

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

*AP++*

## Contents

- I. Introduction
- II. Language Tutorial
- III. Language Reference Manual
- IV. Project Plan
- V. Architectural Design
- VI. Test Plan
- VII. Lessons
- VIII. Appendix

## 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

```
1. int max_value(list<int> a) {
2.     int max_val;
3.     int i;
4.     max_val = -1; // assume all elements are positive
5.     for (i = 0; i < #a; ++i) {
6.         if (a[i] > max_val) {
7.             max_val = a[i];
8.         }
9.     }
10.    return max_val;
11. }
12.
13. int main() {
14.     list<int> a;
15.     a = [4, 10, 40, 23, 85, 2];
16.     printi(max_value(a));
17.     return 0;
18. }
19.
20. 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

`C++` 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	<pre>int x; int y; x = 3; ++x; /* x == 4 */ y = x++; /* y == 4, x == 5 */</pre>
>	greater than	<pre>int x; x = 2; x &gt; 1 /* true */ x &gt; 4 /* false */</pre>
>=	greater than equal to	<pre>int x = 2; x &gt;= 2 /* true */ x &gt;= 4 /* false */</pre>
<	less than	<pre>int x; x = 2; x &lt; 1 /* false */ x &lt; 4 /* true */</pre>
<=	less than equal to	<pre>int x; x = 2; x &lt;= 2 /* true */ x &lt;= 4 /* true */</pre>
&&	boolean AND	<pre>bool b1; bool b2; b1 = true; b2 = false;  (b1 &amp;&amp; b2) /* false */ (true &amp;&amp; true) /* true */ (true &amp;&amp; false) /* false */ (false &amp;&amp; false) /* false */ (false &amp;&amp; true) /* false */</pre>
	boolean OR	<pre>bool b1; bool b2 = false; b1 = true; b2 = false;  (b1    b2) /* false */ (true    true) /* true */ (true    false) /* false */ (false    false) /* false */ (false    true) /* false */</pre>
!	boolean NOT	<pre>bool b1; bool b2; b1 = true; b2 = false;</pre>

		!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 (!expr1) || (!expr2)  
 !expr1 >= expr2 || expr3 && expr4 < expr5 is evaluated as  
 ((!expr1) >= expr2) || (expr3 && (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.

<u>Function</u>	<u>Description</u>
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

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

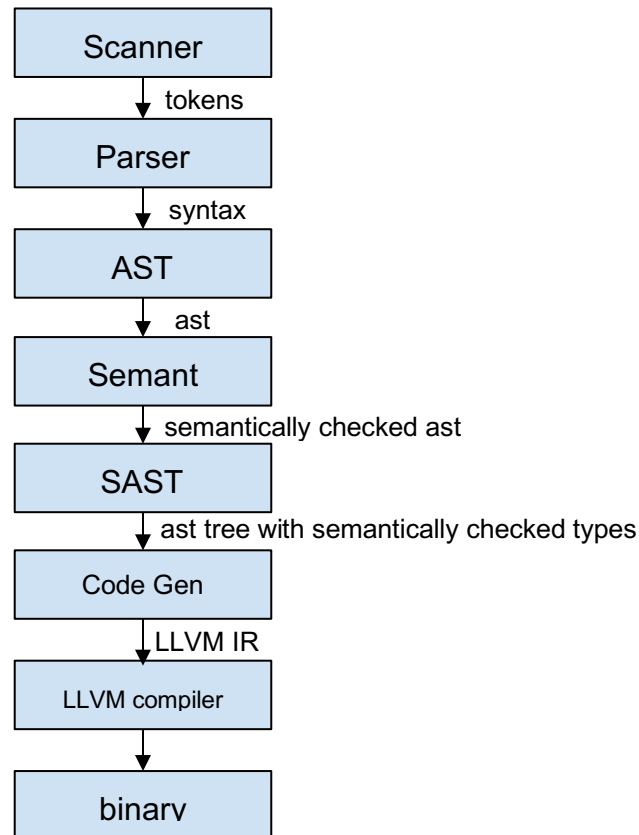
Start Date	Milestone
Sept. 19	Project Proposal
Oct. 15	Project LRM
Nov. 14	Hello World!
Nov. 25	pre/post ++/--
Dec. 6	LLVM representation of a List
Dec. 9	list declaration
Dec. 10	list_push
Dec. 10	list_size
Dec. 11	list_get
Dec. 12	list assignment by ref
Dec. 13	list_pop
Dec. 13	list_set and bracket syntax, i.e. <code>a[i] = x</code>
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

```
1. /* sorts a list in-place with BubbleSort algorithm, */
2.
3. void bubbleSort(list<int> arr) {
4.     int i;
5.     int j;
6.     int tmp;
7.     for (i = 0; i < #arr-1; i++) {
8.         for (j = 0; j < #arr-i-1; j++) {
9.             if (arr[j] > arr[j+1]) {
10.                tmp = arr[j];
11.                arr[j] = arr[j+1];
12.                arr[j+1] = tmp;
13.            }
14.        }
15.    }
16. }
17.
18. int main() {
19.     list<int> a;
20.     int i;
21.     a = [10, 2, 15, 8, 20, 13, 9];
22.     bubbleSort(a);
23.     for(i = 0; i < #a; ++i) {
24.         printi(a[i]);
25.     }
26.     return 0;
27. }
28.
29. output:
30. 2
31. 8
32. 9
33. 13
34. 10
35. 15
36. 13
37. 20
```

#### 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.
11. declare i32 @printf(i8*, ...)
12.
13. define i1 @list_getbool({ i32*, i1* }*, i32) {
14. entry:
15.   %list_ptr_alloc = alloca { i32*, i1* }*
16.   store { i32*, i1* }* %0, { i32*, i1* }** %list_ptr_alloc
17.   %idx_alloc = alloca i32
18.   store i32 %1, i32* %idx_alloc
19.   %list_load = load { i32*, i1* }*, { i32*, i1* }** %list_ptr_alloc
20.   %list_array_ptr = getelementptr inbounds { i32*, i1* }, { i32*, i1* }* %list_load, i32
       0, i32 1
21.   %array_load = load i1*, i1** %list_array_ptr
22.   %idx_load = load i32, i32* %idx_alloc
23.   %list_arr_element_ptr = getelementptr i1, i1* %array_load, i32 %idx_load
24.   %list_array_element_ptr = load i1, i1* %list_arr_element_ptr
25.   ret i1 %list_array_element_ptr
26. }
27.
28. define i32 @list_getint({ i32*, i32* }*, i32) {
29. entry:
30.   %list_ptr_alloc = alloca { i32*, i32* }*
31.   store { i32*, i32* }* %0, { i32*, i32* }** %list_ptr_alloc
32.   %idx_alloc = alloca i32
33.   store i32 %1, i32* %idx_alloc
34.   %list_load = load { i32*, i32* }*, { i32*, i32* }** %list_ptr_alloc
35.   %list_array_ptr = getelementptr inbounds { i32*, i32* }, { i32*, i32* }* %list_load,
       i32 0, i32 1
36.   %array_load = load i32*, i32** %list_array_ptr
37.   %idx_load = load i32, i32* %idx_alloc
38.   %list_arr_element_ptr = getelementptr i32, i32* %array_load, i32 %idx_load
39.   %list_array_element_ptr = load i32, i32* %list_arr_element_ptr
40.   ret i32 %list_array_element_ptr
41. }
42.
43. define double @list_getfloat({ i32*, double* }*, i32) {
44. entry:
45.   %list_ptr_alloc = alloca { i32*, double* }*
46.   store { i32*, double* }* %0, { i32*, double* }** %list_ptr_alloc
47.   %idx_alloc = alloca i32
48.   store i32 %1, i32* %idx_alloc
49.   %list_load = load { i32*, double* }*, { i32*, double* }** %list_ptr_alloc
50.   %list_array_ptr = getelementptr inbounds { i32*, double* }, { i32*,
       double* }* %list_load, i32 0, i32 1
51.   %array_load = load double*, double** %list_array_ptr
52.   %idx_load = load i32, i32* %idx_alloc
53.   %list_arr_element_ptr = getelementptr double, double* %array_load, i32 %idx_load
54.   %list_array_element_ptr = load double, double* %list_arr_element_ptr

```

```

55.  ret double %list_array_element_ptr
56. }
57.
58. define i8* @list_getstr({ i32*, i8** }*, i32) {
59. entry:
60.  %list_ptr_alloc = alloca { i32*, i8** }*
61.  store { i32*, i8** }* %0, { i32*, i8** }** %list_ptr_alloc
62.  %idx_alloc = alloca i32
63.  store i32 %1, i32* %idx_alloc
64.  %list_load = load { i32*, i8** }*, { i32*, i8** }** %list_ptr_alloc
65.  %list_array_ptr = getelementptr inbounds { i32*, i8** }, { i32*, i8** }* %list_load,
    i32 0, i32 1
66.  %array_ptr = load i8**, i8*** %list_array_ptr
67.  %idx_load = load i32, i32* %idx_alloc
68.  %list_array_element_ptr = getelementptr i8*, i8** %array_ptr, i32 %idx_load
69.  %list_array_element_ptr = load i8*, i8** %list_array_element_ptr
70.  ret i8* %list_array_element_ptr
71. }
72.
73. define void @list_setbool({ i32*, i1* }*, i32, i1) {
74. entry:
75.  %list_ptr_alloc = alloca { i32*, i1* }*
76.  store { i32*, i1* }* %0, { i32*, i1* }** %list_ptr_alloc
77.  %list_load = load { i32*, i1* }*, { i32*, i1* }** %list_ptr_alloc
78.  %list_array_ptr = getelementptr inbounds { i32*, i1* }, { i32*, i1* }* %list_load, i32
    0, i32 1
79.  %list_array_load = load i1*, i1** %list_array_ptr
80.  %list_array_next_element_ptr = getelementptr i1, i1* %list_array_load, i32 %1
81.  store i1 %2, i1* %list_array_next_element_ptr
82.  ret void
83. }
84.
85. define void @list_setint({ i32*, i32* }*, i32, i32) {
86. entry:
87.  %list_ptr_alloc = alloca { i32*, i32* }*
88.  store { i32*, i32* }* %0, { i32*, i32* }** %list_ptr_alloc
89.  %list_load = load { i32*, i32* }*, { i32*, i32* }** %list_ptr_alloc
90.  %list_array_ptr = getelementptr inbounds { i32*, i32* }, { i32*, i32* }* %list_load,
    i32 0, i32 1
91.  %list_array_load = load i32*, i32** %list_array_ptr
92.  %list_array_next_element_ptr = getelementptr i32, i32* %list_array_load, i32 %1
93.  store i32 %2, i32* %list_array_next_element_ptr
94.  ret void
95. }
96.
97. define void @list_setfloat({ i32*, double* }*, i32, double) {
98. entry:
99.  %list_ptr_alloc = alloca { i32*, double* }*
100.  store { i32*, double* }* %0, { i32*, double* }** %list_ptr_alloc
101.  %list_load = load { i32*, double* }*, { i32*, double* }** %list_ptr_alloc
102.  %list_array_ptr = getelementptr inbounds { i32*, double* }, { i32*,
    double* }* %list_load, i32 0, i32 1

```

```

103.  %list_array_load = load double*, double** %list_array_ptr
104.  %list_array_next_element_ptr = getelementptr double, double* %list_array_load, i32 %1
105.  store double %2, double* %list_array_next_element_ptr
106.  ret void
107. }
108.
109. define void @list_setstr({ i32*, i8** }*, i32, i8*) {
110. entry:
111.  %list_ptr_alloc = alloca { i32*, i8** }*
112.  store { i32*, i8** }* %0, { i32*, i8** }** %list_ptr_alloc
113.  %list_load = load { i32*, i8** }*, { i32*, i8** }** %list_ptr_alloc
114.  %list_array_ptr = getelementptr inbounds { i32*, i8** }, { i32*, i8** }* %list_load,
    i32 0, i32 1
115.  %list_array_load = load i8**, i8** %list_array_ptr
116.  %list_array_next_element_ptr = getelementptr i8*, i8** %list_array_load, i32 %1
117.  store i8* %2, i8** %list_array_next_element_ptr
118.  ret void
119. }
120.
121. define void @list_pushbool({ i32*, i1* }*, i1) {
122. entry:
123.  %list_ptr_alloc = alloca { i32*, i1* }*
124.  store { i32*, i1* }* %0, { i32*, i1* }** %list_ptr_alloc
125.  %val_alloc = alloca i1
126.  store i1 %1, i1* %val_alloc
127.  %list_load = load { i32*, i1* }*, { i32*, i1* }** %list_ptr_alloc
128.  %list_array_ptr = getelementptr inbounds { i32*, i1* }, { i32*, i1* }* %list_load,
    i32 0, i32 1
129.  %list_array_load = load i1*, i1** %list_array_ptr
130.  %list_size_ptr_ptr = getelementptr inbounds { i32*, i1* }, { i32*, i1* }* %list_load,
    i32 0, i32 0
131.  %list_size_ptr = load i32*, i32** %list_size_ptr_ptr
132.  %list_size = load i32, i32* %list_size_ptr
133.  %list_array_next_element_ptr = getelementptr i1, i1* %list_array_load, i32 %list_size
134.  %inc_size = add i32 %list_size, 1
135.  store i32 %inc_size, i32* %list_size_ptr
136.  %val = load i1, i1* %val_alloc
137.  store i1 %val, i1* %list_array_next_element_ptr
138.  ret void
139. }
140.
141. define void @list_pushint({ i32*, i32* }*, i32) {
142. entry:
143.  %list_ptr_alloc = alloca { i32*, i32* }*
144.  store { i32*, i32* }* %0, { i32*, i32* }** %list_ptr_alloc
145.  %val_alloc = alloca i32
146.  store i32 %1, i32* %val_alloc
147.  %list_load = load { i32*, i32* }*, { i32*, i32* }** %list_ptr_alloc
148.  %list_array_ptr = getelementptr inbounds { i32*, i32* }, { i32*, i32* }* %list_load,
    i32 0, i32 1
149.  %list_array_load = load i32*, i32** %list_array_ptr

```

```

150.  %list_size_ptr_ptr = getelementptr inbounds { i32*, i32* }, { i32*,
    i32* }* %list_load, i32 0, i32 0
151.  %list_size_ptr = load i32*, i32** %list_size_ptr_ptr
152.  %list_size = load i32, i32* %list_size_ptr
153.  %list_array_next_element_ptr = getelementptr i32, i32* %list_array_load,
    i32 %list_size
154.  %inc_size = add i32 %list_size, 1
155.  store i32 %inc_size, i32* %list_size_ptr
156.  %val = load i32, i32* %val_alloc
157.  store i32 %val, i32* %list_array_next_element_ptr
158.  ret void
159. }
160.
161. define void @list_pushfloat({ i32*, double* }*, double) {
162. entry:
163.  %list_ptr_alloc = alloca { i32*, double* }*
164.  store { i32*, double* }* %0, { i32*, double* }** %list_ptr_alloc
165.  %val_alloc = alloca double
166.  store double %1, double* %val_alloc
167.  %list_load = load { i32*, double* }*, { i32*, double* }** %list_ptr_alloc
168.  %list_array_ptr = getelementptr inbounds { i32*, double* }, { i32*,
    double* }* %list_load, i32 0, i32 1
169.  %list_array_load = load double*, double** %list_array_ptr
170.  %list_size_ptr_ptr = getelementptr inbounds { i32*, double* }, { i32*,
    double* }* %list_load, i32 0, i32 0
171.  %list_size_ptr = load i32*, i32** %list_size_ptr_ptr
172.  %list_size = load i32, i32* %list_size_ptr
173.  %list_array_next_element_ptr = getelementptr double, double* %list_array_load,
    i32 %list_size
174.  %inc_size = add i32 %list_size, 1
175.  store i32 %inc_size, i32* %list_size_ptr
176.  %val = load double, double* %val_alloc
177.  store double %val, double* %list_array_next_element_ptr
178.  ret void
179. }
180.
181. define void @list_pushstr({ i32*, i8** }*, i8*) {
182. entry:
183.  %list_ptr_alloc = alloca { i32*, i8** }*
184.  store { i32*, i8** }* %0, { i32*, i8** }** %list_ptr_alloc
185.  %val_alloc = alloca i8*
186.  store i8* %1, i8** %val_alloc
187.  %list_load = load { i32*, i8** }*, { i32*, i8** }** %list_ptr_alloc
188.  %list_array_ptr = getelementptr inbounds { i32*, i8** }, { i32*, i8** }* %list_load,
    i32 0, i32 1
189.  %list_array_load = load i8**, i8*** %list_array_ptr
190.  %list_size_ptr_ptr = getelementptr inbounds { i32*, i8** }, { i32*,
    i8** }* %list_load, i32 0, i32 0
191.  %list_size_ptr = load i32*, i32** %list_size_ptr_ptr
192.  %list_size = load i32, i32* %list_size_ptr
193.  %list_array_next_element_ptr = getelementptr i8*, i8** %list_array_load,
    i32 %list_size
194.  %inc_size = add i32 %list_size, 1

```

```

195.  store i32 %inc_size, i32* %list_size_ptr
196.  %val = load i8*, i8** %val_alloc
197.  store i8* %val, i8** %list_array_next_element_ptr
198.  ret void
199. }
200.
201. define i1 @list_popbool({ i32*, i1* }*) {
202. entry:
203.  %list_ptr_alloc = alloca { i32*, i1* }*
204.  store { i32*, i1* }* %0, { i32*, i1* }** %list_ptr_alloc
205.  %list_load = load { i32*, i1* }*, { i32*, i1* }** %list_ptr_alloc
206.  %list_array_ptr = getelementptr inbounds { i32*, i1* }, { i32*, i1* }* %list_load,
    i32 0, i32 1
207.  %list_array_load = load i1*, i1** %list_array_ptr
208.  %list_size_ptr_ptr = getelementptr inbounds { i32*, i1* }, { i32*, i1* }* %list_load,
    i32 0, i32 0
209.  %list_size_ptr = load i32*, i32** %list_size_ptr_ptr
210.  %list_size = load i32, i32* %list_size_ptr
211.  %dec_size = sub i32 %list_size, 1
212.  %list_array_next_element_ptr = getelementptr i1, i1* %list_array_load, i32 %dec_size
213.  %list_array_next_element = load i1, i1* %list_array_next_element_ptr
214.  store i32 %dec_size, i32* %list_size_ptr
215.  ret i1 %list_array_next_element
216. }
217.
218. define i32 @list_popint({ i32*, i32* }*) {
219. entry:
220.  %list_ptr_alloc = alloca { i32*, i32* }*
221.  store { i32*, i32* }* %0, { i32*, i32* }** %list_ptr_alloc
222.  %list_load = load { i32*, i32* }*, { i32*, i32* }** %list_ptr_alloc
223.  %list_array_ptr = getelementptr inbounds { i32*, i32* }, { i32*, i32* }* %list_load,
    i32 0, i32 1
224.  %list_array_load = load i32*, i32** %list_array_ptr
225.  %list_size_ptr_ptr = getelementptr inbounds { i32*, i32* }, { i32*,
    i32* }* %list_load, i32 0, i32 0
226.  %list_size_ptr = load i32*, i32** %list_size_ptr_ptr
227.  %list_size = load i32, i32* %list_size_ptr
228.  %dec_size = sub i32 %list_size, 1
229.  %list_array_next_element_ptr = getelementptr i32, i32* %list_array_load, i32 %dec_size
230.  %list_array_next_element = load i32, i32* %list_array_next_element_ptr
231.  store i32 %dec_size, i32* %list_size_ptr
232.  ret i32 %list_array_next_element
233. }
234.
235. define double @list_popfloat({ i32*, double* }*) {
236. entry:
237.  %list_ptr_alloc = alloca { i32*, double* }*
238.  store { i32*, double* }* %0, { i32*, double* }** %list_ptr_alloc
239.  %list_load = load { i32*, double* }*, { i32*, double* }** %list_ptr_alloc
240.  %list_array_ptr = getelementptr inbounds { i32*, double* }, { i32*,
    double* }* %list_load, i32 0, i32 1
241.  %list_array_load = load double*, double** %list_array_ptr

```

```

242.  %list_size_ptr_ptr = getelementptr inbounds { i32*, double* }, { i32*,
      double* }* %list_load, i32 0, i32 0
243.  %list_size_ptr = load i32*, i32** %list_size_ptr_ptr
244.  %list_size = load i32, i32* %list_size_ptr
245.  %dec_size = sub i32 %list_size, 1
246.  %list_array_next_element_ptr = getelementptr double, double* %list_array_load,
      i32 %dec_size
247.  %list_array_next_element = load double, double* %list_array_next_element_ptr
248.  store i32 %dec_size, i32* %list_size_ptr
249.  ret double %list_array_next_element
250. }
251.
252. define i8* @list_popstr({ i32*, i8** }*) {
253. entry:
254.  %list_ptr_alloc = alloca { i32*, i8** }*
255.  store { i32*, i8** }* %0, { i32*, i8** }** %list_ptr_alloc
256.  %list_load = load { i32*, i8** }*, { i32*, i8** }** %list_ptr_alloc
257.  %list_array_ptr = getelementptr inbounds { i32*, i8** }, { i32*, i8** }* %list_load,
      i32 0, i32 1
258.  %list_array_load = load i8**, i8*** %list_array_ptr
259.  %list_size_ptr_ptr = getelementptr inbounds { i32*, i8** }, { i32*,
      i8** }* %list_load, i32 0, i32 0
260.  %list_size_ptr = load i32*, i32** %list_size_ptr_ptr
261.  %list_size = load i32, i32* %list_size_ptr
262.  %dec_size = sub i32 %list_size, 1
263.  %list_array_next_element_ptr = getelementptr i8*, i8** %list_array_load, i32 %dec_size
264.  %list_array_next_element = load i8*, i8** %list_array_next_element_ptr
265.  store i32 %dec_size, i32* %list_size_ptr
266.  ret i8* %list_array_next_element
267. }
268.
269. define i32 @list_sizebool({ i32*, i1* }*) {
270. entry:
271.  %list_ptr_alloc = alloca { i32*, i1* }*
272.  store { i32*, i1* }* %0, { i32*, i1* }** %list_ptr_alloc
273.  %list_load = load { i32*, i1* }*, { i32*, i1* }** %list_ptr_alloc
274.  %list_size_ptr_ptr = getelementptr inbounds { i32*, i1* }, { i32*, i1* }* %list_load,
      i32 0, i32 0
275.  %list_size_ptr = load i32*, i32** %list_size_ptr_ptr
276.  %list_size = load i32, i32* %list_size_ptr
277.  ret i32 %list_size
278. }
279.
280. define i32 @list_sizeint({ i32*, i32* }*) {
281. entry:
282.  %list_ptr_alloc = alloca { i32*, i32* }*
283.  store { i32*, i32* }* %0, { i32*, i32* }** %list_ptr_alloc
284.  %list_load = load { i32*, i32* }*, { i32*, i32* }** %list_ptr_alloc
285.  %list_size_ptr_ptr = getelementptr inbounds { i32*, i32* }, { i32*,
      i32* }* %list_load, i32 0, i32 0
286.  %list_size_ptr = load i32*, i32** %list_size_ptr_ptr
287.  %list_size = load i32, i32* %list_size_ptr

```

```

288.  ret i32 %list_size
289.  }
290.
291.  define i32 @list_sizefloat({ i32*, double* }*) {
292.  entry:
293.    %list_ptr_alloc = alloca { i32*, double* }*
294.    store { i32*, double* }* %0, { i32*, double* }** %list_ptr_alloc
295.    %list_load = load { i32*, double* }*, { i32*, double* }** %list_ptr_alloc
296.    %list_size_ptr_ptr = getelementptr inbounds { i32*, double* }, { i32*,
      double* }* %list_load, i32 0, i32 0
297.    %list_size_ptr = load i32*, i32** %list_size_ptr_ptr
298.    %list_size = load i32, i32* %list_size_ptr
299.    ret i32 %list_size
300.  }
301.
302.  define i32 @list_sizestr({ i32*, i8** }*) {
303.  entry:
304.    %list_ptr_alloc = alloca { i32*, i8** }*
305.    store { i32*, i8** }* %0, { i32*, i8** }** %list_ptr_alloc
306.    %list_load = load { i32*, i8** }*, { i32*, i8** }** %list_ptr_alloc
307.    %list_size_ptr_ptr = getelementptr inbounds { i32*, i8** }, { i32*,
      i8** }* %list_load, i32 0, i32 0
308.    %list_size_ptr = load i32*, i32** %list_size_ptr_ptr
309.    %list_size = load i32, i32* %list_size_ptr
310.    ret i32 %list_size
311.  }
312.
313.  define void @list_slicebool({ i32*, i1* }*, { i32*, i1* }*, i32, i32) {
314.  entry:
315.    %list_ptr_alloc = alloca { i32*, i1* }*
316.    store { i32*, i1* }* %0, { i32*, i1* }** %list_ptr_alloc
317.    %list_ptr_ptr = load { i32*, i1* }*, { i32*, i1* }** %list_ptr_alloc
318.    %list_ptr_alloc2 = alloca { i32*, i1* }*
319.    store { i32*, i1* }* %1, { i32*, i1* }** %list_ptr_alloc2
320.    %list_ptr_ptr2 = load { i32*, i1* }*, { i32*, i1* }** %list_ptr_alloc2
321.    %idx_alloc = alloca i32
322.    store i32 %2, i32* %idx_alloc
323.    %idx_load = load i32, i32* %idx_alloc
324.    %idx_alloc1 = alloca i32
325.    store i32 %3, i32* %idx_alloc1
326.    %idx_load2 = load i32, i32* %idx_alloc1
327.    %loop_cnt = alloca i32
328.    store i32 0, i32* %loop_cnt
329.    %loop_upper_bound = sub i32 %idx_load2, %idx_load
330.    br label %while
331.
332.  while:                                     ; preds = %while_body, %entry
333.    %loop_cnt4 = load i32, i32* %loop_cnt
334.    %loop_cond = icmp sle i32 %loop_cnt4, %loop_upper_bound
335.    br i1 %loop_cond, label %while_body, label %merge
336.
337.  while_body:                               ; preds = %while

```



```

338. %to_idx = load i32, i32* %loop_cnt
339. %from_idx = add i32 %to_idx, %idx_load
340. %list_get = call i1 @list_getbool({ i32*, i1* }* %list_ptr_ptr, i32 %from_idx)
341. call void @list_pushbool({ i32*, i1* }* %list_ptr_ptr2, i1 %list_get)
342. %loop_cnt3 = load i32, i32* %loop_cnt
343. %loop_itr = add i32 %loop_cnt3, 1
344. store i32 %loop_itr, i32* %loop_cnt
345. br label %while
346.
347. merge:                                     ; preds = %while
348. ret void
349. }
350.
351. define void @list_sliceint({ i32*, i32* }*, { i32*, i32* }*, i32, i32) {
352. entry:
353. %list_ptr_alloc = alloca { i32*, i32* }*
354. store { i32*, i32* }* %0, { i32*, i32* }** %list_ptr_alloc
355. %list_ptr_ptr = load { i32*, i32* }*, { i32*, i32* }** %list_ptr_alloc
356. %list_ptr_alloc2 = alloca { i32*, i32* }*
357. store { i32*, i32* }* %1, { i32*, i32* }** %list_ptr_alloc2
358. %list_ptr_ptr2 = load { i32*, i32* }*, { i32*, i32* }** %list_ptr_alloc2
359. %idx_alloc = alloca i32
360. store i32 %2, i32* %idx_alloc
361. %idx_load = load i32, i32* %idx_alloc
362. %idx_alloc1 = alloca i32
363. store i32 %3, i32* %idx_alloc1
364. %idx_load2 = load i32, i32* %idx_alloc1
365. %loop_cnt = alloca i32
366. store i32 0, i32* %loop_cnt
367. %loop_upper_bound = sub i32 %idx_load2, %idx_load
368. br label %while
369.
370. while:                                     ; preds = %while_body, %entry
371. %loop_cnt4 = load i32, i32* %loop_cnt
372. %loop_cond = icmp sle i32 %loop_cnt4, %loop_upper_bound
373. br i1 %loop_cond, label %while_body, label %merge
374.
375. while_body:                                 ; preds = %while
376. %to_idx = load i32, i32* %loop_cnt
377. %from_idx = add i32 %to_idx, %idx_load
378. %list_get = call i32 @list_getint({ i32*, i32* }* %list_ptr_ptr, i32 %from_idx)
379. call void @list_pushint({ i32*, i32* }* %list_ptr_ptr2, i32 %list_get)
380. %loop_cnt3 = load i32, i32* %loop_cnt
381. %loop_itr = add i32 %loop_cnt3, 1
382. store i32 %loop_itr, i32* %loop_cnt
383. br label %while
384.
385. merge:                                     ; preds = %while
386. ret void
387. }
388.
389. define void @list_slicefloat({ i32*, double* }*, { i32*, double* }*, i32, i32) {

```

```

390. entry:
391.  %list_ptr_alloc = alloca { i32*, double* }*
392.  store { i32*, double* }* %0, { i32*, double* }** %list_ptr_alloc
393.  %list_ptr_ptr = load { i32*, double* }*, { i32*, double* }** %list_ptr_alloc
394.  %list_ptr_alloc2 = alloca { i32*, double* }*
395.  store { i32*, double* }* %1, { i32*, double* }** %list_ptr_alloc2
396.  %list_ptr_ptr2 = load { i32*, double* }*, { i32*, double* }** %list_ptr_alloc2
397.  %idx_alloc = alloca i32
398.  store i32 %2, i32* %idx_alloc
399.  %idx_load = load i32, i32* %idx_alloc
400.  %idx_alloc1 = alloca i32
401.  store i32 %3, i32* %idx_alloc1
402.  %idx_load2 = load i32, i32* %idx_alloc1
403.  %loop_cnt = alloca i32
404.  store i32 0, i32* %loop_cnt
405.  %loop_upper_bound = sub i32 %idx_load2, %idx_load
406.  br label %while
407.
408. while:                                     ; preds = %while_body, %entry
409.  %loop_cnt4 = load i32, i32* %loop_cnt
410.  %loop_cond = icmp sle i32 %loop_cnt4, %loop_upper_bound
411.  br i1 %loop_cond, label %while_body, label %merge
412.
413. while_body:                               ; preds = %while
414.  %to_idx = load i32, i32* %loop_cnt
415.  %from_idx = add i32 %to_idx, %idx_load
416.  %list_get = call double @list_getfloat({ i32*, double* }* %list_ptr_ptr,
    i32 %from_idx)
417.  call void @list_pushfloat({ i32*, double* }* %list_ptr_ptr2, double %list_get)
418.  %loop_cnt3 = load i32, i32* %loop_cnt
419.  %loop_itr = add i32 %loop_cnt3, 1
420.  store i32 %loop_itr, i32* %loop_cnt
421.  br label %while
422.
423. merge:                                     ; preds = %while
424.  ret void
425. }
426.
427. define void @list_sliceistr({ i32*, i8** }*, { i32*, i8** }*, i32, i32) {
428. entry:
429.  %list_ptr_alloc = alloca { i32*, i8** }*
430.  store { i32*, i8** }* %0, { i32*, i8** }** %list_ptr_alloc
431.  %list_ptr_ptr = load { i32*, i8** }*, { i32*, i8** }** %list_ptr_alloc
432.  %list_ptr_alloc2 = alloca { i32*, i8** }*
433.  store { i32*, i8** }* %1, { i32*, i8** }** %list_ptr_alloc2
434.  %list_ptr_ptr2 = load { i32*, i8** }*, { i32*, i8** }** %list_ptr_alloc2
435.  %idx_alloc = alloca i32
436.  store i32 %2, i32* %idx_alloc
437.  %idx_load = load i32, i32* %idx_alloc
438.  %idx_alloc1 = alloca i32
439.  store i32 %3, i32* %idx_alloc1
440.  %idx_load2 = load i32, i32* %idx_alloc1

```

```

441.  %loop_cnt = alloca i32
442.  store i32 0, i32* %loop_cnt
443.  %loop_upper_bound = sub i32 %idx_load2, %idx_load
444.  br label %while
445.
446. while:                                     ; preds = %while_body, %entry
447.  %loop_cnt4 = load i32, i32* %loop_cnt
448.  %loop_cond = icmp sle i32 %loop_cnt4, %loop_upper_bound
449.  br i1 %loop_cond, label %while_body, label %merge
450.
451. while_body:                               ; preds = %while
452.  %to_idx = load i32, i32* %loop_cnt
453.  %from_idx = add i32 %to_idx, %idx_load
454.  %list_get = call i8* @list_getstr({ i32*, i8** }* %list_ptr_ptr, i32 %from_idx)
455.  call void @list_pushstr({ i32*, i8** }* %list_ptr_ptr2, i8* %list_get)
456.  %loop_cnt3 = load i32, i32* %loop_cnt
457.  %loop_itr = add i32 %loop_cnt3, 1
458.  store i32 %loop_itr, i32* %loop_cnt
459.  br label %while
460.
461. merge:                                     ; preds = %while
462.  ret void
463. }
464.
465. define i32 @list_findbool({ i32*, i1* }*, i1) {
466. entry:
467.  %list_ptr_alloc = alloca { i32*, i1* }*
468.  store { i32*, i1* }* %0, { i32*, i1* }** %list_ptr_alloc
469.  %find_val_alloc = alloca i1
470.  store i1 %1, i1* %find_val_alloc
471.  %find_val = load i1, i1* %find_val_alloc
472.  %list_load = load { i32*, i1* }*, { i32*, i1* }** %list_ptr_alloc
473.  %list_size_ptr_ptr = getelementptr inbounds { i32*, i1* }, { i32*, i1* }* %list_load,
    i32 0, i32 0
474.  %list_size_ptr = load i32*, i32** %list_size_ptr_ptr
475.  %list_size = load i32, i32* %list_size_ptr
476.  %loop_cnt = alloca i32
477.  store i32 0, i32* %loop_cnt
478.  br label %while
479.
480. while:                                     ; preds = %merge, %entry
481.  %loop_iter_cnt = load i32, i32* %loop_cnt
482.  %loop_cond = icmp slt i32 %loop_iter_cnt, %list_size
483.  br i1 %loop_cond, label %while_body, label %merge1
484.
485. while_body:                               ; preds = %while
486.  %to_idx = load i32, i32* %loop_cnt
487.  %list_get = call i1 @list_getbool({ i32*, i1* }* %list_load, i32 %to_idx)
488.  %if_cond = icmp eq i1 %list_get, %find_val
489.  br i1 %if_cond, label %then, label %else
490.
491. merge:                                     ; preds = %else

```

```

492.  %loop_idx = load i32, i32* %loop_cnt
493.  %loop_itr = add i32 %loop_idx, 1
494.  store i32 %loop_itr, i32* %loop_cnt
495.  br label %while

496.
497. then:                                ; preds = %while_body
498.  ret i32 %to_idx
499.
500. else:                                  ; preds = %while_body
501.  br label %merge
502.
503. merge1:                                ; preds = %while
504.  ret i32 -1
505. }
506.
507. define i32 @list_findint({ i32*, i32* }*, i32) {
508. entry:
509.  %list_ptr_alloc = alloca { i32*, i32* }*
510.  store { i32*, i32* }* %0, { i32*, i32* }** %list_ptr_alloc
511.  %find_val_alloc = alloca i32
512.  store i32 %1, i32* %find_val_alloc
513.  %find_val = load i32, i32* %find_val_alloc
514.  %list_load = load { i32*, i32* }*, { i32*, i32* }** %list_ptr_alloc
515.  %list_size_ptr_ptr = getelementptr inbounds { i32*, i32* }, { i32*,
    i32* }* %list_load, i32 0, i32 0
516.  %list_size_ptr = load i32*, i32** %list_size_ptr_ptr
517.  %list_size = load i32, i32* %list_size_ptr
518.  %loop_cnt = alloca i32
519.  store i32 0, i32* %loop_cnt
520.  br label %while

521.
522. while:                                  ; preds = %merge, %entry
523.  %loop_iter_cnt = load i32, i32* %loop_cnt
524.  %loop_cond = icmp slt i32 %loop_iter_cnt, %list_size
525.  br i1 %loop_cond, label %while_body, label %merge1

526.
527. while_body:                              ; preds = %while
528.  %to_idx = load i32, i32* %loop_cnt
529.  %list_get = call i32 @list_getint({ i32*, i32* }* %list_load, i32 %to_idx)
530.  %if_cond = icmp eq i32 %list_get, %find_val
531.  br i1 %if_cond, label %then, label %else

532.
533. merge:                                  ; preds = %else
534.  %loop_idx = load i32, i32* %loop_cnt
535.  %loop_itr = add i32 %loop_idx, 1
536.  store i32 %loop_itr, i32* %loop_cnt
537.  br label %while

538.
539. then:                                    ; preds = %while_body
540.  ret i32 %to_idx
541.

```

```

542. else:                                     ; preds = %while_body
543.   br label %merge
544.
545. merge1:                                     ; preds = %while
546.   ret i32 -1
547. }
548.
549. define i32 @list_findfloat({ i32*, double* }*, double) {
550. entry:
551.   %list_ptr_alloc = alloca { i32*, double* }*
552.   store { i32*, double* }* %0, { i32*, double* }** %list_ptr_alloc
553.   %find_val_alloc = alloca double
554.   store double %1, double* %find_val_alloc
555.   %find_val = load double, double* %find_val_alloc
556.   %list_load = load { i32*, double* }*, { i32*, double* }** %list_ptr_alloc
557.   %list_size_ptr_ptr = getelementptr inbounds { i32*, double* }, { i32*,
double* }* %list_load, i32 0, i32 0
558.   %list_size_ptr = load i32*, i32** %list_size_ptr_ptr
559.   %list_size = load i32, i32* %list_size_ptr
560.   %loop_cnt = alloca i32
561.   store i32 0, i32* %loop_cnt
562.   br label %while
563.
564. while:                                       ; preds = %merge, %entry
565.   %loop_iter_cnt = load i32, i32* %loop_cnt
566.   %loop_cond = icmp slt i32 %loop_iter_cnt, %list_size
567.   br i1 %loop_cond, label %while_body, label %merge1
568.
569. while_body:                                  ; preds = %while
570.   %to_idx = load i32, i32* %loop_cnt
571.   %list_get = call double @list_getfloat({ i32*, double* }* %list_load, i32 %to_idx)
572.   %if_cond = fcmp oeq double %list_get, %find_val
573.   br i1 %if_cond, label %then, label %else
574.
575. merge:                                       ; preds = %else
576.   %loop_idx = load i32, i32* %loop_cnt
577.   %loop_itr = add i32 %loop_idx, 1
578.   store i32 %loop_itr, i32* %loop_cnt
579.   br label %while
580.
581. then:                                       ; preds = %while_body
582.   ret i32 %to_idx
583.
584. else:                                       ; preds = %while_body
585.   br label %merge
586.
587. merge1:                                     ; preds = %while
588.   ret i32 -1
589. }
590.
591. define void @list_removebool({ i32*, i1* }*, i1) {

```

```

592. entry:
593.  %list_ptr_alloc = alloca { i32*, i1* }*
594.  store { i32*, i1* }* %0, { i32*, i1* }** %list_ptr_alloc
595.  %rem_val_ptr = alloca i1
596.  store i1 %1, i1* %rem_val_ptr
597.  %rem_val = load i1, i1* %rem_val_ptr
598.  %list_load = load { i32*, i1* }*, { i32*, i1* }** %list_ptr_alloc
599.  %list_size_ptr_ptr = getelementptr inbounds { i32*, i1* }, { i32*, i1* }* %list_load,
    i32 0, i32 0
600.  %list_size_ptr = load i32*, i32** %list_size_ptr_ptr
601.  %list_size = load i32, i32* %list_size_ptr
602.  %list_find = call i32 @list_findbool({ i32*, i1* }* %list_load, i1 %rem_val)
603.  %loop_cond = icmp sge i32 %list_find, 0
604.  br i1 %loop_cond, label %then, label %else
605.
606. merge:                                ; preds = %else
607.  ret void
608.
609. then:                                  ; preds = %entry
610.  %loop_cnt_ptr = alloca i32
611.  %loop_start_idx = add i32 %list_find, 1
612.  store i32 %loop_start_idx, i32* %loop_cnt_ptr
613.  br label %while
614.
615. while:                                  ; preds = %while_body, %then
616.  %loop_cnt = load i32, i32* %loop_cnt_ptr
617.  %loop_cond1 = icmp slt i32 %loop_cnt, %list_size
618.  br i1 %loop_cond1, label %while_body, label %merge2
619.
620. while_body:                             ; preds = %while
621.  %cur_idx = load i32, i32* %loop_cnt_ptr
622.  %shift_to_idx = sub i32 %cur_idx, 1
623.  %list_get = call i1 @list_getbool({ i32*, i1* }* %list_load, i32 %cur_idx)
624.  call void @list_setbool({ i32*, i1* }* %list_load, i32 %shift_to_idx, i1 %list_get)
625.  %loop_itr = add i32 %cur_idx, 1
626.  store i32 %loop_itr, i32* %loop_cnt_ptr
627.  br label %while
628.
629. merge2:                                  ; preds = %while
630.  %size_dec = sub i32 %list_size, 1
631.  store i32 %size_dec, i32* %list_size_ptr
632.  ret void
633.
634. else:                                    ; preds = %entry
635.  br label %merge
636. }
637.
638. define void @list_removeint({ i32*, i32* }*, i32) {
639. entry:
640.  %list_ptr_alloc = alloca { i32*, i32* }*
641.  store { i32*, i32* }* %0, { i32*, i32* }** %list_ptr_alloc
642.  %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

```



```

743.  %list_size_ptr_ptr = getelementptr inbounds { i32*, i1* }, { i32*, i1* }* %list_load,
      i32 0, i32 0
744.  %list_size_ptr = load i32*, i32** %list_size_ptr_ptr
745.  %list_size = load i32, i32* %list_size_ptr
746.  %loop_cnt_ptr = alloca i32
747.  %last_index = sub i32 %list_size, 1
748.  store i32 %last_index, i32* %loop_cnt_ptr
749.  br label %while
750.
751. while:                                     ; preds = %while_body, %entry
752.  %loop_cnt = load i32, i32* %loop_cnt_ptr
753.  %loop_cond = icmp sge i32 %loop_cnt, %insert_idx
754.  br i1 %loop_cond, label %while_body, label %merge
755.
756. while_body:                               ; preds = %while
757.  %cur_idx = load i32, i32* %loop_cnt_ptr
758.  %shift_to_idx = add i32 %cur_idx, 1
759.  %list_get = call i1 @list_getbool({ i32*, i1* }* %list_load, i32 %cur_idx)
760.  call void @list_setbool({ i32*, i1* }* %list_load, i32 %shift_to_idx, i1 %list_get)
761.  %loop_itr = sub i32 %cur_idx, 1
762.  store i32 %loop_itr, i32* %loop_cnt_ptr
763.  br label %while
764.
765. merge:                                     ; preds = %while
766.  call void @list_setbool({ i32*, i1* }* %list_load, i32 %insert_idx, i1 %insert_val)
767.  %size_inc = add i32 %list_size, 1
768.  store i32 %size_inc, i32* %list_size_ptr
769.  ret void
770. }
771.
772. define void @list_insertint({ i32*, i32* }*, i32, i32) {
773. entry:
774.  %list_ptr_alloc = alloca { i32*, i32* }*
775.  store { i32*, i32* }* %0, { i32*, i32* }** %list_ptr_alloc
776.  %list_load = load { i32*, i32* }*, { i32*, i32* }** %list_ptr_alloc
777.  %insert_idx_ptr = alloca i32
778.  store i32 %1, i32* %insert_idx_ptr
779.  %insert_idx = load i32, i32* %insert_idx_ptr
780.  %insert_val_ptr = alloca i32
781.  store i32 %2, i32* %insert_val_ptr
782.  %insert_val = load i32, i32* %insert_val_ptr
783.  %list_size_ptr_ptr = getelementptr inbounds { i32*, i32* }, { i32*,
      i32* }* %list_load, i32 0, i32 0
784.  %list_size_ptr = load i32*, i32** %list_size_ptr_ptr
785.  %list_size = load i32, i32* %list_size_ptr
786.  %loop_cnt_ptr = alloca i32
787.  %last_index = sub i32 %list_size, 1
788.  store i32 %last_index, i32* %loop_cnt_ptr
789.  br label %while
790.
791. while:                                     ; preds = %while_body, %entry
792.  %loop_cnt = load i32, i32* %loop_cnt_ptr

```

```

793.  %loop_cond = icmp sge i32 %loop_cnt, %insert_idx
794.  br i1 %loop_cond, label %while_body, label %merge
795.
796. while_body:                                     ; preds = %while
797.  %cur_idx = load i32, i32* %loop_cnt_ptr
798.  %shift_to_idx = add i32 %cur_idx, 1
799.  %list_get = call i32 @list_getint({ i32*, i32* }* %list_load, i32 %cur_idx)
800.  call void @list_setint({ i32*, i32* }* %list_load, i32 %shift_to_idx, i32 %list_get)
801.  %loop_itr = sub i32 %cur_idx, 1
802.  store i32 %loop_itr, i32* %loop_cnt_ptr
803.  br label %while
804.
805. merge:                                           ; preds = %while
806.  call void @list_setint({ i32*, i32* }* %list_load, i32 %insert_idx, i32 %insert_val)
807.  %size_inc = add i32 %list_size, 1
808.  store i32 %size_inc, i32* %list_size_ptr
809.  ret void
810. }
811.
812. define void @list_insertfloat({ i32*, double* }*, i32, double) {
813. entry:
814.  %list_ptr_alloc = alloca { i32*, double* }*
815.  store { i32*, double* }* %0, { i32*, double* }** %list_ptr_alloc
816.  %list_load = load { i32*, double* }*, { i32*, double* }** %list_ptr_alloc
817.  %insert_idx_ptr = alloca i32
818.  store i32 %1, i32* %insert_idx_ptr
819.  %insert_idx = load i32, i32* %insert_idx_ptr
820.  %insert_val_ptr = alloca double
821.  store double %2, double* %insert_val_ptr
822.  %insert_val = load double, double* %insert_val_ptr
823.  %list_size_ptr_ptr = getelementptr inbounds { i32*, double* }, { i32*,
      double* }* %list_load, i32 0, i32 0
824.  %list_size_ptr = load i32*, i32** %list_size_ptr_ptr
825.  %list_size = load i32, i32* %list_size_ptr
826.  %loop_cnt_ptr = alloca i32
827.  %last_index = sub i32 %list_size, 1
828.  store i32 %last_index, i32* %loop_cnt_ptr
829.  br label %while
830.
831. while:                                           ; preds = %while_body, %entry
832.  %loop_cnt = load i32, i32* %loop_cnt_ptr
833.  %loop_cond = icmp sge i32 %loop_cnt, %insert_idx
834.  br i1 %loop_cond, label %while_body, label %merge
835.
836. while_body:                                     ; preds = %while
837.  %cur_idx = load i32, i32* %loop_cnt_ptr
838.  %shift_to_idx = add i32 %cur_idx, 1
839.  %list_get = call double @list_getfloat({ i32*, double* }* %list_load, i32 %cur_idx)
840.  call void @list_setfloat({ i32*, double* }* %list_load, i32 %shift_to_idx,
      double %list_get)
841.  %loop_itr = sub i32 %cur_idx, 1
842.  store i32 %loop_itr, i32* %loop_cnt_ptr

```

```

843. br label %while
844.
845. merge:                                     ; preds = %while
846. call void @list_setfloat({ i32*, double* }* %list_load, i32 %insert_idx,
    double %insert_val)
847. %size_inc = add i32 %list_size, 1
848. store i32 %size_inc, i32* %list_size_ptr
849. ret void
850. }
851.
852. define void @list_reversebool({ i32*, i1* }*) {
853. entry:
854. %list_ptr_alloc = alloca { i32*, i1* }*
855. store { i32*, i1* }* %0, { i32*, i1* }** %list_ptr_alloc
856. %list_load = load { i32*, i1* }*, { i32*, i1* }** %list_ptr_alloc
857. %list_size_ptr_ptr = getelementptr inbounds { i32*, i1* }, { i32*, i1* }* %list_load,
    i32 0, i32 0
858. %list_size_ptr = load i32*, i32** %list_size_ptr_ptr
859. %list_size = load i32, i32* %list_size_ptr
860. %left_idx = alloca i32
861. store i32 0, i32* %left_idx
862. %right_idx = alloca i32
863. %tmp = sub i32 %list_size, 1
864. store i32 %tmp, i32* %right_idx
865. br label %while
866.
867. while:                                     ; preds = %while_body, %entry
868. %right_idx6 = load i32, i32* %right_idx
869. %left_idx7 = load i32, i32* %left_idx
870. %while_cond = icmp slt i32 %left_idx7, %right_idx6
871. br i1 %while_cond, label %while_body, label %merge
872.
873. while_body:                               ; preds = %while
874. %left_idx1 = load i32, i32* %left_idx
875. %right_idx2 = load i32, i32* %right_idx
876. %list_get = call i1 @list_getbool({ i32*, i1* }* %list_load, i32 %left_idx1)
877. %list_get3 = call i1 @list_getbool({ i32*, i1* }* %list_load, i32 %right_idx2)
878. call void @list_setbool({ i32*, i1* }* %list_load, i32 %left_idx1, i1 %list_get3)
879. call void @list_setbool({ i32*, i1* }* %list_load, i32 %right_idx2, i1 %list_get)
880. %tmp4 = add i32 %left_idx1, 1
881. store i32 %tmp4, i32* %left_idx
882. %tmp5 = sub i32 %right_idx2, 1
883. store i32 %tmp5, i32* %right_idx
884. br label %while
885.
886. merge:                                     ; preds = %while
887. ret void
888. }
889.
890. define void @list_reverseint({ i32*, i32* }*) {
891. entry:
892. %list_ptr_alloc = alloca { i32*, i32* }*

```

```

893.   store { i32*, i32* }* %0, { i32*, i32* }** %list_ptr_alloc
894.   %list_load = load { i32*, i32* }*, { i32*, i32* }** %list_ptr_alloc
895.   %list_size_ptr_ptr = getelementptr inbounds { i32*, i32* }, { i32*,
      i32* }* %list_load, i32 0, i32 0
896.   %list_size_ptr = load i32*, i32** %list_size_ptr_ptr
897.   %list_size = load i32, i32* %list_size_ptr
898.   %left_idx = alloca i32
899.   store i32 0, i32* %left_idx
900.   %right_idx = alloca i32
901.   %tmp = sub i32 %list_size, 1
902.   store i32 %tmp, i32* %right_idx
903.   br label %while
904.
905. while:                                     ; preds = %while_body, %entry
906.   %right_idx6 = load i32, i32* %right_idx
907.   %left_idx7 = load i32, i32* %left_idx
908.   %while_cond = icmp slt i32 %left_idx7, %right_idx6
909.   br i1 %while_cond, label %while_body, label %merge
910.
911. while_body:                                 ; preds = %while
912.   %left_idx1 = load i32, i32* %left_idx
913.   %right_idx2 = load i32, i32* %right_idx
914.   %list_get = call i32 @list_getint({ i32*, i32* }* %list_load, i32 %left_idx1)
915.   %list_get3 = call i32 @list_getint({ i32*, i32* }* %list_load, i32 %right_idx2)
916.   call void @list_setint({ i32*, i32* }* %list_load, i32 %left_idx1, i32 %list_get3)
917.   call void @list_setint({ i32*, i32* }* %list_load, i32 %right_idx2, i32 %list_get)
918.   %tmp4 = add i32 %left_idx1, 1
919.   store i32 %tmp4, i32* %left_idx
920.   %tmp5 = sub i32 %right_idx2, 1
921.   store i32 %tmp5, i32* %right_idx
922.   br label %while
923.
924. merge:                                     ; preds = %while
925.   ret void
926. }
927.
928. define void @list_reversefloat({ i32*, double* }*) {
929. entry:
930.   %list_ptr_alloc = alloca { i32*, double* }*
931.   store { i32*, double* }* %0, { i32*, double* }** %list_ptr_alloc
932.   %list_load = load { i32*, double* }*, { i32*, double* }** %list_ptr_alloc
933.   %list_size_ptr_ptr = getelementptr inbounds { i32*, double* }, { i32*,
      double* }* %list_load, i32 0, i32 0
934.   %list_size_ptr = load i32*, i32** %list_size_ptr_ptr
935.   %list_size = load i32, i32* %list_size_ptr
936.   %left_idx = alloca i32
937.   store i32 0, i32* %left_idx
938.   %right_idx = alloca i32
939.   %tmp = sub i32 %list_size, 1
940.   store i32 %tmp, i32* %right_idx
941.   br label %while
942.

```

```

943. while:                                     ; preds = %while_body, %entry
944.   %right_idx6 = load i32, i32* %right_idx
945.   %left_idx7 = load i32, i32* %left_idx
946.   %while_cond = icmp slt i32 %left_idx7, %right_idx6
947.   br i1 %while_cond, label %while_body, label %merge
948.
949. while_body:                                 ; preds = %while
950.   %left_idx1 = load i32, i32* %left_idx
951.   %right_idx2 = load i32, i32* %right_idx
952.   %list_get = call double @list_getfloat({ i32*, double* }* %list_load, i32 %left_idx1)
953.   %list_get3 = call double @list_getfloat({ i32*, double* }* %list_load,
     i32 %right_idx2)
954.   call void @list_setfloat({ i32*, double* }* %list_load, i32 %left_idx1,
     double %list_get3)
955.   call void @list_setfloat({ i32*, double* }* %list_load, i32 %right_idx2,
     double %list_get)
956.   %tmp4 = add i32 %left_idx1, 1
957.   store i32 %tmp4, i32* %left_idx
958.   %tmp5 = sub i32 %right_idx2, 1
959.   store i32 %tmp5, i32* %right_idx
960.   br label %while
961.
962. merge:                                       ; preds = %while
963.   ret void
964. }
965.
966. define i32 @main() {
967. entry:
968.   %a = alloca { i32*, i32* }
969.   %list_size_ptr = getelementptr inbounds { i32*, i32* }, { i32*, i32* }* %a, i32 0,
     i32 0
970.   %list_size = alloca i32
971.   store i32 0, i32* %list_size
972.   store i32* %list_size, i32** %list_size_ptr
973.   %list.array = getelementptr inbounds { i32*, i32* }, { i32*, i32* }* %a, i32 0, i32 1
974.   %p = alloca i32, i32 1028
975.   store i32* %p, i32** %list.array
976.   %i = alloca i32
977.   %new_list_ptr = alloca { i32*, i32* }
978.   %list_size_ptr1 = getelementptr inbounds { i32*, i32* }, { i32*,
     i32* }* %new_list_ptr, i32 0, i32 0
979.   %list_size2 = alloca i32
980.   store i32 0, i32* %list_size2
981.   store i32* %list_size2, i32** %list_size_ptr1
982.   %list.array3 = getelementptr inbounds { i32*, i32* }, { i32*, i32* }* %new_list_ptr,
     i32 0, i32 1
983.   %p4 = alloca i32, i32 1028
984.   store i32* %p4, i32** %list.array3
985.   call void @list_pushint({ i32*, i32* }* %new_list_ptr, i32 10)
986.   call void @list_pushint({ i32*, i32* }* %new_list_ptr, i32 2)
987.   call void @list_pushint({ i32*, i32* }* %new_list_ptr, i32 15)
988.   call void @list_pushint({ i32*, i32* }* %new_list_ptr, i32 8)

```

```

989.  call void @list_pushint({ i32*, i32* }* %new_list_ptr, i32 20)
990.  call void @list_pushint({ i32*, i32* }* %new_list_ptr, i32 13)
991.  call void @list_pushint({ i32*, i32* }* %new_list_ptr, i32 9)
992.  %new_list = load { i32*, i32* }, { i32*, i32* }* %new_list_ptr
993.  store { i32*, i32* } %new_list, { i32*, i32* }* %a
994.  %a5 = load { i32*, i32* }, { i32*, i32* }* %a
995.  call void @bubbleSort({ i32*, i32* } %a5)
996.  store i32 0, i32* %i
997.  br label %while
998.
999. while:                                     ; preds = %while_body, %entry
1000.    %i8 = load i32, i32* %i
1001.    %list_size9 = call i32 @list_sizeint({ i32*, i32* }* %a)
1002.    %tmp10 = icmp slt i32 %i8, %list_size9
1003.    br i1 %tmp10, label %while_body, label %merge
1004.
1005.    while_body:                               ; preds = %while
1006.    %i6 = load i32, i32* %i
1007.    %list_get = call i32 @list_getint({ i32*, i32* }* %a, i32 %i6)
1008.    %printf = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([4 x i8], [4
x i8]* @fmt, i32 0, i32 0), i32 %list_get)
1009.    %i7 = load i32, i32* %i
1010.    %tmp = add i32 %i7, 1
1011.    store i32 %tmp, i32* %i
1012.    br label %while
1013.
1014.    merge:                                     ; preds = %while
1015.    ret i32 0
1016.  }
1017.
1018.  define void @bubbleSort({ i32*, i32* } %arr) {
1019.  entry:
1020.    %arr1 = alloca { i32*, i32* }
1021.    %list_size_ptr = getelementptr inbounds { i32*, i32* }, { i32*, i32* }* %arr1,
i32 0, i32 0
1022.    %list_size = alloca i32
1023.    store i32 0, i32* %list_size
1024.    store i32* %list_size, i32** %list_size_ptr
1025.    %list.array = getelementptr inbounds { i32*, i32* }, { i32*, i32* }* %arr1, i32
0, i32 1
1026.    %p = alloca i32, i32 1028
1027.    store i32* %p, i32** %list.array
1028.    store { i32*, i32* } %arr, { i32*, i32* }* %arr1
1029.    %i = alloca i32
1030.    %j = alloca i32
1031.    %tmp = alloca i32
1032.    store i32 0, i32* %i
1033.    br label %while
1034.
1035.    while:                                     ; preds = %merge26, %entry
1036.    %i29 = load i32, i32* %i
1037.    %list_size30 = call i32 @list_sizeint({ i32*, i32* }* %arr1)

```

```

1038.      %tmp31 = sub i32 %list_size30, 1
1039.      %tmp32 = icmp slt i32 %i29, %tmp31
1040.      br i1 %tmp32, label %while_body, label %merge33
1041.
1042.      while_body:                                     ; preds = %while
1043.          store i32 0, i32* %j
1044.          br label %while2
1045.
1046.      while2:                                         ; preds = %merge, %while_body
1047.          %j20 = load i32, i32* %j
1048.          %list_size21 = call i32 @list_sizeint({ i32*, i32* }* %arr1)
1049.          %i22 = load i32, i32* %i
1050.          %tmp23 = sub i32 %list_size21, %i22
1051.          %tmp24 = sub i32 %tmp23, 1
1052.          %tmp25 = icmp slt i32 %j20, %tmp24
1053.          br i1 %tmp25, label %while_body3, label %merge26
1054.
1055.      while_body3:                                    ; preds = %while2
1056.          %j4 = load i32, i32* %j
1057.          %list_get = call i32 @list_getint({ i32*, i32* }* %arr1, i32 %j4)
1058.          %j5 = load i32, i32* %j
1059.          %tmp6 = add i32 %j5, 1
1060.          %list_get7 = call i32 @list_getint({ i32*, i32* }* %arr1, i32 %tmp6)
1061.          %tmp8 = icmp sgt i32 %list_get, %list_get7
1062.          br i1 %tmp8, label %then, label %else
1063.
1064.      merge:                                          ; preds = %else, %then
1065.          %j18 = load i32, i32* %j
1066.          %tmp19 = add i32 %j18, 1
1067.          store i32 %tmp19, i32* %j
1068.          br label %while2
1069.
1070.      then:                                           ; preds = %while_body3
1071.          %j9 = load i32, i32* %j
1072.          %list_get10 = call i32 @list_getint({ i32*, i32* }* %arr1, i32 %j9)
1073.          store i32 %list_get10, i32* %tmp
1074.          %j11 = load i32, i32* %j
1075.          %tmp12 = add i32 %j11, 1
1076.          %list_get13 = call i32 @list_getint({ i32*, i32* }* %arr1, i32 %tmp12)
1077.          %j14 = load i32, i32* %j
1078.          call void @list_setint({ i32*, i32* }* %arr1, i32 %j14, i32 %list_get13)
1079.          %tmp15 = load i32, i32* %tmp
1080.          %j16 = load i32, i32* %j
1081.          %tmp17 = add i32 %j16, 1
1082.          call void @list_setint({ i32*, i32* }* %arr1, i32 %tmp17, i32 %tmp15)
1083.          br label %merge
1084.
1085.      else:                                           ; preds = %while_body3
1086.          br label %merge
1087.
1088.      merge26:                                       ; preds = %while2
1089.          %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

```
1.  /* sorts a list in-place with QuickSort algorithm, */
2.
3.  int partition(list<int> a, int low, int high)
4.  {
5.      int i;
6.      int j;
7.      int pivot;
8.      int temp;
9.
10.     pivot = a[high];
11.     i = (low - 1);
12.     for (j = low; j <= high - 1; j++)
13.     {
14.         if (a[j] <= pivot)
15.         {
16.             i++;
17.             temp = a[i];
18.             a[i] = a[j];
19.             a[j] = temp;
20.         }
21.     }
22.     temp = a[i+1];
23.     a[i+1] = a[high];
24.     a[high] = temp;
25.     return (i + 1);
26. }
27.
28. void quickSortImpl(list<int> a, int low, int high)
29. {
30.     int pi;
31.     if (low < high)
32.     {
33.         pi = partition(a, low, high);
34.         quickSortImpl(a, low, pi - 1);
35.         quickSortImpl(a, pi + 1, high);
36.     }
37.     return;
38. }
39.
40. void quickSort(list<int> a)
41. {
42.     quickSortImpl(a, 0, #a - 1);
43.     return;
44. }
45.
46. int main()
47. {
48.     list<int> a;
49.     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

```

```

25. %list_load = load { i32*, i1* }*, { i32*, i1* }** %list_ptr_alloc
26. %list_array_ptr = getelementptr inbounds { i32*, i1* }, { i32*, i1* }* %list_load, i32
    0, i32 1
27. %array_load = load i1*, i1** %list_array_ptr
28. %idx_load = load i32, i32* %idx_alloc
29. %list_array_element_ptr = getelementptr i1, i1* %array_load, i32 %idx_load
30. %list_array_element_ptr = load i1, i1* %list_array_element_ptr
31. ret i1 %list_array_element_ptr
32. }
33.
34. define i32 @list_getint({ i32*, i32* }*, i32) {
35. entry:
36. %list_ptr_alloc = alloca { i32*, i32* }*
37. store { i32*, i32* }* %0, { i32*, i32* }** %list_ptr_alloc
38. %idx_alloc = alloca i32
39. store i32 %1, i32* %idx_alloc
40. %list_load = load { i32*, i32* }*, { i32*, i32* }** %list_ptr_alloc
41. %list_array_ptr = getelementptr inbounds { i32*, i32* }, { i32*, i32* }* %list_load,
    i32 0, i32 1
42. %array_load = load i32*, i32** %list_array_ptr
43. %idx_load = load i32, i32* %idx_alloc
44. %list_array_element_ptr = getelementptr i32, i32* %array_load, i32 %idx_load
45. %list_array_element_ptr = load i32, i32* %list_array_element_ptr
46. ret i32 %list_array_element_ptr
47. }
48.
49. define double @list_getfloat({ i32*, double* }*, i32) {
50. entry:
51. %list_ptr_alloc = alloca { i32*, double* }*
52. store { i32*, double* }* %0, { i32*, double* }** %list_ptr_alloc
53. %idx_alloc = alloca i32
54. store i32 %1, i32* %idx_alloc
55. %list_load = load { i32*, double* }*, { i32*, double* }** %list_ptr_alloc
56. %list_array_ptr = getelementptr inbounds { i32*, double* }, { i32*,
    double* }* %list_load, i32 0, i32 1
57. %array_load = load double*, double** %list_array_ptr
58. %idx_load = load i32, i32* %idx_alloc
59. %list_array_element_ptr = getelementptr double, double* %array_load, i32 %idx_load
60. %list_array_element_ptr = load double, double* %list_array_element_ptr
61. ret double %list_array_element_ptr
62. }
63.
64. define i8* @list_getstr({ i32*, i8** }*, i32) {
65. entry:
66. %list_ptr_alloc = alloca { i32*, i8** }*
67. store { i32*, i8** }* %0, { i32*, i8** }** %list_ptr_alloc
68. %idx_alloc = alloca i32
69. store i32 %1, i32* %idx_alloc
70. %list_load = load { i32*, i8** }*, { i32*, i8** }** %list_ptr_alloc
71. %list_array_ptr = getelementptr inbounds { i32*, i8** }, { i32*, i8** }* %list_load,
    i32 0, i32 1
72. %array_load = load i8**, i8*** %list_array_ptr

```

```

73. %idx_load = load i32, i32* %idx_alloc
74. %list_array_element_ptr = getelementptr i8*, i8** %array_load, i32 %idx_load
75. %list_array_element_ptr = load i8*, i8** %list_array_element_ptr
76. ret i8* %list_array_element_ptr
77. }
78.
79. define void @list_setbool({ i32*, i1* }*, i32, i1) {
80. entry:
81. %list_ptr_alloc = alloca { i32*, i1* }*
82. store { i32*, i1* }* %0, { i32*, i1* }** %list_ptr_alloc
83. %list_load = load { i32*, i1* }*, { i32*, i1* }** %list_ptr_alloc
84. %list_array_ptr = getelementptr inbounds { i32*, i1* }, { i32*, i1* }* %list_load, i32
    0, i32 1
85. %list_array_load = load i1*, i1** %list_array_ptr
86. %list_array_next_element_ptr = getelementptr i1, i1* %list_array_load, i32 %1
87. store i1 %2, i1* %list_array_next_element_ptr
88. ret void
89. }
90.
91. define void @list_setint({ i32*, i32* }*, i32, i32) {
92. entry:
93. %list_ptr_alloc = alloca { i32*, i32* }*
94. store { i32*, i32* }* %0, { i32*, i32* }** %list_ptr_alloc
95. %list_load = load { i32*, i32* }*, { i32*, i32* }** %list_ptr_alloc
96. %list_array_ptr = getelementptr inbounds { i32*, i32* }, { i32*, i32* }* %list_load,
    i32 0, i32 1
97. %list_array_load = load i32*, i32** %list_array_ptr
98. %list_array_next_element_ptr = getelementptr i32, i32* %list_array_load, i32 %1
99. store i32 %2, i32* %list_array_next_element_ptr
100. ret void
101. }
102.
103. define void @list_setfloat({ i32*, double* }*, i32, double) {
104. entry:
105. %list_ptr_alloc = alloca { i32*, double* }*
106. store { i32*, double* }* %0, { i32*, double* }** %list_ptr_alloc
107. %list_load = load { i32*, double* }*, { i32*, double* }** %list_ptr_alloc
108. %list_array_ptr = getelementptr inbounds { i32*, double* }, { i32*,
    double* }* %list_load, i32 0, i32 1
109. %list_array_load = load double*, double** %list_array_ptr
110. %list_array_next_element_ptr = getelementptr double, double* %list_array_load, i32 %1
111. store double %2, double* %list_array_next_element_ptr
112. ret void
113. }
114.
115. define void @list_setstr({ i32*, i8** }*, i32, i8*) {
116. entry:
117. %list_ptr_alloc = alloca { i32*, i8** }*
118. store { i32*, i8** }* %0, { i32*, i8** }** %list_ptr_alloc
119. %list_load = load { i32*, i8** }*, { i32*, i8** }** %list_ptr_alloc
120. %list_array_ptr = getelementptr inbounds { i32*, i8** }, { i32*, i8** }* %list_load,
    i32 0, i32 1

```

```

121.  %list_array_load = load i8**, i8*** %list_array_ptr
122.  %list_array_next_element_ptr = getelementptr i8*, i8** %list_array_load, i32 %1
123.  store i8* %2, i8** %list_array_next_element_ptr
124.  ret void
125. }
126.
127. define void @list_pushbool({ i32*, i1* }*, i1) {
128. entry:
129.  %list_ptr_alloc = alloca { i32*, i1* }*
130.  store { i32*, i1* }* %0, { i32*, i1* }** %list_ptr_alloc
131.  %val_alloc = alloca i1
132.  store i1 %1, i1* %val_alloc
133.  %list_load = load { i32*, i1* }*, { i32*, i1* }** %list_ptr_alloc
134.  %list_array_ptr = getelementptr inbounds { i32*, i1* }, { i32*, i1* }* %list_load,
    i32 0, i32 1
135.  %list_array_load = load i1*, i1** %list_array_ptr
136.  %list_size_ptr_ptr = getelementptr inbounds { i32*, i1* }, { i32*, i1* }* %list_load,
    i32 0, i32 0
137.  %list_size_ptr = load i32*, i32** %list_size_ptr_ptr
138.  %list_size = load i32, i32* %list_size_ptr
139.  %list_array_next_element_ptr = getelementptr i1, i1* %list_array_load, i32 %list_size
140.  %inc_size = add i32 %list_size, 1
141.  store i32 %inc_size, i32* %list_size_ptr
142.  %val = load i1, i1* %val_alloc
143.  store i1 %val, i1* %list_array_next_element_ptr
144.  ret void
145. }
146.
147. define void @list_pushint({ i32*, i32* }*, i32) {
148. entry:
149.  %list_ptr_alloc = alloca { i32*, i32* }*
150.  store { i32*, i32* }* %0, { i32*, i32* }** %list_ptr_alloc
151.  %val_alloc = alloca i32
152.  store i32 %1, i32* %val_alloc
153.  %list_load = load { i32*, i32* }*, { i32*, i32* }** %list_ptr_alloc
154.  %list_array_ptr = getelementptr inbounds { i32*, i32* }, { i32*, i32* }* %list_load,
    i32 0, i32 1
155.  %list_array_load = load i32*, i32** %list_array_ptr
156.  %list_size_ptr_ptr = getelementptr inbounds { i32*, i32* }, { i32*,
    i32* }* %list_load, i32 0, i32 0
157.  %list_size_ptr = load i32*, i32** %list_size_ptr_ptr
158.  %list_size = load i32, i32* %list_size_ptr
159.  %list_array_next_element_ptr = getelementptr i32, i32* %list_array_load,
    i32 %list_size
160.  %inc_size = add i32 %list_size, 1
161.  store i32 %inc_size, i32* %list_size_ptr
162.  %val = load i32, i32* %val_alloc
163.  store i32 %val, i32* %list_array_next_element_ptr
164.  ret void
165. }
166.
167. define void @list_pushfloat({ i32*, double* }*, double) {

```

```

168. entry:
169.  %list_ptr_alloc = alloca { i32*, double* }*
170.  store { i32*, double* }* %0, { i32*, double* }** %list_ptr_alloc
171.  %val_alloc = alloca double
172.  store double %1, double* %val_alloc
173.  %list_load = load { i32*, double* }*, { i32*, double* }** %list_ptr_alloc
174.  %list_array_ptr = getelementptr inbounds { i32*, double* }, { i32*,
    double* }* %list_load, i32 0, i32 1
175.  %list_array_load = load double*, double** %list_array_ptr
176.  %list_size_ptr_ptr = getelementptr inbounds { i32*, double* }, { i32*,
    double* }* %list_load, i32 0, i32 0
177.  %list_size_ptr = load i32*, i32** %list_size_ptr_ptr
178.  %list_size = load i32, i32* %list_size_ptr
179.  %list_array_next_element_ptr = getelementptr double, double* %list_array_load,
    i32 %list_size
180.  %inc_size = add i32 %list_size, 1
181.  store i32 %inc_size, i32* %list_size_ptr
182.  %val = load double, double* %val_alloc
183.  store double %val, double* %list_array_next_element_ptr
184.  ret void
185. }
186.
187. define void @list_pushstr({ i32*, i8** }*, i8*) {
188. entry:
189.  %list_ptr_alloc = alloca { i32*, i8** }*
190.  store { i32*, i8** }* %0, { i32*, i8** }** %list_ptr_alloc
191.  %val_alloc = alloca i8*
192.  store i8* %1, i8** %val_alloc
193.  %list_load = load { i32*, i8** }*, { i32*, i8** }** %list_ptr_alloc
194.  %list_array_ptr = getelementptr inbounds { i32*, i8** }, { i32*, i8** }* %list_load,
    i32 0, i32 1
195.  %list_array_load = load i8**, i8*** %list_array_ptr
196.  %list_size_ptr_ptr = getelementptr inbounds { i32*, i8** }, { i32*,
    i8** }* %list_load, i32 0, i32 0
197.  %list_size_ptr = load i32*, i32** %list_size_ptr_ptr
198.  %list_size = load i32, i32* %list_size_ptr
199.  %list_array_next_element_ptr = getelementptr i8*, i8** %list_array_load,
    i32 %list_size
200.  %inc_size = add i32 %list_size, 1
201.  store i32 %inc_size, i32* %list_size_ptr
202.  %val = load i8*, i8** %val_alloc
203.  store i8* %val, i8** %list_array_next_element_ptr
204.  ret void
205. }
206.
207. define i1 @list_popbool({ i32*, i1* }*) {
208. entry:
209.  %list_ptr_alloc = alloca { i32*, i1* }*
210.  store { i32*, i1* }* %0, { i32*, i1* }** %list_ptr_alloc
211.  %list_load = load { i32*, i1* }*, { i32*, i1* }** %list_ptr_alloc
212.  %list_array_ptr = getelementptr inbounds { i32*, i1* }, { i32*, i1* }* %list_load,
    i32 0, i32 1
213.  %list_array_load = load i1*, i1** %list_array_ptr

```

```

214.  %list_size_ptr_ptr = getelementptr inbounds { i32*, i1* }, { i32*, i1* }* %list_load,
    i32 0, i32 0
215.  %list_size_ptr = load i32*, i32** %list_size_ptr_ptr
216.  %list_size = load i32, i32* %list_size_ptr
217.  %dec_size = sub i32 %list_size, 1
218.  %list_array_next_element_ptr = getelementptr i1, i1* %list_array_load, i32 %dec_size
219.  %list_array_next_element = load i1, i1* %list_array_next_element_ptr
220.  store i32 %dec_size, i32* %list_size_ptr
221.  ret i1 %list_array_next_element
222. }
223.
224. define i32 @list_popint({ i32*, i32* }*) {
225. entry:
226.  %list_ptr_alloc = alloca { i32*, i32* }*
227.  store { i32*, i32* }* %0, { i32*, i32* }** %list_ptr_alloc
228.  %list_load = load { i32*, i32* }*, { i32*, i32* }** %list_ptr_alloc
229.  %list_array_ptr = getelementptr inbounds { i32*, i32* }, { i32*, i32* }* %list_load,
    i32 0, i32 1
230.  %list_array_load = load i32*, i32** %list_array_ptr
231.  %list_size_ptr_ptr = getelementptr inbounds { i32*, i32* }, { i32*,
    i32* }* %list_load, i32 0, i32 0
232.  %list_size_ptr = load i32*, i32** %list_size_ptr_ptr
233.  %list_size = load i32, i32* %list_size_ptr
234.  %dec_size = sub i32 %list_size, 1
235.  %list_array_next_element_ptr = getelementptr i32, i32* %list_array_load, i32 %dec_size
236.  %list_array_next_element = load i32, i32* %list_array_next_element_ptr
237.  store i32 %dec_size, i32* %list_size_ptr
238.  ret i32 %list_array_next_element
239. }
240.
241. define double @list_popfloat({ i32*, double* }*) {
242. entry:
243.  %list_ptr_alloc = alloca { i32*, double* }*
244.  store { i32*, double* }* %0, { i32*, double* }** %list_ptr_alloc
245.  %list_load = load { i32*, double* }*, { i32*, double* }** %list_ptr_alloc
246.  %list_array_ptr = getelementptr inbounds { i32*, double* }, { i32*,
    double* }* %list_load, i32 0, i32 1
247.  %list_array_load = load double*, double** %list_array_ptr
248.  %list_size_ptr_ptr = getelementptr inbounds { i32*, double* }, { i32*,
    double* }* %list_load, i32 0, i32 0
249.  %list_size_ptr = load i32*, i32** %list_size_ptr_ptr
250.  %list_size = load i32, i32* %list_size_ptr
251.  %dec_size = sub i32 %list_size, 1
252.  %list_array_next_element_ptr = getelementptr double, double* %list_array_load,
    i32 %dec_size
253.  %list_array_next_element = load double, double* %list_array_next_element_ptr
254.  store i32 %dec_size, i32* %list_size_ptr
255.  ret double %list_array_next_element
256. }
257.
258. define i8* @list_popstr({ i32*, i8** }*) {
259. entry:

```

```

260.  %list_ptr_alloc = alloca { i32*, i8** }*
261.  store { i32*, i8** }* %0, { i32*, i8** }** %list_ptr_alloc
262.  %list_load = load { i32*, i8** }*, { i32*, i8** }** %list_ptr_alloc
263.  %list_array_ptr = getelementptr inbounds { i32*, i8** }, { i32*, i8** }* %list_load,
    i32 0, i32 1
264.  %list_array_load = load i8**, i8*** %list_array_ptr
265.  %list_size_ptr_ptr = getelementptr inbounds { i32*, i8** }, { i32*,
    i8** }* %list_load, i32 0, i32 0
266.  %list_size_ptr = load i32*, i32** %list_size_ptr_ptr
267.  %list_size = load i32, i32* %list_size_ptr
268.  %dec_size = sub i32 %list_size, 1
269.  %list_array_next_element_ptr = getelementptr i8*, i8** %list_array_load, i32 %dec_size
270.  %list_array_next_element = load i8*, i8** %list_array_next_element_ptr
271.  store i32 %dec_size, i32* %list_size_ptr
272.  ret i8* %list_array_next_element
273. }
274.
275. define i32 @list_sizebool({ i32*, i1* }*) {
276. entry:
277.  %list_ptr_alloc = alloca { i32*, i1* }*
278.  store { i32*, i1* }* %0, { i32*, i1* }** %list_ptr_alloc
279.  %list_load = load { i32*, i1* }*, { i32*, i1* }** %list_ptr_alloc
280.  %list_size_ptr_ptr = getelementptr inbounds { i32*, i1* }, { i32*, i1* }* %list_load,
    i32 0, i32 0
281.  %list_size_ptr = load i32*, i32** %list_size_ptr_ptr
282.  %list_size = load i32, i32* %list_size_ptr
283.  ret i32 %list_size
284. }
285.
286. define i32 @list_sizeint({ i32*, i32* }*) {
287. entry:
288.  %list_ptr_alloc = alloca { i32*, i32* }*
289.  store { i32*, i32* }* %0, { i32*, i32* }** %list_ptr_alloc
290.  %list_load = load { i32*, i32* }*, { i32*, i32* }** %list_ptr_alloc
291.  %list_size_ptr_ptr = getelementptr inbounds { i32*, i32* }, { i32*,
    i32* }* %list_load, i32 0, i32 0
292.  %list_size_ptr = load i32*, i32** %list_size_ptr_ptr
293.  %list_size = load i32, i32* %list_size_ptr
294.  ret i32 %list_size
295. }
296.
297. define i32 @list_sizefloat({ i32*, double* }*) {
298. entry:
299.  %list_ptr_alloc = alloca { i32*, double* }*
300.  store { i32*, double* }* %0, { i32*, double* }** %list_ptr_alloc
301.  %list_load = load { i32*, double* }*, { i32*, double* }** %list_ptr_alloc
302.  %list_size_ptr_ptr = getelementptr inbounds { i32*, double* }, { i32*,
    double* }* %list_load, i32 0, i32 0
303.  %list_size_ptr = load i32*, i32** %list_size_ptr_ptr
304.  %list_size = load i32, i32* %list_size_ptr
305.  ret i32 %list_size
306. }

```



```

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) {

```

```

358. entry:
359.   %list_ptr_alloc = alloca { i32*, i32* }*
360.   store { i32*, i32* }* %0, { i32*, i32* }** %list_ptr_alloc
361.   %list_ptr_ptr = load { i32*, i32* }*, { i32*, i32* }** %list_ptr_alloc
362.   %list_ptr_alloc2 = alloca { i32*, i32* }*
363.   store { i32*, i32* }* %1, { i32*, i32* }** %list_ptr_alloc2
364.   %list_ptr_ptr2 = load { i32*, i32* }*, { i32*, i32* }** %list_ptr_alloc2
365.   %idx_alloc = alloca i32
366.   store i32 %2, i32* %idx_alloc
367.   %idx_load = load i32, i32* %idx_alloc
368.   %idx_alloc1 = alloca i32
369.   store i32 %3, i32* %idx_alloc1
370.   %idx_load2 = load i32, i32* %idx_alloc1
371.   %loop_cnt = alloca i32
372.   store i32 0, i32* %loop_cnt
373.   %loop_upper_bound = sub i32 %idx_load2, %idx_load
374.   br label %while
375.
376. while:                                     ; preds = %while_body, %entry
377.   %loop_cnt4 = load i32, i32* %loop_cnt
378.   %loop_cond = icmp sle i32 %loop_cnt4, %loop_upper_bound
379.   br i1 %loop_cond, label %while_body, label %merge
380.
381. while_body:                               ; preds = %while
382.   %to_idx = load i32, i32* %loop_cnt
383.   %from_idx = add i32 %to_idx, %idx_load
384.   %list_get = call i32 @list_getint({ i32*, i32* }* %list_ptr_ptr, i32 %from_idx)
385.   call void @list_pushint({ i32*, i32* }* %list_ptr_ptr2, i32 %list_get)
386.   %loop_cnt3 = load i32, i32* %loop_cnt
387.   %loop_itr = add i32 %loop_cnt3, 1
388.   store i32 %loop_itr, i32* %loop_cnt
389.   br label %while
390.
391. merge:                                     ; preds = %while
392.   ret void
393. }
394.
395. define void @list_slice(float({ i32*, double* }*, { i32*, double* }*, i32, i32) {
396. entry:
397.   %list_ptr_alloc = alloca { i32*, double* }*
398.   store { i32*, double* }* %0, { i32*, double* }** %list_ptr_alloc
399.   %list_ptr_ptr = load { i32*, double* }*, { i32*, double* }** %list_ptr_alloc
400.   %list_ptr_alloc2 = alloca { i32*, double* }*
401.   store { i32*, double* }* %1, { i32*, double* }** %list_ptr_alloc2
402.   %list_ptr_ptr2 = load { i32*, double* }*, { i32*, double* }** %list_ptr_alloc2
403.   %idx_alloc = alloca i32
404.   store i32 %2, i32* %idx_alloc
405.   %idx_load = load i32, i32* %idx_alloc
406.   %idx_alloc1 = alloca i32
407.   store i32 %3, i32* %idx_alloc1
408.   %idx_load2 = load i32, i32* %idx_alloc1
409.   %loop_cnt = alloca i32

```

```

410. store i32 0, i32* %loop_cnt
411. %loop_upper_bound = sub i32 %idx_load2, %idx_load
412. br label %while
413.
414. while:                                     ; preds = %while_body, %entry
415. %loop_cnt4 = load i32, i32* %loop_cnt
416. %loop_cond = icmp sle i32 %loop_cnt4, %loop_upper_bound
417. br i1 %loop_cond, label %while_body, label %merge
418.
419. while_body:                                 ; preds = %while
420. %to_idx = load i32, i32* %loop_cnt
421. %from_idx = add i32 %to_idx, %idx_load
422. %list_get = call double @list_getfloat({ i32*, double* }* %list_ptr_ptr,
    i32 %from_idx)
423. call void @list_pushfloat({ i32*, double* }* %list_ptr_ptr2, double %list_get)
424. %loop_cnt3 = load i32, i32* %loop_cnt
425. %loop_itr = add i32 %loop_cnt3, 1
426. store i32 %loop_itr, i32* %loop_cnt
427. br label %while
428.
429. merge:                                     ; preds = %while
430. ret void
431. }
432.
433. define void @list_sliceistr({ i32*, i8** }*, { i32*, i8** }*, i32, i32) {
434. entry:
435. %list_ptr_alloc = alloca { i32*, i8** }*
436. store { i32*, i8** }* %0, { i32*, i8** }** %list_ptr_alloc
437. %list_ptr_ptr = load { i32*, i8** }*, { i32*, i8** }** %list_ptr_alloc
438. %list_ptr_alloc2 = alloca { i32*, i8** }*
439. store { i32*, i8** }* %1, { i32*, i8** }** %list_ptr_alloc2
440. %list_ptr_ptr2 = load { i32*, i8** }*, { i32*, i8** }** %list_ptr_alloc2
441. %idx_alloc = alloca i32
442. store i32 %2, i32* %idx_alloc
443. %idx_load = load i32, i32* %idx_alloc
444. %idx_alloc1 = alloca i32
445. store i32 %3, i32* %idx_alloc1
446. %idx_load2 = load i32, i32* %idx_alloc1
447. %loop_cnt = alloca i32
448. store i32 0, i32* %loop_cnt
449. %loop_upper_bound = sub i32 %idx_load2, %idx_load
450. br label %while
451.
452. while:                                     ; preds = %while_body, %entry
453. %loop_cnt4 = load i32, i32* %loop_cnt
454. %loop_cond = icmp sle i32 %loop_cnt4, %loop_upper_bound
455. br i1 %loop_cond, label %while_body, label %merge
456.
457. while_body:                                 ; preds = %while
458. %to_idx = load i32, i32* %loop_cnt
459. %from_idx = add i32 %to_idx, %idx_load
460. %list_get = call i8* @list_getstr({ i32*, i8** }* %list_ptr_ptr, i32 %from_idx)

```



```

511. }
512.
513. define i32 @list findint({ i32*, i32* }*, i32) {
514. entry:
515.   %list_ptr_alloc = alloca { i32*, i32* }*
516.   store { i32*, i32* }* %0, { i32*, i32* }** %list_ptr_alloc
517.   %find_val_alloc = alloca i32
518.   store i32 %1, i32* %find_val_alloc
519.   %find_val = load i32, i32* %find_val_alloc
520.   %list_load = load { i32*, i32* }*, { i32*, i32* }** %list_ptr_alloc
521.   %list_size_ptr_ptr = getelementptr inbounds { i32*, i32* }, { i32*,
      i32* }* %list_load, i32 0, i32 0
522.   %list_size_ptr = load i32*, i32** %list_size_ptr_ptr
523.   %list_size = load i32, i32* %list_size_ptr
524.   %loop_cnt = alloca i32
525.   store i32 0, i32* %loop_cnt
526.   br label %while
527.
528. while:                                     ; preds = %merge, %entry
529.   %loop_iter_cnt = load i32, i32* %loop_cnt
530.   %loop_cond = icmp slt i32 %loop_iter_cnt, %list_size
531.   br i1 %loop_cond, label %while_body, label %merge1
532.
533. while_body:                               ; preds = %while
534.   %to_idx = load i32, i32* %loop_cnt
535.   %list_get = call i32 @list getint({ i32*, i32* }* %list_load, i32 %to_idx)
536.   %if_cond = icmp eq i32 %list_get, %find_val
537.   br i1 %if_cond, label %then, label %else
538.
539. merge:                                     ; preds = %else
540.   %loop_idx = load i32, i32* %loop_cnt
541.   %loop_itr = add i32 %loop_idx, 1
542.   store i32 %loop_itr, i32* %loop_cnt
543.   br label %while
544.
545. then:                                     ; preds = %while_body
546.   ret i32 %to_idx
547.
548. else:                                     ; preds = %while_body
549.   br label %merge
550.
551. merge1:                                   ; preds = %while
552.   ret i32 -1
553. }
554.
555. define i32 @list findfloat({ i32*, double* }*, double) {
556. entry:
557.   %list_ptr_alloc = alloca { i32*, double* }*
558.   store { i32*, double* }* %0, { i32*, double* }** %list_ptr_alloc
559.   %find_val_alloc = alloca double
560.   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

```



```

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

```



```

711.  %loop_start_idx = add i32 %list_find, 1
712.  store i32 %loop_start_idx, i32* %loop_cnt_ptr
713.  br label %while
714.
715. while:                                     ; preds = %while_body, %then
716.  %loop_cnt = load i32, i32* %loop_cnt_ptr
717.  %loop_cond1 = icmp slt i32 %loop_cnt, %list_size
718.  br i1 %loop_cond1, label %while_body, label %merge2
719.
720. while_body:                                 ; preds = %while
721.  %cur_idx = load i32, i32* %loop_cnt_ptr
722.  %shift_to_idx = sub i32 %cur_idx, 1
723.  %list_get = call double @list_getfloat({ i32*, double* }* %list_load, i32 %cur_idx)
724.  call void @list_setfloat({ i32*, double* }* %list_load, i32 %shift_to_idx,
    double %list_get)
725.  %loop_itr = add i32 %cur_idx, 1
726.  store i32 %loop_itr, i32* %loop_cnt_ptr
727.  br label %while
728.
729. merge2:                                     ; preds = %while
730.  %size_dec = sub i32 %list_size, 1
731.  store i32 %size_dec, i32* %list_size_ptr
732.  ret void
733.
734. else:                                       ; preds = %entry
735.  br label %merge
736. }
737.
738. define void @list_insertbool({ i32*, i1* }*, i32, i1) {
739. entry:
740.  %list_ptr_alloc = alloca { i32*, i1* }*
741.  store { i32*, i1* }* %0, { i32*, i1* }** %list_ptr_alloc
742.  %list_load = load { i32*, i1* }*, { i32*, i1* }** %list_ptr_alloc
743.  %insert_idx_ptr = alloca i32
744.  store i32 %1, i32* %insert_idx_ptr
745.  %insert_idx = load i32, i32* %insert_idx_ptr
746.  %insert_val_ptr = alloca i1
747.  store i1 %2, i1* %insert_val_ptr
748.  %insert_val = load i1, i1* %insert_val_ptr
749.  %list_size_ptr_ptr = getelementptr inbounds { i32*, i1* }, { i32*, i1* }* %list_load,
    i32 0, i32 0
750.  %list_size_ptr = load i32*, i32** %list_size_ptr_ptr
751.  %list_size = load i32, i32* %list_size_ptr
752.  %loop_cnt_ptr = alloca i32
753.  %last_index = sub i32 %list_size, 1
754.  store i32 %last_index, i32* %loop_cnt_ptr
755.  br label %while
756.
757. while:                                     ; preds = %while_body, %entry
758.  %loop_cnt = load i32, i32* %loop_cnt_ptr
759.  %loop_cond = icmp sge i32 %loop_cnt, %insert_idx
760.  br i1 %loop_cond, label %while_body, label %merge

```



```

812.  call void @list_setint({ i32*, i32* }* %list_load, i32 %insert_idx, i32 %insert_val)
813.  %size_inc = add i32 %list_size, 1
814.  store i32 %size_inc, i32* %list_size_ptr
815.  ret void
816. }
817.
818. define void @list_insertfloat({ i32*, double* }*, i32, double) {
819. entry:
820.  %list_ptr_alloc = alloca { i32*, double* }*
821.  store { i32*, double* }* %0, { i32*, double* }** %list_ptr_alloc
822.  %list_load = load { i32*, double* }*, { i32*, double* }** %list_ptr_alloc
823.  %insert_idx_ptr = alloca i32
824.  store i32 %1, i32* %insert_idx_ptr
825.  %insert_idx = load i32, i32* %insert_idx_ptr
826.  %insert_val_ptr = alloca double
827.  store double %2, double* %insert_val_ptr
828.  %insert_val = load double, double* %insert_val_ptr
829.  %list_size_ptr_ptr = getelementptr inbounds { i32*, double* }, { i32*,
      double* }* %list_load, i32 0, i32 0
830.  %list_size_ptr = load i32*, i32** %list_size_ptr_ptr
831.  %list_size = load i32, i32* %list_size_ptr
832.  %loop_cnt_ptr = alloca i32
833.  %last_index = sub i32 %list_size, 1
834.  store i32 %last_index, i32* %loop_cnt_ptr
835.  br label %while
836.
837. while:                                     ; preds = %while_body, %entry
838.  %loop_cnt = load i32, i32* %loop_cnt_ptr
839.  %loop_cond = icmp sge i32 %loop_cnt, %insert_idx
840.  br i1 %loop_cond, label %while_body, label %merge
841.
842. while_body:                                 ; preds = %while
843.  %cur_idx = load i32, i32* %loop_cnt_ptr
844.  %shift_to_idx = add i32 %cur_idx, 1
845.  %list_get = call double @list_getfloat({ i32*, double* }* %list_load, i32 %cur_idx)
846.  call void @list_setfloat({ i32*, double* }* %list_load, i32 %shift_to_idx,
      double %list_get)
847.  %loop_itr = sub i32 %cur_idx, 1
848.  store i32 %loop_itr, i32* %loop_cnt_ptr
849.  br label %while
850.
851. merge:                                     ; preds = %while
852.  call void @list_setfloat({ i32*, double* }* %list_load, i32 %insert_idx,
      double %insert_val)
853.  %size_inc = add i32 %list_size, 1
854.  store i32 %size_inc, i32* %list_size_ptr
855.  ret void
856. }
857.
858. define void @list_reversebool({ i32*, i1* }*) {
859. entry:
860.  %list_ptr_alloc = alloca { i32*, i1* }*

```

```

861.  store { i32*, i1* }* %0, { i32*, i1* }** %list_ptr_alloc
862.  %list_load = load { i32*, i1* }*, { i32*, i1* }** %list_ptr_alloc
863.  %list_size_ptr_ptr = getelementptr inbounds { i32*, i1* }, { i32*, i1* }* %list_load,
    i32 0, i32 0
864.  %list_size_ptr = load i32*, i32** %list_size_ptr_ptr
865.  %list_size = load i32, i32* %list_size_ptr
866.  %left_idx = alloca i32
867.  store i32 0, i32* %left_idx
868.  %right_idx = alloca i32
869.  %tmp = sub i32 %list_size, 1
870.  store i32 %tmp, i32* %right_idx
871.  br label %while

872.
873. while:                                     ; preds = %while_body, %entry
874.  %right_idx6 = load i32, i32* %right_idx
875.  %left_idx7 = load i32, i32* %left_idx
876.  %while_cond = icmp slt i32 %left_idx7, %right_idx6
877.  br i1 %while_cond, label %while_body, label %merge

878.
879. while_body:                               ; preds = %while
880.  %left_idx1 = load i32, i32* %left_idx
881.  %right_idx2 = load i32, i32* %right_idx
882.  %list_get = call i1 @list_getbool({ i32*, i1* }* %list_load, i32 %left_idx1)
883.  %list_get3 = call i1 @list_getbool({ i32*, i1* }* %list_load, i32 %right_idx2)
884.  call void @list_setbool({ i32*, i1* }* %list_load, i32 %left_idx1, i1 %list_get3)
885.  call void @list_setbool({ i32*, i1* }* %list_load, i32 %right_idx2, i1 %list_get3)
886.  %tmp4 = add i32 %left_idx1, 1
887.  store i32 %tmp4, i32* %left_idx
888.  %tmp5 = sub i32 %right_idx2, 1
889.  store i32 %tmp5, i32* %right_idx
890.  br label %while

891.
892. merge:                                     ; preds = %while
893.  ret void
894. }

895.
896. define void @list_reverseint({ i32*, i32* }*) {
897. entry:
898.  %list_ptr_alloc = alloca { i32*, i32* }*
899.  store { i32*, i32* }* %0, { i32*, i32* }** %list_ptr_alloc
900.  %list_load = load { i32*, i32* }*, { i32*, i32* }** %list_ptr_alloc
901.  %list_size_ptr_ptr = getelementptr inbounds { i32*, i32* }, { i32*,
    i32* }* %list_load, i32 0, i32 0
902.  %list_size_ptr = load i32*, i32** %list_size_ptr_ptr
903.  %list_size = load i32, i32* %list_size_ptr
904.  %left_idx = alloca i32
905.  store i32 0, i32* %left_idx
906.  %right_idx = alloca i32
907.  %tmp = sub i32 %list_size, 1
908.  store i32 %tmp, i32* %right_idx
909.  br label %while

910.

```

```

911. while:                                     ; preds = %while_body, %entry
912.   %right_idx6 = load i32, i32* %right_idx
913.   %left_idx7 = load i32, i32* %left_idx
914.   %while_cond = icmp slt i32 %left_idx7, %right_idx6
915.   br i1 %while_cond, label %while_body, label %merge
916.
917. while_body:                                 ; preds = %while
918.   %left_idx1 = load i32, i32* %left_idx
919.   %right_idx2 = load i32, i32* %right_idx
920.   %list_get = call i32 @list_getint({ i32*, i32* }* %list_load, i32 %left_idx1)
921.   %list_get3 = call i32 @list_getint({ i32*, i32* }* %list_load, i32 %right_idx2)
922.   call void @list_setint({ i32*, i32* }* %list_load, i32 %left_idx1, i32 %list_get3)
923.   call void @list_setint({ i32*, i32* }* %list_load, i32 %right_idx2, i32 %list_get)
924.   %tmp4 = add i32 %left_idx1, 1
925.   store i32 %tmp4, i32* %left_idx
926.   %tmp5 = sub i32 %right_idx2, 1
927.   store i32 %tmp5, i32* %right_idx
928.   br label %while
929.
930. merge:                                       ; preds = %while
931.   ret void
932. }
933.
934. define void @list_reversefloat({ i32*, double* }*) {
935. entry:
936.   %list_ptr_alloc = alloca { i32*, double* }*
937.   store { i32*, double* }* %0, { i32*, double* }** %list_ptr_alloc
938.   %list_load = load { i32*, double* }*, { i32*, double* }** %list_ptr_alloc
939.   %list_size_ptr_ptr = getelementptr inbounds { i32*, double* }, { i32*,
     double* }* %list_load, i32 0, i32 0
940.   %list_size_ptr = load i32*, i32** %list_size_ptr_ptr
941.   %list_size = load i32, i32* %list_size_ptr
942.   %left_idx = alloca i32
943.   store i32 0, i32* %left_idx
944.   %right_idx = alloca i32
945.   %tmp = sub i32 %list_size, 1
946.   store i32 %tmp, i32* %right_idx
947.   br label %while
948.
949. while:                                       ; preds = %while_body, %entry
950.   %right_idx6 = load i32, i32* %right_idx
951.   %left_idx7 = load i32, i32* %left_idx
952.   %while_cond = icmp slt i32 %left_idx7, %right_idx6
953.   br i1 %while_cond, label %while_body, label %merge
954.
955. while_body:                                 ; preds = %while
956.   %left_idx1 = load i32, i32* %left_idx
957.   %right_idx2 = load i32, i32* %right_idx
958.   %list_get = call double @list_getfloat({ i32*, double* }* %list_load, i32 %left_idx1)
959.   %list_get3 = call double @list_getfloat({ i32*, double* }* %list_load,
     i32 %right_idx2)

```

```

960.  call void @list_setfloat({ i32*, double* }* %list_load, i32 %left_idx1,
    double %list_get3)
961.  call void @list_setfloat({ i32*, double* }* %list_load, i32 %right_idx2,
    double %list_get)
962.  %tmp4 = add i32 %left_idx1, 1
963.  store i32 %tmp4, i32* %left_idx
964.  %tmp5 = sub i32 %right_idx2, 1
965.  store i32 %tmp5, i32* %right_idx
966.  br label %while
967.
968. merge:                                ; preds = %while
969.  ret void
970. }
971.
972. define i32 @main() {
973. entry:
974.  %a = alloca { i32*, i32* }
975.  %list_size_ptr = getelementptr inbounds { i32*, i32* }, { i32*, i32* }* %a, i32 0,
    i32 0
976.  %list_size = alloca i32
977.  store i32 0, i32* %list_size
978.  store i32* %list_size, i32** %list_size_ptr
979.  %list.array = getelementptr inbounds { i32*, i32* }, { i32*, i32* }* %a, i32 0, i32 1
980.  %p = alloca i32, i32 1028
981.  store i32* %p, i32** %list.array
982.  %i = alloca i32
983.  %new_list_ptr = alloca { i32*, i32* }
984.  %list_size_ptr1 = getelementptr inbounds { i32*, i32* }, { i32*,
    i32* }* %new_list_ptr, i32 0, i32 0
985.  %list_size2 = alloca i32
986.  store i32 0, i32* %list_size2
987.  store i32* %list_size2, i32** %list_size_ptr1
988.  %list.array3 = getelementptr inbounds { i32*, i32* }, { i32*, i32* }* %new_list_ptr,
    i32 0, i32 1
989.  %p4 = alloca i32, i32 1028
990.  store i32* %p4, i32** %list.array3
991.  call void @list_pushint({ i32*, i32* }* %new_list_ptr, i32 10)
992.  call void @list_pushint({ i32*, i32* }* %new_list_ptr, i32 9)
993.  call void @list_pushint({ i32*, i32* }* %new_list_ptr, i32 8)
994.  call void @list_pushint({ i32*, i32* }* %new_list_ptr, i32 7)
995.  call void @list_pushint({ i32*, i32* }* %new_list_ptr, i32 6)
996.  call void @list_pushint({ i32*, i32* }* %new_list_ptr, i32 5)
997.  call void @list_pushint({ i32*, i32* }* %new_list_ptr, i32 4)
998.  call void @list_pushint({ i32*, i32* }* %new_list_ptr, i32 3)
999.  call void @list_pushint({ i32*, i32* }* %new_list_ptr, i32 2)
1000.  call void @list_pushint({ i32*, i32* }* %new_list_ptr, i32 1)
1001.  call void @list_pushint({ i32*, i32* }* %new_list_ptr, i32 0)
1002.  %new_list = load { i32*, i32* }, { i32*, i32* }* %new_list_ptr
1003.  store { i32*, i32* } %new_list, { i32*, i32* }* %a
1004.  %a5 = load { i32*, i32* }, { i32*, i32* }* %a
1005.  call void @quickSort({ i32*, i32* } %a5)
1006.  store i32 0, i32* %i
1007.  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.array = getelementptr inbounds { i32*, i32* }, { i32*, i32* }* %a1, i32
0, i32 1
1036.     %p = alloca i32, i32 1028
1037.     store i32* %p, i32** %list.array
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.array = getelementptr inbounds { i32*, i32* }, { i32*, i32* }* %a1, i32
0, i32 1
1054.     %p = alloca i32, i32 1028

```

```

1055.     store i32* %p, i32** %list.array
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.array = getelementptr inbounds { i32*, i32* }, { i32*, i32* }* %a1, i32
0, i32 1
1100.     %p = alloca i32, i32 1028
1101.     store i32* %p, i32** %list.array
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* %tmp
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.     %tmp34 = load i32, i32* %tmp
1171.     %high35 = load i32, i32* %high3
1172.     call void @list_setint({ i32*, i32* }* %a1, i32 %high35, i32 %tmp34)
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.

```
1. --- SUCCESS TEST CASES --
2. int add(int x, int y)
3. {
4.     return x + y;
5. }
6.
7. int main()
8. {
9.     printi( add(17, 25) );
10.    return 0;
11. }
12.
13. output: 42
14. --- Test Case
15. int main()
16. {
17.     printi(39 + 3);
18.     return 0;
19. }
20. output: 42
21. --- Test Case
22. int main()
23. {
24.     printi(1 + 2 * 3 + 4);
25.     return 0;
26. }
27. output: 11
28. --- Test Case
29. int foo(int a)
30. {
31.     return a;
32. }
33.
34. int main()
35. {
36.     int a;
37.     a = 42;
38.     a = a + 5;
39.     printi(a);
40.     return 0;
41. }
42. output: 47
43. --- Test Case
44. int fib(int x)
```

```
45. {
46.     if (x < 2) return 1;
47.     return fib(x-1) + fib(x-2);
48. }
49.
50. int main()
51. {
52.     printi(fib(0));
53.     printi(fib(1));
54.     printi(fib(2));
55.     printi(fib(3));
56.     printi(fib(4));
57.     printi(fib(5));
58.     return 0;
59. }
60. output:
61. 1
62. 1
63. 2
64. 3
65. 5
66. 8
67. --- Test Case
68. int main()
69. {
70.     float a;
71.     a = 3.14159267;
72.     printf(a);
73.     return 0;
74. }
75. output: 3.14159
76. --- Test Case
77. int main()
78. {
79.     float a;
80.     float b;
81.     float c;
82.     a = 3.14159267;
83.     b = -2.71828;
84.     c = a + b;
85.     printf(c);
86.     return 0;
87. }
88. output: 0.423313
89. --- Test Case
90. void testfloat(float a, float b)
91. {
92.     printf(a + b);
93.     printf(a - b);
94.     printf(a * b);
95.     printf(a / b);
96.     printb(a == b);
97.     printb(a == a);
```

```
98.     printb(a != b);
99.     printb(a != a);
100.         printb(a > b);
101.         printb(a >= b);
102.         printb(a < b);
103.         printb(a <= b);
104.     }
105.
106.     int main()
107.     {
108.         float c;
109.         float d;
110.
111.         c = 42.0;
112.         d = 3.14159;
113.
114.         testfloat(c, d);
115.
116.         testfloat(d, d);
117.
118.         return 0;
119.     }
120.     output:
121.     45.1416
122.     38.8584
123.     131.947
124.     13.369
125.     0
126.     1
127.     1
128.     0
129.     1
130.     1
131.     0
132.     0
133.     6.28318
134.     0
135.     9.86959
136.     1
137.     1
138.     1
139.     0
140.     0
141.     0
142.     1
143.     0
144.     1
145.     --- Test Case
146.     int main()
147.     {
148.         int i;
149.         for (i = 0 ; i < 5 ; i = i + 1) {
```

```
150.     printi(i);
151.     }
152.     printi(42);
153.     return 0;
154.     }
155. output:
156. 0
157. 1
158. 2
159. 3
160. 4
161. 42
162. --- Test Case
163. int main()
164. {
165.     int i;
166.     i = 0;
167.     for ( ; i < 5; ) {
168.         printi(i);
169.         i = i + 1;
170.     }
171.     printi(42);
172.     return 0;
173. }
174. output:
175. 0
176. 1
177. 2
178. 3
179. 4
180. 42
181. --- Test Case
182. int add(int a, int b)
183. {
184.     return a + b;
185. }
186.
187. int main()
188. {
189.     int a;
190.     a = add(39, 3);
191.     printi(a);
192.     return 0;
193. }
194. output: 42
195. --- Test Case
196. /* Bug noticed by Pin-Chin Huang */
197.
198. int fun(int x, int y)
199. {
200.     return 0;
201. }
202.
```

```
203.     int main()
204.     {
205.         int i;
206.         i = 1;
207.
208.         fun(i = 2, i = i+1);
209.
210.         printi(i);
211.         return 0;
212.     }
213.
214.     output: 2
215.     --- Test Case
216.     void printem(int a, int b, int c, int d)
217.     {
218.         printi(a);
219.         printi(b);
220.         printi(c);
221.         printi(d);
222.     }
223.
224.     int main()
225.     {
226.         printem(42,17,192,8);
227.         return 0;
228.     }
229.     output:
230.     42
231.     17
232.     192
233.     8
234.     --- Test Case
235.     int add(int a, int b)
236.     {
237.         int c;
238.         c = a + b;
239.         return c;
240.     }
241.
242.     int main()
243.     {
244.         int d;
245.         d = add(52, 10);
246.         printi(d);
247.         return 0;
248.     }
249.     output: 62
250.     --- Test Case
251.     int foo(int a)
252.     {
253.         return a;
254.     }
```

```
255.
256.     int main()
257.     {
258.         return 0;
259.     }
260.     --- Test Case
261.     void foo() {}
262.
263.     int bar(int a, bool b, int c) { return a + c; }
264.
265.     int main()
266.     {
267.         printi(bar(17, false, 25));
268.         return 0;
269.     }
270.     output: 42
271.     --- Test Case
272.     int a;
273.
274.     void foo(int c)
275.     {
276.         a = c + 42;
277.     }
278.
279.     int main()
280.     {
281.         foo(73);
282.         printi(a);
283.         return 0;
284.     }
285.     output: 115
286.     --- Test Case
287.     void foo(int a)
288.     {
289.         printi(a + 3);
290.     }
291.
292.     int main()
293.     {
294.         foo(40);
295.         return 0;
296.     }
297.     output: 43
298.     --- Test Case
299.     void foo(int a)
300.     {
301.         printi(a + 3);
302.         return;
303.     }
304.
305.     int main()
306.     {
```



```

307.     foo(40);
308.     return 0;
309. }
310. output: 43
311. --- Test Case
312. int gcd(int a, int b) {
313.     while (a != b) {
314.         if (a > b) a = a - b;
315.         else b = b - a;
316.     }
317.     return a;
318. }
319.
320. int main()
321. {
322.     printi(gcd(2,14));
323.     printi(gcd(3,15));
324.     printi(gcd(99,121));
325.     return 0;
326. }
327. output:
328. 2
329. 3
330. 11
331. --- Test Case
332. int gcd(int a, int b) {
333.     while (a != b)
334.         if (a > b) a = a - b;
335.         else b = b - a;
336.     return a;
337. }
338.
339. int main()
340. {
341.     printi(gcd(14,21));
342.     printi(gcd(8,36));
343.     printi(gcd(99,121));
344.     return 0;
345. }
346. output:
347. 7
348. 4
349. 11
350. int a;
351. int b;
352. --- Test Case
353. void printa()
354. {
355.     printi(a);
356. }
357.
358. void printbb()

```

```

359.     {
360.         printi(b);
361.     }
362.
363.     void incab()
364.     {
365.         a = a + 1;
366.         b = b + 1;
367.     }
368.
369.     int main()
370.     {
371.         a = 42;
372.         b = 21;
373.         printa();
374.         printbb();
375.         incab();
376.         printa();
377.         printbb();
378.         return 0;
379.     }
380.     output:
381.     42
382.     21
383.     43
384.     22
385.     bool i;
386.
387.     int main()
388.     {
389.         int i; /* Should hide the global i */
390.
391.         i = 42;
392.         printi(i + i);
393.         return 0;
394.     }
395.     output: 84
396.     --- Test Case
397.     int i;
398.     bool b;
399.     int j;
400.
401.     int main()
402.     {
403.         i = 42;
404.         j = 10;
405.         printi(i + j);
406.         return 0;
407.     }
408.     output: 52
409.     --- Test Case
410.     /*

```

```
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
```

```
464. 17
465. --- Test Case
466. int cond(bool b)
467. {
468.     int x;
469.     if (b)
470.         x = 42;
471.     else
472.         x = 17;
473.     return x;
474. }
475.
476. int main()
477. {
478.     printi(cond(true));
479.     printi(cond(false));
480.     return 0;
481. }
482. output:
483. 42
484. 17
485. --- Test Case
486. int cond(bool b)
487. {
488.     int x;
489.     x = 10;
490.     if (b)
491.         if (x == 10)
492.             x = 42;
493.     else
494.         x = 17;
495.     return x;
496. }
497.
498. int main()
499. {
500.     printi(cond(true));
501.     printi(cond(false));
502.     return 0;
503. }
504. output:
505. 42
506. 10
507. --- Test Case
508. int main()
509. {
510.     list<int> a;
511.     list<float> b;
512.     list<bool> c;
513.     list_push(a, 1);
514.     list_push(b, 3.14);
515.     list_push(c, false);
516.     printi(a[0]);
```

```

517.     printf(b[0]);
518.     printf(c[0]);
519.
520.     return 0;
521. }
522. output:
523. 1
524. 3.14
525. 0
526. --- Test Case
527. int main()
528. {
529.     list<int> a;
530.     list_push(a, 100);
531.     list_push(a, 101);
532.     printi(list_get(a, 0));
533.     printi(a[0]);
534.     printi(list_get(a, 1));
535.     printi(a[1]);
536.     return 0;
537. }
538. output:
539. 100
540. 100
541. 101
542. 101
543. --- Test Case
544. int main()
545. {
546.     list<int> a;
547.     int i;
548.     int x;
549.     for (i=0; i<10; ++i) {
550.         list_push(a, i);
551.     }
552.     x = list_pop(a);
553.     printi(x);
554.     x = list_pop(a);
555.     printi(x);
556.     x = list_size(a);
557.     printi(x);
558.     return 0;
559. }
560. output:
561. 9
562. 8
563. 8
564. --- Test Case
565. int main()
566. {
567.     list<int> a;
568.     list_push(a, 100);
569.     list_push(a, 101);

```

```

570.     printi(list_get(a, 0));
571.     printi(a[0]);
572.     printi(list_get(a, 1));
573.     printi(a[1]);
574.     list_set(a, 0, 102);
575.     list_set(a, 1, 103);
576.     printi(a[0]);
577.     printi(a[1]);
578.     a[0] = 104;
579.     a[1] = 105;
580.     printi(a[0]);
581.     printi(a[1]);
582.     return 0;
583. }
584. output:
585. 100
586. 100
587. 101
588. 101
589. 102
590. 103
591. 104
592. 105
593. --- Test Case
594. int main()
595. {
596.     list<int> a;
597.     list<int> b;
598.     list<int> c;
599.     int i;
600.     c = a;
601.     for (i=0; i<5; ++i) { list_push(a, i); }
602.     b = a;
603.     printi(b[0]);
604.     printi(#b);
605.     list_push(b, 100);
606.     printi(#b);
607.     printi(b[#b-1]);
608.     list_push(c, 101);
609.     printi(#c);
610.     printi(c[#c-1]);
611.     list_push(a, 102);
612.     printi(#a);
613.     printi(a[#a-1]);
614.
615.     return 0;
616. }
617. output:
618. 0
619. 5
620. 6
621. 100
622. 7

```

```
623.     101
624.     8
625.     102
626.     --- Test Case
627.     int main()
628.     {
629.         list<int> a;
630.         list<int> b;
631.         int i;
632.         for (i=0; i<10; ++i) {
633.             list_push(a, i);
634.         }
635.         b = a[2:7];
636.         printi(#b);
637.         for (i=0; i<#b; ++i) {
638.             printi(b[i]);
639.         }
640.
641.         return 0;
642.     }
643.     output:
644.     6
645.     2
646.     3
647.     4
648.     5
649.     6
650.     7
651.     --- Test Case
652.     void list_push2(list<int> l, int val) {
653.         list_push(l, val);
654.         return;
655.     }
656.
657.     int list_size2(list<int> l) {
658.         return #l;
659.     }
660.
661.     int main()
662.     {
663.         list<int> a;
664.         int i;
665.         for (i=0; i<5; ++i) {
666.             list_push(a, i);
667.         }
668.         list_push2(a, 100);
669.         list_push2(a, 101);
670.         for (i=0; i<#a; ++i) {
671.             printi(a[i]);
672.         }
673.         printi(list_size2(a));
674.
```

```

675.     return 0;
676. }
677. output:
678. 0
679. 1
680. 2
681. 3
682. 4
683. 100
684. 101
685. 7
686. --- Test Case
687. int main()
688. {
689.     list<int> a;
690.     int i;
691.     for (i=0; i<5; ++i) {
692.         list_push(a, i);
693.     }
694.     printi(#a);
695.     list_clear(a);
696.     printi(#a);
697.     list_push(a, 10);
698.     list_push(a, 20);
699.     list_push(a, 30);
700.     printi(#a);
701.     for (i=0; i<3; ++i) {
702.         printi(a[i]);
703.     }
704.
705.     return 0;
706. }
707. output:
708. 5
709. 0
710. 3
711. 10
712. 20
713. 30
714. --- Test Case
715. int main()
716. {
717.     list<int> a;
718.     int i;
719.     for (i=0; i<5; ++i) {
720.         list_push(a, i);
721.     }
722.     printi(list_find(a, 2));
723.     printi(list_find(a, 6));
724.     return 0;
725. }
726. output:
727. 2

```



```

728.     -1
729.     --- Test Case
730.     int main()
731.     {
732.         list<int> a;
733.         int i;
734.         for (i=0; i<5; ++i) {
735.             list_push(a, i);
736.         }
737.         list_remove(a, 3);
738.         for (i=0; i<#a; ++i) {
739.             printi(a[i]);
740.         }
741.         list_remove(a, 9);
742.         printi(#a);
743.         list_remove(a, 4);
744.         for (i=0; i<#a; ++i) {
745.             printi(a[i]);
746.         }
747.         return 0;
748.     }
749.     output:
750.     0
751.     1
752.     2
753.     4
754.     4
755.     0
756.     1
757.     2
758.     --- Test Case
759.     int main()
760.     {
761.         list<int> a;
762.         int i;
763.         for (i=0; i<5; ++i) {
764.             list_push(a, i);
765.         }
766.
767.         list_insert(a, 0, 99);
768.         printi(#a);
769.         for (i=0; i<#a; ++i) {
770.             printi(a[i]);
771.         }
772.
773.         list_insert(a, #a-1, 101);
774.         printi(#a);
775.         for (i=0; i<#a; ++i) {
776.             printi(a[i]);
777.         }
778.
779.         list_insert(a, #a, 102);

```

```
780.     printi(#a);
781.     for (i=0; i<#a; ++i) {
782.         printi(a[i]);
783.     }
784.
785.     list_insert(a, 2, 103);
786.     printi(#a);
787.     for (i=0; i<#a; ++i) {
788.         printi(a[i]);
789.     }
790.
791.     return 0;
792. }
793. output:
794. 6
795. 99
796. 0
797. 1
798. 2
799. 3
800. 4
801. 7
802. 99
803. 0
804. 1
805. 2
806. 3
807. 101
808. 4
809. 8
810. 99
811. 0
812. 1
813. 2
814. 3
815. 101
816. 4
817. 102
818. 9
819. 99
820. 0
821. 103
822. 1
823. 2
824. 3
825. 101
826. 4
827. 102
828. --- Test Case
829. int main()
830. {
831.     list<int> a;
832.     int i;
```

```

833.     for (i=0; i<5; ++i) {
834.         list_push(a, i);
835.     }
836.     list_rev(a);
837.     for (i=0; i<#a; ++i) {
838.         printi(a[i]);
839.     }
840.     list_insert(a, 0, 5);
841.     list_rev(a);
842.     for (i=0; i<#a; ++i) {
843.         printi(a[i]);
844.     }
845.
846.     return 0;
847. }
848. output:
849. 4
850. 3
851. 2
852. 1
853. 0
854. 0
855. 1
856. 2
857. 3
858. 4
859. 5
860. --- Test Case
861. int main()
862. {
863.     list<int> a;
864.     list<float> b;
865.     list<bool> c;
866.     int i;
867.
868.     a = [0, 1, 2, 3];
869.     for (i=0; i<#a; ++i) {
870.         printi(a[i]);
871.     }
872.
873.     b = [0.2, 1.1, 2.4, 3.14];
874.     for (i=0; i<#b; ++i) {
875.         printf(b[i]);
876.     }
877.
878.     c = [true, true, false, true];
879.     for (i=0; i<#c; ++i) {
880.         printb(c[i]);
881.     }
882.
883.     return 0;
884. }

```

```
885.     output:
886.     0
887.     1
888.     2
889.     3
890.     0.2
891.     1.1
892.     2.4
893.     3.14
894.     1
895.     1
896.     0
897.     1
898.     --- Test Case
899.     int main()
900.     {
901.         list<int> a;
902.         list<int> b;
903.         int i;
904.
905.         a = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10];
906.
907.         b = a[2:6];
908.         for (i=0; i<#b; ++i) {
909.             printi(b[i]);
910.         }
911.
912.         b = a[:6];
913.         for (i=0; i<#b; ++i) {
914.             printi(b[i]);
915.         }
916.
917.         b = a[6:];
918.         for (i=0; i<#b; ++i) {
919.             printi(b[i]);
920.         }
921.
922.         b = a[:];
923.         for (i=0; i<#b; ++i) {
924.             printi(b[i]);
925.         }
926.
927.         return 0;
928.     }
929.     output:
930.     2
931.     3
932.     4
933.     5
934.     6
935.     0
936.     1
```

```
937.      2
938.      3
939.      4
940.      5
941.      6
942.      6
943.      7
944.      8
945.      9
946.     10
947.      0
948.      1
949.      2
950.      3
951.      4
952.      5
953.      6
954.      7
955.      8
956.      9
957.     10
958.     --- Test Case
959.     void foo(bool i)
960.     {
961.         int i; /* Should hide the formal i */
962.
963.         i = 42;
964.         printi(i + i);
965.     }
966.
967.     int main()
968.     {
969.         foo(true);
970.         return 0;
971.     }
972.     output: 84
973.     --- Test Case
974.     int foo(int a, bool b)
975.     {
976.         int c;
977.         bool d;
978.
979.         c = a;
980.
981.         return c + 10;
982.     }
983.
984.     int main() {
985.         printi(foo(37, false));
986.         return 0;
987.     }
988.     output: 47
```

```
989.     --- Test Case
990.     int main()
991.     {
992.         int x;
993.         x = 1;
994.         printi(x);
995.         printi(--x);
996.         printi(x);
997.         return 0;
998.     }
999.     output:
1000.    1
1001.    0
1002.    0
1003.     --- Test Case
1004.     int main()
1005.     {
1006.         int x;
1007.         x = 1;
1008.         printi(x);
1009.         printi(x--);
1010.         printi(x);
1011.         return 0;
1012.     }
1013.     output:
1014.    1
1015.    1
1016.    0
1017.     --- Test Case
1018.     int main()
1019.     {
1020.         int x;
1021.         int y;
1022.         x = 10;
1023.         y = 3;
1024.         printi(x % y);
1025.         x = 10;
1026.         y = 4;
1027.         printi(x % y);
1028.         x = 10;
1029.         y = 5;
1030.         printi(x % y);
1031.         x = 0;
1032.         y = 10;
1033.         printi(x % y);
1034.     }
1035.     output:
1036.    1
1037.    2
1038.    0
1039.    0
1040.     --- Test Case
1041.     int main()
```

```
1042.  {
1043.    printi(1 + 2);
1044.    printi(1 - 2);
1045.    printi(1 * 2);
1046.    printi(100 / 2);
1047.    printi(99);
1048.    printb(1 == 2);
1049.    printb(1 == 1);
1050.    printi(99);
1051.    printb(1 != 2);
1052.    printb(1 != 1);
1053.    printi(99);
1054.    printb(1 < 2);
1055.    printb(2 < 1);
1056.    printi(99);
1057.    printb(1 <= 2);
1058.    printb(1 <= 1);
1059.    printb(2 <= 1);
1060.    printi(99);
1061.    printb(1 > 2);
1062.    printb(2 > 1);
1063.    printi(99);
1064.    printb(1 >= 2);
1065.    printb(1 >= 1);
1066.    printb(2 >= 1);
1067.    return 0;
1068.  }
1069.  output:
1070.  3
1071.  -1
1072.  2
1073.  50
1074.  99
1075.  0
1076.  1
1077.  99
1078.  1
1079.  0
1080.  99
1081.  1
1082.  0
1083.  99
1084.  1
1085.  1
1086.  0
1087.  99
1088.  0
1089.  1
1090.  99
1091.  0
1092.  1
1093.  1
1094.  --- Test Case
```

```
1095.     int main()
1096.     {
1097.         printb(true);
1098.         printb(false);
1099.         printb(true && true);
1100.         printb(true && false);
1101.         printb(false && true);
1102.         printb(false && false);
1103.         printb(true || true);
1104.         printb(true || false);
1105.         printb(false || true);
1106.         printb(false || false);
1107.         printb(!false);
1108.         printb(!true);
1109.         printi(-10);
1110.     }
```

1111. output:

```
1112.     1
1113.     0
1114.     1
1115.     0
1116.     0
1117.     0
1118.     1
1119.     1
1120.     1
1121.     0
1122.     1
1123.     0
1124.     -10
```

1125. --- Test Case

```
1126.     int main()
1127.     {
1128.         int x;
1129.         x = 0;
1130.         printi(x);
1131.         printi(++x);
1132.         printi(x);
1133.         return 0;
1134.     }
```

1135. output:

```
1136.     0
1137.     1
1138.     1
```

1139. --- Test Case

```
1140.     int main()
1141.     {
1142.         int x;
1143.         x = 0;
1144.         printi(x);
1145.         printi(x++);
1146.         printi(x);
1147.         return 0;
1147.     }
```



```

1148.     }
1149.     output:
1150.     0
1151.     0
1152.     1
1153.     --- Test Case
1154.     /* merges two sorted sublists of arr[] (arr[0..m], arr[m+1..r]) in-place with
        QuickSort algorithm */
1155.
1156.     int partition(list<int> a, int low, int high)
1157.     {
1158.         int i;
1159.         int j;
1160.         int pivot;
1161.         int temp;
1162.
1163.         pivot = a[high];
1164.         i = (low - 1);
1165.         for (j = low; j <= high - 1; j++)
1166.         {
1167.             if (a[j] <= pivot)
1168.             {
1169.                 i++;
1170.                 temp = a[i];
1171.                 a[i] = a[j];
1172.                 a[j] = temp;
1173.             }
1174.         }
1175.         temp = a[i+1];
1176.         a[i+1] = a[high];
1177.         a[high] = temp;
1178.         return (i + 1);
1179.     }
1180.
1181.     void quickSortImpl(list<int> a, int low, int high)
1182.     {
1183.         int pi;
1184.         if (low < high)
1185.         {
1186.             pi = partition(a, low, high);
1187.             quickSortImpl(a, low, pi - 1);
1188.             quickSortImpl(a, pi + 1, high);
1189.         }
1190.         return;
1191.     }
1192.
1193.     void quickSort(list<int> a)
1194.     {
1195.         quickSortImpl(a, 0, #a - 1);
1196.         return;
1197.     }
1198.

```

```
1199.     int main()
1200.     {
1201.         list<int> a;
1202.         int i;
1203.
1204.         a = [10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0];
1205.
1206.         quickSort(a);
1207.
1208.         for (i=0; i<#a; ++i) {
1209.             printi(a[i]);
1210.         }
1211.
1212.         return 0;
1213.     }
1214.     output:
1215.     0
1216.     1
1217.     2
1218.     3
1219.     4
1220.     5
1221.     6
1222.     7
1223.     8
1224.     9
1225.     10
1226.     --- Test Case
1227.     int main()
1228.     {
1229.         int a;
1230.         a = 42;
1231.         printi(a);
1232.         return 0;
1233.     }
1234.     output: 42
1235.     --- Test Case
1236.     int a;
1237.
1238.     void foo(int c)
1239.     {
1240.         a = c + 42;
1241.     }
1242.     --- Test Case
1243.     int main()
1244.     {
1245.         foo(73);
1246.         printi(a);
1247.         return 0;
1248.     }
1249.     output: 115
1250.     --- Test Case
```

```
1251.     int main()
1252.     {
1253.         int i;
1254.         i = 5;
1255.         while (i > 0) {
1256.             printi(i);
1257.             i = i - 1;
1258.         }
1259.         printi(42);
1260.         return 0;
1261.     }
1262.     output:
1263.     5
1264.     4
1265.     3
1266.     2
1267.     1
1268.     42
1269.     --- Test Case
1270.     int foo(int a)
1271.     {
1272.         int j;
1273.         j = 0;
1274.         while (a > 0) {
1275.             j = j + 2;
1276.             a = a - 1;
1277.         }
1278.         return j;
1279.     }
1280.
1281.     int main()
1282.     {
1283.         printi(foo(7));
1284.         return 0;
1285.     }
1286.     output: 14
```

```

1288.      -- FAILURE TEST CASES --
1289.      Fatal error: exception Failure("illegal assignment int = bool in i = false")
1290.      int main()
1291.      {
1292.          int i;
1293.          bool b;
1294.
1295.          i = 42;
1296.          i = 10;
1297.          b = true;
1298.          b = false;
1299.          i = false; /* Fail: assigning a bool to an integer */
1300.      }
1301.      --- Test Case
1302.      Fatal error: exception Failure("illegal assignment bool = int in b = 48")
1303.      int main()
1304.      {
1305.          int i;
1306.          bool b;
1307.
1308.          b = 48; /* Fail: assigning an integer to a bool */
1309.      }
1310.      --- Test Case
1311.      Fatal error: exception Failure("illegal assignment int = void in i = myvoid()")
1312.      void myvoid()
1313.      {
1314.          return;
1315.      }
1316.
1317.      int main()
1318.      {
1319.          int i;
1320.
1321.          i = myvoid(); /* Fail: assigning a void to an integer */
1322.      }
1323.      --- Test Case
1324.      Fatal error: exception Failure("nothing may follow a return")
1325.      int main()
1326.      {
1327.          int i;
1328.
1329.          i = 15;
1330.          return i;
1331.          i = 32; /* Error: code after a return */
1332.      }
1333.      --- Test Case
1334.      Fatal error: exception Failure("nothing may follow a return")
1335.      int main()
1336.      {
1337.          int i;
1338.
1339.          {

```

```
1340.         i = 15;
1341.         return i;
1342.     }
1343.     i = 32; /* Error: code after a return */
1344. }
1345. --- Test Case
1346. Fatal error: exception Failure("illegal binary operator bool + int in d + a")
1347. int a;
1348. bool b;
1349.
1350. void foo(int c, bool d)
1351. {
1352.     int dd;
1353.     bool e;
1354.     a + c;
1355.     c - a;
1356.     a * 3;
1357.     c / 2;
1358.     d + a; /* Error: bool + int */
1359. }
1360.
1361. int main()
1362. {
1363.     return 0;
1364. }
1365. --- Test Case
1366. Fatal error: exception Failure("illegal binary operator bool + int in b + a")
1367. int a;
1368. bool b;
1369.
1370. void foo(int c, bool d)
1371. {
1372.     int d;
1373.     bool e;
1374.     b + a; /* Error: bool + int */
1375. }
1376.
1377. int main()
1378. {
1379.     return 0;
1380. }
1381. --- Test Case
1382. Fatal error: exception Failure("illegal binary operator float + int in b + a")
1383. int a;
1384. float b;
1385.
1386. void foo(int c, float d)
1387. {
1388.     int d;
1389.     float e;
1390.     b + a; /* Error: float + int */
1391. }
```

```

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 &&
1399.         1")
1400.     int main()
1401.     {
1402.         -3.5 && 1; /* Float with AND? */
1403.         return 0;
1404.     }
1405.     --- Test Case
1406.     Fatal error: exception Failure("illegal binary operator float && float in -3.5
1407.         && 2.5")
1408.     int main()
1409.     {
1410.         -3.5 && 2.5; /* Float with AND? */
1411.         return 0;
1412.     }
1413.     --- Test Case
1414.     Fatal error: exception Failure("undeclared identifier j")
1415.     int main()
1416.     {
1417.         int i;
1418.         for ( ; true ; ) {} /* OK: Forever */
1419.
1420.         for (i = 0 ; i < 10 ; i = i + 1) {
1421.             if (i == 3) return 42;
1422.         }
1423.
1424.         for (j = 0; i < 10 ; i = i + 1) {} /* j undefined */
1425.
1426.         return 0;
1427.     }
1428.     --- Test Case
1429.     Fatal error: exception Failure("undeclared identifier j")
1430.     int main()
1431.     {
1432.         int i;
1433.
1434.         for (i = 0; j < 10 ; i = i + 1) {} /* j undefined */
1435.
1436.         return 0;
1437.     }
1438.     --- Test Case
1439.     Fatal error: exception Failure("expected Boolean expression in i")
1440.     int main()
1441.     {
1442.         int i;

```

```

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

```

```

1493. Fatal error: exception Failure("illegal void formal b")
1494. int foo(int a, bool b, int c) { }
1495.
1496. void bar(int a, void b, int c) {} /* Error: illegal void formal b */
1497.
1498. int main()
1499. {
1500.     return 0;
1501. }
1502. --- Test Case
1503. Fatal error: exception Failure("function printi may not be defined")
1504. int foo() {}
1505.
1506. void bar() {}
1507.
1508. int printi() {} /* Should not be able to define print */
1509.
1510. void baz() {}
1511.
1512. int main()
1513. {
1514.     return 0;
1515. }
1516. --- Test Case
1517. Fatal error: exception Failure("illegal void local b")
1518. int foo() {}
1519.
1520. int bar() {
1521.     int a;
1522.     void b; /* Error: illegal void local b */
1523.     bool c;
1524.
1525.     return 0;
1526. }
1527.
1528. int main()
1529. {
1530.     return 0;
1531. }
1532. --- Test Case
1533. Fatal error: exception Failure("expecting 2 arguments in foo(42)")
1534. void foo(int a, bool b)
1535. {
1536. }
1537.
1538. int main()
1539. {
1540.     foo(42, true);
1541.     foo(42); /* Wrong number of arguments */
1542. }
1543. --- 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;
```

```

1595.     int a;
1596.     int b; /* Duplicate global variable */
1597.
1598.     int main()
1599.     {
1600.         return 0;
1601.     }
1602.     --- Test Case
1603.     Fatal error: exception Failure("expected Boolean expression in 42")
1604.     int main()
1605.     {
1606.         if (true) {}
1607.         if (false) {} else {}
1608.         if (42) {} /* Error: non-bool predicate */
1609.     }
1610.     --- Test Case
1611.     Fatal error: exception Failure("undeclared identifier foo")
1612.     int main()
1613.     {
1614.         if (true) {
1615.             foo; /* Error: undeclared variable */
1616.         }
1617.     }
1618.     --- Test Case
1619.     Fatal error: exception Failure("undeclared identifier bar")
1620.     int main()
1621.     {
1622.         if (true) {
1623.             42;
1624.         } else {
1625.             bar; /* Error: undeclared variable */
1626.         }
1627.     }
1628.     --- Test Case
1629.     Fatal error: exception Failure("unrecognized function main")
1630.     Fatal error: exception Failure("return gives bool expected int in true")
1631.     int main()
1632.     {
1633.         return true; /* Should return int */
1634.     }
1635.     --- Test Case
1636.     Fatal error: exception Failure("return gives int expected void in 42")
1637.     void foo()
1638.     {
1639.         if (true) return 42; /* Should return void */
1640.         else return;
1641.     }
1642.
1643.     int main()
1644.     {
1645.         return 42;
1646.     }
1647.     --- Test Case

```

```
1648. Fatal error: exception Failure("expected Boolean expression in 42")
1649. int main()
1650. {
1651.     int i;
1652.
1653.     while (true) {
1654.         i = i + 1;
1655.     }
1656.
1657.     while (42) { /* Should be boolean */
1658.         i = i + 1;
1659.     }
1660.
1661. }
1662. --- Test Case
1663. Fatal error: exception Failure("unrecognized function foo")
1664. int main()
1665. {
1666.     int i;
1667.
1668.     while (true) {
1669.         i = i + 1;
1670.     }
1671.
1672.     while (true) {
1673.         foo(); /* foo undefined */
1674.     }
1675.
1676. }
1677.
```

## 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

```
1. # "make test" Compiles everything and runs the regression tests
2.
3. .PHONY : test
4. test : all testall.sh
5.     ./testall.sh
6.
7. # "make all" builds the executable as well as the "printbig" library designed
8. # to test linking external code
9.
10. .PHONY : all
11. all : ap_plusplus.native
12.
13. # "make ap_plusplus.native" compiles the compiler
14. #
15. # The _tags file controls the operation of ocamlbuild, e.g., by including
16. # packages, enabling warnings
17. #
18. # See https://github.com/ocaml/ocamlbuild/blob/master/manual/manual.adoc
19.
20. ap_plusplus.native :
21.     opam config exec -- \
22.     ocamlbuild -use-ocamlfind ap_plusplus.native
23.
24. # "make clean" removes all generated files
25.
26. .PHONY : clean
27. clean :
28.     ocamlbuild -clean
29.     rm -rf testall.log ocamllvm *.diff
30.
31. printbig : printbig.c
32.     cc -o printbig -DBUILD_TEST printbig.c
33.
34. # Building the tarball
35.
36. TESTS = \
37.     helloworld
38.
39. TESTFILES = $(TESTS:%=test-%.mc) $(TESTS:%=test-%.out)
40.
41. TARFILES = scanner.mll parser.mly ast.ml sast.ml semant.ml codegen.ml Makefile
42.     ap_plusplus.ml testall.sh \
43.     _tags \
```

```
43. printbig.c \  
44. $(TESTFILES:%=tests/%)  
45.  
46. ap++.tar.gz : $(TARFILES)  
47. cd .. && tar czf ap++/ap++.tar.gz \  
48. $(TARFILES:%=ap++/%)
```

## testall.sh

```
1. #!/bin/sh
2.
3. # Regression testing script for `
4. # Step through a list of files
5. # Compile, run, and check the output of each expected-to-work test
6. # Compile and check the error of each expected-to-fail test
7.
8. # Path to the LLVM interpreter
9. #LLI="lli"
10. LLI="/usr/local/opt/llvm/bin/lli"
11.
12. # Path to the LLVM compiler
13. LLC="/usr/local/opt/llvm/bin/llc"
14.
15. # Path to the C compiler
16. CC="cc"
17.
18. # Path to the microc compiler. Usually "./microc.native"
19. # Try "_build/microc.native" if ocamlbuild was unable to create a symbolic link.
20. AP_PLUSPLUS="./ap_plusplus.native"
21. #MICROC="_build/microc.native"
22.
23. # Set time limit for all operations
24. ulimit -t 30
25.
26. globallog=testall.log
27. rm -f $globallog
28. error=0
29. globalerror=0
30.
31. keep=0
32.
33. Usage() {
34.     echo "Usage: testall.sh [options] [.mc files]"
35.     echo "-k    Keep intermediate files"
36.     echo "-h    Print this help"
37.     exit 1
38. }
39.
40. SignalError() {
41.     if [ $error -eq 0 ] ; then
42.         echo "FAILED"
43.         error=1
44.     fi
45.     echo " $1"
46. }
47.
48. # Compare <outfile> <reffile> <difffile>
```

```

49. # Compares the outfile with reffile. Differences, if any, written to difffile
50. Compare() {
51.     generatedfiles="$generatedfiles $3"
52.     echo diff -b $1 $2 ">" $3 1>&2
53.     diff -b "$1" "$2" > "$3" 2>&1 || {
54.     SignalError "$1 differs"
55.     echo "FAILED $1 differs from $2" 1>&2
56.     }
57. }
58.
59. # Run <args>
60. # Report the command, run it, and report any errors
61. Run() {
62.     echo $* 1>&2
63.     eval $* || {
64.     SignalError "$1 failed on $*"
65.     return 1
66.     }
67. }
68.
69. # RunFail <args>
70. # Report the command, run it, and expect an error
71. RunFail() {
72.     echo $* 1>&2
73.     eval $* && {
74.     SignalError "failed: $* did not report an error"
75.     return 1
76.     }
77.     return 0
78. }
79.
80. Check() {
81.     error=0
82.     basename=`echo $1 | sed 's/.*\\\/\\\/
83.                 s/.mc//'\`
84.     reffile=`echo $1 | sed 's/.mc$//'\`
85.     basedir="`echo $1 | sed 's/\/[\^\/]*$//'\`/."
86.
87.     echo -n "$basename..."
88.
89.     echo 1>&2
90.     echo "##### Testing $basename" 1>&2
91.
92.     generatedfiles=""
93.
94.     generatedfiles="$generatedfiles ${basename}.ll ${basename}.s ${basename}.exe
95.     ${basename}.out" &&
96.     Run "$AP_PLUSPLUS" "$1" ">" "${basename}.ll" &&
97.     Run "$LLC" "-relocation-model=pic" "${basename}.ll" ">" "${basename}.s" &&
98.     Run "$CC" "-o" "${basename}.exe" "${basename}.s" "printbig.o" &&
99.     Run "./${basename}.exe" > "${basename}.out" &&
100.    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 $1 | sed 's/\V[^\V]*$//'\`/."
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 kdps c; do
148.     case $c in
149.     k) # Keep intermediate files
150.         keep=1

```

```
151.     ;;
152.     h) # Help
153.         Usage
154.     ;;
155.     esac
156. done
157.
158. shift `expr $OPTIND - 1`
159.
160. LLIFail() {
161.     echo "Could not find the LLVM interpreter \"$LLI\"."
162.     echo "Check your LLVM installation and/or modify the LLI variable in testall.sh"
163.     exit 1
164. }
165.
166. which "$LLI" >> $globallog || LLIFail
167.
168. if [ ! -f printbig.o ]
169. then
170.     echo "Could not find printbig.o"
171.     echo "Try \"make printbig.o\""
172.     exit 1
173. fi
174.
175. if [ $# -ge 1 ]
176. then
177.     files=$@
178. else
179.     files="tests/test-*.mc tests/fail-*.mc"
180. fi
181.
182. for file in $files
183. do
184.     case $file in
185.         *test-*)
186.             Check $file 2>> $globallog
187.             ;;
188.         *fail-*)
189.             CheckFail $file 2>> $globallog
190.             ;;
191.         *)
192.             echo "unknown file type $file"
193.             globalerror=1
194.             ;;
195.     esac
196. done
197.
198. 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

```
1. (* Scanner for AP++ compiler *)
2.
3. { open Parser }
4.
5. let digit = ['0' - '9']
6. let digits = digit+
7. let esc_regex = '\\ ['\\' '\'' '\"' '\n' '\r' '\t']
8. let ascii_regex = ([ ' -'!' '#'-'[' ']'-'~' ])
9.
10. rule token = parse
11. [ ' ' '\t' '\r' '\n' ] { token lexbuf } (* Whitespace *)
12. | "/*" { comment lexbuf } (* Multi-line Comment *)
13. | "//" { line_comment lexbuf } (* Single-line Comment *)
14. | '(' { LPAREN }
15. | ')' { RPAREN }
16. | '{' { LBRACE }
17. | '}' { RBRACE }
18. | '[' { LBRACK }
19. | ']' { RBRACK }
20. | ':' { COLON }
21. | ';' { SEMICOLON }
22. | ',' { COMMA }
23. | "++" { PLUSPLUS }
24. | "--" { MINUSMINUS }
25. | '+' { PLUS }
26. | '-' { MINUS }
27. | '*' { TIMES }
28. | '/' { DIVIDE }
29. | '%' { MOD }
30. | '=' { ASSIGN }
31. | "==" { EQ }
32. | "!=" { NEQ }
33. | "&&" { AND }
34. | "||" { OR }
35. | '!' { NOT }
36. | "<=" { LEQ }
37. | ">=" { GEQ }
38. | "if" { IF }
39. | "else" { ELSE }
40. | "while" { WHILE }
41. | "for" { FOR }
42. | "return" { RETURN }
43. | "int" { INT }
44. | "bool" { BOOL }
45. | "float" { FLOAT }
46. | "void" { VOID }
47. | "<" { LT }
48. | ">" { GT }
49. | "#" { HASH }
50. | "list_push" { LIST_PUSH }
```

```
51. | "list_get"    { LIST_GET }
52. | "list_set"    { LIST_SET }
53. | "list_pop"    { LIST_POP }
54. | "list_size"  { LIST_SIZE }
55. | "list_slice" { LIST_SLICE }
56. | "list_clear" { LIST_CLEAR }
57. | "list_rev"   { LIST_REVERSE }
58. | "list_insert" { LIST_INSERT }
59. | "list_remove" { LIST_REMOVE }
60. | "list_find"  { LIST_FIND }
61. | "list"       { LIST }
62. | digits as lit { ILITERAL(int_of_string lit) }
63. | "true"       { BLITERAL(true) }
64. | "false"      { BLITERAL(false) }
65. | digits '.' digit* ( ['e' 'E'] ['+' '-']? digits )? as lit { FLITERAL(float_of_string
lit) }
66. | ''' ((ascii_regex | esc_regex)* as lit)''' { SLITERAL(lit) }
67. | ['a'-'z' 'A'-'Z']['a'-'z' 'A'-'Z' '0'-'9' '_']* as id_name { ID(id_name) }
68. | eof         { EOF }
69. | _ as char { raise (Failure("illegal character " ^ Char.escaped char)) }
70.
71. and line_comment = parse
72. '\n' { token lexbuf }
73. | _ { line_comment lexbuf }
74.
75. and comment = parse
76. "*/" { token lexbuf }
77. | _ { comment lexbuf }
```

## parser.mly

```
1. /* Ocaml yacc Parser for AP++ compiler */
2.
3. %{
4. open Ast
5. %}
6.
7. %token COLON COMMA SEMICOLON LPAREN RPAREN LBRACE RBRACE LBRACK RBRACK
8. %token PLUS MINUS TIMES DIVIDE ASSIGN MOD
9. %token PLUSPLUS MINUSMINUS
10. %token NOT AND OR
11. %token EQ NEQ LT LEQ GT GEQ
12. %token RETURN IF ELSE WHILE FOR INT BOOL FLOAT STRING VOID
13. %token LIST_PUSH LIST_GET LIST_SET LIST_POP LIST_SIZE HASH LIST_SLICE LIST_CLEAR
    LIST_REVERSE LIST_INSERT LIST_REMOVE LIST_FIND
14. %token <int> ILITERAL
15. %token <bool> BLITERAL
16. %token <string> SLITERAL
17. %token <float> FLITERAL
18. %token <string> ID
19. %token LIST
20. %token EOF
21.
22. %start program
23. %type <Ast.program> program
24.
25. %nonassoc NOELSE
26. %nonassoc ELSEIF
27. %nonassoc ELSE
28. %right ASSIGN
29. %left OR
30. %left AND
31. %left EQ NEQ
32. %left LT GT LEQ GEQ
33. %left PLUS MINUS
34. %left TIMES DIVIDE MOD
35. %right NOT PLUSPLUS MINUSMINUS
36.
37. %%
38.
39. program:
40.   decls EOF { $1 }
41.
42. decls:
43.   /* nothing */ { ([], [])}
44. | decls var_decl { (($2 :: fst $1), snd $1) }
45. | decls func_decl { (fst $1, ($2 :: snd $1)) }
46.
47. /* e.g. int foo(int x, int y) {} */
48. func_decl:
```

```

49.  typ ID LPAREN func_formals_opt RPAREN LBRACE var_decl_list stmt_list RBRACE
50.  { { typ      = $1;
51.      fname    = $2;
52.      formals  = List.rev $4;
53.      locals   = List.rev $7;
54.      body     = List.rev $8 } }
55.
56. func_formals_opt:
57.  /* nothing */ { [] }
58. | func_formals_list { $1 }
59.
60. /* int x, bool y */
61. func_formals_list:
62.  typ ID { [($1, $2)] }
63. | func_formals_list COMMA typ ID { ($3, $4) :: $1 }
64.
65. typ:
66.  INT { Int }
67. | BOOL { Bool }
68. | FLOAT { Float }
69. | STRING { String }
70. | VOID { Void }
71. | LIST LT typ GT { List($3) }
72.
73. /* e.g. int x; int y; */
74. var_decl_list:
75.  /* nothing */ { [] }
76. | var_decl_list var_decl { $2 :: $1 }
77.
78. /* e.g. int x; */
79. var_decl:
80.  typ ID SEMICOLON { ($1, $2) }
81.
82. stmt_list:
83.  /* nothing */ { [] }
84. | stmt_list stmt { $2 :: $1 }
85.
86. /* executes code logic but do not evaluate to any value */
87. stmt:
88.  expr SEMICOLON { Expr $1 }
89. | RETURN expr_opt SEMICOLON { Return $2 }
90. | LBRACE stmt_list RBRACE { Block(List.rev $2) }
91. | IF LPAREN expr RPAREN stmt %prec NOELSE { If($3, $5, Block([])) }
92. | IF LPAREN expr RPAREN stmt ELSE stmt { If($3, $5, $7) }
93. | FOR LPAREN expr_opt SEMICOLON expr SEMICOLON expr_opt RPAREN stmt
94.     { For($3, $5, $7, $9) }
95. | WHILE LPAREN expr RPAREN stmt { While($3, $5) }
96. | LIST_PUSH LPAREN ID COMMA expr RPAREN SEMICOLON { ListPush($3, $5) }
97. | LIST_SET LPAREN ID COMMA expr COMMA expr RPAREN SEMICOLON { ListSet($3, $5, $7) }
98. | ID LBRACK expr RBRACK ASSIGN expr SEMICOLON { ListSet($1, $3, $6) }
99. | LIST_CLEAR LPAREN ID RPAREN SEMICOLON { ListClear($3) }
100. | LIST_REMOVE LPAREN ID COMMA expr RPAREN SEMICOLON { ListRemove($3, $5) }

```

```

101. | LIST_INSERT LPAREN ID COMMA expr COMMA expr RPAREN SEMICOLON { ListInsert($3, $5,
    $7) }
102. | LIST_REVERSE LPAREN ID RPAREN SEMICOLON { ListReverse($3) }
103.
104. expr_opt:
105. /* nothing */ { Noexpr }
106. | expr { $1 }
107.
108. /* executes code logic and evaluates to a value */
109. expr:
110. ILITERAL { ILiteral($1) }
111. | BLITERAL { BLiteral($1) }
112. | SLITERAL { SLiteral($1) }
113. | FLITERAL { FLiteral($1) }
114. | ID { Id($1) }
115. | expr PLUS expr { Binop($1, Add, $3) }
116. | expr MINUS expr { Binop($1, Sub, $3) }
117. | expr TIMES expr { Binop($1, Mult, $3) }
118. | expr DIVIDE expr { Binop($1, Div, $3) }
119. | expr MOD expr { Binop($1, Mod, $3) }
120. | PLUSPLUS ID { Unop(PlusPlusPre, Id($2)) }
121. | MINUSMINUS ID { Unop(MinusMinusPre, Id($2)) }
122. | ID PLUSPLUS { Unop(PlusPlusPost, Id($1)) }
123. | ID MINUSMINUS { Unop(MinusMinusPost, Id($1)) }
124. | expr EQ expr { Binop($1, Equal, $3) }
125. | expr NEQ expr { Binop($1, Neq, $3) }
126. | expr LT expr { Binop($1, Less, $3) }
127. | expr GT expr { Binop($1, Greater, $3) }
128. | expr LEQ expr { Binop($1, Leq, $3) }
129. | expr GEQ expr { Binop($1, Geq, $3) }
130. | MINUS expr %prec NOT { Unop(Neg, $2) }
131. | NOT expr { Unop(Not, $2) }
132. | expr AND expr { Binop($1, And, $3) }
133. | expr OR expr { Binop($1, Or, $3) }
134. | LPAREN expr RPAREN { $2 }
135. | ID ASSIGN expr { Assign($1, $3) }
136. | ID LPAREN args_opt RPAREN { Call($1, $3) }
137. | LIST_GET LPAREN ID COMMA expr RPAREN { ListGet($3, $5) }
138. | ID LBRACK expr RBRACK { ListGet($1, $3) }
139. | LIST_POP LPAREN ID RPAREN { ListPop($3) }
140. | LIST_SIZE LPAREN ID RPAREN { ListSize($3) }
141. | HASH ID { ListSize($2) }
142. | LIST_SLICE LPAREN ID COMMA expr COMMA expr RPAREN { ListSlice($3, $5, $7) }
143. | ID LBRACK expr_opt COLON expr_opt RBRACK { ListSlice($1, $3, $5) }
144. | LIST_FIND LPAREN ID COMMA expr RPAREN { ListFind($3, $5) }
145. | LBRACK args_opt RBRACK { ListLiteral($2) }
146.
147. args_opt:
148. /* nothing */ { [] }
149. | args_list { List.rev $1 }
150.
151. /* args used for function calls */

```



```
152. args_list:
153.     expr          { [$1] }
154. | args_list COMMA expr { $3 :: $1 }
```

## ast.ml

```
1. (* Abstract Syntax Tree (AST) Definitions for AP++ compiler *)
2.
3. type op = Add | Sub | Mult | Div | Mod | Equal | Neq | Less | Leq | Greater | Geq
4.         | And | Or
5.
6. type uop = PlusPlusPre | PlusPlusPost | MinusMinusPre | MinusMinusPost | Neg | Not
7.
8. type typ = Int | Bool | Float | String | Void | List of typ
9.
10. type bind = typ * string
11.
12. type expr =
13.   | ILiteral of int
14.   | BLiteral of bool
15.   | SLiteral of string
16.   | FLiteral of float
17.   | Id of string
18.   | Binop of expr * op * expr
19.   | Unop of uop * expr
20.   | Assign of string * expr
21.   | Call of string * expr list
22.   | ListGet of string * expr
23.   | ListPop of string
24.   | ListSize of string
25.   | ListSlice of string * expr * expr
26.   | ListFind of string * expr
27.   | ListLiteral of expr list
28.   | Noexpr
29.
30. type stmt =
31.   | Block of stmt list
32.   | Expr of expr
33.   | Return of expr
34.   | If of expr * stmt * stmt
35.   | For of expr * expr * expr * stmt
36.   | While of expr * stmt
37.   | ListPush of string * expr
38.   | ListSet of string * expr * expr
39.   | ListClear of string
40.   | ListRemove of string * expr
41.   | ListInsert of string * expr * expr
42.   | ListReverse of string
43.
44. type func_decl = {
45.   typ : typ;
46.   fname : string;
47.   formals : bind list;
48.   locals : bind list;
49.   body : stmt list;
```

```

50. }
51.
52. type program = bind list * func_decl list
53.
54. (* Pretty-printing functions *)
55.
56. let string_of_op = function
57.   Add -> "+"
58. | Sub -> "-"
59. | Mult -> "*"
60. | Div -> "/"
61. | Equal -> "=="
62. | Neq -> "!="
63. | Less -> "<"
64. | Leq -> "<="
65. | Greater -> ">"
66. | Geq -> ">="
67. | And -> "&&"
68. | Or -> "||"
69. | _ -> "?"
70.
71. let string_of_uop = function
72.   Neg -> "-"
73. | Not -> "!"
74. | PlusPlusPre -> "++x"
75. | PlusPlusPost -> "x++"
76. | MinusMinusPre -> "--x"
77. | MinusMinusPost -> "x--"
78.
79. let rec string_of_expr = function
80.   ILiteral(l) -> string_of_int l
81. | FLiteral(l) -> string_of_float l
82. | BLiteral(true) -> "true"
83. | BLiteral(false) -> "false"
84. | SLiteral(l) -> l
85. | ListGet(id, e) -> "list_get " ^ id ^ ", " ^ (string_of_expr e)
86. | ListPop(id) -> "list_pop " ^ id
87. | ListSize(id) -> "list_size " ^ id
88. | ListSlice(id, e1, e2) -> "list_slice " ^ id ^ ", " ^ (string_of_expr e1) ^ ", " ^
(string_of_expr e2)
89. | ListFind(id, e) -> "list_find " ^ id ^ ", " ^ (string_of_expr e)
90. | ListLiteral(_) -> "list_literal"
91. | Id(s) -> s
92. | Binop(e1, o, e2) ->
93.   string_of_expr e1 ^ " " ^ string_of_op o ^ " " ^ string_of_expr e2
94. | Unop(o, e) -> string_of_uop o ^ string_of_expr e
95. | Assign(v, e) -> v ^ " = " ^ string_of_expr e
96. | Call(f, el) ->
97.   f ^ "(" ^ String.concat ", " (List.map string_of_expr el) ^ ")"
98. | Noexpr -> ""
99.
100. 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

```
1. (* Semantic Abstract Syntax Tree (SAST) Definitions for AP++ compiler *)
2.
3. open Ast
4.
5. type sexpr = typ * sx
6. and sx =
7.   SLiteral of int
8. | SBLiteral of bool
9. | SSLiteral of string
10. | SFLiteral of float
11. | SId of string
12. | SBinop of sexpr * op * sexpr
13. | SUnop of uop * sexpr
14. | SAssign of string * sexpr
15. | SCall of string * sexpr list
16. | SListGet of typ * string * sexpr
17. | SListPop of typ * string
18. | SListSize of typ * string
19. | SListSlice of typ * string * sexpr * sexpr
20. | SListFind of typ * string * sexpr
21. | SListLiteral of typ * sexpr list
22. | SNoexpr
23.
24. type sstmt =
25.   SBlock of sstmt list
26. | SExpr of sexpr
27. | SReturn of sexpr
28. | SIf of sexpr * sstmt * sstmt
29. | SFor of sexpr * sexpr * sexpr * sstmt
30. | SWhile of sexpr * sstmt
31. | SListPush of string * sexpr
32. | SListSet of typ * string * sexpr * sexpr
33. | SListClear of typ * string
34. | SListRemove of string * sexpr
35. | SListInsert of string * sexpr * sexpr
36. | SListReverse of typ * string
37.
38. type sfunc_decl = {
39.   styp : typ;
40.   sfname : string;
41.   sformals : bind list;
42.   slocals : bind list;
43.   sbody : sstmt list;
44. }
45.
46. type sprogram = bind list * sfunc_decl list
47.
48.
49. (* Pretty-printing functions *)
```

```

50.
51. let rec string_of_sexpr (t, e) =
52.   "(" ^ string_of_typ t ^ " : " ^ (match e with
53.     SLiteral(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) ^

```

```
99. String.concat "" (List.map string_of_sstmt fdecl.sbody) ^
100.  "}\n"
101.
102. let string_of_sprogram (vars, funcs) =
103.   String.concat "" (List.map string_of_vdecl vars) ^ "\n" ^
104.   String.concat "\n" (List.map string_of_sfdecl funcs)
```

## semant.ml

```
1. (* Semantic checking for the AP++ compiler *)
2.
3. open Ast
4. open Sast
5.
6. module StringMap = Map.Make(String)
7.
8. (* Semantic checking of the AST. Returns an SAST if successful,
9.    throws an exception if something is wrong.
10.
11.    Check each global variable, then check each function *)
12.
13. let check (globals, functions) =
14.
15.    (* Verify a List of bindings has no void types or duplicate names *)
16.    let check_binds (kind : string) (binds : bind list) =
17.        List.iter (function
18.            (Void, b) -> raise (Failure ("illegal void " ^ kind ^ " " ^ b))
19.            | _ -> ()) binds;
20.        let rec dups = function
21.            [] -> ()
22.            | ((_,n1) :: (_,n2) :: _) when n1 = n2 ->
23.                raise (Failure ("duplicate " ^ kind ^ " " ^ n1))
24.            | _ :: t -> dups t
25.        in dups (List.sort (fun (_,a) (_,b) -> compare a b) binds)
26.    in
27.
28.    (**** Check global variables ****)
29.
30.    check_binds "global" globals;
31.
32.    (**** Check functions ****)
33.
34.    (* Collect function declarations for built-in functions: no bodies *)
35.    let built_in_decls =
36.        let add_bind map (name, ty) = StringMap.add name {
37.            typ = Void;
38.            fname = name;
39.            formals = [(ty, "x")];
40.            locals = []; body = [] } map
41.        in List.fold_left add_bind StringMap.empty [ ("printi", Int);
42.            ("printf", Float);
43.            ("prints", String);
44.            ("printb", Bool) ]
45.    in
46.
47.    (* Add function name to symbol table *)
48.    let add_func map fd =
```



```

49.   let built_in_err = "function " ^ fd.fname ^ " may not be defined"
50.   and dup_err = "duplicate function " ^ fd.fname
51.   and make_err er = raise (Failure er)
52.   and n = fd.fname (* Name of the function *)
53.   in match fd with (* No duplicate functions or redefinitions of built-ins *)
54.       _ when StringMap.mem n built_in_decls -> make_err built_in_err
55.       | _ when StringMap.mem n map -> make_err dup_err
56.       | _ -> StringMap.add n fd map
57.   in
58.
59.   (* Collect all function names into one symbol table *)
60.   let function_decls = List.fold_left add_func built_in_decls functions
61.   in
62.
63.   (* Return a function from our symbol table *)
64.   let find_func s =
65.       try StringMap.find s function_decls
66.       with Not_found -> raise (Failure ("unrecognized function " ^ s))
67.   in
68.
69.   let _ = find_func "main" in (* Ensure "main" is defined *)
70.
71.   let check_function func =
72.       (* Make sure no formals or locals are void or duplicates *)
73.       check_binds "formal" func.formals;
74.       check_binds "local" func.locals;
75.
76.       (* Raise an exception if the given rvalue type cannot be assigned to
77.          the given lvalue type *)
78.       let check_assign lvaluet rvaluet err =
79.           if lvaluet = rvaluet then lvaluet else raise (Failure err)
80.       in
81.
82.       (* Build local symbol table of variables for this function *)
83.       let symbols = List.fold_left (fun m (ty, name) -> StringMap.add name ty m)
84.           StringMap.empty (globals @ func.formals @ func.locals )
85.       in
86.
87.       (* Return a variable from our local symbol table *)
88.       let type_of_identifier s =
89.           try StringMap.find s symbols
90.           with Not_found -> raise (Failure ("undeclared identifier " ^ s))
91.       in
92.       (* Check id is a list and return List type *)
93.       let check_list_type id =
94.           match (type_of_identifier id) with
95.           | List t -> t
96.           | t -> raise (Failure ("check list type error, typ: " ^ string_of_type t))
97.       in
98.       (* Return a semantically-checked expression, i.e., with a type *)
99.       let rec expr = function
100.          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_identifiler s, SId s)
106. | Assign(var, e) as ex ->
107.     let lt = type_of_identifiler 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_identifer 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 s1 ->
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 s1 :: ss -> check_stmt_list (s1 @ ss) (* Flatten blocks *)
240.     | s :: ss -> check_stmt s :: check_stmt_list ss
241.     | [] -> []
242.   in SBlock(check_stmt_list s1)
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(s1) -> s1
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);
25.     (L.pointer_type t) |]
26.   and ptr_list_t t = L.pointer_type (L.struct_type context [| L.pointer_type (L.i32_type
27.     context); (L.pointer_type t) |])
28.   in
29.   (* Return the LLVM type for a AP++ type *)
30.   let rec ltype_of_typ = function
31.     A.Int -> i32_t
32.   | A.Bool -> i1_t
33.   | A.Float -> float_t
34.   | A.String -> str_t
35.   | A.Void -> void_t
36.   | A.List t -> list_t (ltype_of_typ t)
37.   in
38.   let type_str t = match t with
39.     A.Int -> "int"
40.   | A.Bool -> "bool"
41.   | A.Float -> "float"
42.   | A.String -> "str"
43.   | _ -> raise (Failure "Invalid string map key type")
44.   in
45.   (* Create a map of global variables after creating each *)
46.   let global_vars : L.llvalue StringMap.t =
47.     let global_var m (t, n) =
48.       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))
build_br_merge;
89.
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))
build_br_merge;
93.
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_array_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_array_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.lvalue 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.array" 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

```

```

232.     let _ = L.build_store (L.param def 2) idx_ptr1 build in
233.     let idx1 = L.build_load idx_ptr1 "idx_load" build in
234.
235.     let idx_ptr2 = L.build_alloca i32_t "idx_alloc" build in
236.     let _ = L.build_store (L.param def 3) idx_ptr2 build in
237.     let idx2 = L.build_load idx_ptr2 "idx_load" build in
238.
239.     (* Loop counter init: 0 *)
240.     let loop_cnt_ptr = L.build_alloca i32_t "loop_cnt" build in
241.     let _ = L.build_store (L.const_int i32_t 0) loop_cnt_ptr build in
242.     (* Loop upper bound: j-i *)
243.     let loop_upper_bound = L.build_sub idx2 idx1 "loop_upper_bound" build in
244.     (* Loop condition: cnt <= j-i *)
245.     let loop_cond_builder =
246.         L.build_icmp L.Icmp.Sle (L.build_load loop_cnt_ptr "loop_cnt" _builder)
loop_upper_bound "loop_cond" _builder
247.     in
248.     (* assignment: b[cnt] = a[cnt + i] *)
249.     let loop_body_builder =
250.         let to_index = L.build_load loop_cnt_ptr "to_idx" _builder in
251.         let from_index = L.build_add to_index idx1 "from_idx" _builder in
252.         let get_val = L.build_call (StringMap.find (type_str typ) list_get) [|
list_ptr; from_index |] "list_get" _builder in
253.         let _ = L.build_call (StringMap.find (type_str typ) list_push) [| list_ptr2;
get_val |] "" _builder in
254.         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
255.         let _ = L.build_store index_incr loop_cnt_ptr _builder in
256.         _builder
257.     in
258.     let while_builder = build_while build loop_cond loop_body def in
259.     ignore(L.build_ret_void while_builder);
260.     StringMap.add def_name def m
261.     in
262.     List.fold_left list_slice_ty StringMap.empty [ A.Bool; A.Int; A.Float; A.String ]
in
263.
264.     (* i32_t list_find(list, val) *)
265.     let list_find : L.lvalue StringMap.t =
266.         let list_find_ty m typ =
267.             let ltype = (ltype_of_typ typ) in
268.             let def_name = (type_str typ) in
269.             let def = L.define_function ("list_find" ^ def_name) (L.function_type i32_t [|
L.pointer_type (list_t ltype); ltype |]) the_module in
270.             let build = L.builder_at_end context (L.entry_block def) in
271.             let list_ptr = L.build_alloca (L.pointer_type (list_t ltype)) "list_ptr_alloc"
build in
272.             ignore(L.build_store (L.param def 0) list_ptr build);
273.             let find_value_ptr = L.build_alloca ltype "find_val_alloc" build in
274.             ignore(L.build_store (L.param def 1) find_value_ptr build);
275.             let find_value = L.build_load find_value_ptr "find_val" build in
276.             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_type 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_type 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
(L.build_load leftPtr "left_idx" _builder)
424.     (L.build_load rightPtr "right_idx" _builder) "while_cond" _builder
425.     in
426.     let while_body_builder =
427.         let left_idx = (L.build_load leftPtr "left_idx" _builder) in
428.         let right_idx = (L.build_load rightPtr "right_idx" _builder) in
429.         let get_left_val = L.build_call (StringMap.find (type_str typ) list_get) [|
list_load; left_idx |] "list_get" _builder in
430.         let get_right_val = L.build_call (StringMap.find (type_str typ) list_get) [|
list_load; right_idx |] "list_get" _builder in
431.         let _ = L.build_call (StringMap.find (type_str typ) list_set) [| list_load;
left_idx; get_right_val |] "" _builder in
432.         let _ = L.build_call (StringMap.find (type_str typ) list_set) [| list_load;
right_idx; get_left_val |] "" _builder in
433.         let _ = L.build_store (L.build_add left_idx (L.const_int i32_t 1) "tmp"
_builder) leftPtr _builder in
434.         let _ = L.build_store (L.build_sub right_idx (L.const_int i32_t 1) "tmp"
_builder) rightPtr _builder in
435.         _builder
436.     in
437.     let while_builder = build_while build while_cond while_body def in
438.     ignore(L.build_ret_void while_builder);
439.     StringMap.add def_name def m in
440. List.fold_left list_reverse_ty StringMap.empty [ A.Bool; A.Int; A.Float ] in
441.
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.    | SLiteral 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_fmuls
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

```



```

550.     | A.Greater -> L.build_icmp L.Icmp.Sgt
551.     | A.Geq     -> L.build_icmp L.Icmp.Sge
552.     ) e1' e2' "tmp" builder
553. | SUnop(op, ((t, e'') as e)) ->
554.     let e' = expr builder e in
555. (match op with
556.   A.Neg when t = A.Float -> L.build_fneg e' "tmp" builder
557. | A.Neg                -> L.build_neg e' "tmp" builder
558. | A.Not                -> L.build_not e' "tmp" builder
559. | A.PlusPlusPre ->
560.   let new_val = (L.build_add e' (L.const_int i32_t 1)) "tmp" builder in
561.   let id = match (e'') with
562.     SId s -> s
563.   | _ -> raise (Failure ("++ operand must be an ID")) in
564.   let var_ptr = (lookup id) in
565.   let _ = L.build_store new_val var_ptr builder in new_val
566. | A.MinusMinusPre ->
567.   let new_val = (L.build_sub e' (L.const_int i32_t 1)) "tmp" builder in
568.   let id = match (e'') with
569.     SId s -> s
570.   | _ -> raise (Failure ("-- operand must be an ID")) in
571.   let var_ptr = (lookup id) in
572.   let _ = L.build_store new_val var_ptr builder in new_val
573. | A.PlusPlusPost ->
574.   let new_val = (L.build_add e' (L.const_int i32_t 1)) "tmp" builder in
575.   let id = match (e'') with
576.     SId s -> s
577.   | _ -> raise (Failure ("++ operand must be an ID")) in
578.   let var_ptr = (lookup id) in
579.   let _ = L.build_store new_val var_ptr builder in e'
580. | A.MinusMinusPost ->
581.   let new_val = (L.build_sub e' (L.const_int i32_t 1)) "tmp" builder in
582.   let id = match (e'') with
583.     SId s -> s
584.   | _ -> raise (Failure ("-- operand must be an ID")) in
585.   let var_ptr = (lookup id) in
586.   let _ = L.build_store new_val var_ptr builder in e')
587.
588. | SListGet (list_type, id, e) ->
589.   L.build_call (StringMap.find (type_str list_type) list_get) [| (lookup id); (expr
builder e) |] "list_get" builder
590. | SListSize (list_type, id) ->
591.   L.build_call ((StringMap.find (type_str list_type)) list_size) [| (lookup id) |]
"list_size" builder
592. | SListPop (list_type, id) ->
593.   L.build_call ((StringMap.find (type_str list_type)) list_pop) [| (lookup id) |]
"list_pop" builder
594. | SListSlice (list_type, id, e1, e2) ->
595.   let ltype = (ltype_of_typ list_type) in
596.   let new_list_ptr = L.build_allca (list_t ltype) "new_list_ptr" builder in
597.   let _ = init_list builder new_list_ptr list_type in
598.   let e' = match (fst e1, fst e2) with
599.     (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 s1 -> List.fold_left stmt builder s1
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 [SEExpr e1 ; SWhile (e2, SBlock [body ; SEExpr 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
```