ->
      f ^ "(" ^ String.concat ", " (List.map string_of_sexpr el) ^ ")"
  | SArrOp(o, s1, s2) ->
      string_of_sarrop o ^ "(" ^ string_of_sexpr s1 ^ ", " ^ s2 ^ ")"
  | SArrUop(o, a) ->
      string_of_sarruop o ^ "(" ^ string_of_sexpr a ^ ")"
  | SArr1DAssign(a, idx, value) ->
      a ^ "[" ^ string_of_sexpr idx ^ "] = " ^ string_of_sexpr value
  | SArr2DAssign(a, idx1, idx2, value) ->
      a ^ "[" ^ string_of_sexpr idx1 ^ "][" ^ string_of_sexpr idx2 ^ "
          ] = " ^ string_of_sexpr value
  | SArr1DAccess(a, idx) ->
      a ^ "[" ^ string_of_sexpr idx ^ "]"
  | SArr2DAccess(a, idx1, idx2) ->
      a ^ "[" ^ string_of_sexpr idx1 ^ "][" ^ string_of_sexpr idx2 ^ "
          ]"
  | SArr1DSlice(a, start_idx, end_idx) ->
      a ^ "[" ^ string_of_int start_idx ^ ":" ^ string_of_int end_idx
          ^ "]"
  | SArr2DSlice(a, start_row, end_row, start_col, end_col) ->
      a ^ "[" ^ string_of_int start_row ^ ":" ^ string_of_int end_row
          ^ ", "
              ^ string_of_int start_col ^ ":" ^ string_of_int end_col
                  ^ "]"
  | SNoExpr -> ""
  ) ^ ")"


let string_of_svdecl = function
SBindDecl(sdecl) -> string_of_typ (fst sdecl) ^ " " ^ (snd sdecl) ^ "
    ;\n"
| SBindInit(sdecl, sexpr) -> string_of_typ (fst sdecl) ^ " " ^ (snd
    sdecl) ^ " = " ^ string_of_sexpr sexpr ^ ";\n"


let rec string_of_sstmt = function
    SBlock(sstmts) ->
    "{\n" ^ String.concat "" (List.map string_of_sstmt sstmts) ^ "}\n"
  | SExpr(sexpr) -> string_of_sexpr sexpr ^ ";\n"
  | SIf(e, s1, s2) ->  "if (" ^ string_of_sexpr e ^ ")\n" ^
                      string_of_sstmt s1 ^ "else\n" ^ string_of_sstmt
                          s2
  | SWhile(e, s) -> "while (" ^ string_of_sexpr e ^ ") " ^
      string_of_sstmt s
  | SFor(s1, e1, e2, s2) -> "for (" ^ string_of_sstmt s1 ^
      string_of_sexpr e1 ^ "; " ^ string_of_sexpr e2 ^ ") " ^
      string_of_sstmt s2
  | SReturn(sexpr) -> "return " ^ string_of_sexpr sexpr ^ ";\n"
  | SBreak -> "break;\n"
  | SVDecl(svdecl) -> string_of_svdecl svdecl

let string_of_sfdecl sfdecl =
  string_of_typ sfdecl.srtyp ^ " " ^
  sfdecl.sfname ^ "(" ^ String.concat ", " (List.map snd sfdecl.
      sformals) ^
  ")\n{\n" ^
  (* String.concat "" (List.map string_of_svdecl sfdecl.locals) ^ *)
  String.concat "" (List.map string_of_sstmt sfdecl.sbody) ^
  "}\n"


  let string_of_sprogram (vars, funcs) =
    "\n\nSemantically-checked program: \n\n" ^
    String.concat "" (List.map string_of_svdecl vars) ^ "\n" ^
    String.concat "\n" (List.map string_of_sfdecl funcs)
```

7.7 semant.ml

```
(* Semantic checking for the CAPybara compiler *)

open Ast
open Sast

module StringMap = Map.Make(String)

(* Semantic checking of the AST. Returns an SAST if successful,
   throws an exception if something is wrong.

   Check each global variable, then check each function *)

let check (globals, functions) =

  (* Ensure global variables do not bind to void type *)
  let check_not_void exceptf = function
    (Void, n) -> raise (Failure (exceptf n))
    | _ -> ()
  in

  (* Verify a list of bindings has no duplicate names *)
  let check_binds (kind : string) (binds : (typ * string) list) =
    List.iter (check_not_void (fun n -> "illegal void variable: " ^ n)
       ) binds;
    let rec dups = function
        [] -> ()
      |  ((_,n1) :: (_,n2) :: _) when n1 = n2 ->
        raise (Failure ("duplicate " ^ kind ^ " " ^ n1))
      | _ :: t -> dups t
    in dups (List.sort (fun (_,a) (_,b) -> compare a b) binds)
  in
```

```
(* Raise an exception if the given rvalue type cannot be assigned to
  the given lvalue type *)
let check_assign lvaluet rvaluet err =
  if lvaluet <> rvaluet then (raise (Failure err));
  lvaluet
in

(* Return a variable from our local symbol table *)
let type_of_identifier s symtab =
  try StringMap.find s symtab
  with Not_found -> raise (Failure ("undeclared identifier " ^ s))
in

(* Extract binds from global vars. We turn them to sbinds at the end
   . *)
let rec extract_global_binds = function
  [] -> []
| hd::tl -> match hd with
  BindDecl(d) -> d::extract_global_binds tl
  | BindInit(d, e) -> d::extract_global_binds tl
in

(* Build list of global binds for symbol table *)
let global_binds = extract_global_binds globals in

(* Check global variables: make sure no globals duplicate, make sure
    bind inits are good *)
check_binds "global" (global_binds);

(* Collect function declarations for built-in functions: no bodies
   *)
let built_in_decls =
  let add_built_in map (name, ty) = StringMap.add name {
    rtyp = Void;
    fname = name;
    formals = [(ty, "x")];
    body = []
  } map
  in List.fold_left add_built_in StringMap.empty [
    ("print_int", Int);
    ("print_double", Double);
    ("print_char", Char);
    ("print_bool", Bool);
    ("print_str", String);
    (* ("print_arr", List) -> can't do this bc don't know size of
       array *)
  ]
in

(* Add function name to symbol table *)
let add_func map fd =
  let built_in_err = "function " ^ fd.fname ^ " may not be defined"
  and dup_err = "duplicate function " ^ fd.fname
  and make_err er = raise (Failure er)
  and n = fd.fname (* Name of the function *)
  in match fd with (* No duplicate functions or redefinitions of
      built-ins *)
     _ when StringMap.mem n built_in_decls -> make_err built_in_err
   | _ when StringMap.mem n map -> make_err dup_err
   | _ ->  StringMap.add n fd map
in

(* Collect all function names into one symbol table *)
let function_decls = List.fold_left add_func built_in_decls
    functions
in

(* Return a function from our symbol table *)
let find_func s =
  try StringMap.find s function_decls
  with Not_found -> raise (Failure ("unrecognized function " ^ s))
in

let _ = find_func "main" in (* Ensure "main" is defined *)

(* Return a semantically-checked expression, i.e., with a type *)
let rec check_expr e symtab =
  match e with
  | IntLit l -> (Int, SIntLit l)
  | DoubleLit l -> (Double, SDoubleLit l)
  | BoolLit l -> (Bool, SBoolLit l)
  | CharLit l -> (Char, SCharLit l)
  | StringLit l -> (String, SStringLit l)
  | Arr1DLit l ->
    let checked_elements = List.map (fun x -> check_expr x symtab) l
        in
    (match checked_elements with
    | [] -> raise (Failure "1D array cannot be empty")
    | (typ, _) :: tail ->
      if (match typ with Arr1D _ | Arr2D _ -> true | _ -> false)
          then
        raise (Failure "1D arrays cannot contain nested arrays")
      else if List.for_all (fun (typ', _) -> typ' = typ) tail then
        (Arr1D (typ, List.length l), SArr1DLit checked_elements)
      else
        raise (Failure ("All elements in 1D array must be the same
            type: " ^ string_of_expr e)))
  | Arr2DLit l ->
    let num_rows = List.length l in
    let row_length = List.length (List.hd l) in
    if not (List.for_all (fun row -> List.length row = row_length) l
       ) then
      raise (Failure ("All rows in a 2D array must have the same
          length: " ^ string_of_expr e))
    else
      let checked_rows =
        List.map (fun row -> List.map (fun x -> check_expr x symtab)
            row) l in
      (match checked_rows with
        | [] -> raise (Failure "2D array cannot be empty")
        | ( (first_typ, _) :: _ ) :: _ ->
          if (match first_typ with Arr1D _ | Arr2D _ -> true | _ ->
              false) then
            raise (Failure "2D arrays cannot contain nested arrays")
          else if List.for_all (fun row -> List.for_all (fun (t, _)
              -> t = first_typ) row) checked_rows then
            (Arr2D (first_typ, num_rows, row_length), SArr2DLit
               checked_rows)
          else
            raise (Failure ("All elements of a 2D array must be of
               the same type" ^ string_of_expr e))
        | _ -> raise (Failure "poop"))
  | Id var -> (type_of_identifier var symtab, SId var)
  | Assign(var, e) as ex ->
    let lt = type_of_identifier var symtab
    and (rt, e') = check_expr e symtab in
    let err = "illegal assignment " ^ string_of_typ lt ^ " = " ^
              string_of_typ rt ^ " in " ^ string_of_expr ex
    in
    (check_assign lt rt err, SAssign(var, (rt, e')))
  | Call(fname, args) as call ->
    let fd = find_func fname in
    let param_length = List.length fd.formals in
    if List.length args != param_length then
      raise (Failure ("expecting " ^ string_of_int param_length ^
                      " arguments in " ^ string_of_expr call))
    else let check_call (ft, _) e =
            let (et, e') = check_expr e symtab in
            let err = "illegal argument found " ^ string_of_typ et ^
                      " expected " ^ string_of_typ ft ^ " in " ^
                          string_of_expr e
            in (check_assign ft et err, e')
      in
      let args' = List.map2 check_call fd.formals args
      in (fd.rtyp, SCall(fname, args'))
  | Uop(op, e1) as e ->
    let (t1, e1') = check_expr e1 symtab in
    let t = (match op with
        Neg when t1 = Int || t1 = Double -> t1
      | Not when t1 = Bool -> Bool
      | _ -> raise (Failure ("illegal unary operator " ^
                            string_of_uop op ^ " on " ^
                                string_of_typ t1 ^
                            " in " ^ string_of_expr e)))
    in (t, SUop(op, (t1, e1')))
  | Binop(e1, op, e2) as e ->
    let (t1, e1') = check_expr e1 symtab
    and (t2, e2') = check_expr e2 symtab in
    let err = "illegal binary operator " ^
              string_of_typ t1 ^ " " ^ string_of_op op ^ " " ^
              string_of_typ t2 ^ " in " ^ string_of_expr e
    in
    (* All binary operators except Mmult require operands of the
       same type*)
    (match op with
      | Mmult ->
        (match (t1, t2) with
          | Arr2D (t1_elem, r1, c1), Arr2D (t2_elem, r2, c2) ->
            if t1_elem <> t2_elem then
              raise (Failure ("matrix elements must be same type for
                   matrix multiplication: " ^
                                string_of_typ t1_elem ^ " vs " ^
                                  string_of_typ t2_elem))
            else if (t1_elem <> Int && t1_elem <> Double) then
              raise (Failure ("matrix elements must be Int or Double
                   for matrix multiplication, got: " ^
                                string_of_typ t1_elem))
            else if c1 <> r2 then
              raise (Failure ("matrix dimensions mismatch for matrix
                   multiplication: " ^
                                Printf.sprintf "(%d x %d) * (%d x %d)"
                                   r1 c1 r2 c2))
            else
              (Arr2D (t1_elem, r1, c2), SBinop((t1, e1'), op, (t2,
                 e2')))
          | _ -> raise (Failure ("Matrix multiplication only
             supported for 2D arrays (matrices), got: " ^
                             string_of_typ t1 ^ " and " ^ string_of_typ
                                t2)))
      | _ ->
        if t1 = t2 then
          (* Determine expression type based on operator and operand
              types *)
          let t =
            (match op, t1 with
            | (Add | Sub | Mult | Div | Mod), Int ->
                Int
            | (Add | Sub | Mult | Div | Mod), Double ->
                Double
            | (Add | Sub | Mult | Div ), Arr1D(Int, n) ->
                Arr1D(Int, n)
            | (Add | Sub | Mult | Div ), Arr1D(Double, n) ->
                Arr1D(Double, n)
            | (Add | Sub | Mult | Div ), Arr2D(Int, m, n) ->
                Arr2D(Int, m, n)
            | (Add | Sub | Mult | Div ), Arr2D(Double, m, n) ->
                Arr2D(Double, m, n)
            | (Equal | Neq), _ ->
                Bool
            | ((Le | Lt | Ge | Gt), (Int | Double)) ->
                Bool
            (* Can include these too:
              | ((Le | Lt | Ge | Gt), Arr1D(Int, n)) ->
                Arr1D(Bool, n)
            | ((Le | Lt | Ge | Gt), Arr1D(Double, n)) ->
                Arr1D(Bool, n)
            | ((Le | Lt | Ge | Gt), Arr2D(Int, m, n)) ->
                  Arr2D(Bool, m, n)
            | ((Le | Lt | Ge | Gt), Arr2D(Double, m, n)) ->
                  Arr2D(Bool, m, n) *)
            | (And | Or), Bool ->
                Bool
            (* | ((And | Or), Arr1D(Bool, n)) ->
              Arr1D(Bool, n)
            | ((And | Or), Arr2D(Bool, m, n)) ->
              Arr2D(Bool, m, n) *)
            | _ -> raise (Failure err))
          in
          (t, SBinop((t1, e1'), op, (t2, e2')))
        else if ((match (t1, t2) with
          | (Int, Arr1D(Int, _))
          | (Arr1D(Int, _), Int)
          | (Int, Arr2D(Int, _, _))
          | (Arr2D(Int, _, _), Int)
          | (Double, Arr1D(Double, _))
          | (Arr1D(Double, _), Double)
          | (Double, Arr2D(Double, _, _))
          | (Arr2D(Double, _, _), Double) -> true
          | _ -> false)
        &&
        op = Mult)
        then
          let t =
            (match t1, t2 with
            | (Int, (Arr1D(Int, n))) | ((Arr1D(Int, n)), Int) ->
                Arr1D(Int, n)
            | (Int, (Arr2D(Int, m, n))) | ((Arr2D(Int, m, n)), Int)
               ->
                Arr2D(Int, m, n)
            | (Double, (Arr1D(Double, n))) | ((Arr1D(Double, n)),
               Double) ->
                Arr1D(Double, n)
            | (Double, (Arr2D(Double, m, n))) | ((Arr2D(Double, m, n
               )), Double) ->
                Arr2D(Double, m, n)
            | _ -> raise (Failure err))
          in (t, SBinop((t1, e1'), op, (t2, e2')))
        else
          raise (Failure err)
      )
  | Arr1DAssign(id, index_expr, value_expr) ->  (* id[index_expr] =
     value_expr *)
   let arr_typ =
     try StringMap.find id symtab
     with Not_found -> raise (Failure ("undeclared array identifier
         : " ^ id))
   in
   (match arr_typ with
     | Arr1D (elem_typ, _) ->
        let (index_type, index_s) = check_expr index_expr symtab in
        let (value_type, value_s) = check_expr value_expr symtab in
        if index_type <> Int then
          raise (Failure ("array index must be of type int, got " ^
                          string_of_typ index_type ^ " in " ^
                            string_of_expr index_expr));
        let _ = check_assign elem_typ value_type
          ("array assignment type mismatch: expected " ^
             string_of_typ elem_typ ^
              ", got " ^ string_of_typ value_type ^ " in " ^
```

```
                 string_of_expr value_expr)
         in (elem_typ, SArr1DAssign (id, (index_type, index_s), (
             value_type, value_s)))
       | _ -> raise (Failure (id ^ " is not a 1D array (Arr1DAssign)"
           )))
 | Arr2DAssign(id, row_expr, col_expr, value_expr) -> (* id[
     row_expr][col_expr] = value_expr *)
     let arr_typ =
       try StringMap.find id symtab
       with Not_found -> raise (Failure ("undeclared array identifier
           : " ^ id))
     in
     (match arr_typ with
       | Arr2D (elem_typ, _, _) ->
         let (row_type, row_s) = check_expr row_expr symtab in
         let (col_type, col_s) = check_expr col_expr symtab in
         let (value_type, value_s) = check_expr value_expr symtab in

         if row_type <> Int || col_type <> Int then
           raise (Failure (
             "array indices must be of type int, got: [" ^
             string_of_typ row_type ^ ", " ^ string_of_typ col_type ^
                 "] " ^
             "in " ^ string_of_expr row_expr ^ " and " ^
                string_of_expr col_expr
           ));

         let _ = check_assign elem_typ value_type
           ("matrix assignment type mismatch: expected " ^
              string_of_typ elem_typ ^
           ", got " ^ string_of_typ value_type ^ " in " ^
              string_of_expr value_expr)
         in (elem_typ, SArr2DAssign (id,
                                    (row_type, row_s),
                                    (col_type, col_s),
                                    (value_type, value_s)))
       | _ -> raise (Failure (id ^ " is not a 2D array")))
 | Arr1DAccess(id, index_expr) -> (* id[index_expr] *)
     let arr_typ =
       try StringMap.find id symtab
       with Not_found -> raise (Failure ("undeclared array identifier
           : " ^ id))
     in
     (match arr_typ with
       | Arr1D (elem_typ, _) ->
         let (index_t, index_s) = check_expr index_expr symtab in
         if index_t <> Int then
           raise (Failure ("array index must be of type int, got " ^
                           string_of_typ index_t ^ " in " ^
                              string_of_expr index_expr));
         (elem_typ, SArr1DAccess (id, (index_t, index_s)))
       (* Arr2D accesses will be parsed as a 1D access, it's the same
           syntax *)
       | Arr2D (elem_typ, _, n) ->
         let (index_t, index_s) = check_expr index_expr symtab in
         if index_t <> Int then
```

```
        raise (Failure ("array index must be of type int, got " ^
                        string_of_typ index_t ^ " in " ^
                           string_of_expr index_expr));
      (Arr1D (elem_typ, n) , SArr1DAccess (id, (index_t, index_s))
         ) (* returns a 1D array *)
    | _ -> raise (Failure (id ^ " is not an array (Arr1DAccess)"))
      )
| Arr2DAccess(id, row_expr, col_expr) -> (* id[row_expr][col_expr]
    *)
  let arr_typ =
    try StringMap.find id symtab
    with Not_found -> raise (Failure ("undeclared array identifier
        : " ^ id))
  in
  (match arr_typ with
    | Arr2D (elem_typ, _, _) ->
      let (row_type, row_s) = check_expr row_expr symtab in
      let (col_type, col_s) = check_expr col_expr symtab in
      if row_type <> Int || col_type <> Int then
        raise (Failure (
          "array indices must be of type int, got: [" ^
          string_of_typ row_type ^ ", " ^ string_of_typ col_type ^
              "] " ^
          "in " ^ string_of_expr row_expr ^ " and " ^
             string_of_expr col_expr
        ));
      (elem_typ, SArr2DAccess (id, (row_type, row_s), (col_type,
         col_s)))
    | _ -> raise (Failure (id ^ " is not a 2D array")))
| ArrUop(op, arr_expr) ->
    let (arr_typ, arr_s) = check_expr arr_expr symtab in
    (match op, arr_typ with
    | Length, Arr1D (_, _) | Length, Arr2D (_, _, _) ->
        (Int, SArrUop (op, (arr_typ, arr_s)))
    | Transpose, Arr2D (elem_typ, rows, cols) ->
        (Arr2D (elem_typ, cols, rows), SArrUop (op, (arr_typ,
           arr_s)))
    | Length, _ ->
        raise (Failure ("'length' applied to non-array type: " ^
           string_of_typ arr_typ))
    | Transpose, _ ->
        raise (Failure ("'transpose' applied to non-2D array type:
            " ^ string_of_typ arr_typ)))
| ArrOp(op, arr_expr, func_name) ->
  let (arr_typ, arr_s) = check_expr arr_expr symtab in
  let fd = find_func func_name in
    (match op, arr_typ with
    | Map, Arr1D (elem_typ, n) ->
      (match fd.formals, fd.rtyp with
      | [(arg_typ, _)], ret_typ when arg_typ = elem_typ ->
          (Arr1D (ret_typ, n), SArrOp (op, (arr_typ, arr_s),
             func_name))
      | _ ->
          raise (Failure (
            "map function must take argument of type " ^
               string_of_typ elem_typ
          )))
    | Map, Arr2D (elem_typ, m, n) ->
      (match fd.formals, fd.rtyp with
      | [(arg_typ, _)], ret_typ when arg_typ = elem_typ ->
          (Arr2D (ret_typ, m, n), SArrOp (op, (arr_typ, arr_s),
             func_name))
      | _ ->
          raise (Failure (
            "map function must take one argument of type " ^
               string_of_typ elem_typ
          )))
    | Reduce, Arr1D (elem_typ, n) ->
      (match fd.formals, fd.rtyp with
      | [(t1, _); (t2, _)], ret_typ when t1 = elem_typ && t2 =
         elem_typ && ret_typ = elem_typ ->
          (elem_typ, SArrOp (op, (arr_typ, arr_s), func_name))
      | _ ->
          raise (Failure (
            "reduce function must take two arguments of type " ^
               string_of_typ elem_typ ^
            " and return " ^ string_of_typ elem_typ ^ " in reduce(
               " ^ string_of_typ arr_typ ^ ", " ^ func_name ^ ")"
          )))
    | Reduce, Arr2D (elem_typ, m, n) ->
      (match fd.formals, fd.rtyp with
      | [(t1, _); (t2, _)], ret_typ when t1 = elem_typ && t2 =
         elem_typ && ret_typ = elem_typ ->
          (Arr1D (elem_typ, m), SArrOp (op, (arr_typ, arr_s),
             func_name))
      | _ ->
          raise (Failure (
            "reduce function must take two arguments of type " ^
               string_of_typ elem_typ ^
            " and return " ^ string_of_typ elem_typ ^ " in reduce(
               " ^ string_of_typ arr_typ ^ ", " ^ func_name ^ ")"
          )))
    | Map, _ | Reduce, _ ->
        raise (Failure (
          string_of_arrop op ^ " expects an array operand, got " ^
              string_of_typ arr_typ ^
          " in " ^ string_of_expr e))
    )
| Arr1DSlice(id, s, e) -> (* id[s:e] *)
  let arr_typ =
    try StringMap.find id symtab
    with Not_found -> raise (Failure ("undeclared array identifier
       : " ^ id))
  in
  (match arr_typ with
    | Arr1D (elem_typ, orig_len) ->
      if s < 0 || e <= 0 || s >= e || e > orig_len then
        raise (Failure (
          "invalid slice bounds [" ^ string_of_int s ^ ":" ^
             string_of_int e ^
          "] for array '" ^ id ^ "' of length " ^ string_of_int
             orig_len
        ))
      else
        let slice_len = e - s in
        (Arr1D (elem_typ, slice_len), SArr1DSlice (id, s, e))
    | _ ->
      raise (Failure (id ^ " is not a 1D array and cannot be
          sliced")))
| Arr2DSlice(id, s1, e1, s2, e2) -> (* id[s1:e1, s2:e2] *)
  let arr_typ =
    try StringMap.find id symtab
    with Not_found -> raise (Failure ("undeclared array identifier
        : " ^ id))
  in
  (match arr_typ with
    | Arr2D (elem_typ, orig_row_len, orig_col_len) ->
      if s1 < 0 || e1 <= 0 || s1 >= e1 || e1 > orig_row_len ||
        s2 < 0 || e2 <= 0 || s2 >= e2 || e2 > orig_col_len then
        raise (Failure (
          "invalid slice bounds [" ^ string_of_int s1 ^ ":" ^
              string_of_int e1 ^
          "][" ^  string_of_int s2 ^ ":" ^ string_of_int e2 ^ "]
              for array '" ^ id ^
          "' of shape " ^ string_of_int orig_row_len ^ "x" ^
              string_of_int orig_col_len
        ))
      else
        let new_rows = e1 - s1 in
        let new_cols = e2 - s2 in
        (Arr2D (elem_typ, new_rows, new_cols), SArr2DSlice (id, s1
            , e1, s2, e2))
    | _ ->
      raise (Failure (id ^ " is not a 2D array and cannot be
          sliced")))
| NoExpr -> (Void, SNoExpr)
in


let check_func func =
(* Make sure no formals or locals are void or duplicates *)
check_binds "formal" func.formals;

(* Build local symbol table of variables for this function *)
let symbols = List.fold_left (fun m (ty, name) -> StringMap.add
    name ty m)
    StringMap.empty ( global_binds @ func.formals )
in

let check_bool_expr e symtab =
  let (t, e') = check_expr e symtab in
  match t with
  | Bool -> (t, e')
  |  _ -> raise (Failure ("expected Boolean expression in " ^
      string_of_expr e))
in

let rec check_stmt_list stmt_list symtab =
  match stmt_list with
  | [] -> []
  | Block sl :: sl'  -> check_stmt_list (sl @ sl') symtab (*
     Flatten blocks *)
  | s :: sl ->
    let s', symtab' = check_stmt s symtab in
        s' :: check_stmt_list sl symtab'
(* Return a semantically-checked statement i.e. containing sexprs
   *)
and check_stmt stmt symtab =
  (* A block is correct if each statement is correct and nothing
     follows any Return statement.  Nested blocks are flattened.
        *)
  match stmt with
  | Block sl -> (SBlock (check_stmt_list sl symtab), symtab)
  | Expr e -> (SExpr (check_expr e symtab), symtab)
  | If(e, st1, st2) ->
    (SIf(check_bool_expr e symtab, fst (check_stmt st1 symtab),
       fst (check_stmt st2 symtab)), symtab)
  | For(s1, e1, e2, st) ->
    let s1', symtab' = check_stmt s1 symtab in
    (* Make sure that s1 is either a VDecl statement or an Exp *)
    let checked_s1' =
      match s1' with
      | SVDecl (SBindInit ((decl_typ, var), init_checked_expr)) ->
          (SVDecl (SBindInit ((decl_typ, var), init_checked_expr))
             )
      | SExpr e ->
          (SExpr (e))
      | _ ->
          raise (Failure "For loop initializer must be either an
             expression or a declaration with initialization")
    in
    let e1' = check_bool_expr e1 symtab' in
    let e2' = check_expr e2 symtab' in
    (SFor(checked_s1', e1', e2', fst (check_stmt st symtab')),
       symtab)
  | While(e, st) ->
   (SWhile(check_bool_expr e symtab, fst (check_stmt st symtab)),
     symtab)
  | Return e ->
    let (t, e') = check_expr e symtab in
    if t = func.rtyp then (SReturn (t, e'), symtab)
    else raise (
        Failure ("return gives " ^ string_of_typ t ^ " expected "
           ^
                string_of_typ func.rtyp ^ " in " ^ string_of_expr
                   e))
  | Break -> (SBreak, symtab)
  | VDecl v ->
    match v with
      | BindDecl(typ, id) ->
        if StringMap.mem id symtab then
          raise (Failure ("Variable " ^ id ^ " is already declared
             "))
        else
```

```
              (SVDecl(SBindDecl(typ, id)), StringMap.add id typ symtab
                 )
          | BindInit((typ, id), e) ->
            let rt, e' = check_expr e symtab in
            let err = "illegal assignment " ^ string_of_typ typ ^ " =
                " ^
                 string_of_typ rt ^ " in " ^ string_of_expr e
            in
            let _ = check_assign typ rt err in
            if StringMap.mem id symtab then
              raise (Failure ("Variable " ^ id ^ " is already declared
                 "))
            else
              (SVDecl(SBindInit((typ, id), (rt, e'))), StringMap.add
                 id typ symtab)
    in (* body of check_func *)
    { srtyp = func.rtyp;
      sfname = func.fname;
      sformals = func.formals;
      sbody = check_stmt_list func.body symbols;
    }
  in

(* check the exprs on RHS of global BindInits are only literals *)
  let check_expr_global e symtab =
    let (t, e') = check_expr e symtab in
    let is_scalar_lit = function
      | SIntLit _ | SDoubleLit _ | SBoolLit _ | SCharLit _ |
         SStringLit _ -> true
      | _ -> false
    in
    let scalar_type = function
      | Int | Double | Bool | Char | String -> true
      | _ -> false
    in
      (match (t, e')  with
        | (Int    , SIntLit     _)
        | (Double , SDoubleLit  _)
        | (Bool   , SBoolLit    _)
        | (Char   , SCharLit    _)
        | (String , SStringLit  _) -> (t, e')
        | (Arr1D (elem_t , _), SArr1DLit els) when scalar_type elem_t
           ->
          if List.for_all (fun (_,e') -> is_scalar_lit e') els
            then (t, e') else raise (Failure ("global initializer for
                arrays must be literals: " ^ string_of_expr e))
        | (Arr2D (elem_t , _, _), SArr2DLit rows) when scalar_type
           elem_t ->
          let row_all_lits row = List.for_all (fun (_,e') ->
              is_scalar_lit e') row in
          if List.for_all row_all_lits rows
          then (t, e') else raise (Failure ("global initializer for
              arrays must be literals: " ^ string_of_expr e))
        | _ -> raise (Failure ("global initializer not supported " ^
           string_of_expr e)))
  in
  (* Create sbinds from all of the globals, checking BindInit types
     while doing so *)
  let check_global_binds global =
    (* Build local symbol table of variables for this function *)
    let global_symtab = List.fold_left (fun m (ty, name) ->
        if StringMap.mem name m then
          raise (Failure ("Variable " ^ name ^ " is already declared")
              )
        else
          StringMap.add name ty m)
        StringMap.empty ( global_binds )
    in
    match global with
      BindDecl(d) -> SBindDecl(d)
      | BindInit(d, e) ->
        let lt = fst d
        and (rt, e') = check_expr_global e global_symtab in
        let err = "illegal global assignment " ^ string_of_typ lt ^ "
            = " ^
                  string_of_typ rt
        in
        let _ = check_assign lt rt err in SBindInit(d, (rt, e'))
  in
  (List.map check_global_binds globals, List.map check_func functions)

(* END *)
```

7.8 codegen.ml

```
(* IR generation: translate takes a semantically checked AST and
   produces LLVM IR

   LLVM tutorial: Make sure to read the OCaml version of the tutorial

   http://llvm.org/docs/tutorial/index.html

   Detailed documentation on the OCaml LLVM library:

   http://llvm.moe/
   http://llvm.moe/ocaml/

*)

module L = Llvm
module A = Ast
open Sast


module StringMap = Map.Make(String)

(* translate : Sast.program -> Llvm.module *)
let translate (globals, functions) =
  let context    = L.global_context () in
```

```
(* Create the LLVM compilation module into which
   we will generate code *)
let the_module = L.create_module context "CAPybara" in

(* Get types from the context *)
let i32_t      = L.i32_type     context    (* ints *)
and double_t   = L.double_type  context    (* doubles *)
and pointer_t  = L.pointer_type            (* function pointers and
    strings *)
and i8_t       = L.i8_type      context    (* chars *)
and i1_t       = L.i1_type      context    (* bool *)
and void_t     = L.void_type    context    (* void for function
    return type *)
and array_t    = L.array_type              (* 1d and 2d arrays *)
in

(* Return the LLVM type for a CAPybara type *)
let rec ltype_of_typ = function
    A.Int    -> i32_t
  | A.Double -> double_t
  | A.Char   -> i8_t
  | A.String -> pointer_t i8_t (* pointer to a char *)
  | A.Bool   -> i1_t
  | A.Arr1D(ty, n) ->
      let elem_ty = ltype_of_typ ty in
        array_t elem_ty n
  | A.Arr2D(ty, m, n) ->
      let elem_ty = ltype_of_typ ty in
        array_t (array_t elem_ty n) m
  | A.Void   -> void_t
in


let map_to_const_llvm ((ty, sx) : sexpr) : L.llvalue =
  match (ty, sx) with
    | (A.Int, SIntLit i) -> L.const_int  (ltype_of_typ ty) i
    | (A.Double, SDoubleLit d) -> L.const_float (ltype_of_typ ty) d
    | (A.Bool, SBoolLit b) -> L.const_int (ltype_of_typ ty) (if b
       then 1 else 0)
    | (A.Char, SCharLit c) -> L.const_int (ltype_of_typ ty) (Char.
       code c)
    | (A.String, SStringLit s) ->
      let str_const = L.const_stringz context s in (* create a char
          array *)
      let global_str_ptr = L.define_global ".str" str_const
          the_module in (* Add our char array to the global context
          *)
      L.const_bitcast global_str_ptr (pointer_t i8_t) (* cast from a
           pointer to char array to a pointer to a char *)
    | _ -> raise (Failure ("global initializer for arrays must be
       literals"))
in


(* Create a map of global variables after creating each *)
let global_vars : L.llvalue StringMap.t =
let global_var m = function
    | SBindDecl(t, na) ->
      let init =
        (
          match t with
          | A.Arr1D(elem_ty, n) ->
            let llvm_elem_ty = ltype_of_typ elem_ty in
            let zero    = L.const_int llvm_elem_ty 0 in
            let array_const   = Array.init n (fun _ -> zero) in
            L.const_array llvm_elem_ty array_const
          | A.Arr2D(elem_ty, m, n) ->
            let llvm_elem_ty = ltype_of_typ elem_ty in
            let zero    = L.const_int llvm_elem_ty 0 in
            let row_ty  = L.array_type llvm_elem_ty n in
            let row     = L.const_array llvm_elem_ty (Array.make n
                zero) in
            L.const_array row_ty (Array.make m row)
          | _ -> L.const_int (ltype_of_typ t) 0
        )

      in StringMap.add na (L.define_global na init the_module) m
    | SBindInit((t, na), (_, e)) ->
      let init =
        (match e with
          | SIntLit i -> L.const_int (ltype_of_typ t) i
          | SDoubleLit d -> L.const_float (ltype_of_typ t) d
          | SBoolLit b -> L.const_int (ltype_of_typ t) (if b then
             1 else 0)
          | SCharLit c -> L.const_int (ltype_of_typ t) (Char.code
             c)
          | SStringLit s ->
            let str_const = L.const_stringz context s in (* create
                a char array *)
            let global_str_ptr = L.define_global ".str" str_const
               the_module in (* Add our char array to the global
               context *)
            L.const_bitcast global_str_ptr (pointer_t i8_t) (*
               cast from a pointer to char array to a pointer to a
                char *)
          | SArr1DLit elems ->
            (* get the types *)
            let (elem_typ, _) = List.hd elems in
            let llvm_elem_ty = ltype_of_typ elem_typ in

            let array_const = Array.map (map_to_const_llvm) (Array
               .of_list elems) in  (* contruct all of the literals
                *)
            L.const_array llvm_elem_ty array_const  (* make the
               array using L.const_array *)
          | SArr2DLit elems_list ->
            (* get all of the types *)
            (match elems_list with
              | ((elem_typ, _) :: _) :: _ ->
                let n  = List.length (List.hd elems_list) in
                let llvm_elem_ty = ltype_of_typ elem_typ in
```

```
                  (* helper func to turn a single row into an LLVM
                     array of constants*)
                  let build_row row =
                    let consts = Array.map map_to_const_llvm (Array.
                        of_list row) in   (* constructed llvalue
                        array *)
                    L.const_array llvm_elem_ty consts          (*
                        array of n x T *)
                  in

                  (* build array of constant rows *)
                  let row_consts : L.llvalue array =
                    Array.map build_row (Array.of_list elems_list)
                          (* m x [n x T] *)
                  in
                  let row_ty = L.array_type llvm_elem_ty n in
                     (* [n x T] type *)
                  L.const_array row_ty row_consts  (* make the array
                      of [m x [n x T]] using row_consts values *)
              | _ -> raise (Failure "empty 2-D array literal")
            )
            | _ -> raise (Failure ("global variable " ^ na ^ " must
               be assigned to a literal"))
          ) in
          StringMap.add na (L.define_global na init the_module) m
  in
  List.fold_left global_var StringMap.empty globals in

(* Add the built-in functions *)
let printf_t : L.lltype =
  L.var_arg_function_type i32_t [| L.pointer_type i8_t |] in
let printf_func : L.llvalue =
  L.declare_function "printf" printf_t the_module in

let arr_1d_memcpy builder len src_body_ptr dst_body_ptr  =
  (* loop *)
  for i = 0 to len - 1 do
    let idx = L.const_int i32_t i in
    (* load element *)
    let src_gep =
      L.build_in_bounds_gep src_body_ptr
        [| L.const_int i32_t 0; idx |]
        "src_gep" builder
    in
    let v = L.build_load src_gep "src_val" builder in

    (* store *)
    let dst_gep =
      L.build_in_bounds_gep dst_body_ptr
        [| L.const_int i32_t 0; idx |]
        "dst_gep" builder
    in
    ignore (L.build_store v dst_gep builder)
  done;

  (* return the ptr to the memcpy'd array *)
  dst_body_ptr
in

let arr_2d_memcpy builder m n src_body_ptr dst_body_ptr  =
  (* loop *)
  for i = 0 to m - 1 do
    let i_idx = L.const_int i32_t i in
    (* pointer to row i of source *)
    let src_row_ptr =
      L.build_in_bounds_gep src_body_ptr
        [| L.const_int i32_t 0; i_idx |]
        "src_row" builder
    in
    (* pointer to row i of dest *)
    let dst_row_ptr =
      L.build_in_bounds_gep dst_body_ptr
        [| L.const_int i32_t 0; i_idx |]
        "dst_row" builder
    in

    for j = 0 to n - 1 do
      let j_idx = L.const_int i32_t j in

      (* load src[i][j] *)
      let src_elem_gep =
        L.build_in_bounds_gep src_row_ptr
          [| L.const_int i32_t 0; j_idx |]
          "src_gep" builder
      in
      let v = L.build_load src_elem_gep "src_val" builder in

      (* store into dest[i][j] *)
      let dst_elem_gep =
        L.build_in_bounds_gep dst_row_ptr
          [| L.const_int i32_t 0; j_idx |]
          "dst_gep" builder
      in
      ignore (L.build_store v dst_elem_gep builder)
    done
  done;

  (* return the ptr to the memcpy'd array *)
  dst_body_ptr
in

(* Define each function (arguments and return type) so we can
   call it even before we've created its body *)
let function_decls : (L.llvalue * sfunc_def) StringMap.t =
  let ll_sig_ty t =
    let base = ltype_of_typ t in
    match t with
    | Arr1D _ | Arr2D _ -> L.pointer_type base
    | _                 -> base
  in
  let function_decl m fdecl =
    fdecl.sformals <- begin
```

```
      match fdecl.srtyp with
      | Arr1D (t, len)  -> fdecl.sformals @ [(Arr1D(t, len), "
         arr1d_return_ptr")]
      | Arr2D (t, row, col) -> fdecl.sformals @ [(Arr2D(t, row, col)
         , "arr2d_return_ptr")]
      | _ -> fdecl.sformals
      end;

    (* helper to get pointers to arrays *)
    let name = fdecl.sfname in

    (* formals: lift any Arr1D/Arr2D to pointers *)
    let formal_types =
      fdecl.sformals
      |> List.map (fun (t,_) -> ll_sig_ty t)
      |> Array.of_list
    in
    let rtyp = (match fdecl.srtyp with
      | Arr1D _ | Arr2D _ -> void_t
      | _ -> ltype_of_typ fdecl.srtyp
    )
    in
    let ftype = L.function_type rtyp formal_types in
    StringMap.add name (L.define_function name ftype the_module,
       fdecl) m in

  List.fold_left function_decl StringMap.empty functions

in

(* Fill in the body of the given function *)
let build_function_body fdecl =
  let (the_function, _) = StringMap.find fdecl.sfname function_decls
      in
  let builder = L.builder_at_end context (L.entry_block the_function
     ) in

  let int_format_str = L.build_global_stringptr "%d\n" "fmt" builder
      in
  let double_format_str = L.build_global_stringptr "%f\n" "fmt"
     builder in

  let bool_fmt = L.build_global_stringptr "%s\n" "bool_fmt" builder
     in
  let true_str = L.build_global_stringptr "true"  "true_lit"
     builder in
  let false_str = L.build_global_stringptr "false" "false_lit"
     builder in

  let char_format_str = L.build_global_stringptr "%c\n" "fmt"
     builder in
  let str_format_str = L.build_global_stringptr "%s\n" "fmt" builder
      in

  (* Construct the function's "formals": formal arguments.
    Allocate each on the stack, initialize their value, if
```

```
      appropriate,
    and remember their values in the "formals" map. Locally declared
    variables in the function will be added to this map as they're
    encountered as statements. Arrays will be passed by reference,
      meaning
    their caller's pointer is used as local parameters. *)
let formal_vars =
  let add_formal m (t, n) p =
    L.set_value_name n p;

    (match n with
    | "arr1d_return_ptr" | "arr2d_return_ptr" ->  (* pass return
        array ptrs as a ptr*)
      StringMap.add n p m
    | _ ->
      let local = L.build_alloca (ltype_of_typ t) n builder in
      (* deep copy when the user passes arrays by value *)
      let () =
        (match t with
          | Arr1D (ty, len) ->
            ignore(arr_1d_memcpy builder len p local);
          | Arr2D (ty, row, col) ->
            ignore(arr_2d_memcpy builder row col p local);
          | _ ->
            ignore (L.build_store p local builder);
        ) in

      StringMap.add n local m
    )
  in
  List.fold_left2 add_formal StringMap.empty fdecl.sformals
      (Array.to_list (L.params the_function))
in

(* Return the value for a variable or formal argument.
   Check local names first, then global names *)
let lookup n local_vars = try StringMap.find n local_vars
  with Not_found -> StringMap.find n global_vars
in

(* Construct code for an expression; return its a pointer to where
    it's value is *)
let rec build_expr builder local_vars ((ty, e) : sexpr) = match e
   with
    SIntLit i  -> L.const_int i32_t i
  | SDoubleLit d -> L.const_float double_t d
  | SBoolLit b  -> L.const_int i1_t (if b then 1 else 0)
  | SCharLit c -> L.const_int i8_t (Char.code c)
  | SStringLit s -> L.build_global_stringptr s ".str" builder
  | SArr1DLit elems ->
     (* get the types *)
    let (elem_typ, _) = List.hd elems in
    let n = List.length elems in
    let llvm_elem_ty = ltype_of_typ elem_typ in
    let arr_ty = L.array_type llvm_elem_ty n in
    (* create the array *)
    let alloca_ptr = L.build_alloca arr_ty "_arr1d_lit" builder in

    List.iteri (fun i (typ, expr) ->
      (* 1) compute the  i th  element *)
      let v = build_expr builder local_vars (typ, expr) in

      (* 2) get pointer to element i *)
      let gep =
        L.build_in_bounds_gep
          alloca_ptr
          [| L.const_int i32_t 0; L.const_int i32_t i |]
          "_arr_lit_gep"
          builder
      in

      (* 3) store the element *)
      ignore (L.build_store v gep builder)
    ) elems;
    alloca_ptr  (* return a pointer to the array *)
  | SArr2DLit elems_list ->
    (* get the types *)
    (match elems_list with
      | ((elems_typ, _) :: _) :: _ ->
        (* m = #rows, n = #cols *)
        let m = List.length elems_list in
        let n  = List.length (List.hd elems_list) in
        let llvm_elem_ty = ltype_of_typ elems_typ in
        let row_ty = L.array_type llvm_elem_ty n in
        let arr_ty = L.array_type row_ty m in

        (* create the array *)
        let alloca_ptr = L.build_alloca arr_ty "_arr2d_lit"
           builder in

        (* for each element, compute and store *)
        List.iteri (fun i row ->
          List.iteri (fun j (ty, expr) ->
            let v = build_expr builder local_vars (ty, expr) in

            let gep = L.build_in_bounds_gep
              alloca_ptr
              [| L.const_int i32_t 0
              ; L.const_int i32_t i
              ; L.const_int i32_t j
              |]
              "_arr2d_gep"
              builder
            in

            ignore (L.build_store v gep builder)
          ) row
        ) elems_list;
        alloca_ptr  (* return a pointer to the array *)
      | _ -> raise (Failure "empty 2-D array literal"))
  | SId s ->
```

```
    let is_arr =
      (match ty with
        | A.Arr1D _ | A.Arr2D _ -> true
        | _ -> false
      )
    in
    let ptr = lookup s local_vars in
    if is_arr then ptr else L.build_load ptr s builder
  | SAssign (s, e) ->
    let e' = build_expr builder local_vars e in
    let body_ptr = lookup s local_vars in

    let () =
      (match ty with
      | A.Arr1D (elem_ty, len) ->
        ignore(arr_1d_memcpy builder len e' body_ptr)
      | A.Arr2D (elem_ty, m, n) ->
        ignore(arr_2d_memcpy builder m n e' body_ptr)
      | _ -> ignore(L.build_store e' (lookup s local_vars) builder
         )
      ) in
    e'
  | SBinop (((A.Arr1D(_, _), _) as e1), op, ((A.Arr1D(_, _), _) as
      e2)) ->
    (match op with
      | A.Add -> build_arr1d_add builder local_vars e1 e2
      | A.Sub -> build_arr1d_sub builder local_vars e1 e2
      | A.Mult -> build_arr1d_mult builder local_vars e1 e2
      | A.Div -> build_arr1d_div builder local_vars e1 e2
      | _ -> raise(Failure "Incompatible operation for Arr1D/Arr1D
         ")
    )
  | SBinop (((A.Arr2D(_, _, _), _) as e1), op, ((A.Arr2D (_, _, _)
     , _) as e2)) ->
      (match op with
        | A.Mmult -> build_arr2d_mmult builder local_vars e1 e2
        | A.Add -> build_arr2d_add builder local_vars e1 e2
        | A.Sub -> build_arr2d_sub builder local_vars e1 e2
        | A.Mult -> build_arr2d_mult builder local_vars e1 e2
        | A.Div -> build_arr2d_div builder local_vars e1 e2
        | _ -> raise(Failure "Incompatible operation for Arr2D/
           Arr2D")
        )
  | SBinop (((A.Double, _) as e1), op, ((A.Double, _) as e2)) ->
    let e1' = build_expr builder local_vars e1
    and e2' = build_expr builder local_vars e2 in
    (match op with
      | A.Add     -> L.build_fadd
      | A.Sub     -> L.build_fsub
      | A.Mult    -> L.build_fmul
      | A.Div     -> L.build_fdiv
      | A.Mod     -> L.build_frem
      | A.Equal   -> L.build_fcmp L.Fcmp.Oeq
      | A.Neq     -> L.build_fcmp L.Fcmp.One
      | A.Lt    -> L.build_fcmp L.Fcmp.Olt
      | A.Gt    -> L.build_fcmp L.Fcmp.Ogt
      | A.Le    -> L.build_fcmp L.Fcmp.Ole
      | A.Ge    -> L.build_fcmp L.Fcmp.Oge
      | _       -> raise (Failure("error: not a viable double-
        double operation"))
      ) e1' e2' "tmp" builder
  (* mismatching types for array * scalar multiplication *)
  | SBinop (((A.Arr1D(_, _), _) as e1), op, e2) ->
    (match (e1, op, e2) with
      | ((A.Arr1D(A.Int, _), _), A.Mult, (A.Int, _)) ->
        build_arr_1d_int_mult builder local_vars e1 e2
      | ((A.Arr1D(A.Double, _), _), A.Mult, (A.Double, _)) ->
        build_arr_1d_double_mult builder local_vars e1 e2
      | _ -> raise (Failure ("build expr: SBinop (((A.Arr1D(_, _),
         _) as e1), op, e2)."))
    )
  | SBinop (e1, op, ((A.Arr1D(_, _), _) as e2)) ->
    (match (e1, op, e2) with
      | ((A.Int, _), A.Mult, (A.Arr1D(A.Int, _), _)) ->
        build_arr_1d_int_mult builder local_vars e2 e1
      | ((A.Double, _), A.Mult, (A.Arr1D(A.Double, _), _)) ->
        build_arr_1d_double_mult builder local_vars e2 e1
      | _ -> raise (Failure ("build expr: SBinop (e1, op, ((A.
        Arr1D(_, _), _) as e2))."))
    )
  | SBinop (((A.Arr2D(_, _, _), _) as e1), op, e2) ->
    (match (e1, op, e2) with
      | ((A.Arr2D(A.Int, _, _), _), A.Mult, (A.Int, _)) ->
        build_arr_2d_int_mult builder local_vars e1 e2
      | ((A.Arr2D(A.Double, _, _), _), A.Mult,(A.Double, _)) ->
        build_arr_2d_double_mult builder local_vars e1 e2
      | _ -> raise (Failure ("build expr: SBinop (((A.Arr2D(_, _,
         _), _) as e1), op, e2)"))
    )
  | SBinop (e1, op, ((A.Arr2D(_, _, _), _) as e2)) ->
    (match (e1, op, e2) with
      | ((A.Int, _), A.Mult, (A.Arr2D(A.Int, _, _), _)) ->
         build_arr_2d_int_mult builder local_vars e2 e1
      | ((A.Double, _), A.Mult, (A.Arr2D(A.Double, _, _), _)) ->
         build_arr_2d_double_mult builder local_vars e2 e1
      | _ -> raise (Failure ("build expr: SBinop (e1, op, ((A.
         Arr2D(_, _, _), _) as e2))"))
    )
  (* General binop case: ints, bools, chars, strings *)
  | SBinop (e1, op, e2) ->
    let e1' = build_expr builder local_vars e1
    and e2' = build_expr builder local_vars e2 in
    (match op with
      | A.Add     -> L.build_add
      | A.Sub     -> L.build_sub
      | A.Mult    -> L.build_mul
      | A.Div     -> L.build_sdiv
      | A.Mod     -> L.build_srem
      | A.Equal   -> L.build_icmp L.Icmp.Eq
      | A.Neq     -> L.build_icmp L.Icmp.Ne
      | A.Lt      -> L.build_icmp L.Icmp.Slt
      | A.Gt      -> L.build_icmp L.Icmp.Sgt
```

```
        | A.Le       -> L.build_icmp L.Icmp.Sle
        | A.Ge       -> L.build_icmp L.Icmp.Sge
        | A.And      -> L.build_and
        | A.Or       -> L.build_or
        | _          -> raise (Failure("error: not a viable int-int
           operation"))
      ) e1' e2' "tmp" builder
    | SUop (op, ((ty, _) as e1)) ->
      let e1' = build_expr builder local_vars e1 in
      (match op with
        | A.Not                     -> L.build_not
        | A.Neg when ty = A.Double -> L.build_fneg
        | A.Neg                     -> L.build_neg
      )  e1' "tmp" builder
    | SArr1DAssign (id, ((index_ty, index_s) as i), ((value_ty,
       value_s) as v)) ->
        (* get ptrs and values *)
        let arr_ptr = lookup id local_vars in
        let index_val = build_expr builder local_vars i in
        let scalar_val = build_expr builder local_vars v in

        (* load arr_ptr[index_val] *)
        let arr_elem_gep =
          L.build_in_bounds_gep arr_ptr
            [| L.const_int i32_t 0; index_val |]
            "arr_gep" builder
        in
        (* store value *)
        ignore (L.build_store scalar_val arr_elem_gep builder);

        (* return the stored value *)
        scalar_val
    | SArr2DAssign (id, ((row_index_ty, row_index_s) as r), ((
       col_index_ty, col_index_s) as c), ((value_ty, value_s) as v))
        ->
      let arr_ptr = lookup id local_vars in
      let row_index_val = build_expr builder local_vars r in
      let col_index_val = build_expr builder local_vars c in
      let scalar_val = build_expr builder local_vars v in

      (* pointer to row i of source *)
       let arr_row_ptr =
        L.build_in_bounds_gep arr_ptr
          [| L.const_int i32_t 0; row_index_val |]
          "arr_row" builder
      in

      (* load arr_ptr[index_val] *)
      let arr_elem_gep =
        L.build_in_bounds_gep arr_row_ptr
          [| L.const_int i32_t 0; col_index_val |]
          "arr_gep" builder
      in
      (* store value *)
      ignore (L.build_store scalar_val arr_elem_gep builder);
```

```
  (* return the stored value *)
  scalar_val
| SArr1DAccess (id, ((index_ty, index_s) as i)) ->
  (* get ptrs and values *)
  let arr_ptr = lookup id local_vars in
  let index_val = build_expr builder local_vars i in

  (* load arr_ptr[index_val] *)
  let arr_elem_gep =
    L.build_in_bounds_gep arr_ptr
      [| L.const_int i32_t 0; index_val |]
      "arr_gep" builder
  in
  (* return the accessed value *)
  L.build_load arr_elem_gep "src_val" builder
| SArr2DAccess (id, ((row_index_ty, row_index_s) as r), ((
   col_index_ty, col_index_s) as c)) ->
  let arr_ptr = lookup id local_vars in
  let row_index_val = build_expr builder local_vars r in
  let col_index_val = build_expr builder local_vars c in

  (* pointer to row i of source *)
   let arr_row_ptr =
    L.build_in_bounds_gep arr_ptr
      [| L.const_int i32_t 0; row_index_val |]
      "arr_row" builder
  in

  (* load arr_ptr[index_val] *)
  let arr_elem_gep =
    L.build_in_bounds_gep arr_row_ptr
      [| L.const_int i32_t 0; col_index_val |]
      "arr_gep" builder
  in
  (* retrieve the value at the index and return *)
  L.build_load arr_elem_gep "src_val" builder
| SArrOp (op, ((arr_ty, arr_se) as arr_expr), func_name) ->
  (match op with
    | A.Map    ->
      (match ty with
      | A.Arr1D (_) -> build_map_arr_1d builder local_vars
         arr_expr func_name ty
      | A.Arr2D (_) -> build_map_arr_2d builder local_vars
         arr_expr func_name ty
      | _ -> raise (Failure("huh"))
      )
    | A.Reduce ->
      (match ty with
      | A.Arr1D (_) -> build_reduce_arr_1d builder local_vars
         arr_expr func_name
      | A.Arr2D (_) -> build_reduce_arr_2d builder local_vars
         arr_expr func_name
      | _ -> raise (Failure("wat"))
      )
  )
| SArrUop (op, ((ty, arr_se) as arr_expr)) ->
```

```
    (match op with
      | A.Length    ->
        (match ty with
        | A.Arr1D (_, n) ->
            L.const_int i32_t n
        | A.Arr2D (_, m, _) ->
            L.const_int i32_t m
        | _ ->
            failwith "Length operator on non-array")
      | A.Transpose -> build_arr_2d_transpose builder local_vars
         arr_expr
    )
  | SArr1DSlice (id, s0, _) ->
      build_arr_1d_slice builder local_vars id ty s0
  | SArr2DSlice (id, s0, _, r0, _) ->
      build_arr_2d_slice builder local_vars id ty s0 r0
  | SNoExpr ->  L.const_int i32_t  0
  | SCall ("print_int", [e]) ->
    L.build_call printf_func [| int_format_str ; (build_expr
       builder local_vars e) |]
      "printf" builder
  | SCall ("print_double", [e]) ->
    L.build_call printf_func [| double_format_str ; (build_expr
       builder local_vars e) |]
      "printf" builder
  | SCall ("print_char", [e]) ->
    L.build_call printf_func [| char_format_str ; (build_expr
       builder local_vars e) |]
      "printf" builder
  | SCall ("print_bool", [e]) ->
    let bool_val = build_expr builder local_vars e in (* i1 result
        of e *)
    let chosen =
      L.build_select bool_val                    (* i1 condition
               *)
                    true_str                     (* i8* if true
                                 *)
                    false_str                    (* i8* if false
                               *)
                    "bool_as_str" builder
    in
    L.build_call printf_func [| bool_fmt ; chosen |] "printf"
       builder
  | SCall ("print_str", [e]) ->
    L.build_call printf_func [| str_format_str ; (build_expr
       builder local_vars e) |]
      "printf" builder
  | SCall (f, args) ->
    let (fdef, fdecl) = StringMap.find f function_decls in
    let llargs = List.rev (List.map (build_expr builder local_vars
       ) (List.rev args)) in
    (match fdecl.srtyp with
      | A.Void -> L.build_call fdef (Array.of_list llargs) ""
         builder
      | A.Arr1D (elem_ty, n) ->
        (* allocate a new array *)
        (* get the types *)
        let llvm_elem_ty = ltype_of_typ elem_ty in
        let arr_ty = L.array_type llvm_elem_ty n in

        (* create the array *)
        let alloca_ptr = L.build_alloca arr_ty "_arr1d_returned"
           builder in

        (* append a ptr to this new array onto list of args passed
            *)
        let appended_llargs = llargs @ [alloca_ptr] in
        ignore(L.build_call fdef (Array.of_list appended_llargs) "
           " builder);
        alloca_ptr
      | A.Arr2D (elem_ty, m, n) ->
        let llvm_elem_ty = ltype_of_typ elem_ty in
        let row_ty = L.array_type llvm_elem_ty n in
        let arr_ty = L.array_type row_ty m in

        (* create the array *)
        let alloca_ptr = L.build_alloca arr_ty "_arr2d_returned"
           builder in
        (* append a ptr to this new array onto list of args passed
            *)
        let appended_llargs = llargs @ [alloca_ptr] in
        ignore(L.build_call fdef (Array.of_list appended_llargs) "
           " builder);
        alloca_ptr
      | _ ->
        let result = f ^ "_result" in
        L.build_call fdef (Array.of_list llargs) result builder
    )
and

build_arr1d_add builder local_vars arr1_se arr2_se =
  (* get the ptrs *)
  let arr1_ptr = build_expr builder local_vars arr1_se in
  let arr2_ptr = build_expr builder local_vars arr2_se in
  let elem_ty, len =
    (match arr1_se with
      | (A.Arr1D (elem_ty, n), _) -> elem_ty, n
      | _ -> raise (Failure "build_arr1d_add: not a 1D array")
    ) in

  (* allocate result [len x i32] *)
  let llvm_arr_ty = ltype_of_typ (A.Arr1D (elem_ty, len)) in
  let res_ptr = L.build_alloca llvm_arr_ty "arr_add_res" builder
     in

  (* loop *)
  for i = 0 to len - 1 do
    let idx = L.const_int i32_t i in
    (* load elements *)
    let src1_gep =
      L.build_in_bounds_gep arr1_ptr
        [| L.const_int i32_t 0; idx |]
        "src1_gep" builder
    in
    let v1 = L.build_load src1_gep "src1_val" builder in
    let src2_gep =
      L.build_in_bounds_gep arr2_ptr
        [| L.const_int i32_t 0; idx |]
        "src2_gep" builder
    in
    let v2 = L.build_load src2_gep "src2_val" builder in

    (* v1 + v2 *)
    let sum =
      (match elem_ty with
      | A.Int -> L.build_add v1 v2 "elt_add" builder
      | A.Double -> L.build_fadd v1 v2 "elt_add" builder
      | _ -> raise (Failure "build_arr1d_add: not an int or double
         ")
      )
    in

    (* store *)
    let dst_gep =
      L.build_in_bounds_gep res_ptr
        [| L.const_int i32_t 0; idx |]
        "dst_gep" builder
    in
    ignore (L.build_store sum dst_gep builder)
  done;

  (* return the ptr to the new array *)
  res_ptr

and

build_arr1d_sub builder local_vars arr1_se arr2_se =
  (* get the ptrs *)
  let arr1_ptr = build_expr builder local_vars arr1_se in
  let arr2_ptr = build_expr builder local_vars arr2_se in
  (* get the length *)
  let elem_ty, len =
    (match arr1_se with
      | (A.Arr1D (elem_ty, n), _) -> elem_ty, n
      | _ -> raise (Failure "build_arr_1d_sub: not an array1D")
    ) in

  (* allocate result [len x i32] *)
  let llvm_arr_ty = ltype_of_typ (A.Arr1D (elem_ty, len)) in
  let res_ptr = L.build_alloca llvm_arr_ty "arr1d_sub_res" builder
      in

  (* loop *)
  for i = 0 to len - 1 do
    let idx = L.const_int i32_t i in
    (* load element *)
    let src_gep =
      L.build_in_bounds_gep arr1_ptr
```

```
        [| L.const_int i32_t 0; idx |]
        "src_gep" builder
    in
    let v1 = L.build_load src_gep "src_val1" builder in
    let src2_gep =
      L.build_in_bounds_gep arr2_ptr
        [| L.const_int i32_t 0; idx |]
        "src2_gep" builder
    in
    let v2 = L.build_load src2_gep "src_val2" builder in

    (* v1 - v2 *)
    let diff =
      (match elem_ty with
      | A.Int -> L.build_sub v1 v2 "elt_sub" builder
      | A.Double -> L.build_fsub v1 v2 "elt_sub" builder
      | _ -> raise (Failure "build_arr1d_sub: not an int or double
         ")
      )
    in

    (* store *)
    let dst_gep =
      L.build_in_bounds_gep res_ptr
        [| L.const_int i32_t 0; idx |]
        "dst_gep" builder
    in
    ignore (L.build_store diff dst_gep builder)
  done;

  (* return the ptr to the new array *)
  res_ptr
and

build_arr1d_mult builder local_vars arr1_se arr2_se =
  (* get the ptrs *)
  let arr1_ptr = build_expr builder local_vars arr1_se in
  let arr2_ptr = build_expr builder local_vars arr2_se in
  (* get the length *)
  let elem_ty, len =
    (match arr1_se with
      | (A.Arr1D (elem_ty, n), _) -> elem_ty, n
      | _ -> raise (Failure "build_arr1d_mult: not an array1D")
    ) in

  (* allocate result [len x i32] *)
  let llvm_arr_ty = ltype_of_typ (A.Arr1D (elem_ty, len)) in
  let res_ptr = L.build_alloca llvm_arr_ty "arr1d_mul_res" builder
      in

  (* loop *)
  for i = 0 to len - 1 do
    let idx = L.const_int i32_t i in
    (* load element *)
    let src_gep =
      L.build_in_bounds_gep arr1_ptr
```

```
        [| L.const_int i32_t 0; idx |]
        "src_gep" builder
    in
    let v1 = L.build_load src_gep "src_val1" builder in
    let src2_gep =
      L.build_in_bounds_gep arr2_ptr
        [| L.const_int i32_t 0; idx |]
        "src2_gep" builder
    in
    let v2 = L.build_load src2_gep "src_val2" builder in

    (* v1 * v2 *)
    let prod =
      (match elem_ty with
      | A.Int -> L.build_mul v1 v2 "elt_mul" builder
      | A.Double -> L.build_fmul v1 v2 "elt_mul" builder
      | _ -> raise (Failure "build_arr1d_mult: not an int or
         double")
      )
    in

    (* store *)
    let dst_gep =
      L.build_in_bounds_gep res_ptr
        [| L.const_int i32_t 0; idx |]
        "dst_gep" builder
    in
    ignore (L.build_store prod dst_gep builder)
  done;

  (* return the ptr to the new array *)
  res_ptr

and

build_arr1d_div builder local_vars arr1_se arr2_se =
  (* get the ptrs *)
  let arr1_ptr = build_expr builder local_vars arr1_se in
  let arr2_ptr = build_expr builder local_vars arr2_se in
  (* get the length *)
  let elem_ty, len =
    (match arr1_se with
      | (A.Arr1D (elem_ty, n), _) -> elem_ty, n
      | _ -> raise (Failure "build_arr1d_div: not an array1D")
    ) in

  (* allocate result [len x i32] *)
  let llvm_arr_ty = ltype_of_typ (A.Arr1D (elem_ty, len)) in
  let res_ptr = L.build_alloca llvm_arr_ty "arr1d_div_res" builder
      in

  (* loop *)
  for i = 0 to len - 1 do
    let idx = L.const_int i32_t i in
    (* load element *)
    let src_gep =
      L.build_in_bounds_gep arr1_ptr
        [| L.const_int i32_t 0; idx |]
        "src_gep" builder
    in
    let v1 = L.build_load src_gep "src_val1" builder in
    let src2_gep =
      L.build_in_bounds_gep arr2_ptr
        [| L.const_int i32_t 0; idx |]
        "src2_gep" builder
    in
    let v2 = L.build_load src2_gep "src_val2" builder in

    (* quotient *)
    let quotient =
      (match elem_ty with
      | A.Int -> L.build_sdiv v1 v2 "elt_div" builder
      | A.Double -> L.build_fdiv v1 v2 "elt_div" builder
      | _ -> raise (Failure "build_arr1d_div: not an int or double
         ")
      )
    in

    (* store *)
    let dst_gep =
      L.build_in_bounds_gep res_ptr
        [| L.const_int i32_t 0; idx |]
        "dst_gep" builder
    in
    ignore (L.build_store quotient dst_gep builder)
  done;

  (* return the ptr to the new array *)
  res_ptr
and

build_arr2d_add builder local_vars arr1_se arr2_se =
  (* get ptrs and values *)
  let arr1_ptr = build_expr builder local_vars arr1_se in
  let arr2_ptr = build_expr builder local_vars arr2_se in

  let elem_ty, m, n =
    (match arr1_se with
      | (A.Arr2D (elem_ty, m, n), _) -> elem_ty, m, n
      | _ -> raise (Failure "build_arr2d_add: not an array2D")
    ) in

  (* allocate result [len x i32] *)
  let llvm_arr_ty = ltype_of_typ (A.Arr2D (elem_ty, m, n)) in
  let res_ptr = L.build_alloca llvm_arr_ty "arr2d_add_res" builder
      in

  (* loop *)
  for i = 0 to m - 1 do
    let i_idx = L.const_int i32_t i in
    (* pointer to row i of source 1 *)
    let src1_row_ptr =
      L.build_in_bounds_gep arr1_ptr
        [| L.const_int i32_t 0; i_idx |]
        "src_row1" builder
    in

    (* pointer to row i of source 2 *)
    let src2_row_ptr =
      L.build_in_bounds_gep arr2_ptr
        [| L.const_int i32_t 0; i_idx |]
        "src_row2" builder
    in

    (* pointer to row i of dest *)
    let dst_row_ptr =
      L.build_in_bounds_gep res_ptr
        [| L.const_int i32_t 0; i_idx |]
        "dst_row" builder
    in

    for j = 0 to n - 1 do
      let j_idx = L.const_int i32_t j in

      (* load src1[i][j] *)
      let src1_elem_gep =
        L.build_in_bounds_gep src1_row_ptr
          [| L.const_int i32_t 0; j_idx |]
          "src_gep1" builder
      in
      let v1 = L.build_load src1_elem_gep "src_val1" builder in

      (* load src2[i][j] *)
      let src2_elem_gep =
        L.build_in_bounds_gep src2_row_ptr
          [| L.const_int i32_t 0; j_idx |]
          "src_gep2" builder
      in
      let v2 = L.build_load src2_elem_gep "src_val2" builder in

      (* v1 + v2 *)
      let sum =
        (match elem_ty with
        | A.Int -> L.build_add v1 v2 "elt_add" builder
        | A.Double -> L.build_fadd v1 v2 "elt_add" builder
        | _ -> raise (Failure "build_arr2d_add: not an int or
           double")
        )
      in

      (* store into dest[i][j] *)
      let dst_elem_gep =
        L.build_in_bounds_gep dst_row_ptr
          [| L.const_int i32_t 0; j_idx |]
          "dst_gep" builder
      in
      ignore (L.build_store sum dst_elem_gep builder)
    done
```

```
    done;

    (* return the ptr to the new array *)
    res_ptr
  and

  build_arr2d_sub builder local_vars arr1_se arr2_se =
    (* get ptrs and values *)
    let arr1_ptr = build_expr builder local_vars arr1_se in
    let arr2_ptr = build_expr builder local_vars arr2_se in

    let elem_ty, m, n =
      (match arr1_se with
        | (A.Arr2D (elem_ty, m, n), _) -> elem_ty, m, n
        | _ -> raise (Failure "build_arr2d_sub: not an array2D")
      ) in

    (* allocate result [len x i32] *)
    let llvm_arr_ty = ltype_of_typ (A.Arr2D (elem_ty, m, n)) in
    let res_ptr = L.build_alloca llvm_arr_ty "arr2d_sub_res" builder
        in

    (* loop *)
    for i = 0 to m - 1 do
      let i_idx = L.const_int i32_t i in
      (* pointer to row i of source 1 *)
      let src1_row_ptr =
        L.build_in_bounds_gep arr1_ptr
          [| L.const_int i32_t 0; i_idx |]
          "src_row1" builder
      in

      (* pointer to row i of source 2 *)
      let src2_row_ptr =
        L.build_in_bounds_gep arr2_ptr
          [| L.const_int i32_t 0; i_idx |]
          "src_row2" builder
      in

      (* pointer to row i of dest *)
      let dst_row_ptr =
        L.build_in_bounds_gep res_ptr
          [| L.const_int i32_t 0; i_idx |]
          "dst_row" builder
      in

      for j = 0 to n - 1 do
        let j_idx = L.const_int i32_t j in

        (* load src1[i][j] *)
        let src1_elem_gep =
          L.build_in_bounds_gep src1_row_ptr
            [| L.const_int i32_t 0; j_idx |]
            "src_gep1" builder
        in
        let v1 = L.build_load src1_elem_gep "src_val1" builder in
        (* load src2[i][j] *)
        let src2_elem_gep =
          L.build_in_bounds_gep src2_row_ptr
            [| L.const_int i32_t 0; j_idx |]
            "src_gep2" builder
        in
        let v2 = L.build_load src2_elem_gep "src_val2" builder in

        (* v1 - v2 *)
        let diff =
          (match elem_ty with
          | A.Int -> L.build_sub v1 v2 "elt_sub" builder
          | A.Double -> L.build_fsub v1 v2 "elt_sub" builder
          | _ -> raise (Failure "build_arr2d_sub: not an int or
             double")
          )
        in

        (* store into dest[i][j] *)
        let dst_elem_gep =
          L.build_in_bounds_gep dst_row_ptr
            [| L.const_int i32_t 0; j_idx |]
            "dst_gep" builder
        in
        ignore (L.build_store diff dst_elem_gep builder)
      done
    done;

    (* return the ptr to the new array *)
    res_ptr

and

build_arr2d_mult builder local_vars arr1_se arr2_se =
    (* get ptrs and values *)
    let arr1_ptr = build_expr builder local_vars arr1_se in
    let arr2_ptr = build_expr builder local_vars arr2_se in

    let elem_ty, m, n =
      (match arr1_se with
        | (A.Arr2D (elem_ty, m, n), _) -> elem_ty, m, n
        | _ -> raise (Failure "build_arr2d_mult: not an array2D")
      ) in

    (* allocate result [len x i32] *)
    let llvm_arr_ty = ltype_of_typ (A.Arr2D (elem_ty, m, n)) in
    let res_ptr = L.build_alloca llvm_arr_ty "arr2d_mul_res" builder
        in

    (* loop *)
    for i = 0 to m - 1 do
      let i_idx = L.const_int i32_t i in
      (* pointer to row i of source 1 *)
      let src1_row_ptr =
        L.build_in_bounds_gep arr1_ptr
          [| L.const_int i32_t 0; i_idx |]
          "src_row1" builder
      in

      (* pointer to row i of source 2 *)
      let src2_row_ptr =
        L.build_in_bounds_gep arr2_ptr
          [| L.const_int i32_t 0; i_idx |]
          "src_row2" builder
      in

      (* pointer to row i of dest *)
      let dst_row_ptr =
        L.build_in_bounds_gep res_ptr
          [| L.const_int i32_t 0; i_idx |]
          "dst_row" builder
      in

      for j = 0 to n - 1 do
        let j_idx = L.const_int i32_t j in

        (* load src1[i][j] *)
        let src1_elem_gep =
          L.build_in_bounds_gep src1_row_ptr
            [| L.const_int i32_t 0; j_idx |]
            "src_gep1" builder
        in
        let v1 = L.build_load src1_elem_gep "src_val1" builder in

        (* load src2[i][j] *)
        let src2_elem_gep =
          L.build_in_bounds_gep src2_row_ptr
            [| L.const_int i32_t 0; j_idx |]
            "src_gep2" builder
        in
        let v2 = L.build_load src2_elem_gep "src_val2" builder in

        (* v1 * v2 *)
        let prod =
          (match elem_ty with
          | A.Int -> L.build_mul v1 v2 "elt_add" builder
          | A.Double -> L.build_fmul v1 v2 "elt_add" builder
          | _ -> raise (Failure "build_arr2d_mult: not an int or
             double")
          )
        in

        (* store into dest[i][j] *)
        let dst_elem_gep =
          L.build_in_bounds_gep dst_row_ptr
            [| L.const_int i32_t 0; j_idx |]
            "dst_gep" builder
        in
        ignore (L.build_store prod dst_elem_gep builder)
      done
    done;
    (* return the ptr to the new array *)
    res_ptr
  and

  build_arr2d_div builder local_vars arr1_se arr2_se =
    (* get ptrs and values *)
    let arr1_ptr = build_expr builder local_vars arr1_se in
    let arr2_ptr = build_expr builder local_vars arr2_se in

    let elem_ty, m, n =
      (match arr1_se with
        | (A.Arr2D (elem_ty, m, n), _) -> elem_ty, m, n
        | _ -> raise (Failure "build_arr2d_div: not an array2D")
      ) in

    (* allocate result [len x i32] *)
    let llvm_arr_ty = ltype_of_typ (A.Arr2D (elem_ty, m, n)) in
    let res_ptr = L.build_alloca llvm_arr_ty "arr2d_div_res" builder
        in

    (* loop *)
    for i = 0 to m - 1 do
      let i_idx = L.const_int i32_t i in
      (* pointer to row i of source 1 *)
      let src1_row_ptr =
        L.build_in_bounds_gep arr1_ptr
          [| L.const_int i32_t 0; i_idx |]
          "src_row1" builder
      in

      (* pointer to row i of source 2 *)
      let src2_row_ptr =
        L.build_in_bounds_gep arr2_ptr
          [| L.const_int i32_t 0; i_idx |]
          "src_row2" builder
      in

      (* pointer to row i of dest *)
      let dst_row_ptr =
        L.build_in_bounds_gep res_ptr
          [| L.const_int i32_t 0; i_idx |]
          "dst_row" builder
      in

      for j = 0 to n - 1 do
        let j_idx = L.const_int i32_t j in

        (* load src1[i][j] *)
        let src1_elem_gep =
          L.build_in_bounds_gep src1_row_ptr
            [| L.const_int i32_t 0; j_idx |]
            "src_gep1" builder
        in
        let v1 = L.build_load src1_elem_gep "src_val1" builder in
        (* load src2[i][j] *)
        let src2_elem_gep =
          L.build_in_bounds_gep src2_row_ptr
            [| L.const_int i32_t 0; j_idx |]
            "src_gep2" builder
        in
        let v2 = L.build_load src2_elem_gep "src_val2" builder in

        (* v1 / v2 *)
        let quotient =
          match elem_ty with
          | A.Int    -> L.build_sdiv  v1 v2 "elt_div" builder
          | A.Double -> L.build_fdiv  v1 v2 "elt_div" builder
          | _       -> raise (Failure "build_arr2d_div: not an int
             or double")
        in

        (* store into dest[i][j] *)
        let dst_elem_gep =
          L.build_in_bounds_gep dst_row_ptr
            [| L.const_int i32_t 0; j_idx |]
            "dst_gep" builder
        in
        ignore (L.build_store quotient dst_elem_gep builder)
      done
    done;

    (* return the ptr to the new array *)
    res_ptr
  and

  build_arr_1d_int_mult builder local_vars arr_se int_se =
    (* get ptrs and values *)
    let arr_ptr = build_expr builder local_vars arr_se in
    let int_val = build_expr builder local_vars int_se in
    let len =
      (match arr_se with
        | (A.Arr1D (_, n), _) -> n
        | _ -> raise (Failure "build_arr_1d_int_mult: not a 1D array
           ")
      ) in
    (* allocate result [len x i32] *)
    let llvm_arr_ty = ltype_of_typ (A.Arr1D (A.Int, len)) in
    let res_ptr = L.build_alloca llvm_arr_ty "arr_mul_res" builder
       in

    (* loop *)
    for i = 0 to len - 1 do
      let idx = L.const_int i32_t i in
      (* load element *)
      let src_gep =
        L.build_in_bounds_gep arr_ptr
          [| L.const_int i32_t 0; idx |]
          "src_gep" builder
      in
      let v = L.build_load src_gep "src_val" builder in
```

```
      (* multiply *)
      let prod = L.build_mul v int_val "elt_mul" builder in

      (* store *)
      let dst_gep =
        L.build_in_bounds_gep res_ptr
          [| L.const_int i32_t 0; idx |]
          "dst_gep" builder
      in
      ignore (L.build_store prod dst_gep builder)
    done;

    (* return the ptr to the new array *)
    res_ptr
  and

  build_arr_2d_int_mult builder local_vars arr_se int_se =
    (* get ptrs and values *)
    let arr_ptr = build_expr builder local_vars arr_se in
    let int_val = build_expr builder local_vars int_se in
    let m, n =
      (match arr_se with
        | (A.Arr2D (_, m, n), _) -> m, n
        | _ -> raise (Failure "build_arr_2d_int_mult: not an array2D
           ")
      ) in

    (* allocate result [len x i32] *)
    let llvm_arr_ty = ltype_of_typ (A.Arr2D (A.Int, m, n)) in
    let res_ptr = L.build_alloca llvm_arr_ty "arr2d_mul_res" builder
        in

    (* loop *)
    for i = 0 to m - 1 do
      let i_idx = L.const_int i32_t i in
      (* pointer to row i of source *)
      let src_row_ptr =
        L.build_in_bounds_gep arr_ptr
          [| L.const_int i32_t 0; i_idx |]
          "src_row" builder
      in
      (* pointer to row i of dest *)
      let dst_row_ptr =
        L.build_in_bounds_gep res_ptr
          [| L.const_int i32_t 0; i_idx |]
          "dst_row" builder
      in

      for j = 0 to n - 1 do
        let j_idx = L.const_int i32_t j in

        (* load src[i][j] *)
        let src_elem_gep =
          L.build_in_bounds_gep src_row_ptr
            [| L.const_int i32_t 0; j_idx |]
            "src_gep" builder
        in
        let v = L.build_load src_elem_gep "src_val" builder in

        (* mul v * scalar *)
        let prod =
          L.build_mul v int_val "elt_mul" builder
        in

        (* store into dest[i][j] *)
        let dst_elem_gep =
          L.build_in_bounds_gep dst_row_ptr
            [| L.const_int i32_t 0; j_idx |]
            "dst_gep" builder
        in
        ignore (L.build_store prod dst_elem_gep builder)
      done
    done;

    (* return the ptr to the new array *)
    res_ptr
  and

  build_arr_1d_double_mult builder local_vars arr_se double_se =
    (* get ptrs and values *)
    let arr_ptr = build_expr builder local_vars arr_se in
    let double_val = build_expr builder local_vars double_se in
    let len =
      (match arr_se with
        | (A.Arr1D (_, n), _) -> n
        | _ -> raise (Failure "build_arr_1d_double_mult: not a 1D
           array")
      ) in

    (* allocate result [len x i32] *)
    let llvm_arr_ty = ltype_of_typ (A.Arr1D (A.Double, len)) in
    let res_ptr = L.build_alloca llvm_arr_ty "arr_mul_res" builder
       in

    (* loop *)
    for i = 0 to len - 1 do
      let idx = L.const_int i32_t i in
      (* load element *)
      let src_gep =
        L.build_in_bounds_gep arr_ptr
          [| L.const_int i32_t 0; idx |]
          "src_gep" builder
      in
      let v = L.build_load src_gep "src_val" builder in

      (* multiply *)
      let prod = L.build_fmul v double_val "elt_mul" builder in

      (* store *)
      let dst_gep =
        L.build_in_bounds_gep res_ptr
```

```
        [| L.const_int i32_t 0; idx |]
        "dst_gep" builder
    in
    ignore (L.build_store prod dst_gep builder)
  done;

  (* return the ptr to the new array *)
  res_ptr
and

build_arr_2d_double_mult builder local_vars arr_se double_se =
   (* get ptrs and values *)
   let arr_ptr = build_expr builder local_vars arr_se in
   let double_val = build_expr builder local_vars double_se in
   let m, n =
    (match arr_se with
      | (A.Arr2D (_, m, n), _) -> m, n
      | _ -> raise (Failure "build_arr_2d_double_mult: not an 2D
         array")
    ) in

   (* allocate result [m x n x i32] *)
   let llvm_arr_ty = ltype_of_typ (A.Arr2D (A.Double, m, n)) in
   let res_ptr = L.build_alloca llvm_arr_ty "arr2d_mul_res"
      builder in

   (* loop *)
   for i = 0 to m - 1 do
     let i_idx = L.const_int i32_t i in
     (* pointer to row i of source *)
     let src_row_ptr =
       L.build_in_bounds_gep arr_ptr
         [| L.const_int i32_t 0; i_idx |]
         "src_row" builder
     in
     (* pointer to row i of dest *)
     let dst_row_ptr =
       L.build_in_bounds_gep res_ptr
         [| L.const_int i32_t 0; i_idx |]
         "dst_row" builder
     in

     for j = 0 to n - 1 do
       let j_idx = L.const_int i32_t j in

       (* load src[i][j] *)
       let src_elem_gep =
         L.build_in_bounds_gep src_row_ptr
           [| L.const_int i32_t 0; j_idx |]
           "src_gep" builder
       in
       let v = L.build_load src_elem_gep "src_val" builder in

       (* mul v * scalar *)
       let prod =
         L.build_fmul v double_val "elt_mul" builder
       in

       (* store into dest[i][j] *)
       let dst_elem_gep =
         L.build_in_bounds_gep dst_row_ptr
           [| L.const_int i32_t 0; j_idx |]
           "dst_gep" builder
       in
       ignore (L.build_store prod dst_elem_gep builder)
     done
   done;

   (* return the ptr to the new array *)
   res_ptr
and

build_arr2d_mmult builder local_vars (ty1, sx1) (ty2, sx2) =
  (* get ptrs to arrays *)
  let arr1_ptr = build_expr builder local_vars (ty1, sx1)in
  let arr2_ptr = build_expr builder local_vars (ty2, sx2) in
  let elem_ty, m, n =
    (match ty1 with
      | A.Arr2D(elem_ty, m, n) -> elem_ty, m, n
      | _ -> raise (Failure "build_arr2d_mmult: not a 2D array:
         ty1")
    ) in

  let p =
    (match ty2 with
      | A.Arr2D(_, _, p) -> p
      | _ -> raise (Failure "build_arr2d_mmult: not a 2D array:
         ty2")
    ) in

  (* allocate result [m x p x i32] *)
  let llvm_arr_ty = ltype_of_typ (A.Arr2D (elem_ty, m, p)) in
  let res_ptr = L.build_alloca llvm_arr_ty "arr2d_mmult_res"
     builder in

  (* a zero value for the accumulator *)
  let zero_acc = match elem_ty with
    | A.Int    -> L.const_int  i32_t 0
    | A.Double -> L.const_float double_t 0.0
    | _        -> failwith "matrix-mul on non-numeric type"
  in

  (* triple nested loop to do the matrix mult *)
  for i = 0 to m - 1 do
    let i_idx = L.const_int i32_t i in

    for j = 0 to p - 1 do
      let j_idx = L.const_int i32_t j in

      (* accumulator register in OCaml *)
      let acc = ref zero_acc in
      (* sum over k *)
      for k = 0 to n - 1 do
        let k_idx = L.const_int i32_t k in

        (* load a = arr1[i][k] *)
        let a_ptr =
          L.build_in_bounds_gep arr1_ptr
            [| L.const_int i32_t 0; i_idx; k_idx |]
            "mm_a_gep" builder
        in
        let a_val = L.build_load a_ptr "mm_a" builder in

        (* load b = arr2[k][j] *)
        let b_ptr =
          L.build_in_bounds_gep arr2_ptr
            [| L.const_int i32_t 0; k_idx; j_idx |]
            "mm_b_gep" builder
        in
        let b_val = L.build_load b_ptr "mm_b" builder in

        (* prod = a * b *)
        let prod = match elem_ty with
          | A.Int    -> L.build_mul  a_val b_val "mm_mul" builder
          | A.Double -> L.build_fmul a_val b_val "mm_fmul" builder
          | _        -> assert false
        in

        (* acc = acc + prod *)
        let sum = match elem_ty with
          | A.Int    -> L.build_add  !acc prod "mm_add"  builder
          | A.Double -> L.build_fadd !acc prod "mm_fadd" builder
          | _        -> assert false
        in

        acc := sum
      done;

      (* store the final acc into res[i][j] *)
      let dst_ptr =
        L.build_in_bounds_gep res_ptr
          [| L.const_int i32_t 0; i_idx; j_idx |]
          "mm_res_gep" builder
      in
      ignore (L.build_store !acc dst_ptr builder)
    done
  done;

  (* return the ptr to the new array *)
  res_ptr
and

build_arr_2d_transpose builder local_vars (ty, sx) =
  (* get the array ptr we're transposing *)
  let arr_ptr = build_expr builder local_vars (ty, sx) in
  let elem_ty, m, n =
    (match ty with
      | A.Arr2D(elem_ty, m, n) -> elem_ty, m, n
      | _ -> raise (Failure "build_arr_2d_transpose: not a 2D
         array: ")
    ) in

  (* get array types *)
  let llvm_out_ty = ltype_of_typ (A.Arr2D(elem_ty, n, m)) in

  (* allocate new transposed array on stack*)
  let res_ptr = L.build_alloca llvm_out_ty "arr2d_transpose"
     builder in

  (* loop to transpose elements *)
  for i = 0 to m - 1 do
    for j = 0 to n - 1 do
      (* a) load element (i,j) from input *)
      let src_gep =
        L.build_in_bounds_gep arr_ptr
          [| L.const_int i32_t 0
           ; L.const_int i32_t i
           ; L.const_int i32_t j
          |]
          "src_gep" builder
      in
      let v = L.build_load src_gep "src" builder in

      (* b) store it into (j,i) of result *)
      let dst_gep =
        L.build_in_bounds_gep res_ptr
          [| L.const_int i32_t 0
           ; L.const_int i32_t j
           ; L.const_int i32_t i
          |]
          "dst_gep" builder
      in
      ignore (L.build_store v dst_gep builder)
    done
  done;

  (* return ptr to transposed array *)
  res_ptr
and


build_map_arr_1d builder local_vars (ty, sx) func_name ret_ty =
  (* get ptrs and values *)
  let arr_ptr = build_expr builder local_vars (ty, sx) in
  let (fn_val, _) = StringMap.find func_name function_decls in
  let n =
    (match ret_ty with
      | A.Arr1D(_, n) -> n
      | _ -> raise (Failure "build_map_arr_1d: not a 1D array: ty:
         ")
    ) in

  (* allocate result [len x i32] *)
  let llvm_arr_ty = ltype_of_typ ret_ty in
  let res_ptr = L.build_alloca llvm_arr_ty (func_name ^ "_map1d")
     builder in

  (* loop *)
  for i = 0 to n - 1 do
    let idx = L.const_int i32_t i in
    (* load element *)
    let src_gep =
      L.build_in_bounds_gep arr_ptr
        [| L.const_int i32_t 0; idx |]
        "src_gep" builder
    in
    let v = L.build_load src_gep "src_val" builder in

    (* apply the map and get a value *)
    let mapped  = L.build_call fn_val [| v |] (func_name ^ "_call"
       ) builder in

    (* store *)
    let dst_gep =
      L.build_in_bounds_gep res_ptr
        [| L.const_int i32_t 0; idx |]
        "dst_gep" builder
    in
    ignore (L.build_store mapped dst_gep builder)
  done;

  (* return the ptr to the new array *)
  res_ptr
and


build_map_arr_2d builder local_vars (ty, sx) func_name ret_ty =
  (* get ptrs and values *)
  let arr_ptr = build_expr builder local_vars (ty, sx) in
  let (fn_val, _) = StringMap.find func_name function_decls in
  let m, n =
    (match ret_ty with
      | A.Arr2D(_, m, n) -> m, n
      | _ -> raise (Failure "build_map_arr_2d: not a 2D array: ty:
           ")
    ) in

  (* allocate result [len x i32] *)
  let llvm_arr_ty = ltype_of_typ ret_ty in
  let res_ptr = L.build_alloca llvm_arr_ty (func_name ^ "_map2d")
     builder in

   (* loop *)
   for i = 0 to m - 1 do
    let i_idx = L.const_int i32_t i in
    (* pointer to row i of source *)
    let src_row_ptr =
      L.build_in_bounds_gep arr_ptr
        [| L.const_int i32_t 0; i_idx |]
```

```
          "src_row" builder
      in
      (* pointer to row i of dest *)
      let dst_row_ptr =
        L.build_in_bounds_gep res_ptr
          [| L.const_int i32_t 0; i_idx |]
          "dst_row" builder
      in

      for j = 0 to n - 1 do
        let j_idx = L.const_int i32_t j in

        (* load src[i][j] *)
        let src_elem_gep =
          L.build_in_bounds_gep src_row_ptr
            [| L.const_int i32_t 0; j_idx |]
            "src_gep" builder
        in
        let v = L.build_load src_elem_gep "src_val" builder in

        (* apply the map and get a value *)
        let mapped  = L.build_call fn_val [| v |] (func_name ^ "
            _call") builder in

        (* store into dest[i][j] *)
        let dst_elem_gep =
          L.build_in_bounds_gep dst_row_ptr
            [| L.const_int i32_t 0; j_idx |]
            "dst_gep" builder
        in
        ignore (L.build_store mapped dst_elem_gep builder)
      done
    done;

    res_ptr

and

build_reduce_arr_1d builder local_vars (ty, sx) func_name =
  (* get ptrs and values *)
  let arr_ptr = build_expr builder local_vars (ty, sx) in
  let (fn_val, _) = StringMap.find func_name function_decls in
  let elem_ty, len =
    (match ty with
      | A.Arr1D(elem_ty, n) -> elem_ty, n
      | _ -> raise (Failure "build_reduce_arr_1d: not a 1D array:
          ty: ")
    ) in

  (* allocate accumulator with initial value of arr_ptr[0] *)
  let zero = L.const_int i32_t 0 in
  let gep0 = L.build_in_bounds_gep arr_ptr [| zero; zero |] "
     red1d_0_gep" builder in
  let acc = ref (L.build_load gep0 "red1d_acc0" builder) in

  (* loop from 1 to len, reducing each element *)
```

```
    for i = 1 to len - 1 do
      let idx = L.const_int i32_t i in
      (* load element *)
      let gep =
        L.build_in_bounds_gep arr_ptr
          [| L.const_int i32_t 0; idx |]
          "red1d_i_gep" builder
      in
      let v = L.build_load gep "red1d_vi" builder in
      let new_acc =
        L.build_call fn_val [| !acc; v |] (func_name ^ "_call")
           builder
      in
      acc := new_acc
    done;
    (* return the final accumulated value *)
    !acc
  and

  build_reduce_arr_2d builder local_vars (ty, sx) func_name =
    (* get ptrs and values *)
    let arr_ptr = build_expr builder local_vars (ty, sx) in
    let (fn_val, _) = StringMap.find func_name function_decls in
    let elem_ty, m, n =
      (match ty with
        | A.Arr2D(elem_ty, m, n) -> elem_ty, m, n
        | _ -> raise (Failure "build_reduce_arr_2d: not a 2D array:
           ty: ")
      ) in

    (* allocate result [len x i32] *)
    let llvm_arr_ty = ltype_of_typ (A.Arr1D(elem_ty, m)) in
    let res_ptr = L.build_alloca llvm_arr_ty (func_name ^ "_reduce2d
       ") builder in
    let zero = L.const_int i32_t 0 in

    (* loop over all rows, for each row loop from 1 to length,
       reducing the elems in each row *)
    for i = 0 to m - 1 do
      (* start-of-row accumulator = arr[i][0] *)
      let i_idx = L.const_int i32_t i in
      let gep00 = L.build_in_bounds_gep arr_ptr [| zero; i_idx; zero
          |] "red2d_src00" builder in
      let acc_row0 = L.build_load gep00 "red2d_acc00" builder in
      (* accumulate row arr[i] into value dst[i] *)
      let dst0 =
        L.build_in_bounds_gep res_ptr
          [| zero; i_idx |]
          "reduce2d_dst0" builder
      in
      ignore (L.build_store acc_row0 dst0 builder);
      let acc = ref acc_row0 in

      (* j = 1..n-1: acc = f(acc, arr[i][j]); out[i][j] = acc *)
      for j = 1 to n - 1 do
        let j_idx = L.const_int i32_t j in
        let gepij =
          L.build_in_bounds_gep arr_ptr [| zero; i_idx; j_idx |] "
              red2d_srcij" builder
        in
        let v = L.build_load gepij "red2d_vij" builder in
        let new_acc =
          L.build_call fn_val [| !acc; v |] (func_name ^ "_call")
              builder
        in
        acc := new_acc;
      done;

      (* store the final accumulator into res[i] *)
      let dst_i =
        L.build_in_bounds_gep res_ptr
          [| zero; i_idx |]
          "reduce2d_dsti" builder
      in
      ignore (L.build_store !acc dst_i builder)
    done;

    (* return ptr to the new array of reduced rows *)
    res_ptr
  and

  build_arr_1d_slice builder local_vars id ty start_index =
    let arr_ptr = lookup id local_vars in
    let elem_ty, len =
      (match ty with
        | A.Arr1D(elem_ty, n) -> elem_ty, n
        | _ -> raise (Failure "build_arr_1d_slice: not a 1D array:
            ty: ")
      ) in

    (* allocate result [len x type_size] *)
    let llvm_arr_ty = ltype_of_typ (Arr1D(elem_ty, len)) in
    let res_ptr = L.build_alloca llvm_arr_ty ("_slice1d") builder in
    let zero   = L.const_int i32_t 0 in

    (* loop to copy elements *)
    for i = 0 to len - 1 do
      (* get element at arr[start_index+i] *)
      let src_gep = L.build_in_bounds_gep arr_ptr
          [| zero; L.const_int i32_t (start_index + i) |]
          "slice1d_src" builder
      in
      let v = L.build_load src_gep "slice1d_v" builder in
      (* store at dest[i]*)
      let dst_gep = L.build_in_bounds_gep res_ptr
          [| zero; L.const_int i32_t i |]
          "slice1d_dst" builder
      in
      ignore (L.build_store v dst_gep builder)
    done;

    (* return pointer to new array *)
    res_ptr
  and

  build_arr_2d_slice builder local_vars id ty start_row_index
     start_col_index =
    let arr_ptr = lookup id local_vars in
    let elem_ty, m, n =
    (match ty with
      | A.Arr2D(elem_ty, m, n) -> elem_ty, m, n
      | _ -> raise (Failure "build_arr_2d_slice: not a 2D array: ty:
          ")
    ) in

    (* allocate result [m x type size][n x type size] *)
    let llvm_arr_ty = ltype_of_typ (Arr2D(elem_ty, m, n)) in
    let res_ptr = L.build_alloca llvm_arr_ty ("_slice2d") builder in
    let zero   = L.const_int i32_t 0 in

    (* loop to copy elements *)
    for i = 0 to m - 1 do
      for j = 0 to n - 1 do
        (* get element at arr[m+i][n+j] *)
        let src_gep = L.build_in_bounds_gep arr_ptr
            [| zero
             ; L.const_int i32_t (start_row_index + i)
             ; L.const_int i32_t (start_col_index + j)
            |]
            "slice2d_src" builder
        in
        let v = L.build_load src_gep "slice2d_v" builder in
        (* store at dest[i][j]*)
        let dst_gep = L.build_in_bounds_gep res_ptr
            [| zero
             ; L.const_int i32_t i
             ; L.const_int i32_t j
            |]
            "slice2d_dst" builder
        in
        ignore (L.build_store v dst_gep builder)
      done
    done;

    (* return pointer to new array *)
    res_ptr
  in

  (* LLVM insists each basic block end with exactly one "terminator"
     instruction that transfers control.  This function runs "instr
        builder"
     if the current block does not already have a terminator.  Used,
     e.g., to handle the "fall off the end of the function" case. *)
  let add_terminal builder instr =
    match L.block_terminator (L.insertion_block builder) with
      Some _ -> ()
    | None -> ignore (instr builder) in
```

```
(* Build the code for the given statement; return the builder for
   the statement's successor (i.e., the next instruction will be
      built
   after the one generated by this call) *)
let rec build_stmt (builder, local_vars, we_bb) = function
    SBlock sl ->
      let build_stmts (builder', local_vars') =
        build_stmt (builder', local_vars', we_bb) in
      (fst (List.fold_left build_stmts (builder, local_vars) sl),
         local_vars)
  | SExpr e -> ignore(build_expr builder local_vars e); (builder,
     local_vars)
  | SReturn e -> ignore(match fdecl.srtyp with
       A.Void -> L.build_ret_void builder
      | A.Arr1D(elem_ty, n) ->
        (* look up the return array ptr formal to get the address
           *)
        let ret_ptr = lookup "arr1d_return_ptr" local_vars in
        (* memcpy e into this address, which is on the caller's
           stack *)
        ignore(arr_1d_memcpy builder n (build_expr builder
           local_vars e) ret_ptr);
        L.build_ret_void builder
      | A.Arr2D(elem_ty, m, n) ->
        (* look up the return array ptr formal to get the address
           *)
        let ret_ptr = lookup "arr2d_return_ptr" local_vars in
        (* memcpy e into this address, which is on the caller's
           stack *)
        ignore(arr_2d_memcpy builder m n (build_expr builder
           local_vars e) ret_ptr);
        L.build_ret_void builder
      | _ -> L.build_ret (build_expr builder local_vars e) builder
         );
      (builder, local_vars)
  | SIf (predicate, then_stmt, else_stmt) ->
    let bool_val = build_expr builder local_vars predicate in

    let then_bb = L.append_block context "then" the_function in
    ignore (build_stmt ((L.builder_at_end context then_bb),
       local_vars, we_bb) then_stmt);
    let else_bb = L.append_block context "else" the_function in
    ignore (build_stmt ((L.builder_at_end context else_bb),
       local_vars, we_bb)  else_stmt);

    let end_bb = L.append_block context "if_end" the_function in
    let build_br_end = L.build_br end_bb in (* partial function *)
    add_terminal (L.builder_at_end context then_bb) build_br_end;
    add_terminal (L.builder_at_end context else_bb) build_br_end;

    ignore(L.build_cond_br bool_val then_bb else_bb builder);
    (L.builder_at_end context end_bb, local_vars)
  | SFor (s1, e1, e2, body) ->
    let builder1, local_vars' = build_stmt (builder, local_vars,
       we_bb) s1 in
    let s2 = SWhile (e1, SBlock [body ; SExpr e2]) in
    let builder2, _ = build_stmt (builder1, local_vars', we_bb) s2
        in
    (builder2, local_vars)
  | SWhile (predicate, body) ->
    let while_bb = L.append_block context "while" the_function in
    let build_br_while = L.build_br while_bb in (* partial
       function *)
    ignore (build_br_while builder);
    let while_builder = L.builder_at_end context while_bb in
    let bool_val = build_expr while_builder local_vars predicate
       in

    let body_bb = L.append_block context "while_body" the_function
        in
    let end_bb = L.append_block context "while_end" the_function
       in

    add_terminal (fst (build_stmt ((L.builder_at_end context
       body_bb), local_vars, Some(end_bb)) body)) build_br_while;

    ignore(L.build_cond_br bool_val body_bb end_bb while_builder);
    (L.builder_at_end context end_bb, local_vars)
  | SBreak ->
    (* we check what the end block of the while loop is to
       unconditionally jump to *)
    let while_end_bb = (match we_bb with
       Some(bb) -> bb
     | None -> raise(Failure ("Break statement not in loop")))
    in ignore(L.build_br while_end_bb builder);
    (builder, local_vars)
  | SVDecl vdecl ->
    let add_array_local local_vars name arr_typ builder =
      (* declare body type *)
      let body_ty   = ltype_of_typ arr_typ in  (* pointer to [m x
         [n x T]] *)
      (* allocate space on stack for body *)
      let body_ptr  = L.build_alloca body_ty name builder in
      (* let handle_ptr  = L.build_alloca handle_ty name builder
         in *)
      let local_vars' = StringMap.add name body_ptr local_vars in
      body_ptr, local_vars'
    in

    (builder, match vdecl with
    | SBindDecl(ty, n) ->
      (* Declaring an array will allocate space on the stack for
         the array itself and its handle.
          The handle is a pointer to the array itself, both are on
              the stack.
         Declaring any other type will allocate space on the stack
             for the type itself.
      *)
      (match ty with
        | A.Arr1D _ | A.Arr2D _ ->
            let _, local_vars' = add_array_local local_vars n ty
                builder in
                local_vars'
              | _ ->
                let local_var = L.build_alloca (ltype_of_typ ty) n
                   builder
                in StringMap.add n local_var local_vars
            )

        | SBindInit((ty, na), e) ->
            (* When declaring and initializing an array, it will
               allocate space on the stack for the
                array itself and its handle. The handle is a pointer to
                   the array itself.  It will
                then deepcopy the array, regardless if there is another
                   variable or an array literal on the RHS.
               When declaring and initialzing any other type, it will
                  allocate space on the stack for
                the type itself. It will then simply store the RHS value
                    into the allocated stack space.
            *)
            let e' = build_expr builder local_vars e
            in L.set_value_name na e';

            (match ty with
              | A.Arr1D(elem_ty, len) ->
                let body_ptr, local_vars' = add_array_local local_vars
                   na ty builder in
                (* always deep-copy literals AND variable sources *)
                ignore(arr_1d_memcpy builder len e' body_ptr);
                local_vars'
              | A.Arr2D (elem_ty, m, n) ->
                let body_ptr, local_vars' = add_array_local local_vars
                   na ty builder in
                (* always deep-copy literals AND variable sources *)
                ignore(arr_2d_memcpy builder m n e' body_ptr);
                local_vars'
              | _ -> (* for regular scalar types *)
                (* allocate the var, store the value, add it's addr to
                   the map *)
                let local_var = L.build_alloca (ltype_of_typ ty) na
                   builder in
                ignore(L.build_store e' local_var builder);
                StringMap.add na local_var local_vars
            )
        )
  (* finished build_stmt *)
  in
  (* Build the code for each statement in the function *)
  let func_builder = fst (build_stmt (builder, formal_vars, None) (
     SBlock fdecl.sbody)) in

  (* Add a return if the last block falls off the end *)
  add_terminal func_builder (L.build_ret_void)

in

List.iter build_function_body functions;
```

```
  the_module
```

7.9 capybara.ml

```
(* Top-level of the CAPybara compiler: scan & parse the input,
   check the resulting AST and generate an SAST from it, generate LLVM
       IR,
   and dump the module *)

   type action = Ast | Sast | LLVM_IR | Compile

   let () =
     let action = ref LLVM_IR in
     let set_action a () = action := a in
     let speclist = [
       ("-a", Arg.Unit (set_action Ast), "Print the AST");
       ("-s", Arg.Unit (set_action Sast), "Print the SAST");
       ("-l", Arg.Unit (set_action LLVM_IR), "Print the generated LLVM
           IR");
       ("-c", Arg.Unit (set_action Compile), "Dump the compiled
          program to the executable file")
     ] in
     let usage_msg = "usage: ./capybara.native [-a|-s|-l|-c] [file.cap
        ]" in
     let channel = ref stdin in
     let filename = ref "" in

     (* Parsing the command-line arguments *)
     Arg.parse speclist (fun fname ->
       channel := open_in fname;
       filename := fname
     ) usage_msg;

     let lexbuf = Lexing.from_channel !channel in

     let ast = Capyparse.program Scanner.token lexbuf in
     match !action with
       Ast -> print_string (Ast.string_of_program ast)
     | _ -> let sast = Semant.check ast in
       match !action with
         Ast     -> ()
       | Sast    -> print_string (Sast.string_of_sprogram sast)
       | LLVM_IR -> print_string (Llvm.string_of_llmodule (Codegen.
          translate sast))
       | Compile ->
          let output_filename = Filename.remove_extension !filename in
          (* open the output file and write the LLVM IR *)
          let oc = open_out output_filename in
          output_string oc (Llvm.string_of_llmodule (Codegen.translate
              sast));
          close_out oc;
```

7.10 Makefile

```
OCB=ocamlbuild -use-ocamlfind -pkg llvm
TARGET=capybara.native

.PHONY: all tests clean

all: $(TARGET)

$(TARGET):
   $(OCB) $(TARGET)

clean:
   $(OCB) -clean
   rm -f parser.ml parser.mli parser.output scanner.ml a.out
```

7.11 test_runner.sh

```
#!/usr/bin/env bash
#
# test-runner.sh
# Runs all CAPybara tests in ./tests
#  - Tests named fail-* must print an error starting with "Fatal error
   :" and exit non-zero
#  - All other tests must compile successfully; we run them and then
   delete the generated binary

TEST_DIR="tests"

for test_path in "$TEST_DIR"/*; do
  test_name=$(basename "$test_path")
  base="${test_name%.cap}"    # strip the .cap suffix

  if [[ "$test_name" == fail-* ]]; then
    echo "=== $test_name (expected failure) ==="
    output=$(./capybara.native -c "$test_path" 2>&1)
    code=$?
    if [[ $code -eq 0 ]]; then
      echo "FAIL: $test_name: expected failure but compilation
          succeeded"
    elif [[ $output == Fatal\ error:* ]]; then
      echo "PASS: $test_name failed as expected"
    else
      echo "FAIL: $test_name did not print 'Fatal error:'"
      echo "Output was:"
      echo "$output"
    fi

  else
    echo "=== $test_name (expected success) ==="
    compile_out=$(./capybara.native -c "$test_path" 2>&1)
    compile_code=$?
    if [[ $compile_code -ne 0 ]]; then
      echo "FAIL: $test_name: compilation failed:"
      echo "$compile_out"
      continue
    fi
    # run the freshly-generated program (named $base)
    echo "Running $base"
    lli "$base" > /dev/null 2>&1
    echo "PASS: $test_name"

    # clean up
    rm -f "$TEST_DIR"/"$base"
  fi
done
```