Amit Shravan Patel
UNI: ap3567
COMS W4115 (CVN)
Prof. Stephen Edwards
Final Project Report

AP++

Contents

I. Introduction

AP++ is designed to be a lightweight programming language with particular emphasis on list syntax and functionality. It is built upon the MicroC language, consisting of a subset of stylistic and syntactic elements from C, along with Python-inspired list support.

Languages such as Python are popular because programmers are able to implement concise programs at a high-level instead of being bogged down by low-level details. Python specifically is well-known for its list data structure, which can be used widely to implement a number of algorithms. Furthermore, Python is notable for its list slicing syntactical sugar, which makes creating sublists very concise and intuitive.

Therefore, I believed it was a notable endeavor to build on top of the existing MicroC framework to support a Python-style list data structure, along with suite of list api functions.

II. Language Tutorial

1. Getting Started

Type *make* inside of the ap++ directory, which will create the compiler to convert *AP++* source code into LLVM IR code. The compiler is run against a target source code file and then piped to lli to convert to machine code and execute:

```
> make
> ./ap_plusplus.native < test_source_code | lli
> test out put here...
```

2. General structure of *AP++* source code

There are 4 basic elements to writing *AP++* programs: variables, functions, statements and expressions.

2.1 Variables

Variables hold two pieces of information: a type and the value that it is assigned. There are 3 primitive types of variables supported in ap++: int, float and bool. Variable declarations are in the following format:
type varName;

e.g.
```
int x;
float y;
bool z;
```

Variable declarations must occur at the top of any function. *AP++* also supports container *list* types that can hold primitive typed elements. They are declared in a similar way:

e.g.
```
list<int> x;
list<int> y;
list<int> z;
```

After variables have been declared, they can be assigned values. Values can either be literals or other variables.

e.g.

```
int x;
int y;

x = 2;
y = z;
y = 1;
```

It's important to note that assignment of primitive types occur by value so in the above example, when y is set to 1, x is still 2. Lists, however, do not function in this way. When assigning a list, any modification made to the list from any variable assigned that list will be seen everywhere. Each variable points to the same underlying list data. This holds true also when lists are used as function arguments.

e.g.

```
list<int> x;
list<int> y;

x = [0, 1, 2, 3];
y = x;
list_push(y, 4);
printi(#x); // prints 5
printi(x[4]); // prints 4
```

There are also a number of list api functions available for use. A comprehensive list is provided in the LRM.

2.2 Functions

Functions are declared and defined with the following format:

```
type functionName(type x1, type x2, ...) {
      // local variable declarations
      // statements
      return type; // if non-void type
}
```

There must be at least one function defined in the source code with the name main(). This is the top-level function that is executed.

2.3 Expressions and Statements

The LRM has a comprehensive list of expressions and statements that can be used to create an *AP++* program. Expressions evaluate to a value, but

statements do not. The main body of a function is made up of permutations of expressions and statements to form logic.

2.4 Printing

Printing to stdout can be done via built-in print functions: printi for printing int types, printf for printing float types, printb for printing bool types and prints for printing string literals.

2.5 Simple example

```
int max_value(list<int> a) {
  int max_val;
  int i;
  max_val = -1; // assume all elements are positive
  for (i = 0; i < #a; ++i) {
    if (a[i] > max_val) {
      max_val = a[i];
    }
  }
  return max_val;
}

int main() {
  list<int> a;
  a = [4, 10, 40, 23, 85, 2];
  printi(max_value(a));
  return 0;
}

output: 85
```

III. Language Reference Manual

1. Lexical Conventions

1.1 Whitespace

Spaces, tabs, return and newlines are ignored during token parsing.

1.2 Comments

Multi-line comments are enclosed in /* */
e.g. /* all code within here is ignored */

Single-line comments are preceded by //
e.g. // all code on this line is ignored

1.3 Reserved words

The following are reserved words and cannot be used as identifiers:

if else for while return void int bool float string true false
list list_push list_get list_set list_pop list_size list_slice list_clear list_rev
list_insert list_remove list_find

1.4 Code Structure

Indentation is not required to parse *AP++ source code*, but it is highly recommended for producing clean, manageable code. Instead, much like C, statement blocks are enclosed by {} braces and statements are terminated by semicolons.

2. Variables

Variables are names that are used to refer to some location in memory.

2.1 Primitive Types

Variables have three primitive types: {*int, float, bool*}.

bool variables can have two possible values {true, false}
int variables are signed 32-bit integer values
float variables are signed 64-bit double values

2.2 Variable Identifiers

Variables are referred by identifier names of unbounded length, which take the regex form: ['a'-'z' 'A'-'Z']['a'-'z' 'A'-'Z' '0-9' '_], i.e. must start with alphabetical characters following by alpha-numeric characters or underscore.

e.g.
Valid: var, var1, var_, Var, v4r_
Invalid: 1var, .var, _var

2.3 Declaration

Variables are declared by the type followed by identifier name and semicolon:

```
int x;
bool y;
float z;
list<int> a;
```

All declarations must occur at the start of any function before other statements and expressions are evaluated.

2.4 Literals

Literals are fixed value constants that do not alter during execution. Variables can be assigned literal values.

int literals have the format: [0-9]+
e.g. 0; 5; 100

float literals have the format:
['0'-'9'] '.' ['0'-'9']* (['e' 'E'] ['+' '-']? ['0'-'9']+)?
e.g. 1., 0.1, 1e4, 1.e5, 2.1E-5

bool literals can have the values {true, false}

string literals are also supported, although string variables are not, and are enclosed by quotes
e.g. "hello world"

list literals can be assigned to list variables and have the format: [literal0, literals 1, ...];
e.g.

```
list<int> a;
a = [0, 1, 2, 3];
list<float> b;
b = [0.1, 1.2, 2.3, 3.4];
list<bool> c;
c = [true, false];
```

2.5 Assignment

In addition to literal assignment, variables can be assigned to other variables as long as that variable is in scope (see Section 3.6). Assignment must occur after the variable declarations at the start of the function. Assignment of primitive typed variables occur by value and are copied to the destination variable's address.

e.g.

```
int x;
int y;
int z;

x = 3;
y = x; /* y == 3, but not the same 3 in memory as the 3 literal in the first line */
```

Variable y is assigned a copy of the value of variable x during assignment. It does not point to x, so any change that occurs to either variable does not reflect the other. All variables of primitive types assign in this manner. Variables of the list type effectively assign by reference and changes made to all assignments are reflected in others.

```
y = 5;
z = x = y; /* x, y and z all have value 5 now */
```

Multiple assignment is possible in the same statement with right-to-left associativity: z = (x = (y)), y returns 5 and is assigned to x, (x=5) returns 5 and is assigned to z. See Section 4.2 for more on operator associativity.

2.6 Scope

AP++ is a statically-scoped language. There are two main scopes - global and local. Global scoped variables are accessible throughout the entire program. Local scoped variables are only accessible within the function they are defined.

e.g.

```
int x; // global scope, accessible from any function

void foo() {
  printi(x);
}

int main() {
   x = 3;
   foo(); // prints 3
}
```

e.g.

```
int main() {
  int x; // local scope, only accessible within this function
}
```

3. Expressions

Expressions consist of a combination of variables, operators and other expressions that return a single value.

3.1 Expression Operators

Operator	**Description**	**Examples**
+	Arithmetic Addition, Binary Operator between two ints	x + y : between 2 vars x + 1 : between var and literal 1 + 2 : between 2 literals (1+2) + (3+4) : between 2 expressions that return type int
-	Arithmetic Subtraction, Binary Operator between two ints	x - y : between 2 vars x - 1 : between var and literal 1 - 2 : between 2 literals (1+2) - (3+4) : between 2 expressions that return type int
/	Arithmetic Division, Binary Operator between two ints	x / y : between 2 vars x / 1 : between var and literal 1 / 2 : between 2 literals (1+2) / (3+4) : between 2 expressions that return type int
*	Arithmetic Multiplication, Binary Operator between two ints	x * y : between 2 vars x * 1 : between var and literal 1 * 2 : between 2 literals (1+2) * (3+4) : between 2 expressions that return type int
%	Modulus	x % y: between 2 int vars x % 2: between int var and literal 1 % 2: between 2 int literals (1+2) % (3+4) : between 2 expressions that return type int
++x	Pre-Increment Operator on variables of int type	int x; int y; x = 3; x++; /* x == 4 */ y = ++x; /* y == 5, x == 5 */

x++	Post-Increment Operator on variable of int type	int x; int y; x = 3; ++x; /* x == 4 */ y = x++; /* y == 4, x == 5 */
>	greater than	int x; x = 2; x > 1 /* true */ x > 4 /* false */
>=	greater than equal to	int x = 2; x >= 2 /* true */ x >= 4 /* false */
<	less than	int x; x = 2; x < 1 /* false */ x < 4 /* true */
<=	less than equal to	int x; x = 2; x <= 2 /* true */ x <= 4 /* true */

&&	boolean AND	bool b1;
		bool b2;
		b1 = true;
		b2 = false;
		
		(b1 && b2) /* false */
		(true && true) /* true */
		(true && false) /* false */
		(false && false) /* false */
		(false && true) /* false */

||	boolean OR	bool b1;
		bool b2 = false;
		b1 = true;
		b2 = false;
		(b1 || b2) /* false */
		(true || true) /* true */
		(true || false) /* false */
		(false || false) /* false */
		(false || true) /* false */

!	boolean NOT	bool b1;
		bool b2;
		b1 = true;
		b2 = false;

		!b1 /* false */ !b2 /* true */

==	Equals	int x;
		x = 2;
		x == 4 /* false */
		true == false /* false */
		int y = 2;
		x == y /* true */
		
		Primitive types compare values, lists compare length and element-element comparison

!=	Not Equals	int x;
		x = 2;
		x != 4 /* true */
		true != false /* true */
		int y;
		y = 2;
		x != y /* false */

=	Assignment	int x;
		int y;
		x = 4;
		y = x;

3.2 Operator Precedence and Associativity

To resolve ambiguity of expressions, we set the following operator precedence and associativity of operators (in order of increasing precedence). Operators on the same level have the same precedence and are evaluated in order according to the defined associativity.

Operator(s)	**Associativity**
=	right-to-left
||	left-to-right
&&	left-to-right
== !=	left-to-right
< <= > >=	left-to-right

+ -	left-to-right
* / %	left-to-right
++ -- !	right-to-left
[:] list subscript () function call	left-to-right

e.g.
int1 + int2 * int3 is evaluated as int1 + (int2 * int3)
expr1 || expr2 && expr3 is evaluated as expr1 || (expr2 && expr3)
!expr1 || !expr2 is evaluated as (!expr) || (!expr2)
!expr1 >= expr2 || expr3 && expr4 < expr5 is evaluated as
((!expr) >= expr2) || (expr && (expr4 < expr5))

3.3 Parenthesis

Parenthesis can be explicitly added to an expression to provide explicit evaluation precedence. The deeper the () nesting, the higher the precedence.

e.g.

40 / 2 + 3 * 4;
default evaluation is (40 / 2) + (3 * 4) = 32
alternative with parens: 40 / ((2 + 3) * 4) = 2

expr1 || expr2 && expr3;
default evaluation is expr1 || (expr2 && expr3)
alternative with parens: (expr1 || expr2) && (expr3)

4. Statements

The bodies of programs in AP++ are made of statements, which occur in the form of simple or compound statements.

4.1 Simple Statements

Simple statements do not contain other statements and are terminated by a semicolon.
E.g.

```
x = 1;       /* assignment */
1+2;         /* in-line expression, which is ignored */
foo(x);      /* function call */
x = foo(x);  /* assignment to value of function call */
```

4.2 Compound Statements

Compound statements contain one or more statements for execution contained in {} blocks and have two general purposes: conditional control flow and iteration.

Conditional Control Flow: *if-else* statements

For conditional control, *AP++* provides *if-else* statements for executing statements upon meeting boolean conditions. both elseif and else blocks are optional.

e.g.

```
if (predicate_expression) {
    /* statements here */
}

if (predicate_expression) {
    /* statements here */
} else (predicate_expression2) {
    /* statements here */
}
```

Iteration:
For iteration, AP++ provides *while* and *for* loop statements for executing statements repeatedly while satisfying a predicate condition.

e.g. *while* loops

```
while (predicate_expression) {
    /* statements here */
}

int x;
int fac;
x = 4;
fac = 1;
while (x > 0) {
    fac = fac * x--;
}
fac /* evaluates to 4! = 24  */
```

e.g. *for* loops

```
for (initial_statement; predicate_expression; iteration_statement) {
    /* statements here */
}

int i;
```

```
int fac;
x = 4;
fac = 1;
for (i=1; i<=x; ++i) {
   fac = fac * i;
}
fac /* evaluates to 4! = 24 */
```

5. Functions

Functions are a group of statements that can be executed via the function identifier and an optional list of function parameters as inputs. A root executing function must exist named *main,* a requirement carried over from the MicroC language.

5.1 Return Types

Functions may return any valid type supported in the language, i.e. *int bool float list<T>* where T is any primitive type. For these types of functions, the last statement must *return* the value of type expected by the function declaration. Functions may also be defined as *void* if they do not expect to return any value.

5.2 Declaration and Definition

Functions are declared and defined in the following format:

```
(type|void) identifier(param0, param1, ...paramN) {
    /* variable declaration statements here */
    /* other statements here */
    return type; // if return type is specific in function declaration
}
```

e.g.

```
int foo() {
    return 3;
}

bool foo(int x) {
    int y;
    y = foo2();
    return x != y;
}

void foo(bool b) {
    /* statements */
}
```

5.3 Parameters

Function parameters are comma delimited and precede by a type identifier. Primitive type parameters are passed by value. That is, the contents of the variable passed into the function call are copied and then passed to the function and assigned to the parameter variable. Changing the value of these parameter variables will not change the value of the original variable. For more complex types such as lists, the variable will still refer to the original value in memory so performing operations on that variable will be reflected in all variables that point to it.

e.g.

```
void square(int x) {
    x = x * x;
}

int z = 3;
foo(z) /* z == 3 after this line */
```

6. Lists

The *list* data structure is an ordered collection of primitive typed elements. Lists in *AP++* are heavily inspired by Python's list syntax and API.

6.1 Declaration

Lists are declared by the *list* keyword along with a primitive type of elements that the list will hold in brackets.

```
list<T> x;                /* default initialization, T is a primitive type */
x = [1, 2, 3];            /* assignment with list literal */
```

The default initialization allocates enough memory to hold 1024 elements, the maximum capacity of a *list* with the current implementation. In the future, this will be a dynamically growing list when capacity is reached.

6.2 List Literals

List literals can be defined with the following format: [literal1, literal2, ...etc] and can be assigned to a list of the same literal type. List literals must have at least one element defined and all elements within the literal must be of the same type.

e.g.

```
list<int> a;
a = [0, 1, 2, 3];

list<float> b;
b = [0.1, 1.2, 2.3, 3.4];

list<bool> c;
c = [true, false, true, false];
```

6.3 List API

List API functions are globally accessible and can be invoked from any scope.

Function	**Description**
int list_size(list a) #a	Returns the number of elements in the list
void list_push(list a)	Pushes element x to the end of the list
T list_get(list a, int i) a[i]	Returns the element at index i

T list_pop(list a)	Removes and returns the last element in the list
void list_insert(list a, int i, T val)	Inserts element val at index i in the list; shifts all elements to the right of the inserted element to the right by 1
int list_find(list a, T val)	Returns index of element val in the list if found, or else -1
void list_remove(list a, T val)	Removes first element that equals val; shifts all elements to the right of the removed element to the left by 1
list<T> list_slice(list a, int i, int j) a[i:j]	Returns a new sublist of a, containing elements a[i] to a[j] inclusively
void list_rev(list a)	Reverses the elements in the list in-place
void list_clear(list a)	Clears all elements in the list in-place

6.4 List Slicing Syntax

With Python-style list slicing syntax, sublists with specified range can be returned. It has the following format: list_identifier[(int):(int)] with the optional ints as starting/ending index (inclusive) span to return from the original list.

e.g.
```
list<int> x;
x = [3, 4, 5, 6, 7, 8];
x[1:4] /* evaluates to [4, 5, 6, 7] */
x[3:]  /* evaluates to [6, 7, 8] */
x[:3]  /* evaluates to [3, 4, 5, 6] */
x[:]   /* returns copy of x, evaluates to [3, 4, 5, 6, 7, 8] */
```

6.5 Local and Function Parameter Variable Assignment

Unlike primitive types, assignment of list types point to the same allocated list in memory. Performing operations on a list will be reflected in all variables that have assigned themselves to that list. Reassigning the variable, however, will not affect the original allocated list's contents, but rather re-point the variable to another allocated list in memory.

e.g.
```
list<int> x;
list<int> y;
list<int> z;
```

```
x = [0, 1, 2];
y = x;
list_pop(y);        /* evaluates to 2, x and y are now both [0, 1] */
x[#x - 1]           /* evaluates to 1 */
z = [3, 4, 5];
z = x;
x                   /* unchanged, still evaluates to [0, 1, 2] */
z                   /* evaluates to [3, 4, 5] */
list_pop(z)         /* evaluates to 5, z is now [3, 4] */
list_append(z, 6);
x                   /* evaluates to [3, 4, 6], same as z

e.g.
void foo(list<int> x) {
    list_pop(x);
    list_append(x, 3);
}

list<int> x;
x = [0, 1, 2];
foo(x);
x; /* evaluates to [0, 1, 3] */
```

IV. Project Plan

The project was broken down into discrete milestones according to the project proposal, apart from the those that that were required by class. Target dates were assigned to each milestone, adding in buffer time for debugging, testing and unknowns. In actuality, the project development timeline did not progress as linearly as outlined by the initial plan. As the milestones progressed, so did the momentum of development as lessons learned from previous milestones carried over. I was able to maintain a decent amount of parity between the final project and the proposal, especially where lists are concerned, and even implemented a couple of extraneous functionality (list_find, list literals). Arbitrary variable declaration w/ assignment and scoping were not ultimately pursued in order to focus the brunt of the efforts on the list API.

1. Style Guide

The style for *AP++* compiler development remained the same that was defined for *MicroC*.

- Variable names are formatted with snake case
- Parser types are upper camel case
- 2-character indentations
- 80 characters per line

2. Planned Timeline

Start Date	Milestone
Sept. 19	Project Proposal
Oct. 15	Project LRM
Nov. 14	Hello World!
Nov. 21	Arbitrary variable declaration w/ assignment
Nov. 24	Scoping
Dec. 1	Lists Phase I: llvm representation, declaration, assignment, push, get
Dec. 8	Lists Phase II: additional list api functions
Dec. 15	cleanup and finalize

2. Project Log

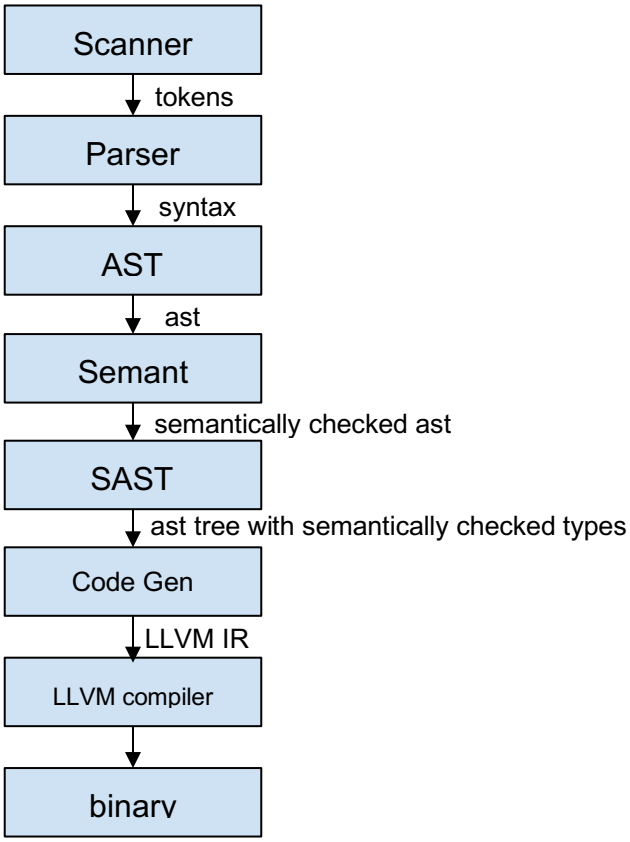
Start Date	Milestone
Sept. 19	Project Proposal
Oct. 15	Project LRM
Nov. 14	Hello World!
Nov. 25	pre/post ++/--
Dec. 6	LLVM representation of a List
Dec. 9	list declaration
Dec. 10	list_push
Dec. 10	list_size
Dec. 11	list_get
Dec. 12	list assignment by ref
Dec. 13	list_pop
Dec. 13	list_set and bracket syntax, i.e. a[i] = x
Dec. 14	list_insert
Dec. 15	list_remove
Dec. 15	list_find (extra)
Dec. 16	list_slice
Dec. 17	slice syntactical sugar
Dec. 17	mod, single-line comments
Dec. 18	list literals (extra)
Dec. 18	cleanup and finalize

3. Environments and Tools

I chose to develop locally instead of using the provided VM or Docker

- Libraries and Languages: OCaml version 4.07.1, OCaml LLVM 7.0.0
- Software: text editor - SublimeText / Vim, source control - GitHub
- OS: Mac OSX 10.13.5

V. Architectural Design



1. Scanner (scanner.mll)
The scanner scans the *AP++* source code to identify and generates tokens from of the input stream. The tokens are discrete sequences of characters that are pattern matched, and generally consists of text that will be assigned a meaning in a later step in the compilation process. Keywords, operators, built-in list functions, variable names and other types of tokens are identified by regex patterns.

2. Parser (parser.mly / ast.ml)
The parser converts the stream of tokens from the scanner into an abstract syntax (AST) tree in order to assign some more structural meaning to the underlying source code. The parser uses context-free grammar rules, token precedence and associativity rules to generate the AST.

3. Semantic Checking (semant.ml / sast.ml)

In this step of the compilation process, the AST is traversed and type-checking is done to ensure the semantic meaning behind the tree makes sense. Many compile-time errors, e.g. type compatibility, definition existence are caught in this step. The AST is then converted into a semantically checked AST called SAST, which holds two pieces of data per node: type and value.

4. Code Generation
The ast is traversed in this step to evaluate the tree and generate an intermediate representation (IR) of lower-level instructions. In our case, we compile *AP++* source code into LLVM IR.

The LLVM IR is then able to be compiled further (e.g. by clang/gcc) into the machine code binary executable.

VI. Test Plan

1. Sample Programs

Sample 1: Bubble Sort

```
/* sorts a list in-place with BubbleSort algorithm, */

void bubbleSort(list<int> arr) {
   int i;
   int j;
   int tmp;
   for (i = 0; i < #arr-1; i++) {
       for (j = 0; j < #arr-i-1; j++) {
           if (arr[j] > arr[j+1]) {
              tmp = arr[j];
              arr[j] = arr[j+1];
              arr[j+1] = tmp;
           }
       }
   }
}

int main() {
    list<int> a;
    int i;
    a = [10, 2, 15, 8, 20, 13, 9];
    bubbleSort(a);
    for(i = 0; i < #a; ++i) {
        printi(a[i]);
    }
    return 0;
}

output:
2
8
9
13
10
15
13
20
```

Target Language Output (LLVM IR):

```
; ModuleID = 'AP_PlusPlus'
source_filename = "AP_PlusPlus"

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

```
@fmt.2 = private unnamed_addr constant [4 x i8] c"%s\0A\00"
@fmt.3 = private unnamed_addr constant [4 x i8] c"%d\0A\00"
@fmt.4 = private unnamed_addr constant [4 x i8] c"%g\0A\00"
@fmt.5 = private unnamed_addr constant [4 x i8] c"%s\0A\00"

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

define i1 @list_getbool({ i32*, i1* }*, i32) {
entry:
  %list_ptr_alloc = alloca { i32*, i1* }*
  store { i32*, i1* }* %0, { i32*, i1* }** %list_ptr_alloc
  %idx_alloc = alloca i32
  store i32 %1, i32* %idx_alloc
  %list_load = load { i32*, i1* }*, { i32*, i1* }** %list_ptr_alloc
  %list_array_ptr = getelementptr inbounds { i32*, i1* }, { i32*, i1* }* %list_load, i32 0, i32 1
  %array_load = load i1*, i1** %list_array_ptr
  %idx_load = load i32, i32* %idx_alloc
  %list_arry_element_ptr = getelementptr i1, i1* %array_load, i32 %idx_load
  %list_array_element_ptr = load i1, i1* %list_arry_element_ptr
  ret i1 %list_array_element_ptr
}

define i32 @list_getint({ i32*, i32* }*, i32) {
entry:
  %list_ptr_alloc = alloca { i32*, i32* }*
  store { i32*, i32* }* %0, { i32*, i32* }** %list_ptr_alloc
  %idx_alloc = alloca i32
  store i32 %1, i32* %idx_alloc
  %list_load = load { i32*, i32* }*, { i32*, i32* }** %list_ptr_alloc
  %list_array_ptr = getelementptr inbounds { i32*, i32* }, { i32*, i32* }* %list_load, i32 0, i32 1
  %array_load = load i32*, i32** %list_array_ptr
  %idx_load = load i32, i32* %idx_alloc
  %list_arry_element_ptr = getelementptr i32, i32* %array_load, i32 %idx_load
  %list_array_element_ptr = load i32, i32* %list_arry_element_ptr
  ret i32 %list_array_element_ptr
}

define double @list_getfloat({ i32*, double* }*, i32) {
entry:
  %list_ptr_alloc = alloca { i32*, double* }*
  store { i32*, double* }* %0, { i32*, double* }** %list_ptr_alloc
  %idx_alloc = alloca i32
  store i32 %1, i32* %idx_alloc
  %list_load = load { i32*, double* }*, { i32*, double* }** %list_ptr_alloc
  %list_array_ptr = getelementptr inbounds { i32*, double* }, { i32*, double* }* %list_load, i32 0, i32 1
  %array_load = load double*, double** %list_array_ptr
  %idx_load = load i32, i32* %idx_alloc
  %list_arry_element_ptr = getelementptr double, double* %array_load, i32 %idx_load
  %list_array_element_ptr = load double, double* %list_arry_element_ptr
```

```
  ret double %list_array_element_ptr
}

define i8* @list_getstr({ i32*, i8** }*, i32) {
entry:
  %list_ptr_alloc = alloca { i32*, i8** }*
  store { i32*, i8** }* %0, { i32*, i8** }** %list_ptr_alloc
  %idx_alloc = alloca i32
  store i32 %1, i32* %idx_alloc
  %list_load = load { i32*, i8** }*, { i32*, i8** }** %list_ptr_alloc
  %list_array_ptr = getelementptr inbounds { i32*, i8** }, { i32*, i8** }* %list_load, i32 0, i32 1
  %array_load = load i8**, i8*** %list_array_ptr
  %idx_load = load i32, i32* %idx_alloc
  %list_arry_element_ptr = getelementptr i8*, i8** %array_load, i32 %idx_load
  %list_array_element_ptr = load i8*, i8** %list_arry_element_ptr
  ret i8* %list_array_element_ptr
}

define void @list_setbool({ i32*, i1* }*, i32, i1) {
entry:
  %list_ptr_alloc = alloca { i32*, i1* }*
  store { i32*, i1* }* %0, { i32*, i1* }** %list_ptr_alloc
  %list_load = load { i32*, i1* }*, { i32*, i1* }** %list_ptr_alloc
  %list_array_ptr = getelementptr inbounds { i32*, i1* }, { i32*, i1* }* %list_load, i32 0, i32 1
  %list_array_load = load i1*, i1** %list_array_ptr
  %list_arry_next_element_ptr = getelementptr i1, i1* %list_array_load, i32 %1
  store i1 %2, i1* %list_arry_next_element_ptr
  ret void
}

define void @list_setint({ i32*, i32* }*, i32, i32) {
entry:
  %list_ptr_alloc = alloca { i32*, i32* }*
  store { i32*, i32* }* %0, { i32*, i32* }** %list_ptr_alloc
  %list_load = load { i32*, i32* }*, { i32*, i32* }** %list_ptr_alloc
  %list_array_ptr = getelementptr inbounds { i32*, i32* }, { i32*, i32* }* %list_load, i32 0, i32 1
  %list_array_load = load i32*, i32** %list_array_ptr
  %list_arry_next_element_ptr = getelementptr i32, i32* %list_array_load, i32 %1
  store i32 %2, i32* %list_arry_next_element_ptr
  ret void
}

define void @list_setfloat({ i32*, double* }*, i32, double) {
entry:
  %list_ptr_alloc = alloca { i32*, double* }*
   store { i32*, double* }* %0, { i32*, double* }** %list_ptr_alloc
   %list_load = load { i32*, double* }*, { i32*, double* }** %list_ptr_alloc
   %list_array_ptr = getelementptr inbounds { i32*, double* }, { i32*, double* }* %list_load, i32 0, i32 1
```

```
  %list_array_load = load double*, double** %list_array_ptr
  %list_arry_next_element_ptr = getelementptr double, double* %list_array_load, i32 %1
  store double %2, double* %list_arry_next_element_ptr
  ret void
}

define void @list_setstr({ i32*, i8** }*, i32, i8*) {
entry:
  %list_ptr_alloc = alloca { i32*, i8** }*
  store { i32*, i8** }* %0, { i32*, i8** }** %list_ptr_alloc
  %list_load = load { i32*, i8** }*, { i32*, i8** }** %list_ptr_alloc
  %list_array_ptr = getelementptr inbounds { i32*, i8** }, { i32*, i8** }* %list_load, i32 0, i32 1
  %list_array_load = load i8**, i8*** %list_array_ptr
  %list_arry_next_element_ptr = getelementptr i8*, i8** %list_array_load, i32 %1
  store i8* %2, i8** %list_arry_next_element_ptr
  ret void
}

define void @list_pushbool({ i32*, i1* }*, i1) {
entry:
  %list_ptr_alloc = alloca { i32*, i1* }*
  store { i32*, i1* }* %0, { i32*, i1* }** %list_ptr_alloc
  %val_alloc = alloca i1
  store i1 %1, i1* %val_alloc
  %list_load = load { i32*, i1* }*, { i32*, i1* }** %list_ptr_alloc
  %list_array_ptr = getelementptr inbounds { i32*, i1* }, { i32*, i1* }* %list_load, i32 0, i32 1
  %list_array_load = load i1*, i1** %list_array_ptr
  %list_size_ptr_ptr = getelementptr inbounds { i32*, i1* }, { i32*, i1* }* %list_load, i32 0, i32 0
  %list_size_ptr = load i32*, i32** %list_size_ptr_ptr
  %list_size = load i32, i32* %list_size_ptr
  %list_arry_next_element_ptr = getelementptr i1, i1* %list_array_load, i32 %list_size
  %inc_size = add i32 %list_size, 1
  store i32 %inc_size, i32* %list_size_ptr
  %val = load i1, i1* %val_alloc
  store i1 %val, i1* %list_arry_next_element_ptr
  ret void
}

define void @list_pushint({ i32*, i32* }*, i32) {
entry:
  %list_ptr_alloc = alloca { i32*, i32* }*
  store { i32*, i32* }* %0, { i32*, i32* }** %list_ptr_alloc
  %val_alloc = alloca i32
  store i32 %1, i32* %val_alloc
  %list_load = load { i32*, i32* }*, { i32*, i32* }** %list_ptr_alloc
  %list_array_ptr = getelementptr inbounds { i32*, i32* }, { i32*, i32* }* %list_load, i32 0, i32 1
  %list_array_load = load i32*, i32** %list_array_ptr
```

```
  %list_size_ptr_ptr = getelementptr inbounds { i32*, i32* }, { i32*, i32* }* %list_load, i32 0, i32 0
  %list_size_ptr = load i32*, i32** %list_size_ptr_ptr
  %list_size = load i32, i32* %list_size_ptr
  %list_arry_next_element_ptr = getelementptr i32, i32* %list_array_load, i32 %list_size
  %inc_size = add i32 %list_size, 1
  store i32 %inc_size, i32* %list_size_ptr
  %val = load i32, i32* %val_alloc
  store i32 %val, i32* %list_arry_next_element_ptr
  ret void
}

define void @list_pushfloat({ i32*, double* }*, double) {
entry:
  %list_ptr_alloc = alloca { i32*, double* }*
  store { i32*, double* }* %0, { i32*, double* }** %list_ptr_alloc
  %val_alloc = alloca double
  store double %1, double* %val_alloc
  %list_load = load { i32*, double* }*, { i32*, double* }** %list_ptr_alloc
  %list_array_ptr = getelementptr inbounds { i32*, double* }, { i32*, double* }* %list_load, i32 0, i32 1
  %list_array_load = load double*, double** %list_array_ptr
  %list_size_ptr_ptr = getelementptr inbounds { i32*, double* }, { i32*, double* }* %list_load, i32 0, i32 0
  %list_size_ptr = load i32*, i32** %list_size_ptr_ptr
  %list_size = load i32, i32* %list_size_ptr
  %list_arry_next_element_ptr = getelementptr double, double* %list_array_load, i32 %list_size
  %inc_size = add i32 %list_size, 1
  store i32 %inc_size, i32* %list_size_ptr
  %val = load double, double* %val_alloc
  store double %val, double* %list_arry_next_element_ptr
  ret void
}

define void @list_pushstr({ i32*, i8** }*, i8*) {
entry:
  %list_ptr_alloc = alloca { i32*, i8** }*
  store { i32*, i8** }* %0, { i32*, i8** }** %list_ptr_alloc
  %val_alloc = alloca i8*
  store i8* %1, i8** %val_alloc
  %list_load = load { i32*, i8** }*, { i32*, i8** }** %list_ptr_alloc
  %list_array_ptr = getelementptr inbounds { i32*, i8** }, { i32*, i8** }* %list_load, i32 0, i32 1
  %list_array_load = load i8**, i8*** %list_array_ptr
  %list_size_ptr_ptr = getelementptr inbounds { i32*, i8** }, { i32*, i8** }* %list_load, i32 0, i32 0
  %list_size_ptr = load i32*, i32** %list_size_ptr_ptr
  %list_size = load i32, i32* %list_size_ptr
  %list_arry_next_element_ptr = getelementptr i8*, i8** %list_array_load, i32 %list_size
  %inc_size = add i32 %list_size, 1
```

```
  store i32 %inc_size, i32* %list_size_ptr
  %val = load i8*, i8** %val_alloc
  store i8* %val, i8** %list_arry_next_element_ptr
  ret void
}

define i1 @list_popbool({ i32*, i1* }*) {
entry:
  %list_ptr_alloc = alloca { i32*, i1* }*
  store { i32*, i1* }* %0, { i32*, i1* }** %list_ptr_alloc
  %list_load = load { i32*, i1* }*, { i32*, i1* }** %list_ptr_alloc
  %list_array_ptr = getelementptr inbounds { i32*, i1* }, { i32*, i1* }* %list_load, i32 0, i32 1
  %list_array_load = load i1*, i1** %list_array_ptr
  %list_size_ptr_ptr = getelementptr inbounds { i32*, i1* }, { i32*, i1* }* %list_load, i32 0, i32 0
  %list_size_ptr = load i32*, i32** %list_size_ptr_ptr
  %list_size = load i32, i32* %list_size_ptr
  %dec_size = sub i32 %list_size, 1
  %list_arry_next_element_ptr = getelementptr i1, i1* %list_array_load, i32 %dec_size
  %list_arry_next_element = load i1, i1* %list_arry_next_element_ptr
  store i32 %dec_size, i32* %list_size_ptr
  ret i1 %list_arry_next_element
}

define i32 @list_popint({ i32*, i32* }*) {
entry:
  %list_ptr_alloc = alloca { i32*, i32* }*
  store { i32*, i32* }* %0, { i32*, i32* }** %list_ptr_alloc
  %list_load = load { i32*, i32* }*, { i32*, i32* }** %list_ptr_alloc
  %list_array_ptr = getelementptr inbounds { i32*, i32* }, { i32*, i32* }* %list_load, i32 0, i32 1
  %list_array_load = load i32*, i32** %list_array_ptr
  %list_size_ptr_ptr = getelementptr inbounds { i32*, i32* }, { i32*, i32* }* %list_load, i32 0, i32 0
  %list_size_ptr = load i32*, i32** %list_size_ptr_ptr
  %list_size = load i32, i32* %list_size_ptr
  %dec_size = sub i32 %list_size, 1
  %list_arry_next_element_ptr = getelementptr i32, i32* %list_array_load, i32 %dec_size
  %list_arry_next_element = load i32, i32* %list_arry_next_element_ptr
  store i32 %dec_size, i32* %list_size_ptr
  ret i32 %list_arry_next_element
}

define double @list_popfloat({ i32*, double* }*) {
entry:
  %list_ptr_alloc = alloca { i32*, double* }*
  store { i32*, double* }* %0, { i32*, double* }** %list_ptr_alloc
  %list_load = load { i32*, double* }*, { i32*, double* }** %list_ptr_alloc
  %list_array_ptr = getelementptr inbounds { i32*, double* }, { i32*, double* }* %list_load, i32 0, i32 1
  %list_array_load = load double*, double** %list_array_ptr
```

```
  %list_size_ptr_ptr = getelementptr inbounds { i32*, double* }, { i32*, double* }* %list_load, i32 0, i32 0
  %list_size_ptr = load i32*, i32** %list_size_ptr_ptr
  %list_size = load i32, i32* %list_size_ptr
  %dec_size = sub i32 %list_size, 1
  %list_arry_next_element_ptr = getelementptr double, double* %list_array_load, i32 %dec_size
  %list_arry_next_element = load double, double* %list_arry_next_element_ptr
  store i32 %dec_size, i32* %list_size_ptr
  ret double %list_arry_next_element
}

define i8* @list_popstr({ i32*, i8** }*) {
entry:
  %list_ptr_alloc = alloca { i32*, i8** }*
  store { i32*, i8** }* %0, { i32*, i8** }** %list_ptr_alloc
  %list_load = load { i32*, i8** }*, { i32*, i8** }** %list_ptr_alloc
  %list_array_ptr = getelementptr inbounds { i32*, i8** }, { i32*, i8** }* %list_load, i32 0, i32 1
  %list_array_load = load i8**, i8*** %list_array_ptr
  %list_size_ptr_ptr = getelementptr inbounds { i32*, i8** }, { i32*, i8** }* %list_load, i32 0, i32 0
  %list_size_ptr = load i32*, i32** %list_size_ptr_ptr
  %list_size = load i32, i32* %list_size_ptr
  %dec_size = sub i32 %list_size, 1
  %list_arry_next_element_ptr = getelementptr i8*, i8** %list_array_load, i32 %dec_size
  %list_arry_next_element = load i8*, i8** %list_arry_next_element_ptr
  store i32 %dec_size, i32* %list_size_ptr
  ret i8* %list_arry_next_element
}

define i32 @list_sizebool({ i32*, i1* }*) {
entry:
  %list_ptr_alloc = alloca { i32*, i1* }*
  store { i32*, i1* }* %0, { i32*, i1* }** %list_ptr_alloc
  %list_load = load { i32*, i1* }*, { i32*, i1* }** %list_ptr_alloc
  %list_size_ptr_ptr = getelementptr inbounds { i32*, i1* }, { i32*, i1* }* %list_load, i32 0, i32 0
  %list_size_ptr = load i32*, i32** %list_size_ptr_ptr
  %list_size = load i32, i32* %list_size_ptr
  ret i32 %list_size
}

define i32 @list_sizeint({ i32*, i32* }*) {
entry:
  %list_ptr_alloc = alloca { i32*, i32* }*
  store { i32*, i32* }* %0, { i32*, i32* }** %list_ptr_alloc
  %list_load = load { i32*, i32* }*, { i32*, i32* }** %list_ptr_alloc
  %list_size_ptr_ptr = getelementptr inbounds { i32*, i32* }, { i32*, i32* }* %list_load, i32 0, i32 0
  %list_size_ptr = load i32*, i32** %list_size_ptr_ptr
  %list_size = load i32, i32* %list_size_ptr
```

```
  ret i32 %list_size
}

define i32 @list_sizefloat({ i32*, double* }*) {
entry:
  %list_ptr_alloc = alloca { i32*, double* }*
  store { i32*, double* }* %0, { i32*, double* }** %list_ptr_alloc
  %list_load = load { i32*, double* }*, { i32*, double* }** %list_ptr_alloc
  %list_size_ptr_ptr = getelementptr inbounds { i32*, double* }, { i32*, double* }* %list_load, i32 0, i32 0
  %list_size_ptr = load i32*, i32** %list_size_ptr_ptr
  %list_size = load i32, i32* %list_size_ptr
  ret i32 %list_size
}

define i32 @list_sizestr({ i32*, i8** }*) {
entry:
  %list_ptr_alloc = alloca { i32*, i8** }*
  store { i32*, i8** }* %0, { i32*, i8** }** %list_ptr_alloc
  %list_load = load { i32*, i8** }*, { i32*, i8** }** %list_ptr_alloc
  %list_size_ptr_ptr = getelementptr inbounds { i32*, i8** }, { i32*, i8** }* %list_load, i32 0, i32 0
  %list_size_ptr = load i32*, i32** %list_size_ptr_ptr
  %list_size = load i32, i32* %list_size_ptr
  ret i32 %list_size
}

define void @list_slicebool({ i32*, i1* }*, { i32*, i1* }*, i32, i32) {
entry:
  %list_ptr_alloc = alloca { i32*, i1* }*
  store { i32*, i1* }* %0, { i32*, i1* }** %list_ptr_alloc
  %list_ptr_ptr = load { i32*, i1* }*, { i32*, i1* }** %list_ptr_alloc
  %list_ptr_alloc2 = alloca { i32*, i1* }*
  store { i32*, i1* }* %1, { i32*, i1* }** %list_ptr_alloc2
  %list_ptr_ptr2 = load { i32*, i1* }*, { i32*, i1* }** %list_ptr_alloc2
  %idx_alloc = alloca i32
  store i32 %2, i32* %idx_alloc
  %idx_load = load i32, i32* %idx_alloc
  %idx_alloc1 = alloca i32
  store i32 %3, i32* %idx_alloc1
  %idx_load2 = load i32, i32* %idx_alloc1
  %loop_cnt = alloca i32
  store i32 0, i32* %loop_cnt
  %loop_upper_bound = sub i32 %idx_load2, %idx_load
  br label %while

while:                                            ; preds = %while_body, %entry
  %loop_cnt4 = load i32, i32* %loop_cnt
  %loop_cond = icmp sle i32 %loop_cnt4, %loop_upper_bound
  br i1 %loop_cond, label %while_body, label %merge

while_body:                                       ; preds = %while
```

```
  %to_idx = load i32, i32* %loop_cnt
  %from_idx = add i32 %to_idx, %idx_load
  %list_get = call i1 @list_getbool({ i32*, i1* }* %list_ptr_ptr, i32 %from_idx)
  call void @list_pushbool({ i32*, i1* }* %list_ptr_ptr2, i1 %list_get)
  %loop_cnt3 = load i32, i32* %loop_cnt
  %loop_itr = add i32 %loop_cnt3, 1
  store i32 %loop_itr, i32* %loop_cnt
  br label %while

merge:                                            ; preds = %while
  ret void
}

define void @list_sliceint({ i32*, i32* }*, { i32*, i32* }*, i32, i32) {
entry:
  %list_ptr_alloc = alloca { i32*, i32* }*
  store { i32*, i32* }* %0, { i32*, i32* }** %list_ptr_alloc
  %list_ptr_ptr = load { i32*, i32* }*, { i32*, i32* }** %list_ptr_alloc
  %list_ptr_alloc2 = alloca { i32*, i32* }*
  store { i32*, i32* }* %1, { i32*, i32* }** %list_ptr_alloc2
  %list_ptr_ptr2 = load { i32*, i32* }*, { i32*, i32* }** %list_ptr_alloc2
  %idx_alloc = alloca i32
  store i32 %2, i32* %idx_alloc
  %idx_load = load i32, i32* %idx_alloc
  %idx_alloc1 = alloca i32
  store i32 %3, i32* %idx_alloc1
  %idx_load2 = load i32, i32* %idx_alloc1
  %loop_cnt = alloca i32
  store i32 0, i32* %loop_cnt
  %loop_upper_bound = sub i32 %idx_load2, %idx_load
  br label %while

while:                                            ; preds = %while_body, %entry
  %loop_cnt4 = load i32, i32* %loop_cnt
  %loop_cond = icmp sle i32 %loop_cnt4, %loop_upper_bound
  br i1 %loop_cond, label %while_body, label %merge

while_body:                                       ; preds = %while
  %to_idx = load i32, i32* %loop_cnt
  %from_idx = add i32 %to_idx, %idx_load
  %list_get = call i32 @list_getint({ i32*, i32* }* %list_ptr_ptr, i32 %from_idx)
  call void @list_pushint({ i32*, i32* }* %list_ptr_ptr2, i32 %list_get)
  %loop_cnt3 = load i32, i32* %loop_cnt
  %loop_itr = add i32 %loop_cnt3, 1
  store i32 %loop_itr, i32* %loop_cnt
  br label %while

merge:                                            ; preds = %while
  ret void
}

define void @list_slicefloat({ i32*, double* }*, { i32*, double* }*, i32, i32) {
```

```
entry:
  %list_ptr_alloc = alloca { i32*, double* }*
  store { i32*, double* }* %0, { i32*, double* }** %list_ptr_alloc
  %list_ptr_ptr = load { i32*, double* }*, { i32*, double* }** %list_ptr_alloc
  %list_ptr_alloc2 = alloca { i32*, double* }*
  store { i32*, double* }* %1, { i32*, double* }** %list_ptr_alloc2
  %list_ptr_ptr2 = load { i32*, double* }*, { i32*, double* }** %list_ptr_alloc2
  %idx_alloc = alloca i32
  store i32 %2, i32* %idx_alloc
  %idx_load = load i32, i32* %idx_alloc
  %idx_alloc1 = alloca i32
  store i32 %3, i32* %idx_alloc1
  %idx_load2 = load i32, i32* %idx_alloc1
  %loop_cnt = alloca i32
  store i32 0, i32* %loop_cnt
  %loop_upper_bound = sub i32 %idx_load2, %idx_load
  br label %while

while:                                            ; preds = %while_body, %entry
  %loop_cnt4 = load i32, i32* %loop_cnt
  %loop_cond = icmp sle i32 %loop_cnt4, %loop_upper_bound
  br i1 %loop_cond, label %while_body, label %merge

while_body:                                       ; preds = %while
  %to_idx = load i32, i32* %loop_cnt
  %from_idx = add i32 %to_idx, %idx_load
  %list_get = call double @list_getfloat({ i32*, double* }* %list_ptr_ptr,
i32 %from_idx)
  call void @list_pushfloat({ i32*, double* }* %list_ptr_ptr2, double %list_get)
  %loop_cnt3 = load i32, i32* %loop_cnt
  %loop_itr = add i32 %loop_cnt3, 1
  store i32 %loop_itr, i32* %loop_cnt
  br label %while

merge:                                            ; preds = %while
  ret void
}

define void @list_slicestr({ i32*, i8** }*, { i32*, i8** }*, i32, i32) {
entry:
  %list_ptr_alloc = alloca { i32*, i8** }*
  store { i32*, i8** }* %0, { i32*, i8** }** %list_ptr_alloc
  %list_ptr_ptr = load { i32*, i8** }*, { i32*, i8** }** %list_ptr_alloc
  %list_ptr_alloc2 = alloca { i32*, i8** }*
  store { i32*, i8** }* %1, { i32*, i8** }** %list_ptr_alloc2
  %list_ptr_ptr2 = load { i32*, i8** }*, { i32*, i8** }** %list_ptr_alloc2
  %idx_alloc = alloca i32
  store i32 %2, i32* %idx_alloc
  %idx_load = load i32, i32* %idx_alloc
  %idx_alloc1 = alloca i32
  store i32 %3, i32* %idx_alloc1
  %idx_load2 = load i32, i32* %idx_alloc1
```

```
  %loop_cnt = alloca i32
  store i32 0, i32* %loop_cnt
  %loop_upper_bound = sub i32 %idx_load2, %idx_load
  br label %while

while:                                            ; preds = %while_body, %entry
  %loop_cnt4 = load i32, i32* %loop_cnt
  %loop_cond = icmp sle i32 %loop_cnt4, %loop_upper_bound
  br i1 %loop_cond, label %while_body, label %merge

while_body:                                       ; preds = %while
  %to_idx = load i32, i32* %loop_cnt
  %from_idx = add i32 %to_idx, %idx_load
  %list_get = call i8* @list_getstr({ i32*, i8** }* %list_ptr_ptr, i32 %from_idx)
  call void @list_pushstr({ i32*, i8** }* %list_ptr_ptr2, i8* %list_get)
  %loop_cnt3 = load i32, i32* %loop_cnt
  %loop_itr = add i32 %loop_cnt3, 1
  store i32 %loop_itr, i32* %loop_cnt
  br label %while

merge:                                            ; preds = %while
  ret void
}

define i32 @list_findbool({ i32*, i1* }*, i1) {
entry:
  %list_ptr_alloc = alloca { i32*, i1* }*
  store { i32*, i1* }* %0, { i32*, i1* }** %list_ptr_alloc
  %find_val_alloc = alloca i1
  store i1 %1, i1* %find_val_alloc
  %find_val = load i1, i1* %find_val_alloc
  %list_load = load { i32*, i1* }*, { i32*, i1* }** %list_ptr_alloc
  %list_size_ptr_ptr = getelementptr inbounds { i32*, i1* }, { i32*, i1* }* %list_load, i32 0, i32 0
  %list_size_ptr = load i32*, i32** %list_size_ptr_ptr
  %list_size = load i32, i32* %list_size_ptr
  %loop_cnt = alloca i32
  store i32 0, i32* %loop_cnt
  br label %while

while:                                            ; preds = %merge, %entry
  %loop_iter_cnt = load i32, i32* %loop_cnt
  %loop_cond = icmp slt i32 %loop_iter_cnt, %list_size
  br i1 %loop_cond, label %while_body, label %merge1

while_body:                                       ; preds = %while
  %to_idx = load i32, i32* %loop_cnt
  %list_get = call i1 @list_getbool({ i32*, i1* }* %list_load, i32 %to_idx)
  %if_cond = icmp eq i1 %list_get, %find_val
  br i1 %if_cond, label %then, label %else

merge:                                            ; preds = %else
```

```
  %loop_idx = load i32, i32* %loop_cnt
  %loop_itr = add i32 %loop_idx, 1
  store i32 %loop_itr, i32* %loop_cnt
  br label %while

then:                                             ; preds = %while_body
  ret i32 %to_idx

else:                                             ; preds = %while_body
  br label %merge

merge1:                                           ; preds = %while
  ret i32 -1
}

define i32 @list_findint({ i32*, i32* }*, i32) {
entry:
  %list_ptr_alloc = alloca { i32*, i32* }*
  store { i32*, i32* }* %0, { i32*, i32* }** %list_ptr_alloc
  %find_val_alloc = alloca i32
  store i32 %1, i32* %find_val_alloc
  %find_val = load i32, i32* %find_val_alloc
  %list_load = load { i32*, i32* }*, { i32*, i32* }** %list_ptr_alloc
  %list_size_ptr_ptr = getelementptr inbounds { i32*, i32* }, { i32*,
i32* }* %list_load, i32 0, i32 0
  %list_size_ptr = load i32*, i32** %list_size_ptr_ptr
  %list_size = load i32, i32* %list_size_ptr
  %loop_cnt = alloca i32
  store i32 0, i32* %loop_cnt
  br label %while

while:                                            ; preds = %merge, %entry
  %loop_iter_cnt = load i32, i32* %loop_cnt
  %loop_cond = icmp slt i32 %loop_iter_cnt, %list_size
  br i1 %loop_cond, label %while_body, label %merge1

while_body:                                       ; preds = %while
  %to_idx = load i32, i32* %loop_cnt
  %list_get = call i32 @list_getint({ i32*, i32* }* %list_load, i32 %to_idx)
  %if_cond = icmp eq i32 %list_get, %find_val
  br i1 %if_cond, label %then, label %else

merge:                                            ; preds = %else
  %loop_idx = load i32, i32* %loop_cnt
  %loop_itr = add i32 %loop_idx, 1
  store i32 %loop_itr, i32* %loop_cnt
  br label %while

then:                                             ; preds = %while_body
  ret i32 %to_idx

```

```
else:                                             ; preds = %while_body
  br label %merge

merge1:                                           ; preds = %while
  ret i32 -1
}

define i32 @list_findfloat({ i32*, double* }*, double) {
entry:
  %list_ptr_alloc = alloca { i32*, double* }*
  store { i32*, double* }* %0, { i32*, double* }** %list_ptr_alloc
  %find_val_alloc = alloca double
  store double %1, double* %find_val_alloc
  %find_val = load double, double* %find_val_alloc
  %list_load = load { i32*, double* }*, { i32*, double* }** %list_ptr_alloc
  %list_size_ptr_ptr = getelementptr inbounds { i32*, double* }, { i32*,
double* }* %list_load, i32 0, i32 0
  %list_size_ptr = load i32*, i32** %list_size_ptr_ptr
  %list_size = load i32, i32* %list_size_ptr
  %loop_cnt = alloca i32
  store i32 0, i32* %loop_cnt
  br label %while

while:                                            ; preds = %merge, %entry
  %loop_iter_cnt = load i32, i32* %loop_cnt
  %loop_cond = icmp slt i32 %loop_iter_cnt, %list_size
  br i1 %loop_cond, label %while_body, label %merge1

while_body:                                       ; preds = %while
  %to_idx = load i32, i32* %loop_cnt
  %list_get = call double @list_getfloat({ i32*, double* }* %list_load, i32 %to_idx)
  %if_cond = fcmp oeq double %list_get, %find_val
  br i1 %if_cond, label %then, label %else

merge:                                            ; preds = %else
  %loop_idx = load i32, i32* %loop_cnt
  %loop_itr = add i32 %loop_idx, 1
  store i32 %loop_itr, i32* %loop_cnt
  br label %while

then:                                             ; preds = %while_body
  ret i32 %to_idx

else:                                             ; preds = %while_body
  br label %merge

merge1:                                           ; preds = %while
  ret i32 -1
}

define void @list_removebool({ i32*, i1* }*, i1) {
```

```
entry:
  %list_ptr_alloc = alloca { i32*, i1* }*
  store { i32*, i1* }* %0, { i32*, i1* }** %list_ptr_alloc
  %rem_val_ptr = alloca i1
  store i1 %1, i1* %rem_val_ptr
  %rem_val = load i1, i1* %rem_val_ptr
  %list_load = load { i32*, i1* }*, { i32*, i1* }** %list_ptr_alloc
  %list_size_ptr_ptr = getelementptr inbounds { i32*, i1* }, { i32*, i1* }* %list_load, i32 0, i32 0
  %list_size_ptr = load i32*, i32** %list_size_ptr_ptr
  %list_size = load i32, i32* %list_size_ptr
  %list_find = call i32 @list_findbool({ i32*, i1* }* %list_load, i1 %rem_val)
  %loop_cond = icmp sge i32 %list_find, 0
  br i1 %loop_cond, label %then, label %else

merge:                                            ; preds = %else
  ret void

then:                                             ; preds = %entry
  %loop_cnt_ptr = alloca i32
  %loop_start_idx = add i32 %list_find, 1
  store i32 %loop_start_idx, i32* %loop_cnt_ptr
  br label %while

while:                                            ; preds = %while_body, %then
  %loop_cnt = load i32, i32* %loop_cnt_ptr
  %loop_cond1 = icmp slt i32 %loop_cnt, %list_size
  br i1 %loop_cond1, label %while_body, label %merge2

while_body:                                       ; preds = %while
  %cur_idx = load i32, i32* %loop_cnt_ptr
  %shift_to_idx = sub i32 %cur_idx, 1
  %list_get = call i1 @list_getbool({ i32*, i1* }* %list_load, i32 %cur_idx)
  call void @list_setbool({ i32*, i1* }* %list_load, i32 %shift_to_idx, i1 %list_get)
  %loop_itr = add i32 %cur_idx, 1
  store i32 %loop_itr, i32* %loop_cnt_ptr
  br label %while

merge2:                                           ; preds = %while
  %size_dec = sub i32 %list_size, 1
  store i32 %size_dec, i32* %list_size_ptr
  ret void

else:                                             ; preds = %entry
  br label %merge
}

define void @list_removeint({ i32*, i32* }*, i32) {
entry:
  %list_ptr_alloc = alloca { i32*, i32* }*
  store { i32*, i32* }* %0, { i32*, i32* }** %list_ptr_alloc
  %rem_val_ptr = alloca i32
```

```
643.   store i32 %1, i32* %rem_val_ptr
644.   %rem_val = load i32, i32* %rem_val_ptr
645.   %list_load = load { i32*, i32* }*, { i32*, i32* }** %list_ptr_alloc
646.   %list_size_ptr_ptr = getelementptr inbounds { i32*, i32* }, { i32*,
    i32* }* %list_load, i32 0, i32 0
647.   %list_size_ptr = load i32*, i32** %list_size_ptr_ptr
648.   %list_size = load i32, i32* %list_size_ptr
649.   %list_find = call i32 @list_findint({ i32*, i32* }* %list_load, i32 %rem_val)
650.   %loop_cond = icmp sge i32 %list_find, 0
651.   br i1 %loop_cond, label %then, label %else
652.
653. merge:                                          ; preds = %else
654.   ret void
655.
656. then:                                           ; preds = %entry
657.   %loop_cnt_ptr = alloca i32
658.   %loop_start_idx = add i32 %list_find, 1
659.   store i32 %loop_start_idx, i32* %loop_cnt_ptr
660.   br label %while
661.
662. while:                                          ; preds = %while_body, %then
663.   %loop_cnt = load i32, i32* %loop_cnt_ptr
664.   %loop_cond1 = icmp slt i32 %loop_cnt, %list_size
665.   br i1 %loop_cond1, label %while_body, label %merge2
666.
667. while_body:                                     ; preds = %while
668.   %cur_idx = load i32, i32* %loop_cnt_ptr
669.   %shift_to_idx = sub i32 %cur_idx, 1
670.   %list_get = call i32 @list_getint({ i32*, i32* }* %list_load, i32 %cur_idx)
671.   call void @list_setint({ i32*, i32* }* %list_load, i32 %shift_to_idx, i32 %list_get)
672.   %loop_itr = add i32 %cur_idx, 1
673.   store i32 %loop_itr, i32* %loop_cnt_ptr
674.   br label %while
675.
676. merge2:                                         ; preds = %while
677.   %size_dec = sub i32 %list_size, 1
678.   store i32 %size_dec, i32* %list_size_ptr
679.   ret void
680.
681. else:                                           ; preds = %entry
682.   br label %merge
683. }
684.
685. define void @list_removefloat({ i32*, double* }*, double) {
686. entry:
687.   %list_ptr_alloc = alloca { i32*, double* }*
688.   store { i32*, double* }* %0, { i32*, double* }** %list_ptr_alloc
689.   %rem_val_ptr = alloca double
690.   store double %1, double* %rem_val_ptr
691.   %rem_val = load double, double* %rem_val_ptr
692.   %list_load = load { i32*, double* }*, { i32*, double* }** %list_ptr_alloc
```

```
693.   %list_size_ptr_ptr = getelementptr inbounds { i32*, double* }, { i32*,
    double* }* %list_load, i32 0, i32 0
694.   %list_size_ptr = load i32*, i32** %list_size_ptr_ptr
695.   %list_size = load i32, i32* %list_size_ptr
696.   %list_find = call i32 @list_findfloat({ i32*, double* }* %list_load, double %rem_val)
697.   %loop_cond = icmp sge i32 %list_find, 0
698.   br i1 %loop_cond, label %then, label %else
699.
700. merge:                                          ; preds = %else
701.   ret void
702.
703. then:                                           ; preds = %entry
704.   %loop_cnt_ptr = alloca i32
705.   %loop_start_idx = add i32 %list_find, 1
706.   store i32 %loop_start_idx, i32* %loop_cnt_ptr
707.   br label %while
708.
709. while:                                          ; preds = %while_body, %then
710.   %loop_cnt = load i32, i32* %loop_cnt_ptr
711.   %loop_cond1 = icmp slt i32 %loop_cnt, %list_size
712.   br i1 %loop_cond1, label %while_body, label %merge2
713.
714. while_body:                                     ; preds = %while
715.   %cur_idx = load i32, i32* %loop_cnt_ptr
716.   %shift_to_idx = sub i32 %cur_idx, 1
717.   %list_get = call double @list_getfloat({ i32*, double* }* %list_load, i32 %cur_idx)
718.   call void @list_setfloat({ i32*, double* }* %list_load, i32 %shift_to_idx,
    double %list_get)
719.   %loop_itr = add i32 %cur_idx, 1
720.   store i32 %loop_itr, i32* %loop_cnt_ptr
721.   br label %while
722.
723. merge2:                                         ; preds = %while
724.   %size_dec = sub i32 %list_size, 1
725.   store i32 %size_dec, i32* %list_size_ptr
726.   ret void
727.
728. else:                                           ; preds = %entry
729.   br label %merge
730. }
731.
732. define void @list_insertbool({ i32*, i1* }*, i32, i1) {
733. entry:
734.   %list_ptr_alloc = alloca { i32*, i1* }*
735.   store { i32*, i1* }* %0, { i32*, i1* }** %list_ptr_alloc
736.   %list_load = load { i32*, i1* }*, { i32*, i1* }** %list_ptr_alloc
737.   %insert_idx_ptr = alloca i32
738.   store i32 %1, i32* %insert_idx_ptr
739.   %insert_idx = load i32, i32* %insert_idx_ptr
740.   %insert_val_ptr = alloca i1
741.   store i1 %2, i1* %insert_val_ptr
742.   %insert_val = load i1, i1* %insert_val_ptr
```

```
  %list_size_ptr_ptr = getelementptr inbounds { i32*, i1* }, { i32*, i1* }* %list_load, i32 0, i32 0
  %list_size_ptr = load i32*, i32** %list_size_ptr_ptr
  %list_size = load i32, i32* %list_size_ptr
  %loop_cnt_ptr = alloca i32
  %last_index = sub i32 %list_size, 1
  store i32 %last_index, i32* %loop_cnt_ptr
  br label %while

while:                                            ; preds = %while_body, %entry
  %loop_cnt = load i32, i32* %loop_cnt_ptr
  %loop_cond = icmp sge i32 %loop_cnt, %insert_idx
  br i1 %loop_cond, label %while_body, label %merge

while_body:                                       ; preds = %while
  %cur_idx = load i32, i32* %loop_cnt_ptr
  %shift_to_idx = add i32 %cur_idx, 1
  %list_get = call i1 @list_getbool({ i32*, i1* }* %list_load, i32 %cur_idx)
  call void @list_setbool({ i32*, i1* }* %list_load, i32 %shift_to_idx, i1 %list_get)
  %loop_itr = sub i32 %cur_idx, 1
  store i32 %loop_itr, i32* %loop_cnt_ptr
  br label %while

merge:                                            ; preds = %while
  call void @list_setbool({ i32*, i1* }* %list_load, i32 %insert_idx, i1 %insert_val)
  %size_inc = add i32 %list_size, 1
  store i32 %size_inc, i32* %list_size_ptr
  ret void
}

define void @list_insertint({ i32*, i32* }*, i32, i32) {
entry:
  %list_ptr_alloc = alloca { i32*, i32* }*
  store { i32*, i32* }* %0, { i32*, i32* }** %list_ptr_alloc
  %list_load = load { i32*, i32* }*, { i32*, i32* }** %list_ptr_alloc
  %insert_idx_ptr = alloca i32
  store i32 %1, i32* %insert_idx_ptr
  %insert_idx = load i32, i32* %insert_idx_ptr
  %insert_val_ptr = alloca i32
  store i32 %2, i32* %insert_val_ptr
  %insert_val = load i32, i32* %insert_val_ptr
  %list_size_ptr_ptr = getelementptr inbounds { i32*, i32* }, { i32*, i32* }* %list_load, i32 0, i32 0
  %list_size_ptr = load i32*, i32** %list_size_ptr_ptr
  %list_size = load i32, i32* %list_size_ptr
  %loop_cnt_ptr = alloca i32
  %last_index = sub i32 %list_size, 1
  store i32 %last_index, i32* %loop_cnt_ptr
  br label %while

while:                                            ; preds = %while_body, %entry
  %loop_cnt = load i32, i32* %loop_cnt_ptr
```

```
  %loop_cond = icmp sge i32 %loop_cnt, %insert_idx
  br i1 %loop_cond, label %while_body, label %merge

while_body:                                       ; preds = %while
  %cur_idx = load i32, i32* %loop_cnt_ptr
  %shift_to_idx = add i32 %cur_idx, 1
  %list_get = call i32 @list_getint({ i32*, i32* }* %list_load, i32 %cur_idx)
  call void @list_setint({ i32*, i32* }* %list_load, i32 %shift_to_idx, i32 %list_get)
  %loop_itr = sub i32 %cur_idx, 1
  store i32 %loop_itr, i32* %loop_cnt_ptr
  br label %while

merge:                                            ; preds = %while
  call void @list_setint({ i32*, i32* }* %list_load, i32 %insert_idx, i32 %insert_val)
  %size_inc = add i32 %list_size, 1
  store i32 %size_inc, i32* %list_size_ptr
  ret void
}

define void @list_insertfloat({ i32*, double* }*, i32, double) {
entry:
  %list_ptr_alloc = alloca { i32*, double* }*
  store { i32*, double* }* %0, { i32*, double* }** %list_ptr_alloc
  %list_load = load { i32*, double* }*, { i32*, double* }** %list_ptr_alloc
  %insert_idx_ptr = alloca i32
  store i32 %1, i32* %insert_idx_ptr
  %insert_idx = load i32, i32* %insert_idx_ptr
  %insert_val_ptr = alloca double
  store double %2, double* %insert_val_ptr
  %insert_val = load double, double* %insert_val_ptr
  %list_size_ptr_ptr = getelementptr inbounds { i32*, double* }, { i32*,
double* }* %list_load, i32 0, i32 0
  %list_size_ptr = load i32*, i32** %list_size_ptr_ptr
  %list_size = load i32, i32* %list_size_ptr
  %loop_cnt_ptr = alloca i32
  %last_index = sub i32 %list_size, 1
  store i32 %last_index, i32* %loop_cnt_ptr
  br label %while

while:                                            ; preds = %while_body, %entry
  %loop_cnt = load i32, i32* %loop_cnt_ptr
  %loop_cond = icmp sge i32 %loop_cnt, %insert_idx
  br i1 %loop_cond, label %while_body, label %merge

while_body:                                       ; preds = %while
  %cur_idx = load i32, i32* %loop_cnt_ptr
  %shift_to_idx = add i32 %cur_idx, 1
  %list_get = call double @list_getfloat({ i32*, double* }* %list_load, i32 %cur_idx)
  call void @list_setfloat({ i32*, double* }* %list_load, i32 %shift_to_idx,
double %list_get)
  %loop_itr = sub i32 %cur_idx, 1
  store i32 %loop_itr, i32* %loop_cnt_ptr
```

```
  br label %while

merge:                                            ; preds = %while
  call void @list_setfloat({ i32*, double* }* %list_load, i32 %insert_idx, double %insert_val)
  %size_inc = add i32 %list_size, 1
  store i32 %size_inc, i32* %list_size_ptr
  ret void
}

define void @list_reversebool({ i32*, i1* }*) {
entry:
  %list_ptr_alloc = alloca { i32*, i1* }*
  store { i32*, i1* }* %0, { i32*, i1* }** %list_ptr_alloc
  %list_load = load { i32*, i1* }*, { i32*, i1* }** %list_ptr_alloc
  %list_size_ptr_ptr = getelementptr inbounds { i32*, i1* }, { i32*, i1* }* %list_load, i32 0, i32 0
  %list_size_ptr = load i32*, i32** %list_size_ptr_ptr
  %list_size = load i32, i32* %list_size_ptr
  %left_idx = alloca i32
  store i32 0, i32* %left_idx
  %right_idx = alloca i32
  %tmp = sub i32 %list_size, 1
  store i32 %tmp, i32* %right_idx
  br label %while

while:                                            ; preds = %while_body, %entry
  %right_idx6 = load i32, i32* %right_idx
  %left_idx7 = load i32, i32* %left_idx
  %while_cond = icmp slt i32 %left_idx7, %right_idx6
  br i1 %while_cond, label %while_body, label %merge

while_body:                                       ; preds = %while
  %left_idx1 = load i32, i32* %left_idx
  %right_idx2 = load i32, i32* %right_idx
  %list_get = call i1 @list_getbool({ i32*, i1* }* %list_load, i32 %left_idx1)
  %list_get3 = call i1 @list_getbool({ i32*, i1* }* %list_load, i32 %right_idx2)
  call void @list_setbool({ i32*, i1* }* %list_load, i32 %left_idx1, i1 %list_get3)
  call void @list_setbool({ i32*, i1* }* %list_load, i32 %right_idx2, i1 %list_get)
  %tmp4 = add i32 %left_idx1, 1
  store i32 %tmp4, i32* %left_idx
  %tmp5 = sub i32 %right_idx2, 1
  store i32 %tmp5, i32* %right_idx
  br label %while

merge:                                            ; preds = %while
  ret void
}

define void @list_reverseint({ i32*, i32* }*) {
entry:
  %list_ptr_alloc = alloca { i32*, i32* }*
```

```
  store { i32*, i32* }* %0, { i32*, i32* }** %list_ptr_alloc
  %list_load = load { i32*, i32* }*, { i32*, i32* }** %list_ptr_alloc
  %list_size_ptr_ptr = getelementptr inbounds { i32*, i32* }, { i32*, i32* }* %list_load, i32 0, i32 0
  %list_size_ptr = load i32*, i32** %list_size_ptr_ptr
  %list_size = load i32, i32* %list_size_ptr
  %left_idx = alloca i32
  store i32 0, i32* %left_idx
  %right_idx = alloca i32
  %tmp = sub i32 %list_size, 1
  store i32 %tmp, i32* %right_idx
  br label %while

while:                                            ; preds = %while_body, %entry
  %right_idx6 = load i32, i32* %right_idx
  %left_idx7 = load i32, i32* %left_idx
  %while_cond = icmp slt i32 %left_idx7, %right_idx6
  br i1 %while_cond, label %while_body, label %merge

while_body:                                       ; preds = %while
  %left_idx1 = load i32, i32* %left_idx
  %right_idx2 = load i32, i32* %right_idx
  %list_get = call i32 @list_getint({ i32*, i32* }* %list_load, i32 %left_idx1)
  %list_get3 = call i32 @list_getint({ i32*, i32* }* %list_load, i32 %right_idx2)
  call void @list_setint({ i32*, i32* }* %list_load, i32 %left_idx1, i32 %list_get3)
  call void @list_setint({ i32*, i32* }* %list_load, i32 %right_idx2, i32 %list_get)
  %tmp4 = add i32 %left_idx1, 1
  store i32 %tmp4, i32* %left_idx
  %tmp5 = sub i32 %right_idx2, 1
  store i32 %tmp5, i32* %right_idx
  br label %while

merge:                                            ; preds = %while
  ret void
}

define void @list_reversefloat({ i32*, double* }*) {
entry:
  %list_ptr_alloc = alloca { i32*, double* }*
  store { i32*, double* }* %0, { i32*, double* }** %list_ptr_alloc
  %list_load = load { i32*, double* }*, { i32*, double* }** %list_ptr_alloc
  %list_size_ptr_ptr = getelementptr inbounds { i32*, double* }, { i32*, double* }* %list_load, i32 0, i32 0
  %list_size_ptr = load i32*, i32** %list_size_ptr_ptr
  %list_size = load i32, i32* %list_size_ptr
  %left_idx = alloca i32
  store i32 0, i32* %left_idx
  %right_idx = alloca i32
  %tmp = sub i32 %list_size, 1
  store i32 %tmp, i32* %right_idx
  br label %while

```

```
while:                                            ; preds = %while_body, %entry
  %right_idx6 = load i32, i32* %right_idx
  %left_idx7 = load i32, i32* %left_idx
  %while_cond = icmp slt i32 %left_idx7, %right_idx6
  br i1 %while_cond, label %while_body, label %merge

while_body:                                       ; preds = %while
  %left_idx1 = load i32, i32* %left_idx
  %right_idx2 = load i32, i32* %right_idx
  %list_get = call double @list_getfloat({ i32*, double* }* %list_load, i32 %left_idx1)
  %list_get3 = call double @list_getfloat({ i32*, double* }* %list_load,
i32 %right_idx2)
  call void @list_setfloat({ i32*, double* }* %list_load, i32 %left_idx1,
double %list_get3)
  call void @list_setfloat({ i32*, double* }* %list_load, i32 %right_idx2,
double %list_get)
  %tmp4 = add i32 %left_idx1, 1
  store i32 %tmp4, i32* %left_idx
  %tmp5 = sub i32 %right_idx2, 1
  store i32 %tmp5, i32* %right_idx
  br label %while

merge:                                            ; preds = %while
  ret void
}

define i32 @main() {
entry:
  %a = alloca { i32*, i32* }
  %list_size_ptr = getelementptr inbounds { i32*, i32* }, { i32*, i32* }* %a, i32 0,
i32 0
  %list_size = alloca i32
  store i32 0, i32* %list_size
  store i32* %list_size, i32** %list_size_ptr
  %list.arry = getelementptr inbounds { i32*, i32* }, { i32*, i32* }* %a, i32 0, i32 1
  %p = alloca i32, i32 1028
  store i32* %p, i32** %list.arry
  %i = alloca i32
  %new_list_ptr = alloca { i32*, i32* }
  %list_size_ptr1 = getelementptr inbounds { i32*, i32* }, { i32*,
i32* }* %new_list_ptr, i32 0, i32 0
  %list_size2 = alloca i32
  store i32 0, i32* %list_size2
  store i32* %list_size2, i32** %list_size_ptr1
  %list.arry3 = getelementptr inbounds { i32*, i32* }, { i32*, i32* }* %new_list_ptr,
i32 0, i32 1
  %p4 = alloca i32, i32 1028
  store i32* %p4, i32** %list.arry3
  call void @list_pushint({ i32*, i32* }* %new_list_ptr, i32 10)
  call void @list_pushint({ i32*, i32* }* %new_list_ptr, i32 2)
  call void @list_pushint({ i32*, i32* }* %new_list_ptr, i32 15)
  call void @list_pushint({ i32*, i32* }* %new_list_ptr, i32 8)
```

```
  call void @list_pushint({ i32*, i32* }* %new_list_ptr, i32 20)
  call void @list_pushint({ i32*, i32* }* %new_list_ptr, i32 13)
  call void @list_pushint({ i32*, i32* }* %new_list_ptr, i32 9)
  %new_list = load { i32*, i32* }, { i32*, i32* }* %new_list_ptr
  store { i32*, i32* } %new_list, { i32*, i32* }* %a
  %a5 = load { i32*, i32* }, { i32*, i32* }* %a
  call void @bubbleSort({ i32*, i32* } %a5)
  store i32 0, i32* %i
  br label %while

while:                                            ; preds = %while_body, %entry
    %i8 = load i32, i32* %i
    %list_size9 = call i32 @list_sizeint({ i32*, i32* }* %a)
    %tmp10 = icmp slt i32 %i8, %list_size9
    br i1 %tmp10, label %while_body, label %merge

  while_body:                                       ; preds = %while
    %i6 = load i32, i32* %i
    %list_get = call i32 @list_getint({ i32*, i32* }* %a, i32 %i6)
    %printf = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([4 x i8], [4 x i8]* @fmt, i32 0, i32 0), i32 %list_get)
    %i7 = load i32, i32* %i
    %tmp = add i32 %i7, 1
    store i32 %tmp, i32* %i
    br label %while

  merge:                                            ; preds = %while
    ret i32 0
  }

  define void @bubbleSort({ i32*, i32* } %arr) {
  entry:
    %arr1 = alloca { i32*, i32* }
    %list_size_ptr = getelementptr inbounds { i32*, i32* }, { i32*, i32* }* %arr1, i32 0, i32 0
    %list_size = alloca i32
    store i32 0, i32* %list_size
    store i32* %list_size, i32** %list_size_ptr
    %list.arry = getelementptr inbounds { i32*, i32* }, { i32*, i32* }* %arr1, i32 0, i32 1
    %p = alloca i32, i32 1028
    store i32* %p, i32** %list.arry
    store { i32*, i32* } %arr, { i32*, i32* }* %arr1
    %i = alloca i32
    %j = alloca i32
    %tmp = alloca i32
    store i32 0, i32* %i
    br label %while

  while:                                            ; preds = %merge26, %entry
    %i29 = load i32, i32* %i
    %list_size30 = call i32 @list_sizeint({ i32*, i32* }* %arr1)
```

```
  %tmp31 = sub i32 %list_size30, 1
  %tmp32 = icmp slt i32 %i29, %tmp31
  br i1 %tmp32, label %while_body, label %merge33

while_body:                                       ; preds = %while
  store i32 0, i32* %j
  br label %while2

while2:                                           ; preds = %merge, %while_body
  %j20 = load i32, i32* %j
  %list_size21 = call i32 @list_sizeint({ i32*, i32* }* %arr1)
  %i22 = load i32, i32* %i
  %tmp23 = sub i32 %list_size21, %i22
  %tmp24 = sub i32 %tmp23, 1
  %tmp25 = icmp slt i32 %j20, %tmp24
  br i1 %tmp25, label %while_body3, label %merge26

while_body3:                                      ; preds = %while2
  %j4 = load i32, i32* %j
  %list_get = call i32 @list_getint({ i32*, i32* }* %arr1, i32 %j4)
  %j5 = load i32, i32* %j
  %tmp6 = add i32 %j5, 1
  %list_get7 = call i32 @list_getint({ i32*, i32* }* %arr1, i32 %tmp6)
  %tmp8 = icmp sgt i32 %list_get, %list_get7
  br i1 %tmp8, label %then, label %else

merge:                                            ; preds = %else, %then
  %j18 = load i32, i32* %j
  %tmp19 = add i32 %j18, 1
  store i32 %tmp19, i32* %j
  br label %while2

then:                                             ; preds = %while_body3
  %j9 = load i32, i32* %j
  %list_get10 = call i32 @list_getint({ i32*, i32* }* %arr1, i32 %j9)
  store i32 %list_get10, i32* %tmp
  %j11 = load i32, i32* %j
  %tmp12 = add i32 %j11, 1
  %list_get13 = call i32 @list_getint({ i32*, i32* }* %arr1, i32 %tmp12)
  %j14 = load i32, i32* %j
  call void @list_setint({ i32*, i32* }* %arr1, i32 %j14, i32 %list_get13)
  %tmp15 = load i32, i32* %tmp
  %j16 = load i32, i32* %j
  %tmp17 = add i32 %j16, 1
  call void @list_setint({ i32*, i32* }* %arr1, i32 %tmp17, i32 %tmp15)
  br label %merge

else:                                             ; preds = %while_body3
  br label %merge

merge26:                                          ; preds = %while2
  %i27 = load i32, i32* %i
```

```
1090.         %tmp28 = add i32 %i27, 1
1091.         store i32 %tmp28, i32* %i
1092.         br label %while
1093.
1094.       merge33:                                          ; preds = %while
1095.         ret void
1096.       }
```

Sample 2: QuickSort

```
/* sorts a list in-place with QuickSort algorithm, */

int partition(list<int> a, int low, int high)
{
    int i;
    int j;
    int pivot;
    int temp;

    pivot = a[high];
    i = (low - 1);
    for (j = low; j <= high - 1; j++)
    {
        if (a[j] <= pivot)
        {
            i++;
            temp = a[i];
            a[i] = a[j];
            a[j] = temp;
        }
    }
    temp = a[i+1];
    a[i+1] = a[high];
    a[high] = temp;
    return (i + 1);
}

void quickSortImpl(list<int> a, int low, int high)
{
    int pi;
    if (low < high)
    {
        pi = partition(a, low, high);
        quickSortImpl(a, low, pi - 1);
        quickSortImpl(a, pi + 1, high);
    }
    return;
}

void quickSort(list<int> a)
{
    quickSortImpl(a, 0, #a - 1);
    return;
}

int main()
{
    list<int> a;
    int i;
```

```
50.
51.     a = [10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0];
52.
53.     quickSort(a);
54.
55.     for (i=0; i<#a; ++i) {
56.       printi(a[i]);
57.     }
58.
59.     return 0;
60. }
61. output:
62. 0
63. 1
64. 2
65. 3
66. 4
67. 5
68. 6
69. 7
70. 8
71. 9
72. 10
```

Target Language Output (LLVM IR):

```
1.  ; ModuleID = 'AP_PlusPlus'
2.  source_filename = "AP_PlusPlus"
3.
4.  @fmt = private unnamed_addr constant [4 x i8] c"%d\0A\00"
5.  @fmt.1 = private unnamed_addr constant [4 x i8] c"%g\0A\00"
6.  @fmt.2 = private unnamed_addr constant [4 x i8] c"%s\0A\00"
7.  @fmt.3 = private unnamed_addr constant [4 x i8] c"%d\0A\00"
8.  @fmt.4 = private unnamed_addr constant [4 x i8] c"%g\0A\00"
9.  @fmt.5 = private unnamed_addr constant [4 x i8] c"%s\0A\00"
10. @fmt.6 = private unnamed_addr constant [4 x i8] c"%d\0A\00"
11. @fmt.7 = private unnamed_addr constant [4 x i8] c"%g\0A\00"
12. @fmt.8 = private unnamed_addr constant [4 x i8] c"%s\0A\00"
13. @fmt.9 = private unnamed_addr constant [4 x i8] c"%d\0A\00"
14. @fmt.10 = private unnamed_addr constant [4 x i8] c"%g\0A\00"
15. @fmt.11 = private unnamed_addr constant [4 x i8] c"%s\0A\00"
16.
17. declare i32 @printf(i8*, ...)
18.
19. define i1 @list_getbool({ i32*, i1* }*, i32) {
20. entry:
21.   %list_ptr_alloc = alloca { i32*, i1* }*
22.   store { i32*, i1* }* %0, { i32*, i1* }** %list_ptr_alloc
23.   %idx_alloc = alloca i32
24.   store i32 %1, i32* %idx_alloc
```

```
  %list_load = load { i32*, i1* }*, { i32*, i1* }** %list_ptr_alloc
  %list_array_ptr = getelementptr inbounds { i32*, i1* }, { i32*, i1* }* %list_load, i32 0, i32 1
  %array_load = load i1*, i1** %list_array_ptr
  %idx_load = load i32, i32* %idx_alloc
  %list_arry_element_ptr = getelementptr i1, i1* %array_load, i32 %idx_load
  %list_array_element_ptr = load i1, i1* %list_arry_element_ptr
  ret i1 %list_array_element_ptr
}

define i32 @list_getint({ i32*, i32* }*, i32) {
entry:
  %list_ptr_alloc = alloca { i32*, i32* }*
  store { i32*, i32* }* %0, { i32*, i32* }** %list_ptr_alloc
  %idx_alloc = alloca i32
  store i32 %1, i32* %idx_alloc
  %list_load = load { i32*, i32* }*, { i32*, i32* }** %list_ptr_alloc
  %list_array_ptr = getelementptr inbounds { i32*, i32* }, { i32*, i32* }* %list_load, i32 0, i32 1
  %array_load = load i32*, i32** %list_array_ptr
  %idx_load = load i32, i32* %idx_alloc
  %list_arry_element_ptr = getelementptr i32, i32* %array_load, i32 %idx_load
  %list_array_element_ptr = load i32, i32* %list_arry_element_ptr
  ret i32 %list_array_element_ptr
}

define double @list_getfloat({ i32*, double* }*, i32) {
entry:
  %list_ptr_alloc = alloca { i32*, double* }*
  store { i32*, double* }* %0, { i32*, double* }** %list_ptr_alloc
  %idx_alloc = alloca i32
  store i32 %1, i32* %idx_alloc
  %list_load = load { i32*, double* }*, { i32*, double* }** %list_ptr_alloc
  %list_array_ptr = getelementptr inbounds { i32*, double* }, { i32*, double* }* %list_load, i32 0, i32 1
  %array_load = load double*, double** %list_array_ptr
  %idx_load = load i32, i32* %idx_alloc
  %list_arry_element_ptr = getelementptr double, double* %array_load, i32 %idx_load
  %list_array_element_ptr = load double, double* %list_arry_element_ptr
  ret double %list_array_element_ptr
}

define i8* @list_getstr({ i32*, i8** }*, i32) {
entry:
  %list_ptr_alloc = alloca { i32*, i8** }*
  store { i32*, i8** }* %0, { i32*, i8** }** %list_ptr_alloc
  %idx_alloc = alloca i32
  store i32 %1, i32* %idx_alloc
  %list_load = load { i32*, i8** }*, { i32*, i8** }** %list_ptr_alloc
  %list_array_ptr = getelementptr inbounds { i32*, i8** }, { i32*, i8** }* %list_load, i32 0, i32 1
  %array_load = load i8**, i8*** %list_array_ptr
```

```
  %idx_load = load i32, i32* %idx_alloc
  %list_arry_element_ptr = getelementptr i8*, i8** %array_load, i32 %idx_load
  %list_array_element_ptr = load i8*, i8** %list_arry_element_ptr
  ret i8* %list_array_element_ptr
}

define void @list_setbool({ i32*, i1* }*, i32, i1) {
entry:
  %list_ptr_alloc = alloca { i32*, i1* }*
  store { i32*, i1* }* %0, { i32*, i1* }** %list_ptr_alloc
  %list_load = load { i32*, i1* }*, { i32*, i1* }** %list_ptr_alloc
  %list_array_ptr = getelementptr inbounds { i32*, i1* }, { i32*, i1* }* %list_load, i32 0, i32 1
  %list_array_load = load i1*, i1** %list_array_ptr
  %list_arry_next_element_ptr = getelementptr i1, i1* %list_array_load, i32 %1
  store i1 %2, i1* %list_arry_next_element_ptr
  ret void
}

define void @list_setint({ i32*, i32* }*, i32, i32) {
entry:
  %list_ptr_alloc = alloca { i32*, i32* }*
  store { i32*, i32* }* %0, { i32*, i32* }** %list_ptr_alloc
  %list_load = load { i32*, i32* }*, { i32*, i32* }** %list_ptr_alloc
  %list_array_ptr = getelementptr inbounds { i32*, i32* }, { i32*, i32* }* %list_load, i32 0, i32 1
  %list_array_load = load i32*, i32** %list_array_ptr
  %list_arry_next_element_ptr = getelementptr i32, i32* %list_array_load, i32 %1
  store i32 %2, i32* %list_arry_next_element_ptr
   ret void
}

define void @list_setfloat({ i32*, double* }*, i32, double) {
entry:
  %list_ptr_alloc = alloca { i32*, double* }*
  store { i32*, double* }* %0, { i32*, double* }** %list_ptr_alloc
  %list_load = load { i32*, double* }*, { i32*, double* }** %list_ptr_alloc
  %list_array_ptr = getelementptr inbounds { i32*, double* }, { i32*, double* }* %list_load, i32 0, i32 1
  %list_array_load = load double*, double** %list_array_ptr
  %list_arry_next_element_ptr = getelementptr double, double* %list_array_load, i32 %1
  store double %2, double* %list_arry_next_element_ptr
  ret void
}

define void @list_setstr({ i32*, i8** }*, i32, i8*) {
entry:
  %list_ptr_alloc = alloca { i32*, i8** }*
  store { i32*, i8** }* %0, { i32*, i8** }** %list_ptr_alloc
  %list_load = load { i32*, i8** }*, { i32*, i8** }** %list_ptr_alloc
  %list_array_ptr = getelementptr inbounds { i32*, i8** }, { i32*, i8** }* %list_load, i32 0, i32 1
```

```
  %list_array_load = load i8**, i8*** %list_array_ptr
  %list_arry_next_element_ptr = getelementptr i8*, i8** %list_array_load, i32 %1
  store i8* %2, i8** %list_arry_next_element_ptr
  ret void
}

define void @list_pushbool({ i32*, i1* }*, i1) {
entry:
  %list_ptr_alloc = alloca { i32*, i1* }*
  store { i32*, i1* }* %0, { i32*, i1* }** %list_ptr_alloc
  %val_alloc = alloca i1
  store i1 %1, i1* %val_alloc
  %list_load = load { i32*, i1* }*, { i32*, i1* }** %list_ptr_alloc
  %list_array_ptr = getelementptr inbounds { i32*, i1* }, { i32*, i1* }* %list_load, i32 0, i32 1
  %list_array_load = load i1*, i1** %list_array_ptr
  %list_size_ptr_ptr = getelementptr inbounds { i32*, i1* }, { i32*, i1* }* %list_load, i32 0, i32 0
  %list_size_ptr = load i32*, i32** %list_size_ptr_ptr
  %list_size = load i32, i32* %list_size_ptr
  %list_arry_next_element_ptr = getelementptr i1, i1* %list_array_load, i32 %list_size
  %inc_size = add i32 %list_size, 1
  store i32 %inc_size, i32* %list_size_ptr
  %val = load i1, i1* %val_alloc
  store i1 %val, i1* %list_arry_next_element_ptr
  ret void
}

define void @list_pushint({ i32*, i32* }*, i32) {
entry:
  %list_ptr_alloc = alloca { i32*, i32* }*
  store { i32*, i32* }* %0, { i32*, i32* }** %list_ptr_alloc
  %val_alloc = alloca i32
  store i32 %1, i32* %val_alloc
  %list_load = load { i32*, i32* }*, { i32*, i32* }** %list_ptr_alloc
  %list_array_ptr = getelementptr inbounds { i32*, i32* }, { i32*, i32* }* %list_load, i32 0, i32 1
  %list_array_load = load i32*, i32** %list_array_ptr
  %list_size_ptr_ptr = getelementptr inbounds { i32*, i32* }, { i32*, i32* }* %list_load, i32 0, i32 0
  %list_size_ptr = load i32*, i32** %list_size_ptr_ptr
  %list_size = load i32, i32* %list_size_ptr
  %list_arry_next_element_ptr = getelementptr i32, i32* %list_array_load, i32 %list_size
  %inc_size = add i32 %list_size, 1
  store i32 %inc_size, i32* %list_size_ptr
  %val = load i32, i32* %val_alloc
  store i32 %val, i32* %list_arry_next_element_ptr
  ret void
}

define void @list_pushfloat({ i32*, double* }*, double) {
```

```
entry:
  %list_ptr_alloc = alloca { i32*, double* }*
  store { i32*, double* }* %0, { i32*, double* }** %list_ptr_alloc
  %val_alloc = alloca double
  store double %1, double* %val_alloc
  %list_load = load { i32*, double* }*, { i32*, double* }** %list_ptr_alloc
  %list_array_ptr = getelementptr inbounds { i32*, double* }, { i32*, double* }* %list_load, i32 0, i32 1
  %list_array_load = load double*, double** %list_array_ptr
  %list_size_ptr_ptr = getelementptr inbounds { i32*, double* }, { i32*, double* }* %list_load, i32 0, i32 0
  %list_size_ptr = load i32*, i32** %list_size_ptr_ptr
  %list_size = load i32, i32* %list_size_ptr
  %list_arry_next_element_ptr = getelementptr double, double* %list_array_load, i32 %list_size
  %inc_size = add i32 %list_size, 1
  store i32 %inc_size, i32* %list_size_ptr
  %val = load double, double* %val_alloc
  store double %val, double* %list_arry_next_element_ptr
  ret void
}

define void @list_pushstr({ i32*, i8** }*, i8*) {
entry:
  %list_ptr_alloc = alloca { i32*, i8** }*
  store { i32*, i8** }* %0, { i32*, i8** }** %list_ptr_alloc
  %val_alloc = alloca i8*
  store i8* %1, i8** %val_alloc
  %list_load = load { i32*, i8** }*, { i32*, i8** }** %list_ptr_alloc
  %list_array_ptr = getelementptr inbounds { i32*, i8** }, { i32*, i8** }* %list_load, i32 0, i32 1
  %list_array_load = load i8**, i8*** %list_array_ptr
  %list_size_ptr_ptr = getelementptr inbounds { i32*, i8** }, { i32*, i8** }* %list_load, i32 0, i32 0
  %list_size_ptr = load i32*, i32** %list_size_ptr_ptr
  %list_size = load i32, i32* %list_size_ptr
  %list_arry_next_element_ptr = getelementptr i8*, i8** %list_array_load, i32 %list_size
  %inc_size = add i32 %list_size, 1
  store i32 %inc_size, i32* %list_size_ptr
  %val = load i8*, i8** %val_alloc
  store i8* %val, i8** %list_arry_next_element_ptr
  ret void
}

define i1 @list_popbool({ i32*, i1* }*) {
entry:
  %list_ptr_alloc = alloca { i32*, i1* }*
  store { i32*, i1* }* %0, { i32*, i1* }** %list_ptr_alloc
  %list_load = load { i32*, i1* }*, { i32*, i1* }** %list_ptr_alloc
  %list_array_ptr = getelementptr inbounds { i32*, i1* }, { i32*, i1* }* %list_load, i32 0, i32 1
  %list_array_load = load i1*, i1** %list_array_ptr
```

```
  %list_size_ptr_ptr = getelementptr inbounds { i32*, i1* }, { i32*, i1* }* %list_load, i32 0, i32 0
  %list_size_ptr = load i32*, i32** %list_size_ptr_ptr
  %list_size = load i32, i32* %list_size_ptr
  %dec_size = sub i32 %list_size, 1
  %list_arry_next_element_ptr = getelementptr i1, i1* %list_array_load, i32 %dec_size
  %list_arry_next_element = load i1, i1* %list_arry_next_element_ptr
  store i32 %dec_size, i32* %list_size_ptr
  ret i1 %list_arry_next_element
}

define i32 @list_popint({ i32*, i32* }*) {
entry:
  %list_ptr_alloc = alloca { i32*, i32* }*
  store { i32*, i32* }* %0, { i32*, i32* }** %list_ptr_alloc
  %list_load = load { i32*, i32* }*, { i32*, i32* }** %list_ptr_alloc
  %list_array_ptr = getelementptr inbounds { i32*, i32* }, { i32*, i32* }* %list_load, i32 0, i32 1
  %list_array_load = load i32*, i32** %list_array_ptr
  %list_size_ptr_ptr = getelementptr inbounds { i32*, i32* }, { i32*, i32* }* %list_load, i32 0, i32 0
  %list_size_ptr = load i32*, i32** %list_size_ptr_ptr
  %list_size = load i32, i32* %list_size_ptr
  %dec_size = sub i32 %list_size, 1
  %list_arry_next_element_ptr = getelementptr i32, i32* %list_array_load, i32 %dec_size
  %list_arry_next_element = load i32, i32* %list_arry_next_element_ptr
  store i32 %dec_size, i32* %list_size_ptr
  ret i32 %list_arry_next_element
}

define double @list_popfloat({ i32*, double* }*) {
entry:
  %list_ptr_alloc = alloca { i32*, double* }*
  store { i32*, double* }* %0, { i32*, double* }** %list_ptr_alloc
  %list_load = load { i32*, double* }*, { i32*, double* }** %list_ptr_alloc
  %list_array_ptr = getelementptr inbounds { i32*, double* }, { i32*, double* }* %list_load, i32 0, i32 1
  %list_array_load = load double*, double** %list_array_ptr
  %list_size_ptr_ptr = getelementptr inbounds { i32*, double* }, { i32*, double* }* %list_load, i32 0, i32 0
  %list_size_ptr = load i32*, i32** %list_size_ptr_ptr
  %list_size = load i32, i32* %list_size_ptr
  %dec_size = sub i32 %list_size, 1
  %list_arry_next_element_ptr = getelementptr double, double* %list_array_load, i32 %dec_size
  %list_arry_next_element = load double, double* %list_arry_next_element_ptr
  store i32 %dec_size, i32* %list_size_ptr
  ret double %list_arry_next_element
}

define i8* @list_popstr({ i32*, i8** }*) {
entry:
```

```
  %list_ptr_alloc = alloca { i32*, i8** }*
  store { i32*, i8** }* %0, { i32*, i8** }** %list_ptr_alloc
  %list_load = load { i32*, i8** }*, { i32*, i8** }** %list_ptr_alloc
  %list_array_ptr = getelementptr inbounds { i32*, i8** }, { i32*, i8** }* %list_load, i32 0, i32 1
  %list_array_load = load i8**, i8*** %list_array_ptr
  %list_size_ptr_ptr = getelementptr inbounds { i32*, i8** }, { i32*, i8** }* %list_load, i32 0, i32 0
  %list_size_ptr = load i32*, i32** %list_size_ptr_ptr
  %list_size = load i32, i32* %list_size_ptr
  %dec_size = sub i32 %list_size, 1
  %list_arry_next_element_ptr = getelementptr i8*, i8** %list_array_load, i32 %dec_size
  %list_arry_next_element = load i8*, i8** %list_arry_next_element_ptr
  store i32 %dec_size, i32* %list_size_ptr
  ret i8* %list_arry_next_element
}

define i32 @list_sizebool({ i32*, i1* }*) {
entry:
  %list_ptr_alloc = alloca { i32*, i1* }*
  store { i32*, i1* }* %0, { i32*, i1* }** %list_ptr_alloc
  %list_load = load { i32*, i1* }*, { i32*, i1* }** %list_ptr_alloc
  %list_size_ptr_ptr = getelementptr inbounds { i32*, i1* }, { i32*, i1* }* %list_load, i32 0, i32 0
  %list_size_ptr = load i32*, i32** %list_size_ptr_ptr
  %list_size = load i32, i32* %list_size_ptr
  ret i32 %list_size
}

define i32 @list_sizeint({ i32*, i32* }*) {
entry:
  %list_ptr_alloc = alloca { i32*, i32* }*
  store { i32*, i32* }* %0, { i32*, i32* }** %list_ptr_alloc
  %list_load = load { i32*, i32* }*, { i32*, i32* }** %list_ptr_alloc
  %list_size_ptr_ptr = getelementptr inbounds { i32*, i32* }, { i32*, i32* }* %list_load, i32 0, i32 0
  %list_size_ptr = load i32*, i32** %list_size_ptr_ptr
  %list_size = load i32, i32* %list_size_ptr
  ret i32 %list_size
}

define i32 @list_sizefloat({ i32*, double* }*) {
entry:
  %list_ptr_alloc = alloca { i32*, double* }*
  store { i32*, double* }* %0, { i32*, double* }** %list_ptr_alloc
  %list_load = load { i32*, double* }*, { i32*, double* }** %list_ptr_alloc
  %list_size_ptr_ptr = getelementptr inbounds { i32*, double* }, { i32*, double* }* %list_load, i32 0, i32 0
  %list_size_ptr = load i32*, i32** %list_size_ptr_ptr
  %list_size = load i32, i32* %list_size_ptr
  ret i32 %list_size
}
```

```
307.
308. define i32 @list_sizestr({ i32*, i8** }*) {
309. entry:
310.   %list_ptr_alloc = alloca { i32*, i8** }*
311.   store { i32*, i8** }* %0, { i32*, i8** }** %list_ptr_alloc
312.   %list_load = load { i32*, i8** }*, { i32*, i8** }** %list_ptr_alloc
313.   %list_size_ptr_ptr = getelementptr inbounds { i32*, i8** }, { i32*,
    i8** }* %list_load, i32 0, i32 0
314.   %list_size_ptr = load i32*, i32** %list_size_ptr_ptr
315.   %list_size = load i32, i32* %list_size_ptr
316.   ret i32 %list_size
317. }
318.
319. define void @list_slicebool({ i32*, i1* }*, { i32*, i1* }*, i32, i32) {
320. entry:
321.   %list_ptr_alloc = alloca { i32*, i1* }*
322.   store { i32*, i1* }* %0, { i32*, i1* }** %list_ptr_alloc
323.   %list_ptr_ptr = load { i32*, i1* }*, { i32*, i1* }** %list_ptr_alloc
324.   %list_ptr_alloc2 = alloca { i32*, i1* }*
325.   store { i32*, i1* }* %1, { i32*, i1* }** %list_ptr_alloc2
326.   %list_ptr_ptr2 = load { i32*, i1* }*, { i32*, i1* }** %list_ptr_alloc2
327.   %idx_alloc = alloca i32
328.   store i32 %2, i32* %idx_alloc
329.   %idx_load = load i32, i32* %idx_alloc
330.   %idx_alloc1 = alloca i32
331.   store i32 %3, i32* %idx_alloc1
332.   %idx_load2 = load i32, i32* %idx_alloc1
333.   %loop_cnt = alloca i32
334.   store i32 0, i32* %loop_cnt
335.   %loop_upper_bound = sub i32 %idx_load2, %idx_load
336.   br label %while
337.
338. while:                                          ; preds = %while_body, %entry
339.   %loop_cnt4 = load i32, i32* %loop_cnt
340.   %loop_cond = icmp sle i32 %loop_cnt4, %loop_upper_bound
341.   br i1 %loop_cond, label %while_body, label %merge
342.
343. while_body:                                     ; preds = %while
344.   %to_idx = load i32, i32* %loop_cnt
345.   %from_idx = add i32 %to_idx, %idx_load
346.   %list_get = call i1 @list_getbool({ i32*, i1* }* %list_ptr_ptr, i32 %from_idx)
347.   call void @list_pushbool({ i32*, i1* }* %list_ptr_ptr2, i1 %list_get)
348.   %loop_cnt3 = load i32, i32* %loop_cnt
349.   %loop_itr = add i32 %loop_cnt3, 1
350.   store i32 %loop_itr, i32* %loop_cnt
351.   br label %while
352.
353. merge:                                          ; preds = %while
354.   ret void
355. }
356.
357. define void @list_sliceint({ i32*, i32* }*, { i32*, i32* }*, i32, i32) {
```

```
entry:
  %list_ptr_alloc = alloca { i32*, i32* }*
  store { i32*, i32* }* %0, { i32*, i32* }** %list_ptr_alloc
  %list_ptr_ptr = load { i32*, i32* }*, { i32*, i32* }** %list_ptr_alloc
  %list_ptr_alloc2 = alloca { i32*, i32* }*
  store { i32*, i32* }* %1, { i32*, i32* }** %list_ptr_alloc2
  %list_ptr_ptr2 = load { i32*, i32* }*, { i32*, i32* }** %list_ptr_alloc2
  %idx_alloc = alloca i32
  store i32 %2, i32* %idx_alloc
  %idx_load = load i32, i32* %idx_alloc
  %idx_alloc1 = alloca i32
  store i32 %3, i32* %idx_alloc1
  %idx_load2 = load i32, i32* %idx_alloc1
  %loop_cnt = alloca i32
  store i32 0, i32* %loop_cnt
  %loop_upper_bound = sub i32 %idx_load2, %idx_load
  br label %while

while:                                            ; preds = %while_body, %entry
  %loop_cnt4 = load i32, i32* %loop_cnt
  %loop_cond = icmp sle i32 %loop_cnt4, %loop_upper_bound
  br i1 %loop_cond, label %while_body, label %merge

while_body:                                       ; preds = %while
  %to_idx = load i32, i32* %loop_cnt
  %from_idx = add i32 %to_idx, %idx_load
  %list_get = call i32 @list_getint({ i32*, i32* }* %list_ptr_ptr, i32 %from_idx)
  call void @list_pushint({ i32*, i32* }* %list_ptr_ptr2, i32 %list_get)
  %loop_cnt3 = load i32, i32* %loop_cnt
  %loop_itr = add i32 %loop_cnt3, 1
  store i32 %loop_itr, i32* %loop_cnt
  br label %while

merge:                                            ; preds = %while
  ret void
}

define void @list_slicefloat({ i32*, double* }*, { i32*, double* }*, i32, i32) {
entry:
  %list_ptr_alloc = alloca { i32*, double* }*
  store { i32*, double* }* %0, { i32*, double* }** %list_ptr_alloc
  %list_ptr_ptr = load { i32*, double* }*, { i32*, double* }** %list_ptr_alloc
  %list_ptr_alloc2 = alloca { i32*, double* }*
  store { i32*, double* }* %1, { i32*, double* }** %list_ptr_alloc2
  %list_ptr_ptr2 = load { i32*, double* }*, { i32*, double* }** %list_ptr_alloc2
  %idx_alloc = alloca i32
  store i32 %2, i32* %idx_alloc
  %idx_load = load i32, i32* %idx_alloc
  %idx_alloc1 = alloca i32
  store i32 %3, i32* %idx_alloc1
  %idx_load2 = load i32, i32* %idx_alloc1
  %loop_cnt = alloca i32
```

```
  store i32 0, i32* %loop_cnt
  %loop_upper_bound = sub i32 %idx_load2, %idx_load
  br label %while

while:                                            ; preds = %while_body, %entry
  %loop_cnt4 = load i32, i32* %loop_cnt
  %loop_cond = icmp sle i32 %loop_cnt4, %loop_upper_bound
  br i1 %loop_cond, label %while_body, label %merge

while_body:                                       ; preds = %while
  %to_idx = load i32, i32* %loop_cnt
  %from_idx = add i32 %to_idx, %idx_load
  %list_get = call double @list_getfloat({ i32*, double* }* %list_ptr_ptr,
i32 %from_idx)
  call void @list_pushfloat({ i32*, double* }* %list_ptr_ptr2, double %list_get)
  %loop_cnt3 = load i32, i32* %loop_cnt
  %loop_itr = add i32 %loop_cnt3, 1
  store i32 %loop_itr, i32* %loop_cnt
  br label %while

merge:                                            ; preds = %while
  ret void
}

define void @list_slicestr({ i32*, i8** }*, { i32*, i8** }*, i32, i32) {
entry:
  %list_ptr_alloc = alloca { i32*, i8** }*
  store { i32*, i8** }* %0, { i32*, i8** }** %list_ptr_alloc
  %list_ptr_ptr = load { i32*, i8** }*, { i32*, i8** }** %list_ptr_alloc
  %list_ptr_alloc2 = alloca { i32*, i8** }*
  store { i32*, i8** }* %1, { i32*, i8** }** %list_ptr_alloc2
  %list_ptr_ptr2 = load { i32*, i8** }*, { i32*, i8** }** %list_ptr_alloc2
  %idx_alloc = alloca i32
  store i32 %2, i32* %idx_alloc
  %idx_load = load i32, i32* %idx_alloc
  %idx_alloc1 = alloca i32
  store i32 %3, i32* %idx_alloc1
  %idx_load2 = load i32, i32* %idx_alloc1
  %loop_cnt = alloca i32
  store i32 0, i32* %loop_cnt
  %loop_upper_bound = sub i32 %idx_load2, %idx_load
  br label %while

while:                                            ; preds = %while_body, %entry
  %loop_cnt4 = load i32, i32* %loop_cnt
  %loop_cond = icmp sle i32 %loop_cnt4, %loop_upper_bound
  br i1 %loop_cond, label %while_body, label %merge

while_body:                                       ; preds = %while
  %to_idx = load i32, i32* %loop_cnt
  %from_idx = add i32 %to_idx, %idx_load
  %list_get = call i8* @list_getstr({ i32*, i8** }* %list_ptr_ptr, i32 %from_idx)
```

```
  call void @list_pushstr({ i32*, i8** }* %list_ptr_ptr2, i8* %list_get)
  %loop_cnt3 = load i32, i32* %loop_cnt
  %loop_itr = add i32 %loop_cnt3, 1
  store i32 %loop_itr, i32* %loop_cnt
  br label %while

merge:                                            ; preds = %while
  ret void
}

define i32 @list_findbool({ i32*, i1* }*, i1) {
entry:
  %list_ptr_alloc = alloca { i32*, i1* }*
  store { i32*, i1* }* %0, { i32*, i1* }** %list_ptr_alloc
  %find_val_alloc = alloca i1
  store i1 %1, i1* %find_val_alloc
  %find_val = load i1, i1* %find_val_alloc
  %list_load = load { i32*, i1* }*, { i32*, i1* }** %list_ptr_alloc
  %list_size_ptr_ptr = getelementptr inbounds { i32*, i1* }, { i32*, i1* }* %list_load, i32 0, i32 0
  %list_size_ptr = load i32*, i32** %list_size_ptr_ptr
  %list_size = load i32, i32* %list_size_ptr
  %loop_cnt = alloca i32
  store i32 0, i32* %loop_cnt
  br label %while

while:                                            ; preds = %merge, %entry
  %loop_iter_cnt = load i32, i32* %loop_cnt
  %loop_cond = icmp slt i32 %loop_iter_cnt, %list_size
  br i1 %loop_cond, label %while_body, label %merge1

while_body:                                       ; preds = %while
  %to_idx = load i32, i32* %loop_cnt
  %list_get = call i1 @list_getbool({ i32*, i1* }* %list_load, i32 %to_idx)
  %if_cond = icmp eq i1 %list_get, %find_val
  br i1 %if_cond, label %then, label %else

merge:                                            ; preds = %else
  %loop_idx = load i32, i32* %loop_cnt
  %loop_itr = add i32 %loop_idx, 1
  store i32 %loop_itr, i32* %loop_cnt
  br label %while

then:                                             ; preds = %while_body
  ret i32 %to_idx

else:                                             ; preds = %while_body
  br label %merge

merge1:                                           ; preds = %while
  ret i32 -1
```

```
}

define i32 @list_findint({ i32*, i32* }*, i32) {
entry:
  %list_ptr_alloc = alloca { i32*, i32* }*
  store { i32*, i32* }* %0, { i32*, i32* }** %list_ptr_alloc
  %find_val_alloc = alloca i32
  store i32 %1, i32* %find_val_alloc
  %find_val = load i32, i32* %find_val_alloc
  %list_load = load { i32*, i32* }*, { i32*, i32* }** %list_ptr_alloc
  %list_size_ptr_ptr = getelementptr inbounds { i32*, i32* }, { i32*, i32* }* %list_load, i32 0, i32 0
  %list_size_ptr = load i32*, i32** %list_size_ptr_ptr
  %list_size = load i32, i32* %list_size_ptr
  %loop_cnt = alloca i32
  store i32 0, i32* %loop_cnt
  br label %while

while:                                            ; preds = %merge, %entry
  %loop_iter_cnt = load i32, i32* %loop_cnt
  %loop_cond = icmp slt i32 %loop_iter_cnt, %list_size
  br i1 %loop_cond, label %while_body, label %merge1

while_body:                                       ; preds = %while
  %to_idx = load i32, i32* %loop_cnt
  %list_get = call i32 @list_getint({ i32*, i32* }* %list_load, i32 %to_idx)
  %if_cond = icmp eq i32 %list_get, %find_val
  br i1 %if_cond, label %then, label %else

merge:                                            ; preds = %else
  %loop_idx = load i32, i32* %loop_cnt
  %loop_itr = add i32 %loop_idx, 1
  store i32 %loop_itr, i32* %loop_cnt
  br label %while

then:                                             ; preds = %while_body
  ret i32 %to_idx

else:                                             ; preds = %while_body
  br label %merge

merge1:                                           ; preds = %while
  ret i32 -1
}

define i32 @list_findfloat({ i32*, double* }*, double) {
entry:
  %list_ptr_alloc = alloca { i32*, double* }*
  store { i32*, double* }* %0, { i32*, double* }** %list_ptr_alloc
  %find_val_alloc = alloca double
  store double %1, double* %find_val_alloc
```

```
561.   %find_val = load double, double* %find_val_alloc
562.   %list_load = load { i32*, double* }*, { i32*, double* }** %list_ptr_alloc
563.   %list_size_ptr_ptr = getelementptr inbounds { i32*, double* }, { i32*,
    double* }* %list_load, i32 0, i32 0
564.   %list_size_ptr = load i32*, i32** %list_size_ptr_ptr
565.   %list_size = load i32, i32* %list_size_ptr
566.   %loop_cnt = alloca i32
567.   store i32 0, i32* %loop_cnt
568.   br label %while
569.
570. while:                                          ; preds = %merge, %entry
571.   %loop_iter_cnt = load i32, i32* %loop_cnt
572.   %loop_cond = icmp slt i32 %loop_iter_cnt, %list_size
573.   br i1 %loop_cond, label %while_body, label %merge1
574.
575. while_body:                                     ; preds = %while
576.   %to_idx = load i32, i32* %loop_cnt
577.   %list_get = call double @list_getfloat({ i32*, double* }* %list_load, i32 %to_idx)
578.   %if_cond = fcmp oeq double %list_get, %find_val
579.   br i1 %if_cond, label %then, label %else
580.
581. merge:                                          ; preds = %else
582.   %loop_idx = load i32, i32* %loop_cnt
583.   %loop_itr = add i32 %loop_idx, 1
584.   store i32 %loop_itr, i32* %loop_cnt
585.   br label %while
586.
587. then:                                           ; preds = %while_body
588.   ret i32 %to_idx
589.
590. else:                                           ; preds = %while_body
591.   br label %merge
592.
593. merge1:                                         ; preds = %while
594.   ret i32 -1
595. }
596.
597. define void @list_removebool({ i32*, i1* }*, i1) {
598. entry:
599.   %list_ptr_alloc = alloca { i32*, i1* }*
600.   store { i32*, i1* }* %0, { i32*, i1* }** %list_ptr_alloc
601.   %rem_val_ptr = alloca i1
602.   store i1 %1, i1* %rem_val_ptr
603.   %rem_val = load i1, i1* %rem_val_ptr
604.   %list_load = load { i32*, i1* }*, { i32*, i1* }** %list_ptr_alloc
605.   %list_size_ptr_ptr = getelementptr inbounds { i32*, i1* }, { i32*, i1* }* %list_load,
    i32 0, i32 0
606.   %list_size_ptr = load i32*, i32** %list_size_ptr_ptr
607.   %list_size = load i32, i32* %list_size_ptr
608.   %list_find = call i32 @list_findbool({ i32*, i1* }* %list_load, i1 %rem_val)
609.   %loop_cond = icmp sge i32 %list_find, 0
610.   br i1 %loop_cond, label %then, label %else
```

```
611.
612. merge:                                          ; preds = %else
613.   ret void
614.
615. then:                                           ; preds = %entry
616.   %loop_cnt_ptr = alloca i32
617.   %loop_start_idx = add i32 %list_find, 1
618.   store i32 %loop_start_idx, i32* %loop_cnt_ptr
619.   br label %while
620.
621. while:                                          ; preds = %while_body, %then
622.   %loop_cnt = load i32, i32* %loop_cnt_ptr
623.   %loop_cond1 = icmp slt i32 %loop_cnt, %list_size
624.   br i1 %loop_cond1, label %while_body, label %merge2
625.
626. while_body:                                     ; preds = %while
627.   %cur_idx = load i32, i32* %loop_cnt_ptr
628.   %shift_to_idx = sub i32 %cur_idx, 1
629.   %list_get = call i1 @list_getbool({ i32*, i1* }* %list_load, i32 %cur_idx)
630.   call void @list_setbool({ i32*, i1* }* %list_load, i32 %shift_to_idx, i1 %list_get)
631.   %loop_itr = add i32 %cur_idx, 1
632.   store i32 %loop_itr, i32* %loop_cnt_ptr
633.   br label %while
634.
635. merge2:                                         ; preds = %while
636.   %size_dec = sub i32 %list_size, 1
637.   store i32 %size_dec, i32* %list_size_ptr
638.   ret void
639.
640. else:                                           ; preds = %entry
641.   br label %merge
642. }
643.
644. define void @list_removeint({ i32*, i32* }*, i32) {
645. entry:
646.   %list_ptr_alloc = alloca { i32*, i32* }*
647.   store { i32*, i32* }* %0, { i32*, i32* }** %list_ptr_alloc
648.   %rem_val_ptr = alloca i32
649.   store i32 %1, i32* %rem_val_ptr
650.   %rem_val = load i32, i32* %rem_val_ptr
651.   %list_load = load { i32*, i32* }*, { i32*, i32* }** %list_ptr_alloc
652.   %list_size_ptr_ptr = getelementptr inbounds { i32*, i32* }, { i32*,
    i32* }* %list_load, i32 0, i32 0
653.   %list_size_ptr = load i32*, i32** %list_size_ptr_ptr
654.   %list_size = load i32, i32* %list_size_ptr
655.   %list_find = call i32 @list_findint({ i32*, i32* }* %list_load, i32 %rem_val)
656.   %loop_cond = icmp sge i32 %list_find, 0
657.   br i1 %loop_cond, label %then, label %else
658.
659. merge:                                          ; preds = %else
660.   ret void
```

```
661.
662. then:                                          ; preds = %entry
663.   %loop_cnt_ptr = alloca i32
664.   %loop_start_idx = add i32 %list_find, 1
665.   store i32 %loop_start_idx, i32* %loop_cnt_ptr
666.   br label %while
667.
668. while:                                          ; preds = %while_body, %then
669.   %loop_cnt = load i32, i32* %loop_cnt_ptr
670.   %loop_cond1 = icmp slt i32 %loop_cnt, %list_size
671.   br i1 %loop_cond1, label %while_body, label %merge2
672.
673. while_body:                                     ; preds = %while
674.   %cur_idx = load i32, i32* %loop_cnt_ptr
675.   %shift_to_idx = sub i32 %cur_idx, 1
676.   %list_get = call i32 @list_getint({ i32*, i32* }* %list_load, i32 %cur_idx)
677.   call void @list_setint({ i32*, i32* }* %list_load, i32 %shift_to_idx, i32 %list_get)
678.   %loop_itr = add i32 %cur_idx, 1
679.   store i32 %loop_itr, i32* %loop_cnt_ptr
680.   br label %while
681.
682. merge2:                                         ; preds = %while
683.   %size_dec = sub i32 %list_size, 1
684.   store i32 %size_dec, i32* %list_size_ptr
685.   ret void
686.
687. else:                                          ; preds = %entry
688.   br label %merge
689. }
690.
691. define void @list_removefloat({ i32*, double* }*, double) {
692. entry:
693.   %list_ptr_alloc = alloca { i32*, double* }*
694.   store { i32*, double* }* %0, { i32*, double* }** %list_ptr_alloc
695.   %rem_val_ptr = alloca double
696.   store double %1, double* %rem_val_ptr
697.   %rem_val = load double, double* %rem_val_ptr
698.   %list_load = load { i32*, double* }*, { i32*, double* }** %list_ptr_alloc
699.   %list_size_ptr_ptr = getelementptr inbounds { i32*, double* }, { i32*,
    double* }* %list_load, i32 0, i32 0
700.   %list_size_ptr = load i32*, i32** %list_size_ptr_ptr
701.   %list_size = load i32, i32* %list_size_ptr
702.   %list_find = call i32 @list_findfloat({ i32*, double* }* %list_load, double %rem_val)
703.   %loop_cond = icmp sge i32 %list_find, 0
704.   br i1 %loop_cond, label %then, label %else
705.
706. merge:                                         ; preds = %else
707.   ret void
708.
709. then:                                          ; preds = %entry
710.   %loop_cnt_ptr = alloca i32
```

```
  %loop_start_idx = add i32 %list_find, 1
  store i32 %loop_start_idx, i32* %loop_cnt_ptr
  br label %while

while:                                            ; preds = %while_body, %then
  %loop_cnt = load i32, i32* %loop_cnt_ptr
  %loop_cond1 = icmp slt i32 %loop_cnt, %list_size
  br i1 %loop_cond1, label %while_body, label %merge2

while_body:                                       ; preds = %while
  %cur_idx = load i32, i32* %loop_cnt_ptr
  %shift_to_idx = sub i32 %cur_idx, 1
  %list_get = call double @list_getfloat({ i32*, double* }* %list_load, i32 %cur_idx)
  call void @list_setfloat({ i32*, double* }* %list_load, i32 %shift_to_idx,
double %list_get)
  %loop_itr = add i32 %cur_idx, 1
  store i32 %loop_itr, i32* %loop_cnt_ptr
  br label %while

merge2:                                           ; preds = %while
  %size_dec = sub i32 %list_size, 1
  store i32 %size_dec, i32* %list_size_ptr
  ret void

else:                                             ; preds = %entry
  br label %merge
}

define void @list_insertbool({ i32*, i1* }*, i32, i1) {
entry:
  %list_ptr_alloc = alloca { i32*, i1* }*
  store { i32*, i1* }* %0, { i32*, i1* }** %list_ptr_alloc
  %list_load = load { i32*, i1* }*, { i32*, i1* }** %list_ptr_alloc
  %insert_idx_ptr = alloca i32
  store i32 %1, i32* %insert_idx_ptr
  %insert_idx = load i32, i32* %insert_idx_ptr
  %insert_val_ptr = alloca i1
  store i1 %2, i1* %insert_val_ptr
  %insert_val = load i1, i1* %insert_val_ptr
  %list_size_ptr_ptr = getelementptr inbounds { i32*, i1* }, { i32*, i1* }* %list_load,
i32 0, i32 0
  %list_size_ptr = load i32*, i32** %list_size_ptr_ptr
  %list_size = load i32, i32* %list_size_ptr
  %loop_cnt_ptr = alloca i32
  %last_index = sub i32 %list_size, 1
  store i32 %last_index, i32* %loop_cnt_ptr
  br label %while

while:                                            ; preds = %while_body, %entry
  %loop_cnt = load i32, i32* %loop_cnt_ptr
  %loop_cond = icmp sge i32 %loop_cnt, %insert_idx
  br i1 %loop_cond, label %while_body, label %merge
```

```
761.
762. while_body:                                        ; preds = %while
763.   %cur_idx = load i32, i32* %loop_cnt_ptr
764.   %shift_to_idx = add i32 %cur_idx, 1
765.   %list_get = call i1 @list_getbool({ i32*, i1* }* %list_load, i32 %cur_idx)
766.   call void @list_setbool({ i32*, i1* }* %list_load, i32 %shift_to_idx, i1 %list_get)
767.   %loop_itr = sub i32 %cur_idx, 1
768.   store i32 %loop_itr, i32* %loop_cnt_ptr
769.   br label %while
770.
771. merge:                                           ; preds = %while
772.   call void @list_setbool({ i32*, i1* }* %list_load, i32 %insert_idx, i1 %insert_val)
773.   %size_inc = add i32 %list_size, 1
774.   store i32 %size_inc, i32* %list_size_ptr
775.   ret void
776. }
777.
778. define void @list_insertint({ i32*, i32* }*, i32, i32) {
779. entry:
780.   %list_ptr_alloc = alloca { i32*, i32* }*
781.   store { i32*, i32* }* %0, { i32*, i32* }** %list_ptr_alloc
782.   %list_load = load { i32*, i32* }*, { i32*, i32* }** %list_ptr_alloc
783.   %insert_idx_ptr = alloca i32
784.   store i32 %1, i32* %insert_idx_ptr
785.   %insert_idx = load i32, i32* %insert_idx_ptr
786.   %insert_val_ptr = alloca i32
787.   store i32 %2, i32* %insert_val_ptr
788.   %insert_val = load i32, i32* %insert_val_ptr
789.   %list_size_ptr_ptr = getelementptr inbounds { i32*, i32* }, { i32*,
    i32* }* %list_load, i32 0, i32 0
790.   %list_size_ptr = load i32*, i32** %list_size_ptr_ptr
791.   %list_size = load i32, i32* %list_size_ptr
792.   %loop_cnt_ptr = alloca i32
793.   %last_index = sub i32 %list_size, 1
794.   store i32 %last_index, i32* %loop_cnt_ptr
795.   br label %while
796.
797. while:                                           ; preds = %while_body, %entry
798.   %loop_cnt = load i32, i32* %loop_cnt_ptr
799.   %loop_cond = icmp sge i32 %loop_cnt, %insert_idx
800.   br i1 %loop_cond, label %while_body, label %merge
801.
802. while_body:                                        ; preds = %while
803.   %cur_idx = load i32, i32* %loop_cnt_ptr
804.   %shift_to_idx = add i32 %cur_idx, 1
805.   %list_get = call i32 @list_getint({ i32*, i32* }* %list_load, i32 %cur_idx)
806.   call void @list_setint({ i32*, i32* }* %list_load, i32 %shift_to_idx, i32 %list_get)
807.   %loop_itr = sub i32 %cur_idx, 1
808.   store i32 %loop_itr, i32* %loop_cnt_ptr
809.   br label %while
810.
811. merge:                                           ; preds = %while
```

```
  call void @list_setint({ i32*, i32* }* %list_load, i32 %insert_idx, i32 %insert_val)
  %size_inc = add i32 %list_size, 1
  store i32 %size_inc, i32* %list_size_ptr
  ret void
}

define void @list_insertfloat({ i32*, double* }*, i32, double) {
entry:
  %list_ptr_alloc = alloca { i32*, double* }*
  store { i32*, double* }* %0, { i32*, double* }** %list_ptr_alloc
  %list_load = load { i32*, double* }*, { i32*, double* }** %list_ptr_alloc
  %insert_idx_ptr = alloca i32
  store i32 %1, i32* %insert_idx_ptr
  %insert_idx = load i32, i32* %insert_idx_ptr
  %insert_val_ptr = alloca double
  store double %2, double* %insert_val_ptr
  %insert_val = load double, double* %insert_val_ptr
  %list_size_ptr_ptr = getelementptr inbounds { i32*, double* }, { i32*, double* }* %list_load, i32 0, i32 0
  %list_size_ptr = load i32*, i32** %list_size_ptr_ptr
  %list_size = load i32, i32* %list_size_ptr
  %loop_cnt_ptr = alloca i32
  %last_index = sub i32 %list_size, 1
  store i32 %last_index, i32* %loop_cnt_ptr
  br label %while

while:                                            ; preds = %while_body, %entry
  %loop_cnt = load i32, i32* %loop_cnt_ptr
  %loop_cond = icmp sge i32 %loop_cnt, %insert_idx
  br i1 %loop_cond, label %while_body, label %merge

while_body:                                       ; preds = %while
  %cur_idx = load i32, i32* %loop_cnt_ptr
  %shift_to_idx = add i32 %cur_idx, 1
  %list_get = call double @list_getfloat({ i32*, double* }* %list_load, i32 %cur_idx)
  call void @list_setfloat({ i32*, double* }* %list_load, i32 %shift_to_idx, double %list_get)
  %loop_itr = sub i32 %cur_idx, 1
  store i32 %loop_itr, i32* %loop_cnt_ptr
  br label %while

merge:                                            ; preds = %while
  call void @list_setfloat({ i32*, double* }* %list_load, i32 %insert_idx, double %insert_val)
  %size_inc = add i32 %list_size, 1
  store i32 %size_inc, i32* %list_size_ptr
  ret void
}

define void @list_reversebool({ i32*, i1* }*) {
entry:
  %list_ptr_alloc = alloca { i32*, i1* }*
```

```
  store { i32*, i1* }* %0, { i32*, i1* }** %list_ptr_alloc
  %list_load = load { i32*, i1* }*, { i32*, i1* }** %list_ptr_alloc
  %list_size_ptr_ptr = getelementptr inbounds { i32*, i1* }, { i32*, i1* }* %list_load, i32 0, i32 0
  %list_size_ptr = load i32*, i32** %list_size_ptr_ptr
  %list_size = load i32, i32* %list_size_ptr
  %left_idx = alloca i32
  store i32 0, i32* %left_idx
  %right_idx = alloca i32
  %tmp = sub i32 %list_size, 1
  store i32 %tmp, i32* %right_idx
  br label %while

while:                                            ; preds = %while_body, %entry
  %right_idx6 = load i32, i32* %right_idx
  %left_idx7 = load i32, i32* %left_idx
  %while_cond = icmp slt i32 %left_idx7, %right_idx6
  br i1 %while_cond, label %while_body, label %merge

while_body:                                       ; preds = %while
  %left_idx1 = load i32, i32* %left_idx
  %right_idx2 = load i32, i32* %right_idx
  %list_get = call i1 @list_getbool({ i32*, i1* }* %list_load, i32 %left_idx1)
  %list_get3 = call i1 @list_getbool({ i32*, i1* }* %list_load, i32 %right_idx2)
  call void @list_setbool({ i32*, i1* }* %list_load, i32 %left_idx1, i1 %list_get3)
  call void @list_setbool({ i32*, i1* }* %list_load, i32 %right_idx2, i1 %list_get)
  %tmp4 = add i32 %left_idx1, 1
  store i32 %tmp4, i32* %left_idx
  %tmp5 = sub i32 %right_idx2, 1
  store i32 %tmp5, i32* %right_idx
  br label %while

merge:                                            ; preds = %while
  ret void
}

define void @list_reverseint({ i32*, i32* }*) {
entry:
  %list_ptr_alloc = alloca { i32*, i32* }*
  store { i32*, i32* }* %0, { i32*, i32* }** %list_ptr_alloc
  %list_load = load { i32*, i32* }*, { i32*, i32* }** %list_ptr_alloc
  %list_size_ptr_ptr = getelementptr inbounds { i32*, i32* }, { i32*, i32* }* %list_load, i32 0, i32 0
  %list_size_ptr = load i32*, i32** %list_size_ptr_ptr
  %list_size = load i32, i32* %list_size_ptr
  %left_idx = alloca i32
  store i32 0, i32* %left_idx
  %right_idx = alloca i32
  %tmp = sub i32 %list_size, 1
  store i32 %tmp, i32* %right_idx
  br label %while

```

```
while:                                            ; preds = %while_body, %entry
  %right_idx6 = load i32, i32* %right_idx
  %left_idx7 = load i32, i32* %left_idx
  %while_cond = icmp slt i32 %left_idx7, %right_idx6
  br i1 %while_cond, label %while_body, label %merge

while_body:                                       ; preds = %while
  %left_idx1 = load i32, i32* %left_idx
  %right_idx2 = load i32, i32* %right_idx
  %list_get = call i32 @list_getint({ i32*, i32* }* %list_load, i32 %left_idx1)
  %list_get3 = call i32 @list_getint({ i32*, i32* }* %list_load, i32 %right_idx2)
  call void @list_setint({ i32*, i32* }* %list_load, i32 %left_idx1, i32 %list_get3)
  call void @list_setint({ i32*, i32* }* %list_load, i32 %right_idx2, i32 %list_get)
  %tmp4 = add i32 %left_idx1, 1
  store i32 %tmp4, i32* %left_idx
  %tmp5 = sub i32 %right_idx2, 1
  store i32 %tmp5, i32* %right_idx
  br label %while

merge:                                            ; preds = %while
  ret void
}

define void @list_reversefloat({ i32*, double* }*) {
entry:
  %list_ptr_alloc = alloca { i32*, double* }*
  store { i32*, double* }* %0, { i32*, double* }** %list_ptr_alloc
  %list_load = load { i32*, double* }*, { i32*, double* }** %list_ptr_alloc
  %list_size_ptr_ptr = getelementptr inbounds { i32*, double* }, { i32*,
double* }* %list_load, i32 0, i32 0
  %list_size_ptr = load i32*, i32** %list_size_ptr_ptr
  %list_size = load i32, i32* %list_size_ptr
  %left_idx = alloca i32
  store i32 0, i32* %left_idx
  %right_idx = alloca i32
  %tmp = sub i32 %list_size, 1
  store i32 %tmp, i32* %right_idx
  br label %while

while:                                            ; preds = %while_body, %entry
  %right_idx6 = load i32, i32* %right_idx
  %left_idx7 = load i32, i32* %left_idx
  %while_cond = icmp slt i32 %left_idx7, %right_idx6
  br i1 %while_cond, label %while_body, label %merge

while_body:                                       ; preds = %while
  %left_idx1 = load i32, i32* %left_idx
  %right_idx2 = load i32, i32* %right_idx
  %list_get = call double @list_getfloat({ i32*, double* }* %list_load, i32 %left_idx1)
  %list_get3 = call double @list_getfloat({ i32*, double* }* %list_load,
i32 %right_idx2)
```

```
  call void @list_setfloat({ i32*, double* }* %list_load, i32 %left_idx1,
double %list_get3)
  call void @list_setfloat({ i32*, double* }* %list_load, i32 %right_idx2,
double %list_get)
  %tmp4 = add i32 %left_idx1, 1
  store i32 %tmp4, i32* %left_idx
  %tmp5 = sub i32 %right_idx2, 1
  store i32 %tmp5, i32* %right_idx
  br label %while

merge:                                            ; preds = %while
  ret void
}

define i32 @main() {
entry:
  %a = alloca { i32*, i32* }
  %list_size_ptr = getelementptr inbounds { i32*, i32* }, { i32*, i32* }* %a, i32 0,
i32 0
  %list_size = alloca i32
  store i32 0, i32* %list_size
  store i32* %list_size, i32** %list_size_ptr
  %list.arry = getelementptr inbounds { i32*, i32* }, { i32*, i32* }* %a, i32 0, i32 1
  %p = alloca i32, i32 1028
  store i32* %p, i32** %list.arry
  %i = alloca i32
  %new_list_ptr = alloca { i32*, i32* }
  %list_size_ptr1 = getelementptr inbounds { i32*, i32* }, { i32*,
i32* }* %new_list_ptr, i32 0, i32 0
  %list_size2 = alloca i32
  store i32 0, i32* %list_size2
  store i32* %list_size2, i32** %list_size_ptr1
  %list.arry3 = getelementptr inbounds { i32*, i32* }, { i32*, i32* }* %new_list_ptr,
i32 0, i32 1
  %p4 = alloca i32, i32 1028
  store i32* %p4, i32** %list.arry3
  call void @list_pushint({ i32*, i32* }* %new_list_ptr, i32 10)
  call void @list_pushint({ i32*, i32* }* %new_list_ptr, i32 9)
  call void @list_pushint({ i32*, i32* }* %new_list_ptr, i32 8)
  call void @list_pushint({ i32*, i32* }* %new_list_ptr, i32 7)
  call void @list_pushint({ i32*, i32* }* %new_list_ptr, i32 6)
  call void @list_pushint({ i32*, i32* }* %new_list_ptr, i32 5)
  call void @list_pushint({ i32*, i32* }* %new_list_ptr, i32 4)
  call void @list_pushint({ i32*, i32* }* %new_list_ptr, i32 3)
  call void @list_pushint({ i32*, i32* }* %new_list_ptr, i32 2)
        call void @list_pushint({ i32*, i32* }* %new_list_ptr, i32 1)
        call void @list_pushint({ i32*, i32* }* %new_list_ptr, i32 0)
        %new_list = load { i32*, i32* }, { i32*, i32* }* %new_list_ptr
        store { i32*, i32* } %new_list, { i32*, i32* }* %a
        %a5 = load { i32*, i32* }, { i32*, i32* }* %a
        call void @quickSort({ i32*, i32* } %a5)
        store i32 0, i32* %i
        br label %while
```

```
1008.
1009.       while:                                            ; preds = %while_body, %entry
1010.         %i8 = load i32, i32* %i
1011.         %list_size9 = call i32 @list_sizeint({ i32*, i32* }* %a)
1012.         %tmp10 = icmp slt i32 %i8, %list_size9
1013.         br i1 %tmp10, label %while_body, label %merge
1014.
1015.       while_body:                                       ; preds = %while
1016.         %i6 = load i32, i32* %i
1017.         %list_get = call i32 @list_getint({ i32*, i32* }* %a, i32 %i6)
1018.         %printf = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([4 x i8], [4
    x i8]* @fmt, i32 0, i32 0), i32 %list_get)
1019.         %i7 = load i32, i32* %i
1020.         %tmp = add i32 %i7, 1
1021.         store i32 %tmp, i32* %i
1022.         br label %while
1023.
1024.       merge:                                            ; preds = %while
1025.         ret i32 0
1026.       }
1027.
1028.       define void @quickSort({ i32*, i32* } %a) {
1029.       entry:
1030.         %a1 = alloca { i32*, i32* }
1031.         %list_size_ptr = getelementptr inbounds { i32*, i32* }, { i32*, i32* }* %a1,
    i32 0, i32 0
1032.         %list_size = alloca i32
1033.         store i32 0, i32* %list_size
1034.         store i32* %list_size, i32** %list_size_ptr
1035.         %list.arry = getelementptr inbounds { i32*, i32* }, { i32*, i32* }* %a1, i32
    0, i32 1
1036.         %p = alloca i32, i32 1028
1037.         store i32* %p, i32** %list.arry
1038.         store { i32*, i32* } %a, { i32*, i32* }* %a1
1039.         %list_size2 = call i32 @list_sizeint({ i32*, i32* }* %a1)
1040.         %tmp = sub i32 %list_size2, 1
1041.         %a3 = load { i32*, i32* }, { i32*, i32* }* %a1
1042.         call void @quickSortImpl({ i32*, i32* } %a3, i32 0, i32 %tmp)
1043.         ret void
1044.       }
1045.
1046.       define void @quickSortImpl({ i32*, i32* } %a, i32 %low, i32 %high) {
1047.       entry:
1048.         %a1 = alloca { i32*, i32* }
1049.         %list_size_ptr = getelementptr inbounds { i32*, i32* }, { i32*, i32* }* %a1,
    i32 0, i32 0
1050.         %list_size = alloca i32
1051.         store i32 0, i32* %list_size
1052.         store i32* %list_size, i32** %list_size_ptr
1053.         %list.arry = getelementptr inbounds { i32*, i32* }, { i32*, i32* }* %a1, i32
    0, i32 1
1054.         %p = alloca i32, i32 1028
```

```
1055.         store i32* %p, i32** %list.arry
1056.         store { i32*, i32* } %a, { i32*, i32* }* %a1
1057.         %low2 = alloca i32
1058.         store i32 %low, i32* %low2
1059.         %high3 = alloca i32
1060.         store i32 %high, i32* %high3
1061.         %pi = alloca i32
1062.         %low4 = load i32, i32* %low2
1063.         %high5 = load i32, i32* %high3
1064.         %tmp = icmp slt i32 %low4, %high5
1065.         br i1 %tmp, label %then, label %else
1066.
1067.       merge:                                            ; preds = %else, %then
1068.         ret void
1069.
1070.       then:                                             ; preds = %entry
1071.         %high6 = load i32, i32* %high3
1072.         %low7 = load i32, i32* %low2
1073.         %a8 = load { i32*, i32* }, { i32*, i32* }* %a1
1074.         %partition_result = call i32 @partition({ i32*, i32* } %a8, i32 %low7,
    i32 %high6)
1075.         store i32 %partition_result, i32* %pi
1076.         %pi9 = load i32, i32* %pi
1077.         %tmp10 = sub i32 %pi9, 1
1078.         %low11 = load i32, i32* %low2
1079.         %a12 = load { i32*, i32* }, { i32*, i32* }* %a1
1080.         call void @quickSortImpl({ i32*, i32* } %a12, i32 %low11, i32 %tmp10)
1081.         %high13 = load i32, i32* %high3
1082.         %pi14 = load i32, i32* %pi
1083.         %tmp15 = add i32 %pi14, 1
1084.         %a16 = load { i32*, i32* }, { i32*, i32* }* %a1
1085.         call void @quickSortImpl({ i32*, i32* } %a16, i32 %tmp15, i32 %high13)
1086.         br label %merge
1087.
1088.       else:                                             ; preds = %entry
1089.         br label %merge
1090.       }
1091.
1092.       define i32 @partition({ i32*, i32* } %a, i32 %low, i32 %high) {
1093.       entry:
1094.         %a1 = alloca { i32*, i32* }
1095.         %list_size_ptr = getelementptr inbounds { i32*, i32* }, { i32*, i32* }* %a1,
    i32 0, i32 0
1096.         %list_size = alloca i32
1097.         store i32 0, i32* %list_size
1098.         store i32* %list_size, i32** %list_size_ptr
1099.         %list.arry = getelementptr inbounds { i32*, i32* }, { i32*, i32* }* %a1, i32
    0, i32 1
1100.         %p = alloca i32, i32 1028
1101.         store i32* %p, i32** %list.arry
1102.         store { i32*, i32* } %a, { i32*, i32* }* %a1
1103.         %low2 = alloca i32
```

```
1104.         store i32 %low, i32* %low2
1105.         %high3 = alloca i32
1106.         store i32 %high, i32* %high3
1107.         %i = alloca i32
1108.         %j = alloca i32
1109.         %pivot = alloca i32
1110.         %temp = alloca i32
1111.         %high4 = load i32, i32* %high3
1112.         %list_get = call i32 @list_getint({ i32*, i32* }* %a1, i32 %high4)
1113.         store i32 %list_get, i32* %pivot
1114.         %low5 = load i32, i32* %low2
1115.         %tmp = sub i32 %low5, 1
1116.         store i32 %tmp, i32* %i
1117.         %low6 = load i32, i32* %low2
1118.         store i32 %low6, i32* %j
1119.         br label %while
1120.
1121.       while:                                            ; preds = %merge, %entry
1122.         %j22 = load i32, i32* %j
1123.         %high23 = load i32, i32* %high3
1124.         %tmp24 = sub i32 %high23, 1
1125.         %tmp25 = icmp sle i32 %j22, %tmp24
1126.         br i1 %tmp25, label %while_body, label %merge26
1127.
1128.       while_body:                                       ; preds = %while
1129.         %j7 = load i32, i32* %j
1130.         %list_get8 = call i32 @list_getint({ i32*, i32* }* %a1, i32 %j7)
1131.         %pivot9 = load i32, i32* %pivot
1132.         %tmp10 = icmp sle i32 %list_get8, %pivot9
1133.         br i1 %tmp10, label %then, label %else
1134.
1135.       merge:                                            ; preds = %else, %then
1136.         %j20 = load i32, i32* %j
1137.         %tmp21 = add i32 %j20, 1
1138.         store i32 %tmp21, i32* %j
1139.         br label %while
1140.
1141.       then:                                             ; preds = %while_body
1142.         %i11 = load i32, i32* %i
1143.         %tmp12 = add i32 %i11, 1
1144.         store i32 %tmp12, i32* %i
1145.         %i13 = load i32, i32* %i
1146.         %list_get14 = call i32 @list_getint({ i32*, i32* }* %a1, i32 %i13)
1147.         store i32 %list_get14, i32* %temp
1148.         %j15 = load i32, i32* %j
1149.         %list_get16 = call i32 @list_getint({ i32*, i32* }* %a1, i32 %j15)
1150.         %i17 = load i32, i32* %i
1151.         call void @list_setint({ i32*, i32* }* %a1, i32 %i17, i32 %list_get16)
1152.         %temp18 = load i32, i32* %temp
1153.         %j19 = load i32, i32* %j
1154.         call void @list_setint({ i32*, i32* }* %a1, i32 %j19, i32 %temp18)
1155.         br label %merge
```

```
1156.
1157.       else:                                             ; preds = %while_body
1158.         br label %merge
1159.
1160.       merge26:                                          ; preds = %while
1161.         %i27 = load i32, i32* %i
1162.         %tmp28 = add i32 %i27, 1
1163.         %list_get29 = call i32 @list_getint({ i32*, i32* }* %a1, i32 %tmp28)
1164.         store i32 %list_get29, i32* %temp
1165.         %high30 = load i32, i32* %high3
1166.         %list_get31 = call i32 @list_getint({ i32*, i32* }* %a1, i32 %high30)
1167.         %i32 = load i32, i32* %i
1168.         %tmp33 = add i32 %i32, 1
1169.         call void @list_setint({ i32*, i32* }* %a1, i32 %tmp33, i32 %list_get31)
1170.         %temp34 = load i32, i32* %temp
1171.         %high35 = load i32, i32* %high3
1172.         call void @list_setint({ i32*, i32* }* %a1, i32 %high35, i32 %temp34)
1173.         %i36 = load i32, i32* %i
1174.         %tmp37 = add i32 %i36, 1
1175.         ret i32 %tmp37
1176.       }
```

2. Test Suite

Failures and Success unit test cases were created to ensure reliability of the functionality implemented. On top of the existing MicroC tests, there were several new test cases written, especially around the new list data structure. Special attention was given to edge cases and potential off-by-one errors.

```
--- SUCCESS TEST CASES --
int add(int x, int y)
{
  return x + y;
}

int main()
{
  printi( add(17, 25) );
  return 0;
}

output: 42
--- Test Case
int main()
{
  printi(39 + 3);
  return 0;
}
output: 42
--- Test Case
int main()
{
  printi(1 + 2 * 3 + 4);
  return 0;
}
output: 11
--- Test Case
int foo(int a)
{
  return a;
}

int main()
{
  int a;
  a = 42;
  a = a + 5;
  printi(a);
  return 0;
}
output: 47
--- Test Case
int fib(int x)
```

```
{
  if (x < 2) return 1;
  return fib(x-1) + fib(x-2);
}

int main()
{
  printi(fib(0));
  printi(fib(1));
  printi(fib(2));
  printi(fib(3));
  printi(fib(4));
  printi(fib(5));
  return 0;
}
output:
1
1
2
3
5
8
--- Test Case
int main()
{
  float a;
  a = 3.14159267;
  printf(a);
  return 0;
}
output: 3.14159
--- Test Case
int main()
{
  float a;
  float b;
  float c;
  a = 3.14159267;
  b = -2.71828;
  c = a + b;
  printf(c);
  return 0;
}
output: 0.423313
--- Test Case
void testfloat(float a, float b)
{
  printf(a + b);
  printf(a - b);
  printf(a * b);
  printf(a / b);
  printb(a == b);
  printb(a == a);
```

```
printb(a != b);
printb(a != a);
        printb(a > b);
        printb(a >= b);
        printb(a < b);
        printb(a <= b);
      }

      int main()
      {
        float c;
        float d;

        c = 42.0;
        d = 3.14159;

        testfloat(c, d);

        testfloat(d, d);

        return 0;
      }
      output:
      45.1416
      38.8584
      131.947
      13.369
      0
      1
      1
      0
      1
      1
      0
      0
      6.28318
      0
      9.86959
      1
      1
      1
      0
      0
      0
      1
      0
      1
      --- Test Case
      int main()
      {
        int i;
        for (i = 0 ; i < 5 ; i = i + 1) {
```

```
    printi(i);
  }
  printi(42);
  return 0;
}
output:
0
1
2
3
4
42
--- Test Case
int main()
{
  int i;
  i = 0;
  for ( ; i < 5; ) {
    printi(i);
    i = i + 1;
  }
  printi(42);
  return 0;
}
output:
0
1
2
3
4
42
--- Test Case
int add(int a, int b)
{
  return a + b;
}

int main()
{
  int a;
  a = add(39, 3);
  printi(a);
  return 0;
}
output: 42
--- Test Case
/* Bug noticed by Pin-Chin Huang */

int fun(int x, int y)
{
  return 0;
}

```

```
int main()
{
  int i;
  i = 1;

  fun(i = 2, i = i+1);

  printi(i);
  return 0;
}

output: 2
--- Test Case
void printem(int a, int b, int c, int d)
{
  printi(a);
  printi(b);
  printi(c);
  printi(d);
}

int main()
{
  printem(42,17,192,8);
  return 0;
}
output:
42
17
192
8
--- Test Case
int add(int a, int b)
{
  int c;
  c = a + b;
  return c;
}

int main()
{
  int d;
  d = add(52, 10);
  printi(d);
  return 0;
}
output: 62
--- Test Case
int foo(int a)
{
  return a;
}
```

```
int main()
{
  return 0;
}
--- Test Case
void foo() {}

int bar(int a, bool b, int c) { return a + c; }

int main()
{
  printi(bar(17, false, 25));
  return 0;
}
output: 42
--- Test Case
int a;

void foo(int c)
{
  a = c + 42;
}

int main()
{
  foo(73);
  printi(a);
  return 0;
}
output: 115
--- Test Case
void foo(int a)
{
  printi(a + 3);
}

int main()
{
  foo(40);
  return 0;
}
output: 43
--- Test Case
void foo(int a)
{
  printi(a + 3);
  return;
}

int main()
{
```

```
  foo(40);
  return 0;
}
output: 43
--- Test Case
int gcd(int a, int b) {
  while (a != b) {
    if (a > b) a = a - b;
    else b = b - a;
  }
  return a;
}

int main()
{
  printi(gcd(2,14));
  printi(gcd(3,15));
  printi(gcd(99,121));
  return 0;
}
output:
2
3
11
--- Test Case
int gcd(int a, int b) {
  while (a != b)
    if (a > b) a = a - b;
    else b = b - a;
  return a;
}

int main()
{
  printi(gcd(14,21));
  printi(gcd(8,36));
  printi(gcd(99,121));
  return 0;
}
output:
7
4
11
int a;
int b;
--- Test Case
void printa()
{
  printi(a);
}

void printbb()
```

```
{
  printi(b);
}

void incab()
{
  a = a + 1;
  b = b + 1;
}

int main()
{
  a = 42;
  b = 21;
  printa();
  printbb();
  incab();
  printa();
  printbb();
  return 0;
}
output:
42
21
43
22
bool i;

int main()
{
  int i; /* Should hide the global i */

  i = 42;
  printi(i + i);
  return 0;
}
output: 84
--- Test Case
int i;
bool b;
int j;

int main()
{
  i = 42;
  j = 10;
  printi(i + j);
  return 0;
}
output: 52
--- Test Case
/*
```

```
411.        * "Hello World" - Literal Print Test
412.        */
413.
414.       int main()
415.       {
416.         printi(1234);
417.         printf(123.4);
418.         printb(true);
419.         prints("Hello World!");
420.         return 0;
421.       }
422.       output:
423.       1234
424.       123.4
425.       1
426.       Hello World!
427.       --- Test Case
428.       int main()
429.       {
430.         if (true) printi(42);
431.         printi(17);
432.         return 0;
433.       }
434.       output:
435.       42
436.       17
437.       --- Test Case
438.       int main()
439.       {
440.         if (true) printi(42); else printi(8);
441.         printi(17);
442.         return 0;
443.       }
444.       output:
445.       42
446.       17
447.       --- Test Case
448.       int main()
449.       {
450.         if (false) printi(42);
451.         printi(17);
452.         return 0;
453.       }
454.       output: 17
455.       --- Test Case
456.       int main()
457.       {
458.         if (false) printi(42); else printi(8);
459.         printi(17);
460.         return 0;
461.       }
462.       output:
463.       8
```

```
17
--- Test Case
int cond(bool b)
{
  int x;
  if (b)
    x = 42;
  else
    x = 17;
  return x;
}

int main()
{
 printi(cond(true));
 printi(cond(false));
 return 0;
}
output:
42
17
--- Test Case
int cond(bool b)
{
  int x;
  x = 10;
  if (b)
    if (x == 10)
      x = 42;
  else
    x = 17;
  return x;
}

int main()
{
 printi(cond(true));
 printi(cond(false));
 return 0;
}
output:
42
10
--- Test Case
int main()
{
  list<int> a;
  list<float> b;
  list<bool> c;
  list_push(a, 1);
  list_push(b, 3.14);
  list_push(c, false);
  printi(a[0]);
```

```
  printf(b[0]);
  printb(c[0]);

  return 0;
}
output:
1
3.14
0
--- Test Case
int main()
{
  list<int> a;
  list_push(a, 100);
  list_push(a, 101);
  printi(list_get(a, 0));
  printi(a[0]);
  printi(list_get(a, 1));
  printi(a[1]);
  return 0;
}
output:
100
100
101
101
--- Test Case
int main()
{
  list<int> a;
  int i;
  int x;
  for (i=0; i<10; ++i) {
    list_push(a, i);
  }
  x = list_pop(a);
  printi(x);
  x = list_pop(a);
  printi(x);
  x = list_size(a);
  printi(x);
  return 0;
}
output:
9
8
8
--- Test Case
int main()
{
  list<int> a;
  list_push(a, 100);
  list_push(a, 101);
```

```
  printi(list_get(a, 0));
  printi(a[0]);
  printi(list_get(a, 1));
  printi(a[1]);
  list_set(a, 0, 102);
  list_set(a, 1, 103);
  printi(a[0]);
  printi(a[1]);
  a[0] = 104;
  a[1] = 105;
  printi(a[0]);
  printi(a[1]);
  return 0;
}
output:
100
100
101
101
102
103
104
105
--- Test Case
int main()
{
  list<int> a;
  list<int> b;
  list<int> c;
  int i;
  c = a;
  for (i=0; i<5; ++i) { list_push(a, i); }
  b = a;
  printi(b[0]);
  printi(#b);
  list_push(b, 100);
  printi(#b);
  printi(b[#b-1]);
  list_push(c, 101);
  printi(#c);
  printi(c[#c-1]);
  list_push(a, 102);
  printi(#a);
  printi(a[#a-1]);

  return 0;
}
output:
0
5
6
100
7
```

```
101
8
102
--- Test Case
int main()
{
  list<int> a;
  list<int> b;
  int i;
  for (i=0; i<10; ++i) {
    list_push(a, i);
  }
  b = a[2:7];
  printi(#b);
  for (i=0; i<#b; ++i) {
    printi(b[i]);
  }

  return 0;
}
output:
6
2
3
4
5
6
7
--- Test Case
void list_push2(list<int> l, int val) {
  list_push(l, val);
  return;
}

int list_size2(list<int> l) {
  return #l;
}

int main()
{
  list<int> a;
  int i;
  for (i=0; i<5; ++i) {
    list_push(a, i);
  }
  list_push2(a, 100);
  list_push2(a, 101);
  for (i=0; i<#a; ++i) {
    printi(a[i]);
  }
  printi(list_size2(a));

```

```
  return 0;
}
output:
0
1
2
3
4
100
101
7
--- Test Case
int main()
{
  list<int> a;
  int i;
  for (i=0; i<5; ++i) {
    list_push(a, i);
  }
  printi(#a);
  list_clear(a);
  printi(#a);
  list_push(a, 10);
  list_push(a, 20);
  list_push(a, 30);
  printi(#a);
  for (i=0; i<3; ++i) {
    printi(a[i]);
  }

  return 0;
}
output:
5
0
3
10
20
30
--- Test Case
int main()
{
  list<int> a;
  int i;
  for (i=0; i<5; ++i) {
    list_push(a, i);
  }
  printi(list_find(a, 2));
  printi(list_find(a, 6));
  return 0;
}
output:
2
```

```
-1
--- Test Case
int main()
{
  list<int> a;
  int i;
  for (i=0; i<5; ++i) {
    list_push(a, i);
  }
  list_remove(a, 3);
  for (i=0; i<#a; ++i) {
    printi(a[i]);
  }
  list_remove(a, 9);
  printi(#a);
  list_remove(a, 4);
  for (i=0; i<#a; ++i) {
    printi(a[i]);
  }
  return 0;
}
output:
0
1
2
4
4
0
1
2
--- Test Case
int main()
{
  list<int> a;
  int i;
  for (i=0; i<5; ++i) {
    list_push(a, i);
  }

  list_insert(a, 0, 99);
  printi(#a);
  for (i=0; i<#a; ++i) {
    printi(a[i]);
  }

  list_insert(a, #a-1, 101);
  printi(#a);
  for (i=0; i<#a; ++i) {
    printi(a[i]);
  }

  list_insert(a, #a, 102);
```

```
  printi(#a);
  for (i=0; i<#a; ++i) {
    printi(a[i]);
  }

  list_insert(a, 2, 103);
  printi(#a);
  for (i=0; i<#a; ++i) {
    printi(a[i]);
  }

  return 0;
}
output:
6
99
0
1
2
3
4
7
99
0
1
2
3
101
4
8
99
0
1
2
3
101
4
102
9
99
0
103
1
2
3
101
4
102
--- Test Case
int main()
{
  list<int> a;
  int i;
```

```
  for (i=0; i<5; ++i) {
    list_push(a, i);
  }
  list_rev(a);
  for (i=0; i<#a; ++i) {
    printi(a[i]);
  }
  list_insert(a, 0, 5);
  list_rev(a);
  for (i=0; i<#a; ++i) {
    printi(a[i]);
  }

  return 0;
}
output:
4
3
2
1
0
0
1
2
3
4
5
--- Test Case
int main()
{
  list<int> a;
  list<float> b;
  list<bool> c;
  int i;

  a = [0, 1, 2, 3];
  for (i=0; i<#a; ++i) {
    printi(a[i]);
  }

  b = [0.2, 1.1, 2.4, 3.14];
  for (i=0; i<#b; ++i) {
    printf(b[i]);
  }

  c = [true, true, false, true];
  for (i=0; i<#c; ++i) {
    printb(c[i]);
  }

  return 0;
}
```

```
output:
0
1
2
3
0.2
1.1
2.4
3.14
1
1
0
1
--- Test Case
int main()
{
  list<int> a;
  list<int> b;
  int i;

  a = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10];

  b = a[2:6];
  for (i=0; i<#b; ++i) {
    printi(b[i]);
  }

  b = a[:6];
  for (i=0; i<#b; ++i) {
    printi(b[i]);
  }

  b = a[6:];
  for (i=0; i<#b; ++i) {
    printi(b[i]);
  }

  b = a[:];
  for (i=0; i<#b; ++i) {
    printi(b[i]);
  }

  return 0;
}
output:
2
3
4
5
6
0
1
```

```
2
3
4
5
6
6
7
8
9
10
0
1
2
3
4
5
6
7
8
9
10
--- Test Case
void foo(bool i)
{
  int i; /* Should hide the formal i */

  i = 42;
  printi(i + i);
}

int main()
{
  foo(true);
  return 0;
}
output: 84
--- Test Case
int foo(int a, bool b)
{
  int c;
  bool d;

  c = a;

  return c + 10;
}

int main() {
 printi(foo(37, false));
 return 0;
}
output: 47
```

```
--- Test Case
int main()
{
  int x;
  x = 1;
  printi(x);
  printi(--x);
  printi(x);
  return 0;
}
output:
1
0
0
--- Test Case
int main()
{
  int x;
  x = 1;
  printi(x);
  printi(x--);
  printi(x);
  return 0;
}
output:
1
1
0
--- Test Case
int main()
{
  int x;
  int y;
  x = 10;
  y = 3;
  printi(x % y);
  x = 10;
  y = 4;
  printi(x % y);
  x = 10;
  y = 5;
  printi(x % y);
  x = 0;
  y = 10;
  printi(x % y);
}
output:
1
2
0
0
--- Test Case
int main()
```

```
{
  printi(1 + 2);
  printi(1 - 2);
  printi(1 * 2);
  printi(100 / 2);
  printi(99);
  printb(1 == 2);
  printb(1 == 1);
  printi(99);
  printb(1 != 2);
  printb(1 != 1);
  printi(99);
  printb(1 < 2);
  printb(2 < 1);
  printi(99);
  printb(1 <= 2);
  printb(1 <= 1);
  printb(2 <= 1);
  printi(99);
  printb(1 > 2);
  printb(2 > 1);
  printi(99);
  printb(1 >= 2);
  printb(1 >= 1);
  printb(2 >= 1);
  return 0;
}
output:
3
-1
2
50
99
0
1
99
1
0
99
1
0
99
1
1
0
99
0
1
99
0
1
1
--- Test Case
```

```
int main()
{
  printb(true);
  printb(false);
  printb(true && true);
  printb(true && false);
  printb(false && true);
  printb(false && false);
  printb(true || true);
  printb(true || false);
  printb(false || true);
  printb(false || false);
  printb(!false);
  printb(!true);
  printi(-10);
}
output:
1
0
1
0
0
0
1
1
1
0
1
0
-10
--- Test Case
int main()
{
  int x;
  x = 0;
  printi(x);
  printi(++x);
  printi(x);
  return 0;
}
output:
0
1
1
--- Test Case
int main()
{
  int x;
  x = 0;
  printi(x);
  printi(x++);
  printi(x);
  return 0;
```

```
}
output:
0
0
1
--- Test Case
/* merges two sorted sublists of arr[] (arr[0..m], arr[m+1..r]) in-place with QuickSort algorithm */

int partition(list<int> a, int low, int high)
{
    int i;
    int j;
    int pivot;
    int temp;

    pivot = a[high];
    i = (low - 1);
    for (j = low; j <= high - 1; j++)
    {
        if (a[j] <= pivot)
        {
            i++;
            temp = a[i];
            a[i] = a[j];
            a[j] = temp;
        }
    }
    temp = a[i+1];
    a[i+1] = a[high];
    a[high] = temp;
    return (i + 1);
}

void quickSortImpl(list<int> a, int low, int high)
{
    int pi;
    if (low < high)
    {
        pi = partition(a, low, high);
        quickSortImpl(a, low, pi - 1);
        quickSortImpl(a, pi + 1, high);
    }
    return;
}

void quickSort(list<int> a)
{
    quickSortImpl(a, 0, #a - 1);
    return;
}

```

```
int main()
{
    list<int> a;
    int i;

    a = [10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0];

    quickSort(a);

    for (i=0; i<#a; ++i) {
      printi(a[i]);
    }

    return 0;
}
output:
0
1
2
3
4
5
6
7
8
9
10
--- Test Case
int main()
{
  int a;
  a = 42;
  printi(a);
  return 0;
}
output: 42
--- Test Case
int a;

void foo(int c)
{
  a = c + 42;
}
--- Test Case
int main()
{
  foo(73);
  printi(a);
  return 0;
}
output: 115
--- Test Case
```

```
int main()
{
  int i;
  i = 5;
  while (i > 0) {
    printi(i);
    i = i - 1;
  }
  printi(42);
  return 0;
}
output:
5
4
3
2
1
42
--- Test Case
int foo(int a)
{
  int j;
  j = 0;
  while (a > 0) {
    j = j + 2;
    a = a - 1;
  }
  return j;
}

int main()
{
  printi(foo(7));
  return 0;
}
output: 14
```

```
-- FAILURE TEST CASES --
Fatal error: exception Failure("illegal assignment int = bool in i = false")
int main()
{
  int i;
  bool b;

  i = 42;
  i = 10;
  b = true;
  b = false;
  i = false; /* Fail: assigning a bool to an integer */
}
--- Test Case
Fatal error: exception Failure("illegal assignment bool = int in b = 48")
int main()
{
  int i;
  bool b;

  b = 48; /* Fail: assigning an integer to a bool */
}
--- Test Case
Fatal error: exception Failure("illegal assignment int = void in i = myvoid()")
void myvoid()
{
  return;
}

int main()
{
  int i;

  i = myvoid(); /* Fail: assigning a void to an integer */
}
--- Test Case
Fatal error: exception Failure("nothing may follow a return")
int main()
{
  int i;

  i = 15;
  return i;
  i = 32; /* Error: code after a return */
}
--- Test Case
Fatal error: exception Failure("nothing may follow a return")
int main()
{
  int i;

  {
```

```
      i = 15;
      return i;
    }
    i = 32; /* Error: code after a return */
}
--- Test Case
Fatal error: exception Failure("illegal binary operator bool + int in d + a")
int a;
bool b;

void foo(int c, bool d)
{
  int dd;
  bool e;
  a + c;
  c - a;
  a * 3;
  c / 2;
  d + a; /* Error: bool + int */
}

int main()
{
  return 0;
}
--- Test Case
Fatal error: exception Failure("illegal binary operator bool + int in b + a")
int a;
bool b;

void foo(int c, bool d)
{
  int d;
  bool e;
  b + a; /* Error: bool + int */
}

int main()
{
  return 0;
}
--- Test Case
Fatal error: exception Failure("illegal binary operator float + int in b + a")
int a;
float b;

void foo(int c, float d)
{
  int d;
  float e;
  b + a; /* Error: float + int */
}
```

```
1392.
1393.       int main()
1394.       {
1395.         return 0;
1396.       }
1397.       --- Test Case
1398.       Fatal error: exception Failure("illegal binary operator float && int in -3.5 &&
    1")
1399.       int main()
1400.       {
1401.         -3.5 && 1; /* Float with AND? */
1402.         return 0;
1403.       }
1404.       --- Test Case
1405.       Fatal error: exception Failure("illegal binary operator float && float in -3.5
    && 2.5")
1406.       int main()
1407.       {
1408.         -3.5 && 2.5; /* Float with AND? */
1409.         return 0;
1410.       }
1411.       --- Test Case
1412.       Fatal error: exception Failure("undeclared identifier j")
1413.       int main()
1414.       {
1415.         int i;
1416.         for ( ; true ; ) {} /* OK: Forever */
1417.
1418.         for (i = 0 ; i < 10 ; i = i + 1) {
1419.           if (i == 3) return 42;
1420.         }
1421.
1422.         for (j = 0; i < 10 ; i = i + 1) {} /* j undefined */
1423.
1424.         return 0;
1425.       }
1426.       --- Test Case
1427.       Fatal error: exception Failure("undeclared identifier j")
1428.       int main()
1429.       {
1430.         int i;
1431.
1432.         for (i = 0; j < 10 ; i = i + 1) {} /* j undefined */
1433.
1434.         return 0;
1435.       }
1436.       --- Test Case
1437.       Fatal error: exception Failure("expected Boolean expression in i")
1438.       int main()
1439.       {
1440.         int i;
1441.
```

```
1442.         for (i = 0; i ; i = i + 1) {} /* i is an integer, not Boolean */
1443.
1444.         return 0;
1445.       }
1446.       --- Test Case
1447.       Fatal error: exception Failure("undeclared identifier j")
1448.       int main()
1449.       {
1450.         int i;
1451.
1452.         for (i = 0; i < 10 ; i = j + 1) {} /* j undefined */
1453.
1454.         return 0;
1455.       }
1456.       --- Test Case
1457.       Fatal error: exception Failure("unrecognized function foo")
1458.       int main()
1459.       {
1460.         int i;
1461.
1462.         for (i = 0; i < 10 ; i = i + 1) {
1463.           foo(); /* Error: no function foo */
1464.         }
1465.
1466.         return 0;
1467.       }
1468.       --- Test Case
1469.       Fatal error: exception Failure("duplicate function bar")
1470.       int foo() {}
1471.
1472.       int bar() {}
1473.
1474.       int baz() {}
1475.
1476.       void bar() {} /* Error: duplicate function bar */
1477.
1478.       int main()
1479.       {
1480.         return 0;
1481.       }
1482.       --- Test Case
1483.       Fatal error: exception Failure("duplicate formal a")
1484.       int foo(int a, bool b, int c) { }
1485.
1486.       void bar(int a, bool b, int a) {} /* Error: duplicate formal a in bar */
1487.
1488.       int main()
1489.       {
1490.         return 0;
1491.       }
1492.       --- Test Case
```

```
Fatal error: exception Failure("illegal void formal b")
int foo(int a, bool b, int c) { }

void bar(int a, void b, int c) {} /* Error: illegal void formal b */

int main()
{
  return 0;
}
--- Test Case
Fatal error: exception Failure("function printi may not be defined")
int foo() {}

void bar() {}

int printi() {} /* Should not be able to define print */

void baz() {}

int main()
{
  return 0;
}
--- Test Case
Fatal error: exception Failure("illegal void local b")
int foo() {}

int bar() {
  int a;
  void b; /* Error: illegal void local b */
  bool c;

  return 0;
}

int main()
{
  return 0;
}
--- Test Case
Fatal error: exception Failure("expecting 2 arguments in foo(42)")
void foo(int a, bool b)
{
}

int main()
{
  foo(42, true);
  foo(42); /* Wrong number of arguments */
}
--- Test Case
```

```
1544.       Fatal error: exception Failure("expecting 2 arguments in foo(42, true, false)")
1545.       void foo(int a, bool b)
1546.       {
1547.       }
1548.
1549.       int main()
1550.       {
1551.         foo(42, true);
1552.         foo(42, true, false); /* Wrong number of arguments */
1553.       }
1554.       --- Test Case
1555.       Fatal error: exception Failure("illegal argument found void expected bool in
    bar()")
1556.       void foo(int a, bool b)
1557.       {
1558.       }
1559.
1560.       void bar()
1561.       {
1562.       }
1563.
1564.       int main()
1565.       {
1566.         foo(42, true);
1567.         foo(42, bar()); /* int and void, not int and bool */
1568.       }
1569.       --- Test Case
1570.       Fatal error: exception Failure("illegal argument found int expected bool in 42")
1571.       void foo(int a, bool b)
1572.       {
1573.       }
1574.
1575.       int main()
1576.       {
1577.         foo(42, true);
1578.         foo(42, 42); /* Fail: int, not bool */
1579.       }
1580.       --- Test Case
1581.       Fatal error: exception Failure("illegal void global a")
1582.       int c;
1583.       bool b;
1584.       void a; /* global variables should not be void */
1585.
1586.
1587.       int main()
1588.       {
1589.         return 0;
1590.       }
1591.       --- Test Case
1592.       Fatal error: exception Failure("duplicate global b")
1593.       int b;
1594.       bool c;
```

```
int a;
int b; /* Duplicate global variable */

int main()
{
  return 0;
}
--- Test Case
Fatal error: exception Failure("expected Boolean expression in 42")
int main()
{
  if (true) {}
  if (false) {} else {}
  if (42) {} /* Error: non-bool predicate */
}
--- Test Case
Fatal error: exception Failure("undeclared identifier foo")
int main()
{
  if (true) {
    foo; /* Error: undeclared variable */
  }
}
--- Test Case
Fatal error: exception Failure("undeclared identifier bar")
int main()
{
  if (true) {
    42;
  } else {
    bar; /* Error: undeclared variable */
  }
}
--- Test Case
Fatal error: exception Failure("unrecognized function main")
Fatal error: exception Failure("return gives bool expected int in true")
int main()
{
  return true; /* Should return int */
}
--- Test Case
Fatal error: exception Failure("return gives int expected void in 42")
void foo()
{
  if (true) return 42; /* Should return void */
  else return;
}

int main()
{
  return 42;
}
--- Test Case
```

```
      Fatal error: exception Failure("expected Boolean expression in 42")
      int main()
      {
        int i;

        while (true) {
          i = i + 1;
        }

        while (42) { /* Should be boolean */
          i = i + 1;
        }

      }
      --- Test Case
      Fatal error: exception Failure("unrecognized function foo")
      int main()
      {
        int i;

        while (true) {
          i = i + 1;
        }

        while (true) {
          foo(); /* foo undefined */
        }

      }

```

VII. Lessons Learned

Although the final product maintains parity with the proposal pretty well, there were a few things that were not implemented due to the complexity of the problem being an unknown at the time. It was much easier to propose having unlimited scope blocks that can define arbitrary variables at any point in the code, than to come up with a strategy to implement it in a timely manner. One lesson learned is that limiting scope more aggressively to try to account for unknowns is probably a good idea.

Also, I learned much about low-level instructions, specifically with memory management, and realized how verbose the details are. To implement something that looks as innocuous as a list get, a[i], is 20+ lines of code to generate the LLVM for it. This process has taught me how to be more meticulous especially when considering multiple levels of indirection with pointers.

VIII. Appendix

1. Source Code

Makefile

```
# "make test" Compiles everything and runs the regression tests

.PHONY : test
test : all testall.sh
    ./testall.sh

# "make all" builds the executable as well as the "printbig" library designed
# to test linking external code

.PHONY : all
all : ap_plusplus.native

# "make ap_plusplus.native" compiles the compiler
#
# The _tags file controls the operation of ocamlbuild, e.g., by including
# packages, enabling warnings
#
# See https://github.com/ocaml/ocamlbuild/blob/master/manual/manual.adoc

ap_plusplus.native :
    opam config exec -- \
    ocamlbuild -use-ocamlfind ap_plusplus.native

# "make clean" removes all generated files

.PHONY : clean
clean :
    ocamlbuild -clean
    rm -rf testall.log ocamlllvm *.diff

printbig : printbig.c
    cc -o printbig -DBUILD_TEST printbig.c

# Building the tarball

TESTS = \
  helloworld

TESTFILES = $(TESTS:%=test-%.mc) $(TESTS:%=test-%.out)

TARFILES = scanner.mll parser.mly ast.ml sast.ml semant.ml codegen.ml Makefile
ap_plusplus.ml testall.sh \
    _tags \
```

```
    printbig.c \
    $(TESTFILES:%=tests/%)

ap++.tar.gz : $(TARFILES)
    cd .. && tar czf ap++/ap++.tar.gz \
        $(TARFILES:%=ap++/%)
```

testall.sh

```
#!/bin/sh

# Regression testing script for `
# Step through a list of files
#  Compile, run, and check the output of each expected-to-work test
#  Compile and check the error of each expected-to-fail test

# Path to the LLVM interpreter
#LLI="lli"
LLI="/usr/local/opt/llvm/bin/lli"

# Path to the LLVM compiler
LLC="/usr/local/opt/llvm/bin/llc"

# Path to the C compiler
CC="cc"

# Path to the microc compiler.  Usually "./microc.native"
# Try "_build/microc.native" if ocamlbuild was unable to create a symbolic link.
AP_PLUSPLUS="./ap_plusplus.native"
#MICROC="_build/microc.native"

# Set time limit for all operations
ulimit -t 30

globallog=testall.log
rm -f $globallog
error=0
globalerror=0

keep=0

Usage() {
    echo "Usage: testall.sh [options] [.mc files]"
    echo "-k    Keep intermediate files"
    echo "-h    Print this help"
    exit 1
}

SignalError() {
    if [ $error -eq 0 ] ; then
    echo "FAILED"
    error=1
    fi
    echo "  $1"
}

# Compare <outfile> <reffile> <difffile>
```

```
# Compares the outfile with reffile.  Differences, if any, written to difffile
Compare() {
    generatedfiles="$generatedfiles $3"
    echo diff -b $1 $2 ">" $3 1>&2
    diff -b "$1" "$2" > "$3" 2>&1 || {
    SignalError "$1 differs"
    echo "FAILED $1 differs from $2" 1>&2
    }
}

# Run <args>
# Report the command, run it, and report any errors
Run() {
    echo $* 1>&2
    eval $* || {
    SignalError "$1 failed on $*"
    return 1
    }
}

# RunFail <args>
# Report the command, run it, and expect an error
RunFail() {
    echo $* 1>&2
    eval $* && {
    SignalError "failed: $* did not report an error"
    return 1
    }
    return 0
}

Check() {
    error=0
    basename=`echo $1 | sed 's/.*\\///
                            s/.mc//'`
    reffile=`echo $1 | sed 's/.mc$//'`
    basedir="`echo $1 | sed 's/\/[^\/]*$//'`/."

    echo -n "$basename..."

    echo 1>&2
    echo "###### Testing $basename" 1>&2

    generatedfiles=""

    generatedfiles="$generatedfiles ${basename}.ll ${basename}.s ${basename}.exe
${basename}.out" &&
    Run "$AP_PLUSPLUS" "$1" ">" "${basename}.ll" &&
    Run "$LLC" "-relocation-model=pic" "${basename}.ll" ">" "${basename}.s" &&
    Run "$CC" "-o" "${basename}.exe" "${basename}.s" "printbig.o" &&
    Run "./${basename}.exe" > "${basename}.out" &&
    Compare ${basename}.out ${reffile}.out ${basename}.diff
```

```
100.
101.     # Report the status and clean up the generated files
102.
103.     if [ $error -eq 0 ] ; then
104.     if [ $keep -eq 0 ] ; then
105.         rm -f $generatedfiles
106.     fi
107.     echo "OK"
108.     echo "###### SUCCESS" 1>&2
109.     else
110.     echo "###### FAILED" 1>&2
111.     globalerror=$error
112.     fi
113. }
114.
115. CheckFail() {
116.     error=0
117.     basename=`echo $1 | sed 's/.*\\///
118.                               s/.mc//'`
119.     reffile=`echo $1 | sed 's/.mc$//'`
120.     basedir="`echo $1 | sed 's/\/[^\/]*$//'`/."
121.
122.     echo -n "$basename..."
123.
124.     echo 1>&2
125.     echo "###### Testing $basename" 1>&2
126.
127.     generatedfiles=""
128.
129.     generatedfiles="$generatedfiles ${basename}.err ${basename}.diff" &&
130.     RunFail "$AP_PLUSPLUS" "<" $1 "2>" "${basename}.err" ">>" $globallog &&
131.     Compare ${basename}.err ${reffile}.err ${basename}.diff
132.
133.     # Report the status and clean up the generated files
134.
135.     if [ $error -eq 0 ] ; then
136.     if [ $keep -eq 0 ] ; then
137.         rm -f $generatedfiles
138.     fi
139.     echo "OK"
140.     echo "###### SUCCESS" 1>&2
141.     else
142.     echo "###### FAILED" 1>&2
143.     globalerror=$error
144.     fi
145. }
146.
147. while getopts kdpsh c; do
148.     case $c in
149.     k) # Keep intermediate files
150.         keep=1
```

```
        ;;
    h) # Help
        Usage
        ;;
    esac
done

shift `expr $OPTIND - 1`

LLIFail() {
  echo "Could not find the LLVM interpreter \"$LLI\"."
  echo "Check your LLVM installation and/or modify the LLI variable in testall.sh"
  exit 1
}

which "$LLI" >> $globallog || LLIFail

if [ ! -f printbig.o ]
then
    echo "Could not find printbig.o"
    echo "Try \"make printbig.o\""
    exit 1
fi

if [ $# -ge 1 ]
then
    files=$@
else
    files="tests/test-*.mc tests/fail-*.mc"
fi

for file in $files
do
    case $file in
    *test-*)
        Check $file 2>> $globallog
        ;;
    *fail-*)
        CheckFail $file 2>> $globallog
        ;;
    *)
        echo "unknown file type $file"
        globalerror=1
        ;;
    esac
done

exit $globalerror
```

Top-Level ap_plusplus.ml

```
1.  (* Top-level of the AP++ compiler: scan & parse the input,
2.     check the resulting AST and generate an SAST from it, generate LLVM IR,
3.     and dump the module *)
4.
5.  type action = Ast | Sast | LLVM_IR | Compile
6.
7.  let () =
8.    let action = ref Compile in
9.    let set_action a () = action := a in
10.   let speclist = [
11.     ("-a", Arg.Unit (set_action Ast), "Print the AST");
12.     ("-s", Arg.Unit (set_action Sast), "Print the SAST");
13.     ("-l", Arg.Unit (set_action LLVM_IR), "Print the generated LLVM IR");
14.     ("-c", Arg.Unit (set_action Compile),
15.       "Check and print the generated LLVM IR (default)");
16.   ] in
17.   let usage_msg = "usage: ./ap++.native [-a|-s|-l|-c] [file.mc]" in
18.   let channel = ref stdin in
19.   Arg.parse speclist (fun filename -> channel := open_in filename) usage_msg;
20.
21.   let lexbuf = Lexing.from_channel !channel in
22.   let ast = Parser.program Scanner.token lexbuf in
23.   match !action with
24.     Ast -> print_string (Ast.string_of_program ast)
25.   | _ -> let sast = Semant.check ast in
26.     match !action with
27.       Ast     -> ()
28.     | Sast    -> print_string (Sast.string_of_sprogram sast)
29.     | LLVM_IR -> print_string (Llvm.string_of_llmodule (Codegen.translate sast))
30.     | Compile -> let m = Codegen.translate sast in
31.     Llvm_analysis.assert_valid_module m;
32.     print_string (Llvm.string_of_llmodule m)
```

scanner.mll

```
(* Scanner for AP++ compiler *)

{ open Parser }

let digit = ['0' - '9']
let digits = digit+
let esc_regex = '\\' ['\\' ''' '"' 'n' 'r' 't']
let ascii_regex = ([' '-'!' '#'-'[' ']'-'~'])

rule token = parse
 [' ' '\t' '\r' '\n'] { token lexbuf } (* Whitespace *)
| "/*"    { comment lexbuf }           (* Multi-line Comment *)
| "//" { line_comment lexbuf }         (* Single-line Comment *)
| '('     { LPAREN }
| ')'     { RPAREN }
| '{'     { LBRACE }
| '}'     { RBRACE }
| '['     { LBRACK }
| ']'     { RBRACK }
| ':'     { COLON }
| ';'     { SEMICOLON }
| ','     { COMMA }
| "++"    { PLUSPLUS }
| "--"    { MINUSMINUS }
| '+'     { PLUS }
| '-'     { MINUS }
| '*'     { TIMES }
| '/'     { DIVIDE }
| '%'     { MOD }
| '='     { ASSIGN }
| "=="    { EQ }
| "!="    { NEQ }
| "&&"    { AND }
| "||"    { OR }
| '!'     { NOT }
| "<="    { LEQ }
| ">="    { GEQ }
| "if"    { IF }
| "else"  { ELSE }
| "while" { WHILE }
| "for"   { FOR }
| "return" { RETURN }
| "int"   { INT }
| "bool"  { BOOL }
| "float" { FLOAT }
| "void"  { VOID }
| "<"     { LT }
| ">"     { GT }
| "#"     { HASH }
| "list_push"   { LIST_PUSH }
```

```
| "list_get"    { LIST_GET }
| "list_set"    { LIST_SET }
| "list_pop"    { LIST_POP }
| "list_size"   { LIST_SIZE }
| "list_slice"  { LIST_SLICE }
| "list_clear"  { LIST_CLEAR }
| "list_rev"    { LIST_REVERSE }
| "list_insert" { LIST_INSERT }
| "list_remove" { LIST_REMOVE }
| "list_find"   { LIST_FIND }
| "list"        { LIST }
| digits as lit { ILITERAL(int_of_string lit) }
| "true"    { BLITERAL(true) }
| "false"   { BLITERAL(false) }
| digits '.'  digit* ( ['e' 'E'] ['+' '-']? digits )? as lit { FLITERAL(float_of_string
  lit) }
| '"' ((ascii_regex | esc_regex)* as lit)'"' { SLITERAL(lit) }
| ['a'-'z' 'A'-'Z']['a'-'z' 'A'-'Z' '0'-'9' '_']* as id_name { ID(id_name) }
| eof     { EOF }
| _ as char { raise (Failure("illegal character " ^ Char.escaped char)) }

and line_comment = parse
'\n' { token lexbuf }
| _ { line_comment lexbuf }

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

parser.mly

```
/* Ocamlyacc Parser for AP++ compiler */

%{
open Ast
%}

%token COLON COMMA SEMICOLON LPAREN RPAREN LBRACE RBRACE LBRACK RBRACK
%token PLUS MINUS TIMES DIVIDE ASSIGN MOD
%token PLUSPLUS MINUSMINUS
%token NOT AND OR
%token EQ NEQ LT LEQ GT GEQ
%token RETURN IF ELSE WHILE FOR INT BOOL FLOAT STRING VOID
%token LIST_PUSH LIST_GET LIST_SET LIST_POP LIST_SIZE HASH LIST_SLICE LIST_CLEAR
LIST_REVERSE LIST_INSERT LIST_REMOVE LIST_FIND
%token <int> ILITERAL
%token <bool> BLITERAL
%token <string> SLITERAL
%token <float> FLITERAL
%token <string> ID
%token LIST
%token EOF

%start program
%type <Ast.program> program

%nonassoc NOELSE
%nonassoc ELSEIF
%nonassoc ELSE
%right ASSIGN
%left OR
%left AND
%left EQ NEQ
%left LT GT LEQ GEQ
%left PLUS MINUS
%left TIMES DIVIDE MOD
%right NOT PLUSPLUS MINUSMINUS

%%

program:
  decls EOF { $1 }

decls:
  /* nothing */   { ([], [])}
| decls var_decl  { (($2 :: fst $1), snd $1) }
| decls func_decl { (fst $1, ($2 :: snd $1)) }

/* e.g. int foo(int x, int y) {} */
func_decl:
```

```
  typ ID LPAREN func_formals_opt RPAREN LBRACE var_decl_list stmt_list RBRACE
  { { typ     = $1;
      fname   = $2;
      formals = List.rev $4;
      locals  = List.rev $7;
      body    = List.rev $8 } }

func_formals_opt:
   /* nothing */    { [] }
|  func_formals_list { $1 }

/* int x, bool y */
func_formals_list:
  typ ID                          { [($1, $2)] }
| func_formals_list COMMA typ ID { ($3, $4) :: $1 }

typ:
  INT             { Int }
| BOOL            { Bool }
| FLOAT           { Float }
| STRING          { String }
| VOID            { Void }
| LIST LT typ GT  { List($3) }

/* e.g. int x; int y; */
var_decl_list:
  /* nothing */ { [] }
|  var_decl_list var_decl { $2 :: $1 }

/* e.g. int x; */
var_decl:
  typ ID SEMICOLON { ($1, $2) }

stmt_list:
  /* nothing */  { [] }
| stmt_list stmt { $2 :: $1 }

/* executes code logic but do not evaluate to any value */
stmt:
  expr SEMICOLON                              { Expr $1 }
| RETURN expr_opt SEMICOLON                   { Return $2 }
| LBRACE stmt_list RBRACE                     { Block(List.rev $2) }
| IF LPAREN expr RPAREN stmt %prec NOELSE      { If($3, $5, Block([])) }
| IF LPAREN expr RPAREN stmt ELSE stmt        { If($3, $5, $7) }
| FOR LPAREN expr_opt SEMICOLON expr SEMICOLON expr_opt RPAREN stmt
                                              { For($3, $5, $7, $9) }
| WHILE LPAREN expr RPAREN stmt                { While($3, $5) }
| LIST_PUSH LPAREN ID COMMA expr RPAREN SEMICOLON { ListPush($3, $5) }
| LIST_SET LPAREN ID COMMA expr COMMA expr RPAREN SEMICOLON { ListSet($3, $5, $7) }
| ID LBRACK expr RBRACK ASSIGN expr SEMICOLON         { ListSet($1, $3, $6) }
| LIST_CLEAR LPAREN ID RPAREN SEMICOLON       { ListClear($3) }
| LIST_REMOVE LPAREN ID COMMA expr RPAREN SEMICOLON { ListRemove($3, $5) }
```

```
| LIST_INSERT LPAREN ID COMMA expr COMMA expr RPAREN SEMICOLON { ListInsert($3, $5, $7) }
| LIST_REVERSE LPAREN ID RPAREN SEMICOLON       { ListReverse($3) }

expr_opt:
  /* nothing */ { Noexpr }
| expr          { $1 }

/* executes code logic and evaluates to a value */
expr:
  ILITERAL                                { ILiteral($1) }
| BLITERAL                                { BLiteral($1) }
| SLITERAL                                { SLiteral($1) }
| FLITERAL                                { FLiteral($1) }
| ID                                      { Id($1) }
| expr PLUS   expr                        { Binop($1, Add, $3) }
| expr MINUS  expr                        { Binop($1, Sub, $3) }
| expr TIMES  expr                        { Binop($1, Mult, $3) }
| expr DIVIDE expr                        { Binop($1, Div, $3) }
| expr MOD expr                           { Binop($1, Mod, $3) }
| PLUSPLUS ID                             { Unop(PlusPlusPre, Id($2)) }
| MINUSMINUS ID                           { Unop(MinusMinusPre, Id($2)) }
| ID PLUSPLUS                             { Unop(PlusPlusPost, Id($1)) }
| ID MINUSMINUS                           { Unop(MinusMinusPost, Id($1)) }
| expr EQ expr                            { Binop($1, Equal, $3) }
| expr NEQ expr                           { Binop($1, Neq, $3) }
| expr LT expr                            { Binop($1, Less, $3) }
| expr GT expr                            { Binop($1, Greater, $3) }
| expr LEQ expr                           { Binop($1, Leq, $3) }
| expr GEQ expr                           { Binop($1, Geq, $3) }
| MINUS expr %prec NOT                    { Unop(Neg, $2)      }
| NOT expr                                { Unop(Not, $2) }
| expr AND expr                           { Binop($1, And, $3) }
| expr OR expr                            { Binop($1, Or, $3) }
| LPAREN expr RPAREN                      { $2 }
| ID ASSIGN expr                          { Assign($1, $3) }
| ID LPAREN args_opt RPAREN                 { Call($1, $3) }
| LIST_GET LPAREN ID COMMA expr RPAREN    { ListGet($3, $5) }
| ID LBRACK expr RBRACK                   { ListGet($1, $3) }
| LIST_POP LPAREN ID RPAREN               { ListPop($3) }
| LIST_SIZE LPAREN ID RPAREN              { ListSize($3) }
| HASH ID                                 { ListSize($2) }
| LIST_SLICE LPAREN ID COMMA expr COMMA expr RPAREN { ListSlice($3, $5, $7) }
| ID LBRACK expr_opt COLON expr_opt RBRACK        { ListSlice($1, $3, $5) }
| LIST_FIND LPAREN ID COMMA expr RPAREN   { ListFind($3, $5) }
| LBRACK args_opt RBRACK                  { ListLiteral($2) }

args_opt:
    /* nothing */ { [] }
  | args_list  { List.rev $1 }

/* args used for function calls */
```

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

ast.ml

```
(* Abstract Syntax Tree (AST) Definitions for AP++ compiler *)

type op = Add | Sub | Mult | Div | Mod | Equal | Neq | Less | Leq | Greater | Geq
          | And | Or

type uop = PlusPlusPre | PlusPlusPost | MinusMinusPre | MinusMinusPost | Neg | Not

type typ = Int | Bool | Float | String | Void | List of typ

type bind = typ * string

type expr =
    ILiteral of int
  | BLiteral of bool
  | SLiteral of string
  | FLiteral of float
  | Id of string
  | Binop of expr * op * expr
  | Unop of uop * expr
  | Assign of string * expr
  | Call of string * expr list
  | ListGet of string * expr
  | ListPop of string
  | ListSize of string
  | ListSlice of string * expr * expr
  | ListFind of string * expr
  | ListLiteral of expr list
  | Noexpr

type stmt =
  Block of stmt list
| Expr of expr
| Return of expr
| If of expr * stmt * stmt
| For of expr * expr * expr * stmt
| While of expr * stmt
| ListPush of string * expr
| ListSet of string * expr * expr
| ListClear of string
| ListRemove of string * expr
| ListInsert of string * expr * expr
| ListReverse of string

type func_decl = {
    typ : typ;
    fname : string;
    formals : bind list;
    locals : bind list;
    body : stmt list;
```

```
  }

type program = bind list * func_decl list

(* Pretty-printing functions *)

let string_of_op = function
    Add -> "+"
  | Sub -> "-"
  | Mult -> "*"
  | Div -> "/"
  | Equal -> "=="
  | Neq -> "!="
  | Less -> "<"
  | Leq -> "<="
  | Greater -> ">"
  | Geq -> ">="
  | And -> "&&"
  | Or -> "||"
  | _ -> "?"

let string_of_uop = function
    Neg -> "-"
  | Not -> "!"
  | PlusPlusPre -> "++x"
  | PlusPlusPost -> "x++"
  | MinusMinusPre -> "--x"
  | MinusMinusPost -> "x++"

let rec string_of_expr = function
    ILiteral(l) -> string_of_int l
  | FLiteral(l) -> string_of_float l
  | BLiteral(true) -> "true"
  | BLiteral(false) -> "false"
  | SLiteral(l) -> l
  | ListGet(id, e) -> "list_get " ^ id ^ ", " ^ (string_of_expr e)
  | ListPop (id) -> "list_pop " ^ id
  | ListSize(id) -> "list_size " ^ id
  | ListSlice(id, e1, e2) -> "list_slice " ^ id ^ ", " ^ (string_of_expr e1) ^ ", " ^
(string_of_expr e2)
  | ListFind(id, e) -> "list_find " ^ id ^ ", " ^ (string_of_expr e)
  | ListLiteral(_) -> "list_literal"
  | Id(s) -> s
  | Binop(e1, o, e2) ->
      string_of_expr e1 ^ " " ^ string_of_op o ^ " " ^ string_of_expr e2
  | Unop(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) ^ ")"
  | Noexpr -> ""

let rec string_of_stmt = function
```

```
101.     Block(stmts) ->
102.       "{\n" ^ String.concat "" (List.map string_of_stmt stmts) ^ "}\n"
103.   | Expr(expr) -> string_of_expr expr ^ ";\n";
104.   | Return(expr) -> "return " ^ string_of_expr expr ^ ";\n";
105.   | If(e, s, Block([])) -> "if (" ^ string_of_expr e ^ ")\n" ^ string_of_stmt s
106.   | If(e, s1, s2) ->  "if (" ^ string_of_expr e ^ ")\n" ^
107.       string_of_stmt s1 ^ "else\n" ^ string_of_stmt s2
108.   | For(e1, e2, e3, s) ->
109.       "for (" ^ string_of_expr e1  ^ " ; " ^ string_of_expr e2 ^ " ; " ^
110.       string_of_expr e3  ^ ") " ^ string_of_stmt s
111.   | While(e, s) -> "while (" ^ string_of_expr e ^ ") " ^ string_of_stmt s
112.   | ListPush(id, e) -> "list_push " ^ id ^ ", " ^ string_of_expr e
113.   | ListSet(id, e1, e2) -> "list_set " ^ id ^ ", " ^ (string_of_expr e1) ^ ", " ^
    (string_of_expr e2)
114.   | ListClear(id) -> "list_clear " ^ id
115.   | ListRemove(id, e) -> "list remove " ^ id ^ ", " ^ (string_of_expr e)
116.   | ListInsert(id, e1, e2) -> "list_insert " ^ id ^ ", " ^ (string_of_expr e1) ^ ", " ^
    (string_of_expr e2)
117.   | ListReverse(id) -> "list_rev " ^ id
118.
119. let rec string_of_typ = function
120.     Int -> "int"
121.   | Bool -> "bool"
122.   | Float -> "float"
123.   | Void -> "void"
124.   | String -> "string"
125.   | List x -> "list<" ^ (string_of_typ x) ^ ">"
126.
127. let string_of_vdecl (t, id) = string_of_typ t ^ " " ^ id ^ ";\n"
128.
129. let string_of_fdecl fdecl =
130.   string_of_typ fdecl.typ ^ " " ^
131.   fdecl.fname ^ "(" ^ String.concat ", " (List.map snd fdecl.formals) ^
132.   ")\n{\n" ^
133.   String.concat "" (List.map string_of_vdecl fdecl.locals) ^
134.   String.concat "" (List.map string_of_stmt fdecl.body) ^
135.   "}\n"
136.
137. let string_of_program (vars, funcs) =
138.   String.concat "" (List.map string_of_vdecl vars) ^ "\n" ^
139.   String.concat "\n" (List.map string_of_fdecl funcs)
```

sast.ml

```
(* Semantic Abstract Syntax Tree (SAST) Definitions for AP++ compiler *)

open Ast

type sexpr = typ * sx
and sx =
    SILiteral of int
  | SBLiteral of bool
  | SSLiteral of string
  | SFLiteral of float
  | SId of string
  | SBinop of sexpr * op * sexpr
  | SUnop of uop * sexpr
  | SAssign of string * sexpr
  | SCall of string * sexpr list
  | SListGet of typ * string * sexpr
  | SListPop of typ * string
  | SListSize of typ * string
  | SListSlice of typ * string * sexpr * sexpr
  | SListFind of typ * string * sexpr
  | SListLiteral of typ * sexpr list
  | SNoexpr

type sstmt =
  SBlock of sstmt list
| SExpr of sexpr
| SReturn of sexpr
| SIf of sexpr * sstmt * sstmt
| SFor of sexpr * sexpr * sexpr * sstmt
| SWhile of sexpr * sstmt
| SListPush of string * sexpr
| SListSet of typ * string * sexpr * sexpr
| SListClear of typ * string
| SListRemove of string * sexpr
| SListInsert of string * sexpr * sexpr
| SListReverse of typ * string

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

type sprogram = bind list * sfunc_decl list


(* Pretty-printing functions *)
```

```
50.
51. let rec string_of_sexpr (t, e) =
52.   "(" ^ string_of_typ t ^ " : " ^ (match e with
53.     SILiteral(l) -> string_of_int l
54.   | SFLiteral(l) -> string_of_float l
55.   | SBLiteral(true) -> "true"
56.   | SBLiteral(false) -> "false"
57.   | SSLiteral(l) -> l
58.   | SListGet(_, id, e) -> "list_get " ^ id ^ ", " ^ (string_of_sexpr e)
59.   | SListPop (_, id) -> "list_pop " ^ id
60.   | SListSize(_, id) -> "list_size " ^ id
61.   | SListSlice(_, id, e1, e2) -> "list_slice " ^ id ^ ", " ^ (string_of_sexpr e1) ^ ", "
    ^ (string_of_sexpr e2)
62.   | SListFind(_, id, e) -> "list_find " ^ id ^ ", " ^ (string_of_sexpr e)
63.   | SListLiteral(_) -> "list_literal"
64.   | SId(s) -> s
65.   | SBinop(e1, o, e2) ->
66.       string_of_sexpr e1 ^ " " ^ string_of_op o ^ " " ^ string_of_sexpr e2
67.   | SUnop(o, e) -> string_of_uop o ^ string_of_sexpr e
68.   | SAssign(v, e) -> v ^ " = " ^ string_of_sexpr e
69.   | SCall(f, el) ->
70.       f ^ "(" ^ String.concat ", " (List.map string_of_sexpr el) ^ ")"
71.   | SNoexpr -> ""
72.         ) ^ ")"
73.
74. let rec string_of_sstmt = function
75.     SBlock(stmts) ->
76.       "{\n" ^ String.concat "" (List.map string_of_sstmt stmts) ^ "}\n"
77.   | SExpr(expr) -> string_of_sexpr expr ^ ";\n";
78.   | SReturn(expr) -> "return " ^ string_of_sexpr expr ^ ";\n";
79.   | SIf(e, s, SBlock([])) ->
80.       "if (" ^ string_of_sexpr e ^ ")\n" ^ string_of_sstmt s
81.   | SIf(e, s1, s2) ->  "if (" ^ string_of_sexpr e ^ ")\n" ^
82.       string_of_sstmt s1 ^ "else\n" ^ string_of_sstmt s2
83.   | SFor(e1, e2, e3, s) ->
84.       "for (" ^ string_of_sexpr e1  ^ " ; " ^ string_of_sexpr e2 ^ " ; " ^
85.       string_of_sexpr e3  ^ ") " ^ string_of_sstmt s
86.   | SWhile(e, s) -> "while (" ^ string_of_sexpr e ^ ") " ^ string_of_sstmt s
87.   | SListPush(id, e) -> "list_push " ^ id ^ ", " ^ string_of_sexpr e
88.   | SListSet(_, id, e1, e2) -> "list_set " ^ id ^ ", " ^ (string_of_sexpr e1) ^ ", " ^
    (string_of_sexpr e2)
89.   | SListClear(_, id) -> "list_clear " ^ id
90.   | SListRemove(id, e) -> "list remove " ^ id ^ ", " ^ (string_of_sexpr e)
91.   | SListInsert(id, e1, e2) -> "list_insert " ^ id ^ ", " ^ (string_of_sexpr e1) ^ ", "
    ^ (string_of_sexpr e2)
92.   | SListReverse(_, id) -> "list_rev " ^ id
93.
94. let string_of_sfdecl fdecl =
95.   string_of_typ fdecl.styp ^ " " ^
96.   fdecl.sfname ^ "(" ^ String.concat ", " (List.map snd fdecl.sformals) ^
97.   ")\n{\n" ^
98.   String.concat "" (List.map string_of_vdecl fdecl.slocals) ^
```

```
  String.concat "" (List.map string_of_sstmt fdecl.sbody) ^
   "}\n"

let string_of_sprogram (vars, funcs) =
   String.concat "" (List.map string_of_vdecl vars) ^ "\n" ^
   String.concat "\n" (List.map string_of_sfdecl funcs)
```

semant.ml

```
(* Semantic checking for the AP++ 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) =

  (* Verify a list of bindings has no void types or duplicate names *)
  let check_binds (kind : string) (binds : bind list) =
    List.iter (function
  (Void, b) -> raise (Failure ("illegal void " ^ kind ^ " " ^ b))
      | _ -> ()) 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

  (**** Check global variables ****)

  check_binds "global" globals;

  (**** Check functions ****)

  (* Collect function declarations for built-in functions: no bodies *)
  let built_in_decls =
    let add_bind map (name, ty) = StringMap.add name {
      typ = Void;
      fname = name;
      formals = [(ty, "x")];
      locals = []; body = [] } map
    in List.fold_left add_bind StringMap.empty [ ("printi", Int);
                                                 ("printf", Float);
                                                 ("prints", String);
                                                 ("printb", Bool) ]
  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 *)

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

    (* 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 lvaluet else raise (Failure err)
    in

    (* 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 (globals @ func.formals @ func.locals )
    in

    (* Return a variable from our local symbol table *)
    let type_of_identifier s =
      try StringMap.find s symbols
      with Not_found -> raise (Failure ("undeclared identifier " ^ s))
    in
    (* Check id is a list and return list type *)
    let check_list_type id =
       match (type_of_identifier id) with
          List t -> t
        | t -> raise (Failure ("check list type error, typ: " ^ string_of_typ t))
    in
    (* Return a semantically-checked expression, i.e., with a type *)
    let rec expr = function
         ILiteral l -> (Int, SILiteral l)
```

```
101.       | BLiteral l  -> (Bool, SBLiteral l)
102.       | FLiteral l -> (Float, SFLiteral l)
103.       | SLiteral l -> (String, SSLiteral l)
104.       | Noexpr     -> (Void, SNoexpr)
105.       | Id s       -> (type_of_identifier s, SId s)
106.       | Assign(var, e) as ex ->
107.           let lt = type_of_identifier var
108.           and (rt, e') = expr e in
109.           let err = "illegal assignment " ^ string_of_typ lt ^ " = " ^
110.             string_of_typ rt ^ " in " ^ string_of_expr ex
111.           in (check_assign lt rt err, SAssign(var, (rt, e')))
112.       | Unop(op, e) as ex ->
113.           let (t, e') = expr e in
114.           let ty = match op with
115.             Neg when t = Int || t = Float -> t
116.           | Not when t = Bool -> Bool
117.           | PlusPlusPre when t = Int -> t
118.           | MinusMinusPre when t = Int -> t
119.           | PlusPlusPost when t = Int -> t
120.           | MinusMinusPost when t = Int -> t
121.           | _ -> raise (Failure ("illegal unary operator " ^
122.                                  string_of_uop op ^ string_of_typ t ^
123.                                  " in " ^ string_of_expr ex))
124.           in (ty, SUnop(op, (t, e')))
125.       | Binop(e1, op, e2) as e ->
126.           let (t1, e1') = expr e1
127.           and (t2, e2') = expr e2 in
128.           (* All binary operators require operands of the same type *)
129.           let same = t1 = t2 in
130.           (* Determine expression type based on operator and operand types *)
131.           let ty = match op with
132.             Add | Sub | Mult | Div | Mod when same && t1 = Int   -> Int
133.           | Add | Sub | Mult | Div when same && t1 = Float -> Float
134.           | Equal | Neq            when same               -> Bool
135.           | Less | Leq | Greater | Geq
136.                      when same && (t1 = Int || t1 = Float) -> Bool
137.           | And | Or when same && t1 = Bool -> Bool
138.           | _ -> raise (
139.                  Failure ("illegal binary operator " ^
140.                      string_of_typ t1 ^ " " ^ string_of_op op ^ " " ^
141.                      string_of_typ t2 ^ " in " ^ string_of_expr e))
142.           in (ty, SBinop((t1, e1'), op, (t2, e2')))
143.       | ListGet (var, e) ->
144.          let (t, e') = expr e in
145.          let ty = match t with
146.              Int -> Int
147.              | _ -> raise (Failure ("list_get index must be integer, not " ^
    string_of_typ t))
148.          in let list_type = check_list_type var
149.          in (list_type, SListGet(list_type, var, (ty, e')))
150.       | ListPop var ->
151.          let list_type = check_list_type var
152.          in (list_type, SListPop(list_type, var))
```

```
153.       | ListSize var ->
154.           (Int, SListSize(check_list_type var, var))
155.       | ListSlice (var, e1, e2) ->
156.           let e1' = expr e1
157.           and e2' = expr e2 in
158.           let _ = match (fst e1', fst e2') with
159.               (Int, Int)
160.             | (Void, Int)
161.             | (Int, Void)
162.             | (Void, Void) -> Int
163.             | _ -> raise (Failure ("invalid list index arguments for list slice: " ^
    string_of_sexpr e1' ^ ", " ^ string_of_sexpr e2'))
164.           in (type_of_identifier var, SListSlice(check_list_type var, var, e1', e2'))
165.       | ListFind (var, e) ->
166.         let (t, e') = expr e in
167.         (Int, SListFind(check_list_type var, var, (t, e')))
168.       | ListLiteral vals ->
169.         let (t', _) = expr (List.hd vals) in
170.         let map_func lit = expr lit in
171.         let vals' = List.map map_func vals in
172.         (* TODO: check that all vals are of the same type *)
173.         (List t', SListLiteral(t', vals'))
174.       | Call(fname, args) as call ->
175.           let fd = find_func fname in
176.           let param_length = List.length fd.formals in
177.           if List.length args != param_length then
178.             raise (Failure ("expecting " ^ string_of_int param_length ^
179.                             " arguments in " ^ string_of_expr call))
180.           else let check_call (ft, _) e =
181.             let (et, e') = expr e in
182.             let err = "illegal argument found " ^ string_of_typ et ^
183.               " expected " ^ string_of_typ ft ^ " in " ^ string_of_expr e
184.             in (check_assign ft et err, e')
185.           in
186.           let args' = List.map2 check_call fd.formals args
187.           in (fd.typ, SCall(fname, args'))
188.     in
189.
190.     let check_bool_expr e =
191.       let (t', e') = expr e
192.       and err = "expected Boolean expression in " ^ string_of_expr e
193.       in if t' != Bool then raise (Failure err) else (t', e')
194.
195.     in
196.     let check_int_expr e =
197.       let (t', e') = expr e
198.       and err = "expected Integer expression in " ^ string_of_expr e
199.       in if t' != Int then raise (Failure err) else (t', e')
200.     in
201.     let check_match_list_type_expr l e =
202.      let (t', e') as e'' = expr e
203.       in let err = "list type and expression type do not match " ^ (string_of_typ t') ^
    ", " ^ (string_of_sexpr e'')
```

```
204.       in if t' != (check_list_type l) then raise (Failure err) else (t', e')
205.     in
206.     (* Return a semantically-checked statement i.e. containing sexprs *)
207.     let rec check_stmt = function
208.         Expr e -> SExpr (expr e)
209.       | ListPush (var, e) ->
210.         let _ = check_list_type var in
211.         SListPush(var, check_match_list_type_expr var e)
212.       | ListSet (var, e1, e2) ->
213.           SListSet(check_list_type var, var, check_int_expr e1,
    check_match_list_type_expr var e2)
214.       | ListClear var ->
215.           SListClear(check_list_type var, var)
216.       | ListRemove (var, e) ->
217.          let _ = check_list_type var in
218.          SListRemove(var, check_match_list_type_expr var e)
219.       | ListInsert (var, e1, e2) ->
220.          let _ = check_list_type var in
221.          SListInsert(var, check_int_expr e1, check_match_list_type_expr var e2)
222.       | ListReverse var ->
223.          SListReverse(check_list_type var, var)
224.       | If(p, b1, b2) -> SIf(check_bool_expr p, check_stmt b1, check_stmt b2)
225.       | For(e1, e2, e3, st) -> SFor(expr e1, check_bool_expr e2, expr e3, check_stmt
    st)
226.       | While(p, s) -> SWhile(check_bool_expr p, check_stmt s)
227.       | Return e -> let (t, e') = expr e in
228.         if t = func.typ then SReturn (t, e')
229.         else raise (
230.           Failure ("return gives " ^ string_of_typ t ^ " expected " ^
231.              string_of_typ func.typ ^ " in " ^ string_of_expr e))
232.
233.         (* A block is correct if each statement is correct and nothing
234.            follows any Return statement.  Nested blocks are flattened. *)
235.       | Block sl ->
236.           let rec check_stmt_list = function
237.               [Return _ as s] -> [check_stmt s]
238.             | Return _ :: _   -> raise (Failure "nothing may follow a return")
239.             | Block sl :: ss  -> check_stmt_list (sl @ ss) (* Flatten blocks *)
240.             | s :: ss         -> check_stmt s :: check_stmt_list ss
241.             | []              -> []
242.           in SBlock(check_stmt_list sl)
243.
244.     in (* body of check_function *)
245.     { styp = func.typ;
246.       sfname = func.fname;
247.       sformals = func.formals;
248.       slocals  = func.locals;
249.       sbody = match check_stmt (Block func.body) with
250.     SBlock(sl) -> sl
251.       | _ -> raise (Failure ("internal error: block didn't become a block?"))
252.     }
253.   in (globals, List.map check_function functions)
```

codegen.ml

```
1.  (* generates LLVM IR from AP++ source code *)
2.
3.  module L = Llvm
4.  module A = Ast
5.  open Sast
6.
7.  module StringMap = Map.Make(String)
8.
9.  (* translate : Sast.program -> Llvm.module *)
10. let translate (globals, functions) =
11.   let context    = L.global_context () in
12.
13.   (* Create the LLVM compilation module into which
14.      we will generate code *)
15.   let the_module = L.create_module context "AP_PlusPlus" in
16.
17.   (* Get types from the context *)
18.   let i32_t      = L.i32_type    context
19.   and i8_t       = L.i8_type     context
20.   and i1_t       = L.i1_type     context
21.   and float_t    = L.double_type context
22.   and str_t      = L.pointer_type (L.i8_type context)
23.   and void_t     = L.void_type   context
24.   and list_t t   = L.struct_type context [| L.pointer_type (L.i32_type context);
    (L.pointer_type t) |]
25.   and ptr_list_t t = L.pointer_type (L.struct_type context [| L.pointer_type (L.i32_type
    context); (L.pointer_type t) |])
26.   in
27.
28.   (* Return the LLVM type for a AP++ type *)
29.   let rec ltype_of_typ = function
30.       A.Int   -> i32_t
31.     | A.Bool  -> i1_t
32.     | A.Float -> float_t
33.     | A.String -> str_t
34.     | A.Void  -> void_t
35.     | A.List t -> list_t (ltype_of_typ t)
36.   in
37.   let type_str t = match t with
38.        A.Int -> "int"
39.      | A.Bool -> "bool"
40.      | A.Float -> "float"
41.      | A.String -> "str"
42.      | _ -> raise (Failure "Invalid string map key type")
43.   in
44.   (* Create a map of global variables after creating each *)
45.   let global_vars : L.llvalue StringMap.t =
46.     let global_var m (t, n) =
47.       let init = match t with
```

```
48.         A.Float -> L.const_float (ltype_of_typ t) 0.0
49.       | _ -> L.const_int (ltype_of_typ t) 0
50.      in StringMap.add n (L.define_global n init the_module) m in
51.    List.fold_left global_var StringMap.empty globals in
52.
53.  let printf_t : L.lltype =
54.      L.var_arg_function_type i32_t [| L.pointer_type i8_t |] in
55.  let printf_func : L.llvalue =
56.      L.declare_function "printf" printf_t the_module in
57.
58.   (* LLVM insists each basic block end with exactly one "terminator"
59.    instruction that transfers control.  This function runs "instr builder"
60.    if the current block does not already have a terminator.  Used,
61.    e.g., to handle the "fall off the end of the function" case. *)
62.   let add_terminal builder instr =
63.     match L.block_terminator (L.insertion_block builder) with
64.       Some _ -> ()
65.     | None -> ignore (instr builder) in
66.
67.    let build_while builder build_predicate build_body func_def =
68.       let pred_bb = L.append_block context "while" func_def in
69.       ignore(L.build_br pred_bb builder);
70.
71.       let body_bb = L.append_block context "while_body" func_def in
72.       add_terminal (build_body (L.builder_at_end context body_bb)) (L.build_br
    pred_bb);
73.
74.       let pred_builder = L.builder_at_end context pred_bb in
75.       let bool_val = build_predicate pred_builder in
76.
77.       let merge_bb = L.append_block context "merge" func_def in
78.       ignore(L.build_cond_br bool_val body_bb merge_bb pred_builder);
79.       L.builder_at_end context merge_bb
80.   in
81.
82.   let build_if builder build_predicate build_then_stmt build_else_stmt func_def =
83.       let bool_val = build_predicate builder in
84.       let merge_bb = L.append_block context "merge" func_def in
85.       let build_br_merge = L.build_br merge_bb in (* partial function *)
86.
87.       let then_bb = L.append_block context "then" func_def in
88.       add_terminal (build_then_stmt (L.builder_at_end context then_bb))
89.       build_br_merge;
90.
91.       let else_bb = L.append_block context "else" func_def in
92.       add_terminal (build_else_stmt (L.builder_at_end context else_bb))
93.       build_br_merge;
94.
95.       ignore(L.build_cond_br bool_val then_bb else_bb builder);
96.       L.builder_at_end context merge_bb
97.  in
```

```
98.   (* ltype list_get(list a, i32_t index) *)
99.   let list_get : L.llvalue StringMap.t =
100.     let list_get_ty m typ =
101.       let ltype = (ltype_of_typ typ) in
102.       let def_name = (type_str typ) in
103.       let def = L.define_function ("list_get" ^ def_name) (L.function_type ltype [|
    L.pointer_type (list_t ltype); i32_t |]) the_module in
104.       let build = L.builder_at_end context (L.entry_block def) in
105.       let list_ptr = L.build_alloca (L.pointer_type (list_t ltype)) "list_ptr_alloc"
    build in
106.       let _ = L.build_store (L.param def 0) list_ptr build in
107.       let idx_ptr = L.build_alloca i32_t "idx_alloc" build in
108.       let _ = L.build_store (L.param def 1) idx_ptr build in
109.       let list_load = L.build_load list_ptr "list_load" build in
110.       let list_array_ptr = L.build_struct_gep list_load 1 "list_array_ptr" build in
111.       let list_array_load = L.build_load list_array_ptr "array_load" build in
112.       let idx = L.build_load idx_ptr "idx_load" build in
113.       let list_array_element_ptr = L.build_gep list_array_load [| idx |]
    "list_arry_element_ptr" build in
114.       let element_val = L.build_load list_array_element_ptr "list_array_element_ptr"
    build in
115.       let _ = L.build_ret element_val build in
116.       StringMap.add def_name def m in
117.    List.fold_left list_get_ty StringMap.empty [ A.Bool; A.Int; A.Float; A.String ] in
118.
119.   (* void list_set(list a, i32_t idx, ltype value) *)
120.   let list_set : L.llvalue StringMap.t =
121.     let list_set_ty m typ =
122.       let ltype = (ltype_of_typ typ) in
123.       let def_name = (type_str typ) in
124.       let def = L.define_function ("list_set" ^ def_name) (L.function_type void_t [|
    L.pointer_type (list_t ltype); i32_t; ltype |]) the_module in
125.       let build = L.builder_at_end context (L.entry_block def) in
126.       let list_ptr = L.build_alloca (L.pointer_type (list_t ltype)) "list_ptr_alloc"
    build in
127.       ignore(L.build_store (L.param def 0) list_ptr build);
128.       let list_load = L.build_load list_ptr "list_load" build in
129.       let list_array_ptr = L.build_struct_gep list_load 1 "list_array_ptr" build in
130.       let list_array_load = L.build_load list_array_ptr "list_array_load" build in
131.       let idx_element_ptr = L.build_gep list_array_load [| L.param def 1 |]
    "list_arry_next_element_ptr" build in
132.       let _ = L.build_store (L.param def 2) idx_element_ptr build in
133.       let _ = L.build_ret_void build in
134.       StringMap.add def_name def m in
135.    List.fold_left list_set_ty StringMap.empty [ A.Bool; A.Int; A.Float; A.String ] in
136.
137.   (* void list_push(list, ltype value) *)
138.   let list_push : L.llvalue StringMap.t =
139.     let list_push_ty m typ =
140.       let ltype = (ltype_of_typ typ) in
141.       let def_name = (type_str typ) in
142.       let def = L.define_function ("list_push" ^ def_name) (L.function_type void_t [|
    L.pointer_type (list_t ltype); ltype |]) the_module in
```

```
143.      let build = L.builder_at_end context (L.entry_block def) in
144.      let list_ptr = L.build_alloca (L.pointer_type (list_t ltype)) "list_ptr_alloc"
    build in
145.      ignore(L.build_store (L.param def 0) list_ptr build);
146.      let valPtr = L.build_alloca ltype "val_alloc" build in
147.      ignore(L.build_store (L.param def 1) valPtr build);
148.      let list_load = L.build_load list_ptr "list_load" build in
149.      let list_array_ptr = L.build_struct_gep list_load 1 "list_array_ptr" build in
150.      let list_array_load = L.build_load list_array_ptr "list_array_load" build in
151.      let list_size_ptr_ptr = L.build_struct_gep list_load 0 "list_size_ptr_ptr" build
    in
152.      let list_size_ptr = L.build_load list_size_ptr_ptr "list_size_ptr" build in
153.      let list_size = L.build_load list_size_ptr "list_size" build in
154.      let next_index = list_size in
155.      let next_element_ptr = L.build_gep list_array_load [| next_index |]
    "list_arry_next_element_ptr" build in
156.      let next_size = L.build_add list_size (L.const_int i32_t 1) "inc_size" build in
157.      let _ = L.build_store next_size list_size_ptr build in
158.      let _ = L.build_store (L.build_load valPtr "val" build) next_element_ptr build in
159.      let _ = L.build_ret_void build in
160.      StringMap.add def_name def m in
161.    List.fold_left list_push_ty StringMap.empty [ A.Bool; A.Int; A.Float; A.String ] in
162.
163.    (* ltype list_pop(list a) *)
164.    let list_pop : L.llvalue StringMap.t =
165.      let list_pop_ty m typ =
166.        let ltype = (ltype_of_typ typ) in
167.        let def_name = (type_str typ) in
168.        let def = L.define_function ("list_pop" ^ def_name) (L.function_type ltype [|
    L.pointer_type (list_t ltype) |]) the_module in
169.        let build = L.builder_at_end context (L.entry_block def) in
170.        let list_ptr = L.build_alloca (L.pointer_type (list_t ltype)) "list_ptr_alloc"
    build in
171.        ignore(L.build_store (L.param def 0) list_ptr build);
172.        let list_load = L.build_load list_ptr "list_load" build in
173.        let list_array_ptr = L.build_struct_gep list_load 1 "list_array_ptr" build in
174.        let list_array_load = L.build_load list_array_ptr "list_array_load" build in
175.        let list_size_ptr_ptr = L.build_struct_gep list_load 0 "list_size_ptr_ptr" build
    in
176.        let list_size_ptr = L.build_load list_size_ptr_ptr "list_size_ptr" build in
177.        let list_size = L.build_load list_size_ptr "list_size" build in
178.        let list_sizeMin1 = L.build_sub list_size (L.const_int i32_t 1) "dec_size" build
    in
179.        let last_element_ptr = L.build_gep list_array_load [| list_sizeMin1 |]
    "list_arry_next_element_ptr" build in
180.        let last_element_val = L.build_load last_element_ptr "list_arry_next_element"
    build in
181.        let _ = L.build_store list_sizeMin1 list_size_ptr build in
182.        let _ = L.build_ret last_element_val build in
183.      StringMap.add def_name def m in
184.    List.fold_left list_pop_ty StringMap.empty [ A.Bool; A.Int; A.Float; A.String ] in
185.
186.    (* i32_t list_size(list a) *)
```

```
187.   let list_size : L.llvalue StringMap.t =
188.     let list_size_ty m typ =
189.      let ltype = (ltype_of_typ typ) in
190.      let def_name = (type_str typ) in
191.      let def = L.define_function ("list_size" ^ def_name) (L.function_type i32_t [|
    L.pointer_type (list_t ltype) |]) the_module in
192.      let build = L.builder_at_end context (L.entry_block def) in
193.      let list_ptr = L.build_alloca (L.pointer_type (list_t ltype)) "list_ptr_alloc"
    build in
194.      ignore(L.build_store (L.param def 0) list_ptr build);
195.      let list_load = L.build_load list_ptr "list_load" build in
196.      let list_size_ptr_ptr = L.build_struct_gep list_load 0 "list_size_ptr_ptr" build
    in
197.      let list_size_ptr = L.build_load list_size_ptr_ptr "list_size_ptr" build in
198.      let list_size = L.build_load list_size_ptr "list_size" build in
199.      ignore(L.build_ret list_size build);
200.      StringMap.add def_name def m in
201.   List.fold_left list_size_ty StringMap.empty [ A.Bool; A.Int; A.Float; A.String ] in
202.
203.   let init_list builder list_ptr list_type =
204.     (* initialize size to 0 *)
205.     let sizePtrPtr = L.build_struct_gep list_ptr 0 "list_size_ptr" builder in
206.        let sizePtr = L.build_alloca i32_t "list_size" builder in
207.        let _ = L.build_store (L.const_int i32_t 0) sizePtr builder in
208.        ignore(L.build_store sizePtr sizePtrPtr builder);
209.     (* initialize array *)
210.     let list_array_ptr = L.build_struct_gep list_ptr 1 "list.arry" builder in
211.      (* TODO: allocate nothing and have list grow dynamically as necessary when pushing
    into the list *)
212.       let p = L.build_array_alloca (ltype_of_typ list_type) (L.const_int i32_t 1028)
    "p" builder in
213.       ignore(L.build_store p list_array_ptr builder);
214.   in
215.
216.   let list_slice : L.llvalue StringMap.t =
217.      let list_slice_ty m typ =
218.         let ltype = (ltype_of_typ typ) in
219.         let def_name = (type_str typ) in
220.         let def = L.define_function ("list_slice" ^ def_name) (L.function_type void_t
    [| ptr_list_t ltype; ptr_list_t ltype; i32_t; i32_t |]) the_module in
221.         let build = L.builder_at_end context (L.entry_block def) in
222.
223.         let list_ptr_ptr = L.build_alloca (ptr_list_t ltype) "list_ptr_alloc" build in
224.         let _ = L.build_store (L.param def 0) list_ptr_ptr build in
225.         let list_ptr = L.build_load list_ptr_ptr "list_ptr_ptr" build in
226.
227.         let list_ptr_ptr2 = L.build_alloca (ptr_list_t ltype) "list_ptr_alloc2" build
    in
228.         let _ = L.build_store (L.param def 1) list_ptr_ptr2 build in
229.         let list_ptr2 = L.build_load list_ptr_ptr2 "list_ptr_ptr2" build in
230.
231.         let idx_ptr1 = L.build_alloca i32_t "idx_alloc" build in
```

```
        let _ = L.build_store (L.param def 2) idx_ptr1 build in
        let idx1 = L.build_load idx_ptr1 "idx_load" build in

        let idx_ptr2 = L.build_alloca i32_t "idx_alloc" build in
        let _ = L.build_store (L.param def 3) idx_ptr2 build in
        let idx2 = L.build_load idx_ptr2 "idx_load" build in

        (* loop counter init: 0 *)
        let loop_cnt_ptr = L.build_alloca i32_t "loop_cnt" build in
        let _ = L.build_store (L.const_int i32_t 0) loop_cnt_ptr build in
        (* loop upper bound: j-i *)
        let loop_upper_bound = L.build_sub idx2 idx1 "loop_upper_bound" build in
        (* loop condition: cnt <= j-i *)
        let loop_cond _builder =
            L.build_icmp L.Icmp.Sle (L.build_load loop_cnt_ptr "loop_cnt" _builder)
loop_upper_bound "loop_cond" _builder
        in
        (* assignment: b[cnt] = a[cnt + i] *)
        let loop_body _builder =
           let to_index = L.build_load loop_cnt_ptr "to_idx" _builder in
           let from_index = L.build_add to_index idx1 "from_idx" _builder in
           let get_val = L.build_call (StringMap.find (type_str typ) list_get) [|
list_ptr; from_index |] "list_get" _builder in
           let _ = L.build_call (StringMap.find (type_str typ) list_push) [| list_ptr2;
get_val |] "" _builder in
           let index_incr = L.build_add (L.build_load loop_cnt_ptr "loop_cnt" _builder)
(L.const_int i32_t 1) "loop_itr" _builder in
           let _ = L.build_store index_incr loop_cnt_ptr _builder in
           _builder
        in
        let while_builder = build_while build loop_cond loop_body def in
        ignore(L.build_ret_void while_builder);
        StringMap.add def_name def m
     in
     List.fold_left list_slice_ty StringMap.empty [ A.Bool; A.Int; A.Float; A.String ]
in

  (* i32_t list_find(list, val) *)
  let list_find : L.llvalue StringMap.t =
    let list_find_ty m typ =
       let ltype = (ltype_of_typ typ) in
       let def_name = (type_str typ) in
       let def = L.define_function ("list_find" ^ def_name) (L.function_type i32_t [|
L.pointer_type (list_t ltype); ltype |]) the_module in
       let build = L.builder_at_end context (L.entry_block def) in
       let list_ptr = L.build_alloca (L.pointer_type (list_t ltype)) "list_ptr_alloc"
build in
       ignore(L.build_store (L.param def 0) list_ptr build);
       let find_value_ptr = L.build_alloca ltype "find_val_alloc" build in
       ignore(L.build_store (L.param def 1) find_value_ptr build);
       let find_value = L.build_load find_value_ptr "find_val" build in
       let list_load = L.build_load list_ptr "list_load" build in
```

```
277.        let list_size_ptr_ptr = L.build_struct_gep list_load 0 "list_size_ptr_ptr" build
    in
278.        let list_size_ptr = L.build_load list_size_ptr_ptr "list_size_ptr" build in
279.        let list_size = L.build_load list_size_ptr "list_size" build in
280.        let loop_idx_ptr = L.build_alloca i32_t "loop_cnt" build in
281.        let _ = L.build_store (L.const_int i32_t 0) loop_idx_ptr build in
282.        let loop_upper_bound = list_size in
283.        let loop_cond _builder =
284.           L.build_icmp L.Icmp.Slt (L.build_load loop_idx_ptr "loop_iter_cnt" _builder)
    loop_upper_bound "loop_cond" _builder
285.        in
286.        let loop_body _builder =
287.          let index = L.build_load loop_idx_ptr "to_idx" _builder in
288.          let get_val = L.build_call (StringMap.find (type_str typ) list_get) [|
    list_load; index |] "list_get" _builder in
289.          let if_cond _builder2 =
290.             (match typ with
291.                 A.Int | A.Bool -> L.build_icmp L.Icmp.Eq
292.               | A.Float -> L.build_fcmp L.Fcmp.Oeq
293.               | _ -> raise (Failure ("list_find does not support this list type"))
294.             ) get_val find_value "if_cond" _builder2
295.          in
296.          let if_body _builder2 = ignore(L.build_ret index _builder2); _builder2 in
297.          let else_body _builder2 = ignore(L.const_int i32_t 0); _builder2 in
298.          let if_builder = build_if _builder if_cond if_body else_body def in
299.          let index_incr = L.build_add (L.build_load loop_idx_ptr "loop_idx" if_builder)
    (L.const_int i32_t 1) "loop_itr" if_builder in
300.          let _ = L.build_store index_incr loop_idx_ptr if_builder in
301.          if_builder
302.        in
303.        let while_builder = build_while build loop_cond loop_body def in
304.        ignore(L.build_ret (L.const_int i32_t (-1)) while_builder);
305.        StringMap.add def_name def m in
306.      List.fold_left list_find_ty StringMap.empty [ A.Bool; A.Int; A.Float ] in
307.
308.    (* void list_remove(list, typ value) *)
309.    let list_remove : L.llvalue StringMap.t =
310.      let list_remove_ty m typ =
311.       let ltype = (ltype_of_typ typ) in
312.       let def_name = (type_str typ) in
313.       let def = L.define_function ("list_remove" ^ def_name) (L.function_type void_t [|
    L.pointer_type (list_t ltype); ltype |]) the_module in
314.       let build = L.builder_at_end context (L.entry_block def) in
315.       let list_ptr = L.build_alloca (L.pointer_type (list_t ltype)) "list_ptr_alloc"
    build in
316.       ignore(L.build_store (L.param def 0) list_ptr build);
317.       let remove_value_ptr = L.build_alloca ltype "rem_val_ptr" build in
318.       ignore(L.build_store (L.param def 1) remove_value_ptr build);
319.       let remove_value = L.build_load remove_value_ptr "rem_val" build in
320.       let list_load = L.build_load list_ptr "list_load" build in
321.       let list_size_ptr_ptr = L.build_struct_gep list_load 0 "list_size_ptr_ptr" build
    in
322.       let list_size_ptr = L.build_load list_size_ptr_ptr "list_size_ptr" build in
```

```
323.      let list_size = L.build_load list_size_ptr "list_size" build in
324.      let listFindIndex = L.build_call (StringMap.find (type_str typ) list_find) [|
    list_load; remove_value |] "list_find" build in
325.      let list_find_if_cond _builder =
326.          L.build_icmp L.Icmp.Sge listFindIndex (L.const_int i32_t 0) "loop_cond"
    _builder in
327.      let list_else_body _builder = ignore(L.const_int i32_t 0); _builder in
328.      let list_find_if_body _builder =
329.         let loop_idx_ptr = L.build_alloca i32_t "loop_cnt_ptr" _builder in
330.         let loop_start_idx = L.build_add listFindIndex (L.const_int i32_t 1)
    "loop_start_idx" _builder in
331.         let _ = L.build_store loop_start_idx loop_idx_ptr _builder in
332.         let loop_upper_bound = list_size in
333.         let loop_cond _builder =
334.            L.build_icmp L.Icmp.Slt (L.build_load loop_idx_ptr "loop_cnt" _builder)
    loop_upper_bound "loop_cond" _builder
335.         in
336.         let loop_body _builder =
337.           let cur_index = L.build_load loop_idx_ptr "cur_idx" _builder in
338.           let shiftto_index = L.build_sub cur_index (L.const_int i32_t 1)
    "shift_to_idx" _builder in
339.           let get_val = L.build_call (StringMap.find (type_str typ) list_get) [|
    list_load; cur_index |] "list_get" _builder in
340.           let _ = L.build_call (StringMap.find (type_str typ) list_set) [| list_load;
    shiftto_index; get_val |] "" _builder in
341.           let index_incr = L.build_add cur_index (L.const_int i32_t 1) "loop_itr"
    _builder in
342.           let _ = L.build_store index_incr loop_idx_ptr _builder in
343.           _builder
344.         in
345.         let while_builder = build_while _builder loop_cond loop_body def in
346.         let size_dec = L.build_sub list_size (L.const_int i32_t 1) "size_dec"
    while_builder in
347.         let _ = L.build_store size_dec list_size_ptr while_builder in
348.         ignore(L.build_ret_void while_builder); while_builder
349.      in
350.      let if_builder = build_if build list_find_if_cond list_find_if_body list_else_body
    def in
351.      let _ = L.build_ret_void if_builder in
352.      StringMap.add def_name def m in
353.   List.fold_left list_remove_ty StringMap.empty [ A.Bool; A.Int; A.Float ] in
354.
355.   (* void list_insert(list, int idx, typ value) *)
356.   let list_insert : L.llvalue StringMap.t =
357.     let list_insert_ty m typ =
358.      let ltype = (ltype_of_typ typ) in
359.      let def_name = (type_str typ) in
360.      let def = L.define_function ("list_insert" ^ def_name) (L.function_type void_t [|
    L.pointer_type (list_t ltype); i32_t; ltype |]) the_module in
361.      let build = L.builder_at_end context (L.entry_block def) in
362.
363.      let list_ptr = L.build_alloca (L.pointer_type (list_t ltype)) "list_ptr_alloc"
    build in
```

```
364.      ignore(L.build_store (L.param def 0) list_ptr build);
365.      let list_load = L.build_load list_ptr "list_load" build in
366.
367.      let insertidx_ptr = L.build_alloca i32_t "insert_idx_ptr" build in
368.      ignore(L.build_store (L.param def 1) insertidx_ptr build);
369.      let insertIdx = L.build_load insertidx_ptr "insert_idx" build in
370.
371.      let insertValPtr = L.build_alloca ltype "insert_val_ptr" build in
372.      ignore(L.build_store (L.param def 2) insertValPtr build);
373.      let insertVal = L.build_load insertValPtr "insert_val" build in
374.
375.      let list_size_ptr_ptr = L.build_struct_gep list_load 0 "list_size_ptr_ptr" build
    in
376.      let list_size_ptr = L.build_load list_size_ptr_ptr "list_size_ptr" build in
377.      let list_size = L.build_load list_size_ptr "list_size" build in
378.      let loop_idx_ptr = L.build_alloca i32_t "loop_cnt_ptr" build in
379.      let lastIndex = L.build_sub list_size (L.const_int i32_t 1) "last_index" build in
380.      let _ = L.build_store lastIndex loop_idx_ptr build in
381.      let decto_index = insertIdx in
382.      let loop_cond _builder =
383.         L.build_icmp L.Icmp.Sge (L.build_load loop_idx_ptr "loop_cnt" _builder)
    decto_index "loop_cond" _builder
384.      in
385.      let loop_body _builder =
386.        let cur_index = L.build_load loop_idx_ptr "cur_idx" _builder in
387.        let shiftto_index = L.build_add cur_index (L.const_int i32_t 1) "shift_to_idx"
    _builder in
388.        let get_val = L.build_call (StringMap.find (type_str typ) list_get) [|
    list_load; cur_index |] "list_get" _builder in
389.        let _ = L.build_call (StringMap.find (type_str typ) list_set) [| list_load;
    shiftto_index; get_val |] "" _builder in
390.        let indexDec = L.build_sub cur_index (L.const_int i32_t 1) "loop_itr" _builder
    in
391.        let _ = L.build_store indexDec loop_idx_ptr _builder in
392.        _builder
393.      in
394.      let while_builder = build_while build loop_cond loop_body def in
395.      let _ = L.build_call (StringMap.find (type_str typ) list_set) [| list_load;
    insertIdx; insertVal |] "" while_builder in
396.      let sizeInc = L.build_add list_size (L.const_int i32_t 1) "size_inc" while_builder
    in
397.      let _ = L.build_store sizeInc list_size_ptr while_builder in
398.      ignore(L.build_ret_void while_builder);
399.      StringMap.add def_name def m in
400.   List.fold_left list_insert_ty StringMap.empty [ A.Bool; A.Int; A.Float ] in
401.
402.  (* void list_insert(list, int idx, typ value) *)
403.   let list_reverse : L.llvalue StringMap.t =
404.     let list_reverse_ty m typ =
405.      let ltype = (ltype_of_typ typ) in
406.      let def_name = (type_str typ) in
```

```
407.      let def = L.define_function ("list_reverse" ^ def_name) (L.function_type void_t [|
    L.pointer_type (list_t ltype) |]) the_module in
408.      let build = L.builder_at_end context (L.entry_block def) in
409.
410.      let list_ptr = L.build_alloca (L.pointer_type (list_t ltype)) "list_ptr_alloc"
    build in
411.      ignore(L.build_store (L.param def 0) list_ptr build);
412.      let list_load = L.build_load list_ptr "list_load" build in
413.
414.      let list_size_ptr_ptr = L.build_struct_gep list_load 0 "list_size_ptr_ptr" build
    in
415.      let list_size_ptr = L.build_load list_size_ptr_ptr "list_size_ptr" build in
416.      let list_size = L.build_load list_size_ptr "list_size" build in
417.
418.      let leftPtr = L.build_alloca i32_t "left_idx" build in
419.      let _ = L.build_store (L.const_int i32_t 0) leftPtr build in
420.      let rightPtr = L.build_alloca i32_t "right_idx" build in
421.      let _ = L.build_store (L.build_sub list_size (L.const_int i32_t 1) "tmp" build)
    rightPtr build in
422.
423.      let while_cond _builder = L.build_icmp L.Icmp.Slt
424.         (L.build_load leftPtr "left_idx" _builder)
425.         (L.build_load rightPtr "right_idx" _builder) "while_cond" _builder
426.      in
427.      let while_body _builder =
428.         let left_idx = (L.build_load leftPtr "left_idx" _builder) in
429.         let right_idx = (L.build_load rightPtr "right_idx" _builder) in
430.         let get_left_val = L.build_call (StringMap.find (type_str typ) list_get) [|
    list_load; left_idx |] "list_get" _builder in
431.         let get_right_val = L.build_call (StringMap.find (type_str typ) list_get) [|
    list_load; right_idx |] "list_get" _builder in
432.         let _ = L.build_call (StringMap.find (type_str typ) list_set) [| list_load;
    left_idx; get_right_val |] "" _builder in
433.         let _ = L.build_call (StringMap.find (type_str typ) list_set) [| list_load;
    right_idx; get_left_val |] "" _builder in
434.         let _ = L.build_store (L.build_add left_idx (L.const_int i32_t 1) "tmp"
    _builder) leftPtr _builder in
435.         let _ = L.build_store (L.build_sub right_idx (L.const_int i32_t 1) "tmp"
    _builder) rightPtr _builder in
436.         _builder
437.       in
438.       let while_builder = build_while build while_cond while_body def in
439.       ignore(L.build_ret_void while_builder);
440.       StringMap.add def_name def m in
441.   List.fold_left list_reverse_ty StringMap.empty [ A.Bool; A.Int; A.Float ] in
442.
443.   (* Define each function (arguments and return type) so we can
444.      call it even before we've created its body *)
445.   let function_decls : (L.llvalue * sfunc_decl) StringMap.t =
446.     let function_decl m fdecl =
447.       let name = fdecl.sfname
```

```
448.       and formal_types = Array.of_list (List.map (fun (t,_) -> ltype_of_typ t)
    fdecl.sformals)
449.       in let ftype = L.function_type (ltype_of_typ fdecl.styp) formal_types in
450.       StringMap.add name (L.define_function name ftype the_module, fdecl) m in
451.       List.fold_left function_decl StringMap.empty functions in
452.
453.   (* Fill in the body of the given function *)
454.   let build_function_body fdecl =
455.     let (the_function, _) = StringMap.find fdecl.sfname function_decls in
456.     let builder = L.builder_at_end context (L.entry_block the_function) in
457.
458.     let int_format_str = L.build_global_stringptr "%d\n" "fmt" builder in
459.     let float_format_str = L.build_global_stringptr "%g\n" "fmt" builder in
460.     let str_format_str = L.build_global_stringptr "%s\n" "fmt" builder in
461.
462.     (* Construct the function's "locals": formal arguments and locally
463.        declared variables.  Allocate each on the stack, initialize their
464.        value, if appropriate, and remember their values in the "locals" map *)
465.     let local_vars =
466.       let add_formal m (t, n) p =
467.         L.set_value_name n p;
468.     let local = L.build_alloca (ltype_of_typ t) n builder in
469.     ignore(
470.       match t with
471.         A.List list_type -> init_list builder local list_type
472.       | _ -> ()
473.     );
474.     ignore (L.build_store p local builder);
475.     StringMap.add n local m
476.
477.       (* Allocate space for any locally declared variables and add the
478.        * resulting registers to our map *)
479.       and add_local m (t, n) =
480.   let local_var = L.build_alloca (ltype_of_typ t) n builder in
481.   ignore(
482.     match t with
483.       A.List list_type -> init_list builder local_var list_type
484.     | _ -> ()
485.   );
486.   StringMap.add n local_var m
487.       in
488.       let formals = List.fold_left2 add_formal StringMap.empty fdecl.sformals
489.           (Array.to_list (L.params the_function)) in
490.       List.fold_left add_local formals fdecl.slocals
491.     in
492.
493.     (* Return the value for a variable or formal argument.
494.        Check local names first, then global names *)
495.     let lookup n = try StringMap.find n local_vars
496.                    with Not_found -> StringMap.find n global_vars
497.
498.     in
```

```
499.     (* Construct code for an expression; return its value *)
500.     let rec expr builder ((_, e) : sexpr) = match e with
501.         SILiteral i  -> L.const_int i32_t i
502.       | SBLiteral b  -> L.const_int i1_t (if b then 1 else 0)
503.       | SFLiteral l -> L.const_float float_t l
504.       | SSLiteral s -> L.build_global_stringptr (s^"\x00") "strptr" builder
505.       | SNoexpr     -> L.const_int i32_t 0
506.       | SId s       -> L.build_load (lookup s) s builder
507.       | SAssign (s, e) -> let e' = expr builder e in
508.                           ignore(L.build_store e' (lookup s) builder); e'
509.       | SBinop ((A.Float, _ ) as e1, op, e2) ->
510.         let e1' = expr builder e1
511.         and e2' = expr builder e2 in
512.         (match op with
513.           A.Add     -> L.build_fadd
514.         | A.Sub     -> L.build_fsub
515.         | A.Mult    -> L.build_fmul
516.         | A.Div     -> L.build_fdiv
517.         | A.Equal   -> L.build_fcmp L.Fcmp.Oeq
518.         | A.Neq     -> L.build_fcmp L.Fcmp.One
519.         | A.Less    -> L.build_fcmp L.Fcmp.Olt
520.         | A.Leq     -> L.build_fcmp L.Fcmp.Ole
521.         | A.Greater -> L.build_fcmp L.Fcmp.Ogt
522.         | A.Geq     -> L.build_fcmp L.Fcmp.Oge
523.         | A.And | A.Or | A.Mod ->
524.             raise (Failure "internal error: semant should have rejected and/or on
    float")
525.         ) e1' e2' "tmp" builder
526.        (* TODO: list equality check *)
527.        (* | SBinop ((A.List, _) as e1, op, e2) ->
528.         let e1' = expr builder e1
529.         and e2' = expr builder e2 in
530.         (match op with
531.           A.Equal   ->
532.         | A.Neq     ->
533.         | _ -> raise (Failure "internal error: semant should have rejected and/or on
    float")
534.         ) e1' e2' "tmp" builder *)
535.       | SBinop (e1, op, e2) ->
536.         let e1' = expr builder e1
537.         and e2' = expr builder e2 in
538.         (match op with
539.           A.Add     -> L.build_add
540.         | A.Sub     -> L.build_sub
541.         | A.Mult    -> L.build_mul
542.         | A.Div     -> L.build_sdiv
543.         | A.Mod     -> L.build_srem
544.         | A.And     -> L.build_and
545.         | A.Or      -> L.build_or
546.         | A.Equal   -> L.build_icmp L.Icmp.Eq
547.         | A.Neq     -> L.build_icmp L.Icmp.Ne
548.         | A.Less    -> L.build_icmp L.Icmp.Slt
549.         | A.Leq     -> L.build_icmp L.Icmp.Sle
```

```
        | A.Greater -> L.build_icmp L.Icmp.Sgt
        | A.Geq     -> L.build_icmp L.Icmp.Sge
        ) e1' e2' "tmp" builder
    | SUnop(op, ((t, e'') as e)) ->
        let e' = expr builder e in
      (match op with
        A.Neg when t = A.Float -> L.build_fneg e' "tmp" builder
      | A.Neg                  -> L.build_neg e' "tmp" builder
      | A.Not                  -> L.build_not e' "tmp" builder
      | A.PlusPlusPre ->
        let new_val = (L.build_add e' (L.const_int i32_t 1)) "tmp" builder in
        let id = match (e'') with
          SId s -> s
          | _ -> raise (Failure ("++ operand must be an ID")) in
        let var_ptr = (lookup id) in
        let _ = L.build_store new_val var_ptr builder in new_val
      | A.MinusMinusPre ->
        let new_val = (L.build_sub e' (L.const_int i32_t 1)) "tmp" builder in
        let id = match (e'') with
          SId s -> s
          | _ -> raise (Failure ("-- operand must be an ID")) in
        let var_ptr = (lookup id) in
        let _ = L.build_store new_val var_ptr builder in new_val
      | A.PlusPlusPost ->
        let new_val = (L.build_add e' (L.const_int i32_t 1)) "tmp" builder in
        let id = match (e'') with
          SId s -> s
          | _ -> raise (Failure ("++ operand must be an ID")) in
        let var_ptr = (lookup id) in
        let _ = L.build_store new_val var_ptr builder in e'
      | A.MinusMinusPost ->
        let new_val = (L.build_sub e' (L.const_int i32_t 1)) "tmp" builder in
        let id = match (e'') with
          SId s -> s
          | _ -> raise (Failure ("-- operand must be an ID")) in
        let var_ptr = (lookup id) in
        let _ = L.build_store new_val var_ptr builder in e')

    | SListGet (list_type, id, e) ->
      L.build_call (StringMap.find (type_str list_type) list_get) [| (lookup id); (expr
builder e) |] "list_get" builder
    | SListSize (list_type, id) ->
      L.build_call ((StringMap.find (type_str list_type)) list_size) [| (lookup id) |]
"list_size" builder
    | SListPop (list_type, id) ->
      L.build_call ((StringMap.find (type_str list_type)) list_pop) [| (lookup id) |]
"list_pop" builder
    | SListSlice (list_type, id, e1, e2) ->
       let ltype = (ltype_of_typ list_type) in
       let new_list_ptr = L.build_alloca (list_t ltype) "new_list_ptr" builder in
       let _ = init_list builder new_list_ptr list_type in
       let e' = match (fst e1, fst e2) with
           (A.Int, A.Int) -> (expr builder e1, expr builder e2)
```

```
600.           | (A.Void, A.Int) -> (L.const_int i32_t 0, expr builder e2)
601.           | (A.Int, A.Void) ->  (expr builder e1, L.build_sub (expr builder (A.Int,
    SListSize(list_type, id))) (L.const_int i32_t 1) "size_min_one" builder)
602.           | (A.Void, A.Void) -> (L.const_int i32_t 0, L.build_sub (expr builder (A.Int,
    SListSize(list_type, id))) (L.const_int i32_t 1) "size_min_one" builder)
603.           | _ -> raise (Failure ("illegal list slice arguments"))
604.         in
605.         let _ = L.build_call ((StringMap.find (type_str list_type)) list_slice) [|
    (lookup id); new_list_ptr; fst e'; snd e' |] "" builder in
606.         L.build_load new_list_ptr "new_list" builder
607.       | SListFind (list_type, id, e) ->
608.         L.build_call (StringMap.find (type_str list_type) list_find) [| (lookup id);
    (expr builder e) |] "list_find" builder
609.       | SListLiteral (list_type, literals) ->
610.         let ltype = (ltype_of_typ list_type) in
611.         let new_list_ptr = L.build_alloca (list_t ltype) "new_list_ptr" builder in
612.         let _ = init_list builder new_list_ptr list_type in
613.         let map_func literal =
614.           ignore(L.build_call (StringMap.find (type_str list_type) list_push) [|
    new_list_ptr; (expr builder literal) |] "" builder);
615.         in
616.         let _ = List.rev (List.map map_func literals) in
617.         L.build_load new_list_ptr "new_list" builder
618.       | SCall ("prints", [e]) ->
619.         L.build_call printf_func [| str_format_str ; (expr builder e) |] "printf" builder
620.       | SCall ("printi", [e]) ->
621.         L.build_call printf_func [| int_format_str ; (expr builder e) |] "printf" builder
622.       | SCall ("printb", [e]) ->
623.         L.build_call printf_func [| int_format_str; (expr builder e) |] "printf" builder
624.       | SCall ("printf", [e]) ->
625.         L.build_call printf_func [| float_format_str ; (expr builder e) |] "printf"
    builder
626.       | SCall (f, args) ->
627.           let (fdef, fdecl) = StringMap.find f function_decls in
628.           let llargs = List.rev (List.map (expr builder) (List.rev args)) in
629.           let result = (match fdecl.styp with
630.                               A.Void -> ""
631.                             | _ -> f ^ "_result") in
632.                 L.build_call fdef (Array.of_list llargs) result builder
633.           in
634.
635.       (* Build the code for the given statement; return the builder for
636.          the statement's successor (i.e., the next instruction will be built
637.          after the one generated by this call) *)
638.
639.       let rec stmt builder = function
640.           SBlock sl -> List.fold_left stmt builder sl
641.         | SListPush (id, e) ->
642.           ignore(L.build_call (StringMap.find (type_str (fst e)) list_push) [| (lookup
    id); (expr builder e) |] "" builder); builder
643.         | SListSet (list_type, id, e1, e2) ->
644.           ignore(L.build_call (StringMap.find (type_str list_type) list_set) [| (lookup
    id); (expr builder e1); (expr builder e2) |] "" builder); builder
```

```
645.       | SListClear (list_type, id) ->
646.           ignore(init_list builder (lookup id) list_type); builder
647.       | SListRemove (id, e) ->
648.           ignore(L.build_call (StringMap.find (type_str (fst e)) list_remove) [|
   (lookup id); (expr builder e) |] "" builder); builder
649.       | SListInsert (id, e1, e2) ->
650.           ignore(L.build_call (StringMap.find (type_str (fst e2)) list_insert) [|
   (lookup id); (expr builder e1); (expr builder e2) |] "" builder); builder
651.       | SListReverse (list_type, id) ->
652.           ignore(L.build_call (StringMap.find (type_str list_type) list_reverse) [|
   (lookup id) |] "" builder); builder
653.       | SExpr e -> ignore(expr builder e); builder
654.       | SReturn e -> ignore(match fdecl.styp with
655.                             (* Special "return nothing" instr *)
656.                             A.Void -> L.build_ret_void builder
657.                             (* Build return statement *)
658.                           | _ -> L.build_ret (expr builder e) builder );
659.                     builder
660.       | SIf (predicate, then_stmt, else_stmt) ->
661.         let bool_val = expr builder predicate in
662.         let merge_bb = L.append_block context "merge" the_function in
663.         let build_br_merge = L.build_br merge_bb in (* partial function *)
664.
665.         let then_bb = L.append_block context "then" the_function in
666.         add_terminal (stmt (L.builder_at_end context then_bb) then_stmt)
667.         build_br_merge;
668.
669.         let else_bb = L.append_block context "else" the_function in
670.         add_terminal (stmt (L.builder_at_end context else_bb) else_stmt)
671.         build_br_merge;
672.
673.         ignore(L.build_cond_br bool_val then_bb else_bb builder);
674.         L.builder_at_end context merge_bb
675.
676.       | SWhile (predicate, body) ->
677.         let pred_bb = L.append_block context "while" the_function in
678.         ignore(L.build_br pred_bb builder);
679.
680.         let body_bb = L.append_block context "while_body" the_function in
681.         add_terminal (stmt (L.builder_at_end context body_bb) body)
682.           (L.build_br pred_bb);
683.
684.         let pred_builder = L.builder_at_end context pred_bb in
685.         let bool_val = expr pred_builder predicate in
686.
687.         let merge_bb = L.append_block context "merge" the_function in
688.         ignore(L.build_cond_br bool_val body_bb merge_bb pred_builder);
689.         L.builder_at_end context merge_bb
690.
691.       (* Implement for loops as while loops *)
692.       | SFor (e1, e2, e3, body) -> stmt builder
```

```
693.        ( SBlock [SExpr e1 ; SWhile (e2, SBlock [body ; SExpr e3]) ] )
694.
695.      in
696.
697.    (* Build the code for each statement in the function *)
698.    let builder = stmt builder (SBlock fdecl.sbody) in
699.
700.    (* Add a return if the last block falls off the end *)
701.    add_terminal builder (match fdecl.styp with
702.        A.Void -> L.build_ret_void
703.      | A.Float -> L.build_ret (L.const_float float_t 0.0)
704.      | t -> L.build_ret (L.const_int (ltype_of_typ t) 0))
705.  in
706.
707.  List.iter build_function_body functions;
708.  the_module
```