BlueShell Final Report

Kenneth Lin (kenneth.lin@tufts.edu)
Alan Luc (alan.luc@tufts.edu)
Tina Ma (tina.ma@tufts.edu)
Mary-Joy Sidhom (mary-joy.sidhom@tufts.edu)

May 5, 2023

1 Introduction

BlueShell is a programming language designed to facilitate interaction with the shell. The syntax of classic shell scripting is verbose and very different from other programming languages, resulting in a steep learning curve. BlueShell solves this by using C-like syntax combined with the functionality of shell scripts.

In BlueShell, executables are native types that can be run using BlueShell code. Executables allow the user to encapsulate an executable path and its arguments into a single variable. In addition, Blueshell provides built-in operators on executables which can be used to combine multiple executables at runtime. BlueShell also offers its users monomorphic lists and function pointers, which can be used to operate on multiple executables.

Contents

2 Tutorial

2.1 Environment Requirements

BlueShell requires the user to have OCaml, LLVM, the gcc compiler, and a Unix operating system.

2.2 Creating a BlueShell Program

BlueShell programs are denoted with the `.bs` file extension.

Blueshell's syntax is loosely based off of a combination of the C programming language and bash shell language. There is no need to create a `main` function to start execution. The following lines of code:

```
/* This is valid BlueShell! */

/* Declare and intialize a string variable */
string str = "Hello World!";

/* Declare and intialize an integer variable */
int x = 2;
```

are a valid BlueShell Program.

2.3 Using our Compiler

To compile a BlueShell program, run the included bash script as follows:

```
./compile.sh [filename]
```

This will generate an executable named `filename.exe`. To execute the BlueShell program, run the executable with the following syntax:

```
./filename.exe
```

2.4 Lists

BlueShell features built-in monomorphic lists. In the code snippet below, we create a `string` list of two elements. We then add another element to the front of list using the cons, or `::`, operator.

```
/* creates a string list with two elements */
list of string l = ["world", "!"];

/* adds an element to the front of the list */
"hello" :: l;
```

2.5 Executables: Using the Shell

BlueShell provides executables as a built-in type. Executables comprise of a executable's relative path in the file system and an optional list of arguments.

Below we create two executable variables named `e1` and `e2`. Assuming the system has the path

to Unix commands set in the default PATH, `e1` is an executable representing the `ls` command. To use executables with arguments, add the `withargs` keyword and a list of arguments following the path. `e2` is the Unix `echo` command with a single string argument.

```
/* creating two executables */

/**/
exec e1 = <"ls">;
exec e2 = <"echo" withargs ["Hello World!"]>;
```

However, to produce visible output, the executables need to be run. To run executables in the local shell environment, use the `./` operator.

```
/* creating two executables */
exec e1 = <"ls">;
exec e2 = <"echo" withargs ["Hello World!"]>;

/* running the two executables */
./e1;
./e2;
```

Executables can also be run without assigning them to a variable by adding the `./` operator before the declaration.

```
/* Create an exeuctable to run echo with argument "Hello
   world", then run it */

./<"echo" withargs ["Hello World"]>;
```

3 Language Reference Manual

3.1 Reading our LRM: Notation

We denote prose with the standard LaTeX font:

This is prose.

We denote code with typewriter font:

`This is code.`

We denote grammar rules with italics:

This is a grammar rule.

Additionally, if a part of a grammar rule is optional (i.e. can be empty), we label it with *opt* in subscript, as such:

This is an optional grammar rule$_{opt}$.

3.2 Lexical Convention

Comments

Single line comments begin with the `//` symbol with no intervening blank space and terminate with the newline character (`\n`) or EOF (end-of-file).

```
// this is a comment
```

Multi-line comments begin with the `/*` symbol and terminate with the first occurrence of the `*/` symbol that appears after.

```
/* this
   is
   also
   a
   comment */
```

During compilation, comments are ignored.

Blank space

Spaces, newlines, and tabs are ignored during compilation except for as tools for separating tokens. Users can use newlines and tabs to style their code.

Identifiers (names)

Identifiers (variable and function names) in BlueShell can consist of letters (both upper and lower case), digits [0 - 9], and underscores. The first character in an identifier must be a letter. Anything in the list of keywords below must not be used as an identifier.

Keywords

The following keywords are reserved.

`and, or, not, if, else, for, while, return, int, bool, float, void, exec, char, string, list, true, false, function, len, withargs, of`

3.3 Types

⟨*typ*⟩ ::= `int`
| `bool`
| `float`
| `void`
| `exec`
| `char`
| `string`
| `list of` *typ*
| `function (` *(typ* `->` *)* typ*`)`

Primitive Types

- Integers, or `int`, are 32-bit signed integers. Leading 0's in front of an integer value are ignored and do not change the value of the integer.
- Floating-point values, or `float`, are numbers represented by a floating-point representation. BlueShell uses the IEEE 754 standard for double-precision floating-point numbers which comprises of a sign-bit, an 11-bit exponent, and a 53-bit mantissa. The exponent range is from -1022 to 1023. Leading zeroes are NOT ignored.
- Booleans, or `bool`, either hold the value `true` or `false`.
- `void` is a type that represents no value. Its sole purpose is to represent the return type of a function that does not return a value (see Section 3.6 on Functions). Void types cannot be assigned (see Section 3.4 on assignment).
- Characters, or `char`, are a single symbol enclosed within single quotes.
- Strings, or type `string`, are a sequence of zero or more characters enclosed within double quotes. On 64-bit platforms, strings of up to $2^{57} - 9$ characters are supported.
- The following escape sequences can be used to denote special characters. Unless otherwise specified, all of these escape sequences apply to both characters and strings.
  1. `'\n'`
  2. `'\t'`
  3. `'\\'`
  4. `'\r'`
  5. `'\''` (Characters only)
  6. `'\"'` (Strings only)

Lists

Lists are a monomorphic built-in type that can store any number of elements. When a list is declared, the type of its elements needs to be specified.

In BlueShell, lists act similarly to vectors that are in other languages. BlueShell lists have operations to grow and determine the number of elements in the list. Additionally, they have the indexing operator to access and change elements in the list. BlueShell lists are 0-indexed, and accessing an index less than 0 or greater than or equal to the list's length will result in a runtime error.

List Creation
BlueShell lists can be created using the following syntax:

`list of` *typ identifier* `= [` {*(expr* `,`*)* expr*}*?* `];`

A type *typ* is required after the keyword `of`; this specifies the type of each element of the list. Each expression between the square brackets must be of type *typ*. If there are nothing between the square brackets on the right side of the assign, then the list can be assigned to any list type.

Executables

Executables, or `exec`, are a built-in type that represent an executable binary and a list of command line arguments to be run with that binary. Executables can either be simple or complex.

Simple Executables are comprised of:

1. A path of type `string` representing a path to a executable file relative to the directory of the program (See section 3.3 for more).
2. A possibly empty list of command line arguments to pass into the executable.

The arguments of a simple executable cannot be changed.

Complex Executables are executables chained together using executable operators. **The path and arguments of a complex executable cannot be changed.** More information about executable operators can be found in 3.4.

Executable creation
There are two ways to declare a simple executable. The first way, which is more general, is as follows:

`exec` *identifier* `= <` *expr* `withargs` *expr* `>;`

The first expression is a string or string variable representing the path. The second expression is a potentially empty list of either strings, chars, ints, floats, or bools representing the arguments. Using a list of executables, lists, or function pointers will result in an error.

At runtime, elements of the executable's arguments list are cast to strings. Boolean values are cast to the strings "true" and "false" respectively.

The second way, which is shorthand for defining a executable with no arguments, is as follows:

`exec` *identifier* `= <` *expr* `>;`

where the expression is a string or string variable representing the path.

As an example, the following code creates two executables using both of the methods described above:

```
/* Create an executable with a path to the cat binary
   in"/bin/cat"with two command line arguments: string
   "file1.txt" and string "file2.txt" */

exec cat = < "/bin/cat" withargs ["file.txt", "file2.txt"] >; }

/* create an executable with a path to the ls binary at
   "/bin/ls" with no command line arguments */

exec ls = < "/bin/ls" >;
```

Complex executables can only be created through using executable operations (see Section 3.4).

Functions

The syntax for a general function declaration is as follows:

typ identifier(*(expr* ,)* *expr?*) { *statement-list* }

where *typ* is the return type of the function, the expressions inside the parentheses are variable declarations that define the parameters of the function, and the lists of statements making up the function body.

Functions Pointers

The syntax to create a function pointer is as follows:

`function` ((*typ* `->`) * *typ*) *identifier* `=` *identifier*

where the last type is the return type (must be non void), all prior types are the types of the parameters, and the identifier on the right hand side of the equals sign is the name of either an existing function or an existing function variable in the current scope.

For more information on functions refer to section 3.6

3.4 Expressions

⟨*expr*⟩ ::= *lit*
| *identifier*
| *vdecl*
| *expr binop expr*
| *preop expr*
| *expr*`$`
| *expr*`[`*expr*`]`
| `[]`
| `[` *cont-list*

⟨*cont-list*⟩ ::= *expr* `,` *cont-list*
  | *expr* `]`

⟨*lit*⟩ ::= *literal*
  | *float-literal*
  | *boolean-literal*
  | *char-literal*
  | *string-literal*
  | *exec-literal*
  | *list-literal*

⟨*vdecl*⟩ ::= *typ identifier*

⟨*binop*⟩ ::= `+`
  | `-`
  | `*`
  | `|`
  | `/`
  | `==`
  | `=`
  | `!=`
  | `>`
  | `<`
  | `>=`
  | `<=`
  | `&&`
  | `||`
  | `and`
  | `or`

⟨*preop*⟩ ::= `-`
  | `!`
  | `./`
  | `len`

⟨*exec-literal*⟩ ::= `<` *string-literal* `>`
  | `<` *string-literal* `withargs` *list-literal* `>`

⟨*list-literal*⟩ ::= `[` *[literal,]** `]`

⟨*bool-literal*⟩ ::= `true`
  | `false`

⟨*char-literal*⟩ ::= `'[''A''-''Z' 'a''-''z'']'`

⟨*string-literal*⟩ ::= `"[''A''-''Z' 'a''-''z'']*"`

⟨*float-literal*⟩ ::= *digits* `.` *digit**

⟨*int-literal*⟩ ::= *digits*

⟨*digits*⟩ ::= *digit+*

⟨*digit*⟩ ::= `[0-9]`

Precedence

BlueShell has the following precedence rules. Rules towards the top have higher precedence than those below them. Rules on the same line have equal precedence. All of the operators below are left associative besides `len`, `!`, `::`, and `=`, which are right associative.

1. `$` operator
2. List Indexing(`[]`)
3. `::` operator
4. `len` operator
5. `!` operator
6. `|` operator
7. `*, /` operators
8. `+, -` operators
9. `<, >, <=, >=` operators
10. `==, !=` operators
11. `&&` operator
12. `||` operator
13. `=` operator
14. `./` operator

Primary Expressions

Note: Undefined behavior will occur if any of these expressions result in a value that exceeds the bounds of the type specified in Section 3.3.

Literals

1. An `int` literal consists of a sequence of digits and represents an integer value.
2. A `bool` literal can be either `true` or `false`.
3. A `float` literal consists of a sequence of digits representing an integer value, a period, and a sequence of digits representing a fractional value. There must be at least one digit to the left of the period. There can be zero or more digits to the right.
4. A `char` literal consists of a single character enclosed between single quotes (see Section 3.7 for exceptions).
5. A `string` literal consists of a sequence of characters enclosed between double quotes (see Section 3.7 for exceptions).
6. An `exec` literal consists of the left angle bracket `<` followed by a string literal representing the path, the keyword `withargs` followed by a list literal consisting of string literals representing arguments (optionally), and the right angle bracket `>`.
7. A `list` literal consists of a series of (potentially none) comma separated literals between `[` and `]`. All the literals must be of the same type. Exec literals and list literals are the only literals that cannot be elements of a list.

Identifier

identifier
An identifier as an expression will evaluate to the value bound to the identifier.

Parenthesized Expression

`(`*expr*`)`
An expression placed between parentheses is given top precedence, regardless of the operations within it.

Function Call

identifier`(` *[(expr* `,`*)* expr]?* `)`
A function call consists of two components:

1. An identifier, representing the name of a function.
2. A (possible empty) series of comma separated expressions contained between parentheses representing the arguments to the function. (see Section 3.6 on Functions)

Variable Declaration

typ identifier
This creates a variable named *identifier* of type *typ*. The scope of this variable will follow the rules in Section 3.7.

Assignment

identifier `=` *expr*
This evaluates the expression and assigns its value to the specified identifier. The expression **cannot** be a variable declaration.

Expression Assignment

expr `=` *expr*
There are only three types of expressions that can be on the left of the `=` symbol in expression assignment:

1. **List Indexing Operator** follows the form `a[x] =` *expr*. This reassigns the element in list `a` at index `x` to be of value `expr`. The type of *expr* must be the same type the elements in list `a`.
2. **Executable Path Operator** follows the form `e$ =` *expr*. The path operator allows access to the path of an executable. This reassigns the path of executable `e` to be the value of *expr*. *expr* must be of type `string`. See Section 5 for more information about the path operator.
3. **Variable Declaration** follows the form *typ identifier* = *expr*. It declares and initializes the variable *identifier* in one line.

The *expr* on the right of the symbol can be any expression **except for** variable declaration.

Operators

Arithmetic Operators

Operands of arithmetic operations can be either `int` or `float` types. Both operands must be of the same type. The result has the same type as the operands.

Symbol	Operation	Example	Result
`*`	Multiplication	`2.5 * 3`	7.5
/	Division	`15 / 4`	3
+	Addition	`1.1 + 2.1`	3.3
`-`	Subtraction	`4 - 5`	-1

An error is raised if division by zero occurs. If the result of integer division is not an integer, the result is floored.

Negation

`-`, or negation, is a unary operator that can only be used on `int` or `float` types.

Symbol	Operation	Example	Result
`-`	Numerical Negation	`-1`	-1

Equality Operators

Equality operators are binary operators that can only be used on `float` and `int` types. The operands must be of the same type. The resulting type is a boolean.

Symbol	Operation	Example	Result
`==`	Equal to	`5 == 2`	false
`!=`	Not Equal to	`false != true`	true

Comparison Operators

Comparison operators are binary operators. They can be performed on floats and integers. Both operands must be of the same type. The resulting type of the operation is a boolean.

Symbol	Operation	Example	Result
`>`	Greater than	`5 > 2`	true
`<`	Less than	`5 < 2`	false
`>=`	Greater Than or Equal to	`7.3 >= 6.5`	true
`<=`	Less than or Equal to	`3.5 <= 3.5`	true

Logical Operators

The logical binary operators can only be performed on boolean values. The resulting type is a boolean.

Symbol	Operation	Example	Result
`||`	Logical Or	`true || false`	true
`or`	Logical Or	`true or false`	true
`&&`	Logical And	`true && false`	false
`and`	Logical And	`true and false`	false
`!`	Boolean Negation	`!false`	true
`not`	Boolean Negation	`not false`	true

List Operators

Cons

Cons, or `::`, appends an element to the beginning of the list. The first expression can evaluate to any type, and the second expression is expected to be a list. If the list is empty, the resulting list will be of the type of the first expression. If the list is not empty, the type of the first expression must match the type of the list.

```
/* Use cons on lists */

list of int l = [];

/* l is now [1] */
l = 1 :: l;

list of int l2 = [1, 2, 3];

/* l2 is now [0, 1, 2, 3] */
l2 = 0 :: l2;
```

Length

Length, or `len`, takes in a list and returns an `int` representing number of elements in the list. If the list is empty, length returns zero.

```
/* Use len on lists*/

list of int l = [1, 2, 3];
/* length is 3 */
int length = len l;


list of int l2 = [];
/* length2 is 0*/
int length2 = len l2;
```

Note that len cannot be used in one of the expressions of a for loop.

List Indexing

Individual elements of a list can be read and manipulated with the indexing, or `[]`, operator. The syntax of indexing is as follows:

`a[x]`

where `a` is a list and `x` is an integer. If the integer is greater than the length of a list, a runtime error occurs. Lists in BlueShell are zero-indexed.

Note that list indexing cannot be used in one of the expressions in a for loop.

Executable Operators

NOTE: The amount of space we have allocated for each executable output is 16,384 bytes. Running an executable with output greater than this size may result in undefined behavior.

Executable operators provide a way to work with executables, including executing, manipulating, and retrieving values from the executable. The binary operators can also be used to create complex executables.

1. **Executable Concatenation**

   The executable concatenation, or `+` operator, takes two executables as operands. These can be either simple or complex executables. The result is a complex executable that binds the two operand executables together. When the resulting executable is run, the standard output of the operand executables are concatenated.

```
/* Assume e1 and e2 executables exist and are defined. */

/* Assign resulting complex executable to a variable */
exec e = e1 + e2;

/* the string that is output when running is the concatenated
   standard outputs from executing e1 and e2*/
string out = ./e;
/* out will contain the concatenated outputs of e1 and e2 */
```

2. **Executable Star**

   The executable star or `*` operator, takes two executables as operands. These can be either simple or complex executables. The result is a complex executable that, when run, executes the executables passed in as operands in sequence from left to right. Only the output from the right operand will be printed and visible.

```
/* Assume e1 and e2 executables exist and are defined. */

/* Assign resulting complex executable to a variable */
exec e = e1 * e2;

/* out will contain the output of e2 but
e1 will also have been executed */
string out = ./e;
```

3. **Pipe**

   The pipe, or `|` operator, takes two executables as operands. These can be either simple or complex executables. When the resulting executable is run, the standard output of the left operand will be piped into the standard intput of the right operand. If the right operand is a complex executable, the standard output from the left operand is piped into the right operand's left-most simple executable.

```
/* Assume e1, e2, and e3 are simple executables that exist and
   are defined. */

exec e = e1 | e2;

/* out will contain the output of e2 where the output of e1
   has been taken in as input */
string out = ./e;

/* the standard output of e3 will be piped into the standard
   input of e1 since e1 is the leftmost simple executable to
   the right of the pipe */
string out1 = ./(e3 | e);
```

4. **Run**
   The run, or `./` operator can only be applied to an executable type. The behavior of run depends on the type of executable:

   (a) If the executable is simple, the executable and its arguments will be run as if run on the command line.

   (b) If the executable is complex, any simple executables that built up the complex executable will be run. The effects of the operators used to bind the simple executables will be applied.

   In both cases, the result is a `string` representing the standard output of the executable(s). See section 3.4 for more about how executable binary operators affect the output of the run operator.

```

exec e = < "/bin/cat" withargs ["file.txt"] >;
/* will run "cat file.txt" as if it were run on the command
   line and save the output into string s1 */
string s1 =./e;

exec e1 = < "/bin/ls" withargs ["-l"] >;
exec e2 = < "/bin/wc" >;

/* execute e1 and e2 in accordance to the pipe operator.
   string s contains the output of e2 in accordance to the
   pipe operator */
string s = ./(e1 | e2);
```

5. **Path**
   This path, or `$` operator can only be applied to a **simple executable**. Path can be used to access and manipulate the path to an executable. Changing of the path is done in-place and does not create a new executable.

```
/* Create an executable with the path to the "cat" binary*/

exec e = < "/bin/cat" withargs ["file.txt"] >;

/* output holds "/bin/cat" */
string output = e$;
```

3.5 Statements

⟨*stmt*⟩ ::= *expr* ;
| `return;`
| `return` *expr* ;
| `if (` *expr* `) {` *stmt* `} [ else {` *stmt* `} ]`
| `for (` $expr_{opt}$ `;` *expr* `;` $expr_{opt}$ `) {` *stmt* `}`
| `while (` *expr* `) {` *stmt* `}`
| *statement-list*

Statements are executed in sequence. Most statements are of expressions in the form:

expr;

These are often seen as assignments or function calls. A list of statements enclosed in two curly braces also reduces to a statement:

statement-list:
{ *stmt** }

The statements are executed one by one in the order in which they appear.

Conditional Statements

Conditional statements take one of two forms:

1. `if (` *expr* `) {` *stmt* `}`
2. `if (` *expr* `) {` *stmt* `} else {` *stmt* `}`

For both cases, if *expr* evaluates to `true`, the first statement list is executed. In the case of the second form, if *expr* evaluates to `false`, the second statement list is executed. In the case of the first form, if *expr* evaluates to `false`, the program continues without executing any of the *statement-lists*.

`while` Statement

`while` statements take the form of:

`while (` *expr* `) {` *stmt* `}`

stmt is repeatedly executed while the *expr* evaluates to `true`. The *expr* is tested before *stmt* is evaluated.

`for` Statement

A `for` statement follows the following format:

`for (` $expr_{opt}$ `;` *expr* `;` $expr_{opt}$ `) {` *stmt* `}`

The first expression initializes the loop and is optional. The second expression is a test which occurs before each execution of the loop. If the test evaluates to `true`, the statement is executed. The third expression is also optional and specifies an increment performed after each execution of the statement.

Note that using length or index on a list in one of the expressions in a for loop causes a compilation error.

Return Statements

A function returns to its caller with a `return` statement. This has one of the following forms:

```
return;
| return expr;
```

In the first form, nothing is returned. In the second, the value of the expression is returned to the caller.

3.6 Functions

Function Declarations
A function declaration is as follows:

type identifier `(` *[(type identifier* `,`*)* type identifer]?* `)`
statement-list

The function declaration must be followed immediately by the function body, which is represented by the *statement-list*. Functions can only be declared at the top level.

Function Bodies
A function body is comprised of a *statement-list*. Any declarations which occur within the function body are local to the function body (See Section 3.7 for more information on scope). The function body must include at least one `return` statement where the following *expr* evaluates to a type matching *typ*. In the case where *typ* is `void`, a return with no expression attached is expected. There should not be any statements after a `return` statement.

Function Calls
To invoke a function, the syntax is as follows:

identifier`(`*[(expr* `,`*)* expr]?*`)`

The function call must come after the function declaration and will evaluate to the expression returned by the function. If the function is void, then there will not be an expression returned.

The number of expressions provided to the function call must match the number of arguments specified in the function declaration. Each expression must evaluate to the same type as its respective argument.

Function Pointers

In BlueShell, we have the notion of function pointers, which are typed variables that can be linked to a function. The syntax to create a function pointer is as follows:

$$\texttt{function ((}\textit{typ}\texttt{ ->) * }\textit{typ}\texttt{) }\textit{identifier}\texttt{ = }\textit{identifier}$$

where the rightmost *typ* is the return type (must be non void), all prior types are the types of the parameters, and the identifier on the right hand side of the equals sign is either an existing function or an existing function variable in the current scope.

Here is an example of how function pointers can be used:

```
/* function pointer example */

/* simple function that adds two floats */
float addf(float a, float b)  {
    ./<"echo" withargs [a + b]>;
    return a + b;
}

/* creating a function pointer */
function (float -> float -> float) add_func;

/* setting the function pointer to point to addf */
add_func = addf;
/* calling addf using the add_func pointer */
add_func(1.2, 3.4);
```

3.7 Scope

This section details the rules in which variables are in scope:

1. All variables must be declared before they are used.
2. Global variables are defined in statements outside a block or function. These variables will always remain in scope once they are declared until the program ends.
3. A variable declared in a block remains in scope for the entirety of the block once declared.
4. If a variable in a block has the same identifier as a global, it shadows the global for the rest of the block.
5. Global variable names must all be unique. Variable names within a single block must all be unique.

4 Project Plan

4.1 Planning

Our project timeline and plan adhered closely to the course layout. After brainstorming project ideas, we eventually settled on a shell scripting language. We all agreed that bash scripting can be fairly unintuitive for programmers coming from languages like C and Java. We therefore tried to create a language that would have both a user-friendly syntax and interesting scripting functionality.

With those goals in mind, we chose a C-style syntax. We hoped this would make our language syntax familiar and allow us to borrow from MicroC's implementation. We also originally planned to make executables their own type and include many operators, include polymorphic lists, and function pointers. However, after discovering some issues with polymorphic lists, we decided to switch to monomorphic lists. We also planned for executables to have more features, but we decided to focus on the core functionality instead.

When coding, we frequently used pair-programming. This helped us catch bugs earlier and develop workarounds faster.

4.2 Project Timeline

Important Features & Deliverables	**Completion Date**
Proposal	Feb 01
Scanner & Parser	Feb 23
Hello World	Mar 29
Extended Testsuite	Apr 19
Executable Autocasting	Apr 20
Function Pointers	Apr 27
Indexing	Apr 30
Executable Operators	May 02
Final Compiler	May 05

4.3 Roles and Responsibilities

Alan and Mary-Joy spearheaded development of semantic checking and code generation. Together they mastered LLVM and did much of the coding in the second half of the project.

Tina managed all aspects of testing for the project. She developed and managed the testing scripts and framework used throughout the project.

Kenny managed deliverables for the group and led work on the final report. He also contributed to debugging elements of semantic analysis and code generation.

All group members contributed to language design and the scanner/parser.

Although we did not strictly follow these roles, team members generally gravitated towards these areas.

- Alan and Mary-Joy: Lead Developers
- Tina: Testing Lead
- Kenny: Project Manager

4.4 Development Tools

Programming Languages

Our compiler is written in OCaml with some functionality delegated to C. Our testing scripts are implemented with bash scripts.

Dependencies

- opam 2.1.4
- LLVM 14.0.0
- gcc
- Make

Tools

- VSCode (our main IDE)
    - LiveShare Extension (for collaboration)
- Overleaf (for documentation)
- GitHub (for version control)

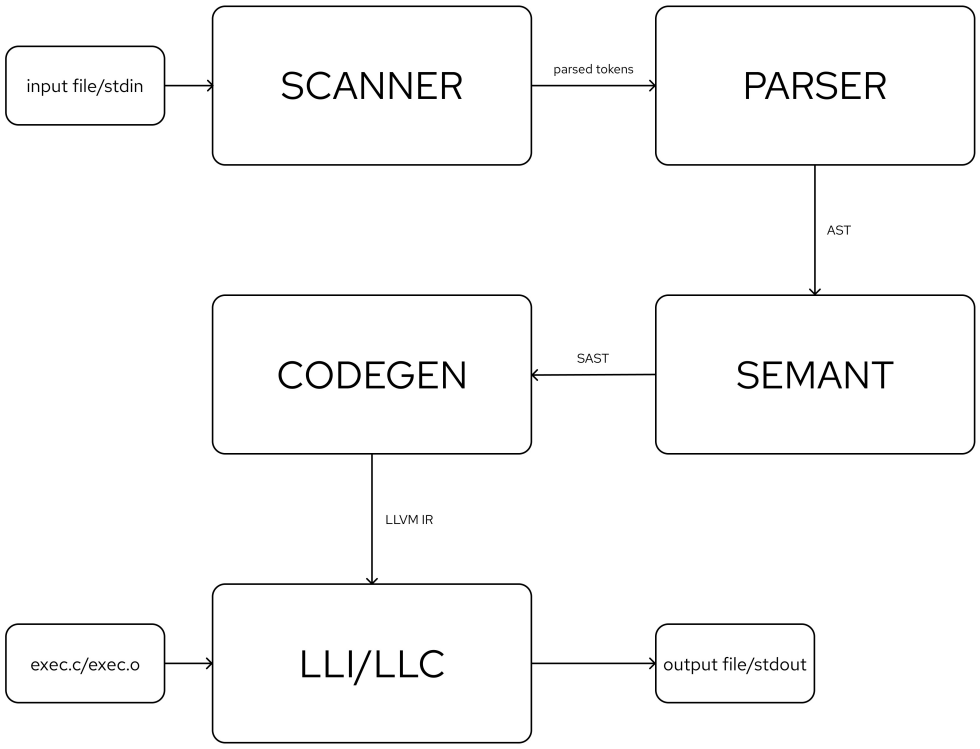
4.5 Project Log

See Git log in Appendix Section 8.4.

5 Architectural Design

5.1 Interfaces

BlueShell utilized four main modules (more below) with a toplevel to call each component. Below is a block diagram of how our modules interact.



Lexical Analysis

This module is implemented in `scanner.mll`. The role of the scanner is to tokenize input and check that all tokens are valid BlueShell tokens.

Parser

This module is implemented across `parser.mly` and `ast.ml`. The parser is responsible for generating an abstract syntax tree (AST) from the input tokens. This ensures that the BlueShell programs adhere to the BlueShell grammar.

Semantic Analysis

This module is implemented across `semant.ml` and `sast.ml`. The semantic analyzer converts the AST into a semantically-checked AST (SAST). This step peforms type-checking and ensures functions have valid return statements.

Code Generation

This module is implemented in `codegen.ml` and `exec.c`. The code generator converts the SAST into LLVM IR and generates calls to the external C code located in `exec.c`. `exec.c` performs `execvp` calls to allow for shell interaction. `exec.c` also handles the recursive nature of executable operators.

5.2 Implementation Details

Lists

Lists are implemented as boxed linked lists. For every element of the list, there is a linked list node that contains a pointer to the value and a pointer to the next element. Additionally, to help with the casting of elements when we pass them to executable functions, we created a type enum to map BlueShell types to numbers to help with executable argument casting. Because executable, list, and function types are not allowed in executable arguments, they fall under the "other" category.

```
// struct to represent a list node (linked list under the hood)
struct list {
    void **val;
    struct list* next;

    // type to cast arguments to (see Type enum below)
    int typ;
};

//how types are mapped to enums
enum Type { INT = 0, FLOAT = 1, BOOL = 2, CHAR = 3, STRING =
   4, OTHER = 5 };
```

LLVM Definitions

There are three lists operators in our language:

1. **Cons:** Adds an element to the beginning of a list of the same type. Note that you can cons an element of any type onto an empty list.
2. **Index:** Users can access any element of a list.
3. **Length:** Users can get the length of a list.

Function Pointers

When the user declare a function at the top-level scope, the address of where those instructions are stored is retrieved. In our symbol table, those addresses are mapped to the names of the functions. When function pointers are created, the new identifiers are mapped to these addresses in our symbol table.

Because all addresses have a mapping from a function name, anonymous functions are not allowed in BlueShell. Users cannot call functions from lists or functions returned from other functions. Additionally, users cannot assign function bodies directly to function pointers. The user can only set function pointers to existing identifiers, which are either a top-level function name or another function pointer.

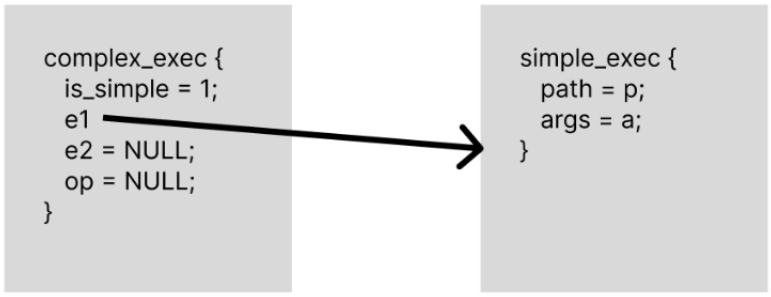
Executables

Executables are implemented using two different structs that are linked together in a binary tree structure. The first struct is called `simple_exec`, which contains an executable's path and arguments. The second struct is called `complex_exec`, which is used to link executables together as a result of a executable binary operator. Every leaf of this tree is a `simple_exec` struct, and every non-leaf is a `complex_exec` struct.

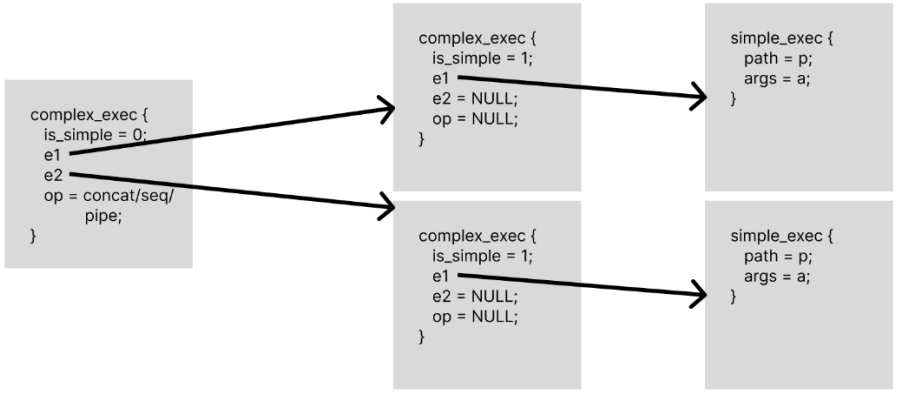
One thing to note is that simple executables in BlueShell are wrapped within a `complex_exec` struct where `is_simple = 1`. With this in mind, the fields of a `complex_exec` struct are a boolean representing whether it is simple or not, an operator enum (null if `is_simple = 1`) that can either represent concatenation, sequencing, or piping, and two pointers that work as follows:

1. If `is_simple = 1`, then `e1` points to a `simple_exec` struct and `e2` is null.
2. If `is_simple = 0`, then `e1` and `e2` will both point to `complex_exec` structs.

The below diagrams show how simple and complex executables are represented:



This is a simple executable.



This is a complex executable pointing to 2 simple executables. Note that only one `op` can be present at a time.

Executable Binary Operators

Note that in this section we use C struct notation to denote which field of a complex executable we are referring to.

Suppose we have a complex executable struct `e`, where `e.is_simple = 0` (not simple). To evaluate `./e`, we use the following algorithm:

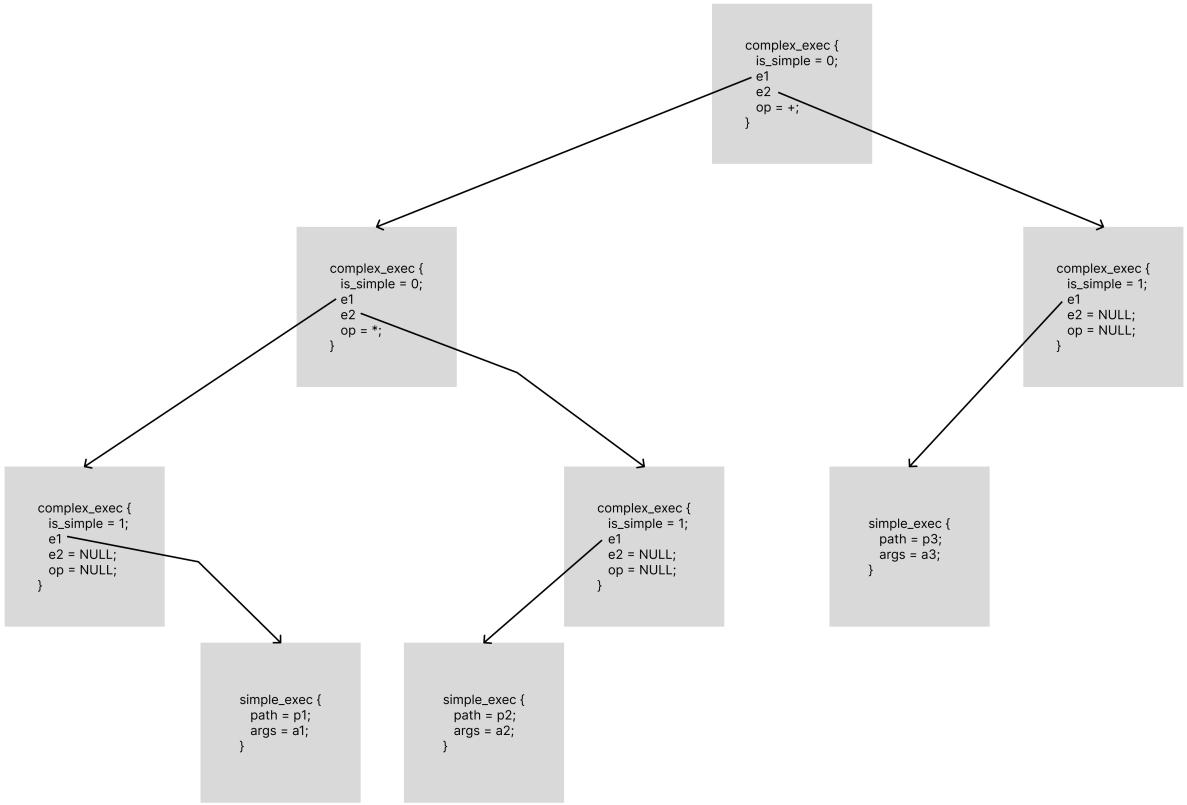
1. Regardless of what `e.op` is, we will first recursively evaluate `e.e1`, or the left child of `e`. This will evaluate to a string.
2. Depending on what `e.op` is, we will proceed as follows:
   (a) If the operation is concatenation, we will recursively evaluate `e.e2` and return the result of calling `strcat` on the left string followed by the right string.
   (b) If the operation is sequencing, we will recursively evaluate `e.e2` and return it, discarding the result of `e.e1`.
   (c) If the operation is piping, we will cache the result of `e.e1` to a file. The name of this file will change as the number of pipes increases, avoiding file conflicts if multiple pipes are encountered. We will then recursively evaluate `e.e2` with one caveat: the cached file will replace stdin on the left-most child of `e.e2`. The result of `e.e2` will be returned.

On the next two pages, we have provided some examples of complex executables along with an explanation of how they run.

Example 1: (e1 * e2) + e3

First we encounter a complex executable with a '+' operator. Since add is left associative, we first execute the executable on the left. This is also a complex executable with the operator '*'. Since '*' is also left associative, we first execute the executable on the left. This is a simple executable so we run the executable as normal and return the output as a string. We then return to the complex executable with the '*' operator.

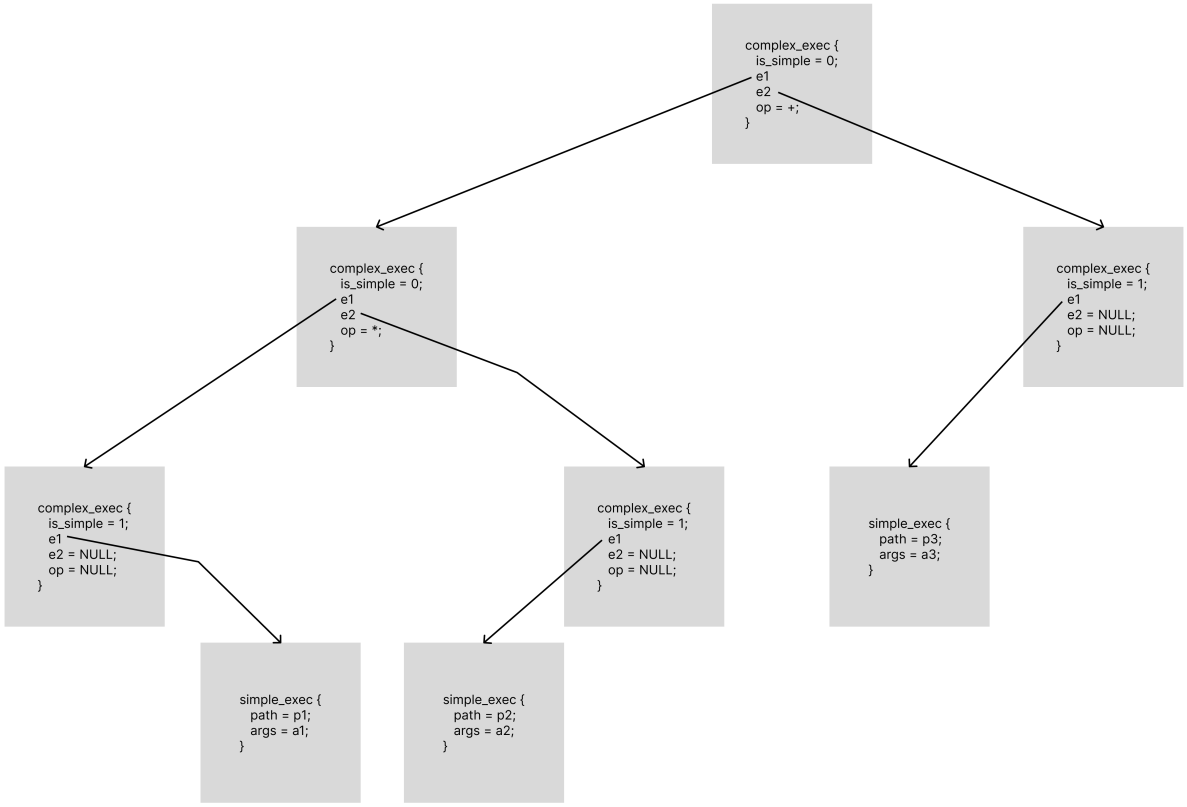
We have evaluated the left side so we will now evaluate the right side. This is a simple executable so we will run it as normal and return the output as a string. The '*' operator only returns the right executable output so we only return that. We are back to the complex executable with the '+' operator. We now run the executable on the right side which is a simple executable so we execute it as normal, returning the output as a string. We are then back at the '+' complex executable which concatenates the two strings together and returns the resulting string.



Example 2: e1 |(e2 + e3)

First we encounter a complex executable with an '|' operator. Since pipe is left associative, we first execute the executable on the left. This is a simple executable so we execute it as normal and return the resulting string. We then return to the complex executable with the '|' operator.

We have evaluated the left side so we will now evaluate the right side. This is a complex executable so with the '+' operator. Since '+' is left associative, we evaluate the left side first. This is a simple executable so we will execute it. However, because we are on the right side of a pipe and have also reached the left-most node of the right-side, we will pipe the input from the left side of the executable into this simple executable. We will then return the output and return, We have returned to the complex executable with the '+' operator. Now we will evaluate the right side which is a simple executable so we will run it normally and return the output. Then the strings returned from both sides of the concatenation are concatenated and returned.



5.3 Contributions

Component	Contributions
Scanner & Parser	Team
AST	Team
SAST	Alan & Mary-Joy
Semantic Analyzer	Kenny, Alan, and Mary-Joy
Code Generation	Team
C Helper	Alan & Mary-Joy

6 Test Plan

Our general plan for testing was to write unit and integration tests for features as we developed them. The general flow was as follows:

1. Write unit and integration tests for a feature. Passing tests begin with `test-` and failing tests with `fail-`.
2. Develop the feature.
3. Use our `compile.sh` script to generate an executable for the test file.
4. Run the executable and manually verify the output is correct or that the correct compiler error is raised.
5. Use our `make-gsts.sh` script to generate a gold-standard (`.gst`) file.

If we completed a larger feature or made changes to an existing feature, we would generally use our `testall.sh` script . This would run all of our tests and compare them against the gold-standard.

One problem we encountered was testing commands such as `ls` which rely on the local environment. We would try to avoid these if possible, but if we did have to run these types of tests, we would manually regenerate gold-standards when necessary.

6.1 Test Scripts and Automation

The full code listings for our testing scripts are located in Appendix 8.2

Makefile
The Makefile includes compilation commands and rules used in the testing scripts.

testall.sh
This scripts runs all our tests and compares the output to our gold standards. Any differences will be printed out. A flag can be passed in to specify the type of tests - either scanner-parser or SAST. If no test type flag is passed in, the script automatically runs our end-to-end tests.

make-gsts.sh
This script creates gold-standard output files for tests. It takes a required parameter specifying what kind of tests to make the gold-standards for, and an optional parameter of a filename, which if passed in, creates the gold standard for the specific tests. If no arguments are given, it makes the gold standard for every test.

compile.sh
This script takes in a BlueShell filename as an argument and compiles it to create an executable.

6.2 Contributions

Tina authored the `Makefile` and all of the bash scripts utilized in the testing framework. Kenny also assisted with some of the `Makefile` commands.
All team members assisted in writing tests.

6.3 Example Program 1: run-programs.bs

BlueShell Code

```
/* run-programs.bs
   compiles and runs some BlueShell test programs */

// Executes an executable type
string execute_exec(exec e) {
    return ./e;
}

// Executes an executable type
exec create_execs(string s) {
    return <s>;
}

//compiles a bs executable
exec create_compile_execs(string s) {
    return <"./compile.sh" withargs [s]>;
}

//maps a list of executables over a function and outputs a
   list of strings
list of exec map_string_to_exec(function (string -> exec)
   func, list of string strings) {
    int l = len strings;

    list of exec new_execs = [];
    for (int i = l - 1; i >= 0; i = i - 1) {
        new_execs = func(strings[i]) :: new_execs;
    }
    return new_execs;
}

// maps a list of strings over a function and outputs a list
   of executables
list of string map_exec_to_string(function (exec -> string)
   func, list of exec execs) {
    int l = len execs;
    list of string new_strings = [];
    for (int i = l - 1; i >= 0; i = i - 1) {
        new_strings = func(execs[i]) :: new_strings;
    }
    return new_strings;
}

//creates exeutables that compiles the following tests
list of string execs_to_compile =
   ["tests/test-echo1.bs","tests/test-pipe1.bs",
   "tests/test-concatseq1.bs"];
```

```
list of exec compile_execs =
    map_string_to_exec(create_compile_execs, execs_to_compile);

// compiles the scripts
map_exec_to_string(execute_exec, compile_execs);

// executes the compiled bs programs
list of string run_execs = ["./test-echo1.exe",
    "./test-pipe1.exe", "./test-concatseq1.exe"];
list of exec final_execs = map_string_to_exec(create_execs,
    run_execs);
map_exec_to_string(execute_exec, final_execs);
```

LLVM Code

```
; ModuleID = 'BlueShell'
source_filename = "BlueShell"

@string = private unnamed_addr constant [13 x i8]
    c"./compile.sh\00", align 1
@string.1 = private unnamed_addr constant [20 x i8]
    c"tests/test-echo1.bs\00", align 1
@string.2 = private unnamed_addr constant [20 x i8]
    c"tests/test-pipe1.bs\00", align 1
@string.3 = private unnamed_addr constant [25 x i8]
    c"tests/test-concatseq1.bs\00", align 1
@string.4 = private unnamed_addr constant [17 x i8]
    c"./test-echo1.exe\00", align 1
@string.5 = private unnamed_addr constant [17 x i8]
    c"./test-pipe1.exe\00", align 1
@string.6 = private unnamed_addr constant [22 x i8]
    c"./test-concatseq1.exe\00", align 1

declare i8* @execvp_helper(i8*, { i8*, i8*, i32 }*, ...)

declare i8* @recurse_exec({ i1, i8*, i8*, i32 }*, ...)

define i32 @main() {
entry:
  %malloccall = tail call i8* @malloc(i32 ptrtoint ({ i1, i8*,
    i8*, i32 }* (i8**)** getelementptr ({ i1, i8*, i8*, i32 }*
    (i8**)*, { i1, i8*, i8*, i32 }* (i8**)** null, i32 1) to
    i32))
  %"function def" = bitcast i8* %malloccall to { i1, i8*, i8*,
    i32 }* (i8**)**
  store { i1, i8*, i8*, i32 }* (i8**)* @create_compile_execs,
    { i1, i8*, i8*, i32 }* (i8**)** %"function def", align 8
  %malloccall1 = tail call i8* @malloc(i32 ptrtoint ({ i1,
    i8*, i8*, i32 }* (i8**)** getelementptr ({ i1, i8*, i8*,
    i32 }* (i8**)*, { i1, i8*, i8*, i32 }* (i8**)** null, i32
    1) to i32))
```

```
22  %"function def2" = bitcast i8* %malloccall1 to { i1, i8*,
     i8*, i32 }* (i8**)**
23  store { i1, i8*, i8*, i32 }* (i8**)* @create_execs, { i1,
     i8*, i8*, i32 }* (i8**)** %"function def2", align 8
24  %malloccall3 = tail call i8* @malloc(i32 ptrtoint (i8** ({
     i1, i8*, i8*, i32 }*)** getelementptr (i8** ({ i1, i8*,
     i8*, i32 }*)*, i8** ({ i1, i8*, i8*, i32 }*)** null, i32 1)
     to i32))
25  %"function def4" = bitcast i8* %malloccall3 to i8** ({ i1,
     i8*, i8*, i32 }*)**
26  store i8** ({ i1, i8*, i8*, i32 }*)* @execute_exec, i8** ({
     i1, i8*, i8*, i32 }*)** %"function def4", align 8
27  %malloccall5 = tail call i8* @malloc(i32 ptrtoint ({ i8*,
     i8*, i32 }* (i8** ({ i1, i8*, i8*, i32 }*)*, { i8*, i8*,
     i32 }*)** getelementptr ({ i8*, i8*, i32 }* (i8** ({ i1,
     i8*, i8*, i32 }*)*, { i8*, i8*, i32 }*)*, { i8*, i8*, i32
     }* (i8** ({ i1, i8*, i8*, i32 }*)*, { i8*, i8*, i32 }*)**
     null, i32 1) to i32))
28  %"function def6" = bitcast i8* %malloccall5 to { i8*, i8*,
     i32 }* (i8** ({ i1, i8*, i8*, i32 }*)*, { i8*, i8*, i32
     }*)**
29  store { i8*, i8*, i32 }* (i8** ({ i1, i8*, i8*, i32 }*)*, {
     i8*, i8*, i32 }*)* @map_exec_to_string, { i8*, i8*, i32 }*
     (i8** ({ i1, i8*, i8*, i32 }*)*, { i8*, i8*, i32 }*)**
     %"function def6", align 8
30  %malloccall7 = tail call i8* @malloc(i32 ptrtoint ({ i8*,
     i8*, i32 }* ({ i1, i8*, i8*, i32 }* (i8**)*, { i8*, i8*,
     i32 }*)** getelementptr ({ i8*, i8*, i32 }* ({ i1, i8*,
     i8*, i32 }* (i8**)*, { i8*, i8*, i32 }*)*, { i8*, i8*, i32
     }* ({ i1, i8*, i8*, i32 }* (i8**)*, { i8*, i8*, i32 }*)**
     null, i32 1) to i32))
31  %"function def8" = bitcast i8* %malloccall7 to { i8*, i8*,
     i32 }* ({ i1, i8*, i8*, i32 }* (i8**)*, { i8*, i8*, i32
     }*)**
32  store { i8*, i8*, i32 }* ({ i1, i8*, i8*, i32 }* (i8**)*, {
     i8*, i8*, i32 }*)* @map_string_to_exec, { i8*, i8*, i32 }*
     ({ i1, i8*, i8*, i32 }* (i8**)*, { i8*, i8*, i32 }*)**
     %"function def8", align 8
33  %malloccall9 = tail call i8* @malloc(i32 ptrtoint (i8**
     getelementptr (i8*, i8** null, i32 1) to i32))
34  %double_string_ptr = bitcast i8* %malloccall9 to i8**
35  store i8* getelementptr inbounds ([20 x i8], [20 x i8]*
     @string.1, i32 0, i32 0), i8** %double_string_ptr, align 8
36  %malloccall10 = tail call i8* @malloc(i32 ptrtoint (i8***
     getelementptr (i8**, i8*** null, i32 1) to i32))
37  %value_ptr = bitcast i8* %malloccall10 to i8***
38  store i8** %double_string_ptr, i8*** %value_ptr, align 8
39  %malloccall11 = tail call i8* @malloc(i32 ptrtoint ({ i8*,
     i8*, i32 }* getelementptr ({ i8*, i8*, i32 }, { i8*, i8*,
     i32 }* null, i32 1) to i32))
40  %list_node = bitcast i8* %malloccall11 to { i8*, i8*, i32 }*
```

```
41   %struct_val_ptr = getelementptr inbounds { i8*, i8*, i32 },
      { i8*, i8*, i32 }* %list_node, i32 0, i32 0
42   %struct_ptr_ptr = getelementptr inbounds { i8*, i8*, i32 },
      { i8*, i8*, i32 }* %list_node, i32 0, i32 1
43   %struct_ty_ptr = getelementptr inbounds { i8*, i8*, i32 }, {
      i8*, i8*, i32 }* %list_node, i32 0, i32 2
44   %malloccall12 = tail call i8* @malloc(i32 ptrtoint (i8**
      getelementptr (i8*, i8** null, i32 1) to i32))
45   %double_string_ptr13 = bitcast i8* %malloccall12 to i8**
46   store i8* getelementptr inbounds ([20 x i8], [20 x i8]*
      @string.2, i32 0, i32 0), i8** %double_string_ptr13, align 8
47   %malloccall14 = tail call i8* @malloc(i32 ptrtoint (i8***
      getelementptr (i8**, i8*** null, i32 1) to i32))
48   %value_ptr15 = bitcast i8* %malloccall14 to i8***
49   store i8** %double_string_ptr13, i8*** %value_ptr15, align 8
50   %malloccall16 = tail call i8* @malloc(i32 ptrtoint ({ i8*,
      i8*, i32 }* getelementptr ({ i8*, i8*, i32 }, { i8*, i8*,
      i32 }* null, i32 1) to i32))
51   %list_node17 = bitcast i8* %malloccall16 to { i8*, i8*, i32
      }*
52   %struct_val_ptr18 = getelementptr inbounds { i8*, i8*, i32
      }, { i8*, i8*, i32 }* %list_node17, i32 0, i32 0
53   %struct_ptr_ptr19 = getelementptr inbounds { i8*, i8*, i32
      }, { i8*, i8*, i32 }* %list_node17, i32 0, i32 1
54   %struct_ty_ptr20 = getelementptr inbounds { i8*, i8*, i32 },
      { i8*, i8*, i32 }* %list_node17, i32 0, i32 2
55   %malloccall21 = tail call i8* @malloc(i32 ptrtoint (i8**
      getelementptr (i8*, i8** null, i32 1) to i32))
56   %double_string_ptr22 = bitcast i8* %malloccall21 to i8**
57   store i8* getelementptr inbounds ([25 x i8], [25 x i8]*
      @string.3, i32 0, i32 0), i8** %double_string_ptr22, align 8
58   %malloccall23 = tail call i8* @malloc(i32 ptrtoint (i8***
      getelementptr (i8**, i8*** null, i32 1) to i32))
59   %value_ptr24 = bitcast i8* %malloccall23 to i8***
60   store i8** %double_string_ptr22, i8*** %value_ptr24, align 8
61   %malloccall25 = tail call i8* @malloc(i32 ptrtoint ({ i8*,
      i8*, i32 }* getelementptr ({ i8*, i8*, i32 }, { i8*, i8*,
      i32 }* null, i32 1) to i32))
62   %list_node26 = bitcast i8* %malloccall25 to { i8*, i8*, i32
      }*
63   %struct_val_ptr27 = getelementptr inbounds { i8*, i8*, i32
      }, { i8*, i8*, i32 }* %list_node26, i32 0, i32 0
64   %struct_ptr_ptr28 = getelementptr inbounds { i8*, i8*, i32
      }, { i8*, i8*, i32 }* %list_node26, i32 0, i32 1
65   %struct_ty_ptr29 = getelementptr inbounds { i8*, i8*, i32 },
      { i8*, i8*, i32 }* %list_node26, i32 0, i32 2
66   %casted_ptr_ptr = bitcast i8** %struct_ptr_ptr28 to { i8*,
      i8*, i32 }**
67   store { i8*, i8*, i32 }* null, { i8*, i8*, i32 }**
      %casted_ptr_ptr, align 8
68   %casted_val = bitcast i8*** %value_ptr24 to i8*
69   store i8* %casted_val, i8** %struct_val_ptr27, align 8
```

```
70  store i32 4, i32* %struct_ty_ptr29, align 4
71  %casted_ptr_ptr30 = bitcast i8** %struct_ptr_ptr19 to { i8*,
     i8*, i32 }**
72  store { i8*, i8*, i32 }* %list_node26, { i8*, i8*, i32 }**
     %casted_ptr_ptr30, align 8
73  %casted_val31 = bitcast i8*** %value_ptr15 to i8*
74  store i8* %casted_val31, i8** %struct_val_ptr18, align 8
75  store i32 4, i32* %struct_ty_ptr20, align 4
76  %casted_ptr_ptr32 = bitcast i8** %struct_ptr_ptr to { i8*,
     i8*, i32 }**
77  store { i8*, i8*, i32 }* %list_node17, { i8*, i8*, i32 }**
     %casted_ptr_ptr32, align 8
78  %casted_val33 = bitcast i8*** %value_ptr to i8*
79  store i8* %casted_val33, i8** %struct_val_ptr, align 8
80  store i32 4, i32* %struct_ty_ptr, align 4
81  %malloccall34 = tail call i8* @malloc(i32 ptrtoint ({ i8*,
     i8*, i32 }** getelementptr ({ i8*, i8*, i32 }*, { i8*, i8*,
     i32 }** null, i32 1) to i32))
82  %"variable ptr" = bitcast i8* %malloccall34 to { i8*, i8*,
     i32 }**
83  store { i8*, i8*, i32 }* %list_node, { i8*, i8*, i32 }**
     %"variable ptr", align 8
84  %fval = load { i8*, i8*, i32 }* ({ i1, i8*, i8*, i32 }*
     (i8**)*, { i8*, i8*, i32 }*)*, { i8*, i8*, i32 }* ({ i1,
     i8*, i8*, i32 }* (i8**)*, { i8*, i8*, i32 }*)** %"function
     def8", align 8
85  %execs_to_compile = load { i8*, i8*, i32 }*, { i8*, i8*, i32
     }** %"variable ptr", align 8
86  %create_compile_execs = load { i1, i8*, i8*, i32 }* (i8**)*,
     { i1, i8*, i8*, i32 }* (i8**)** %"function def", align 8
87  %map_string_to_exec_result = call { i8*, i8*, i32 }* %fval({
     i1, i8*, i8*, i32 }* (i8**)* %create_compile_execs, { i8*,
     i8*, i32 }* %execs_to_compile)
88  %malloccall35 = tail call i8* @malloc(i32 ptrtoint ({ i8*,
     i8*, i32 }** getelementptr ({ i8*, i8*, i32 }*, { i8*, i8*,
     i32 }** null, i32 1) to i32))
89  %"variable ptr36" = bitcast i8* %malloccall35 to { i8*, i8*,
     i32 }**
90  store { i8*, i8*, i32 }* %map_string_to_exec_result, { i8*,
     i8*, i32 }** %"variable ptr36", align 8
91  %fval37 = load { i8*, i8*, i32 }* (i8** ({ i1, i8*, i8*, i32
     }*)*, { i8*, i8*, i32 }*)*, { i8*, i8*, i32 }* (i8** ({ i1,
     i8*, i8*, i32 }*)*, { i8*, i8*, i32 }*)** %"function def6",
     align 8
92  %compile_execs = load { i8*, i8*, i32 }*, { i8*, i8*, i32
     }** %"variable ptr36", align 8
93  %execute_exec = load i8** ({ i1, i8*, i8*, i32 }*)*, i8** ({
     i1, i8*, i8*, i32 }*)** %"function def4", align 8
94  %map_exec_to_string_result = call { i8*, i8*, i32 }*
     %fval37(i8** ({ i1, i8*, i8*, i32 }*)* %execute_exec, {
     i8*, i8*, i32 }* %compile_execs)
```

```
95  %malloccall38 = tail call i8* @malloc(i32 ptrtoint (i8**
     getelementptr (i8*, i8** null, i32 1) to i32))
96  %double_string_ptr39 = bitcast i8* %malloccall38 to i8**
97  store i8* getelementptr inbounds ([17 x i8], [17 x i8]*
     @string.4, i32 0, i32 0), i8** %double_string_ptr39, align 8
98  %malloccall40 = tail call i8* @malloc(i32 ptrtoint (i8***
     getelementptr (i8**, i8*** null, i32 1) to i32))
99  %value_ptr41 = bitcast i8* %malloccall40 to i8***
100 store i8** %double_string_ptr39, i8*** %value_ptr41, align 8
101 %malloccall42 = tail call i8* @malloc(i32 ptrtoint ({ i8*,
     i8*, i32 }* getelementptr ({ i8*, i8*, i32 }, { i8*, i8*,
     i32 }* null, i32 1) to i32))
102 %list_node43 = bitcast i8* %malloccall42 to { i8*, i8*, i32
     }*
103 %struct_val_ptr44 = getelementptr inbounds { i8*, i8*, i32
     }, { i8*, i8*, i32 }* %list_node43, i32 0, i32 0
104 %struct_ptr_ptr45 = getelementptr inbounds { i8*, i8*, i32
     }, { i8*, i8*, i32 }* %list_node43, i32 0, i32 1
105 %struct_ty_ptr46 = getelementptr inbounds { i8*, i8*, i32 },
     { i8*, i8*, i32 }* %list_node43, i32 0, i32 2
106 %malloccall47 = tail call i8* @malloc(i32 ptrtoint (i8**
     getelementptr (i8*, i8** null, i32 1) to i32))
107 %double_string_ptr48 = bitcast i8* %malloccall47 to i8**
108 store i8* getelementptr inbounds ([17 x i8], [17 x i8]*
     @string.5, i32 0, i32 0), i8** %double_string_ptr48, align 8
109 %malloccall49 = tail call i8* @malloc(i32 ptrtoint (i8***
     getelementptr (i8**, i8*** null, i32 1) to i32))
110 %value_ptr50 = bitcast i8* %malloccall49 to i8***
111 store i8** %double_string_ptr48, i8*** %value_ptr50, align 8
112 %malloccall51 = tail call i8* @malloc(i32 ptrtoint ({ i8*,
     i8*, i32 }* getelementptr ({ i8*, i8*, i32 }, { i8*, i8*,
     i32 }* null, i32 1) to i32))
113 %list_node52 = bitcast i8* %malloccall51 to { i8*, i8*, i32
     }*
114 %struct_val_ptr53 = getelementptr inbounds { i8*, i8*, i32
     }, { i8*, i8*, i32 }* %list_node52, i32 0, i32 0
115 %struct_ptr_ptr54 = getelementptr inbounds { i8*, i8*, i32
     }, { i8*, i8*, i32 }* %list_node52, i32 0, i32 1
116 %struct_ty_ptr55 = getelementptr inbounds { i8*, i8*, i32 },
     { i8*, i8*, i32 }* %list_node52, i32 0, i32 2
117 %malloccall56 = tail call i8* @malloc(i32 ptrtoint (i8**
     getelementptr (i8*, i8** null, i32 1) to i32))
118 %double_string_ptr57 = bitcast i8* %malloccall56 to i8**
119 store i8* getelementptr inbounds ([22 x i8], [22 x i8]*
     @string.6, i32 0, i32 0), i8** %double_string_ptr57, align 8
120 %malloccall58 = tail call i8* @malloc(i32 ptrtoint (i8***
     getelementptr (i8**, i8*** null, i32 1) to i32))
121 %value_ptr59 = bitcast i8* %malloccall58 to i8***
122 store i8** %double_string_ptr57, i8*** %value_ptr59, align 8
123 %malloccall60 = tail call i8* @malloc(i32 ptrtoint ({ i8*,
     i8*, i32 }* getelementptr ({ i8*, i8*, i32 }, { i8*, i8*,
     i32 }* null, i32 1) to i32))
```

```
124   %list_node61 = bitcast i8* %malloccall60 to { i8*, i8*, i32
       }*
125   %struct_val_ptr62 = getelementptr inbounds { i8*, i8*, i32
       }, { i8*, i8*, i32 }* %list_node61, i32 0, i32 0
126   %struct_ptr_ptr63 = getelementptr inbounds { i8*, i8*, i32
       }, { i8*, i8*, i32 }* %list_node61, i32 0, i32 1
127   %struct_ty_ptr64 = getelementptr inbounds { i8*, i8*, i32 },
       { i8*, i8*, i32 }* %list_node61, i32 0, i32 2
128   %casted_ptr_ptr65 = bitcast i8** %struct_ptr_ptr63 to { i8*,
       i8*, i32 }**
129   store { i8*, i8*, i32 }* null, { i8*, i8*, i32 }**
       %casted_ptr_ptr65, align 8
130   %casted_val66 = bitcast i8*** %value_ptr59 to i8*
131   store i8* %casted_val66, i8** %struct_val_ptr62, align 8
132   store i32 4, i32* %struct_ty_ptr64, align 4
133   %casted_ptr_ptr67 = bitcast i8** %struct_ptr_ptr54 to { i8*,
       i8*, i32 }**
134   store { i8*, i8*, i32 }* %list_node61, { i8*, i8*, i32 }**
       %casted_ptr_ptr67, align 8
135   %casted_val68 = bitcast i8*** %value_ptr50 to i8*
136   store i8* %casted_val68, i8** %struct_val_ptr53, align 8
137   store i32 4, i32* %struct_ty_ptr55, align 4
138   %casted_ptr_ptr69 = bitcast i8** %struct_ptr_ptr45 to { i8*,
       i8*, i32 }**
139   store { i8*, i8*, i32 }* %list_node52, { i8*, i8*, i32 }**
       %casted_ptr_ptr69, align 8
140   %casted_val70 = bitcast i8*** %value_ptr41 to i8*
141   store i8* %casted_val70, i8** %struct_val_ptr44, align 8
142   store i32 4, i32* %struct_ty_ptr46, align 4
143   %malloccall71 = tail call i8* @malloc(i32 ptrtoint ({ i8*,
       i8*, i32 }** getelementptr ({ i8*, i8*, i32 }*, { i8*, i8*,
       i32 }** null, i32 1) to i32))
144   %"variable ptr72" = bitcast i8* %malloccall71 to { i8*, i8*,
       i32 }**
145   store { i8*, i8*, i32 }* %list_node43, { i8*, i8*, i32 }**
       %"variable ptr72", align 8
146   %fval73 = load { i8*, i8*, i32 }* ({ i1, i8*, i8*, i32 }*
       (i8**)*, { i8*, i8*, i32 }*)*, { i8*, i8*, i32 }* ({ i1,
       i8*, i8*, i32 }* (i8**)*, { i8*, i8*, i32 }*)** %"function
       def8", align 8
147   %run_execs = load { i8*, i8*, i32 }*, { i8*, i8*, i32 }**
       %"variable ptr72", align 8
148   %create_execs = load { i1, i8*, i8*, i32 }* (i8**)*, { i1,
       i8*, i8*, i32 }* (i8**)** %"function def2", align 8
149   %map_string_to_exec_result74 = call { i8*, i8*, i32 }*
       %fval73({ i1, i8*, i8*, i32 }* (i8**)* %create_execs, {
       i8*, i8*, i32 }* %run_execs)
150   %malloccall75 = tail call i8* @malloc(i32 ptrtoint ({ i8*,
       i8*, i32 }** getelementptr ({ i8*, i8*, i32 }*, { i8*, i8*,
       i32 }** null, i32 1) to i32))
151   %"variable ptr76" = bitcast i8* %malloccall75 to { i8*, i8*,
       i32 }**
```

```
  store { i8*, i8*, i32 }* %map_string_to_exec_result74, { i8*, i8*, i32 }** %"variable ptr76", align 8
  %fval77 = load { i8*, i8*, i32 }* (i8** ({ i1, i8*, i8*, i32 }*)*, { i8*, i8*, i32 }*)*, { i8*, i8*, i32 }* (i8** ({ i1, i8*, i8*, i32 }*)*, { i8*, i8*, i32 }*)** %"function def6", align 8
  %final_execs = load { i8*, i8*, i32 }*, { i8*, i8*, i32 }** %"variable ptr76", align 8
  %execute_exec78 = load i8** ({ i1, i8*, i8*, i32 }*)*, i8** ({ i1, i8*, i8*, i32 }*)** %"function def4", align 8
  %map_exec_to_string_result79 = call { i8*, i8*, i32 }* %fval77(i8** ({ i1, i8*, i8*, i32 }*)* %execute_exec78, { i8*, i8*, i32 }* %final_execs)
  ret i32 0
}

define { i8*, i8*, i32 }* @map_exec_to_string(i8** ({ i1, i8*, i8*, i32 }*)* %0, { i8*, i8*, i32 }* %1) {
entry:
  %malloccall = tail call i8* @malloc(i32 ptrtoint ({ i1, i8*, i8*, i32 }* (i8**)** getelementptr ({ i1, i8*, i8*, i32 }* (i8**)*, { i1, i8*, i8*, i32 }* (i8**)** null, i32 1) to i32))
  %"function def" = bitcast i8* %malloccall to { i1, i8*, i8*, i32 }* (i8**)**
  store { i1, i8*, i8*, i32 }* (i8**)* @create_compile_execs, { i1, i8*, i8*, i32 }* (i8**)** %"function def", align 8
  %malloccall1 = tail call i8* @malloc(i32 ptrtoint ({ i1, i8*, i8*, i32 }* (i8**)** getelementptr ({ i1, i8*, i8*, i32 }* (i8**)*, { i1, i8*, i8*, i32 }* (i8**)** null, i32 1) to i32))
  %"function def2" = bitcast i8* %malloccall1 to { i1, i8*, i8*, i32 }* (i8**)**
  store { i1, i8*, i8*, i32 }* (i8**)* @create_execs, { i1, i8*, i8*, i32 }* (i8**)** %"function def2", align 8
  %malloccall3 = tail call i8* @malloc(i32 ptrtoint (i8** ({ i1, i8*, i8*, i32 }*)** getelementptr (i8** ({ i1, i8*, i8*, i32 }*)*, i8** ({ i1, i8*, i8*, i32 }*)** null, i32 1) to i32))
  %"function def4" = bitcast i8* %malloccall3 to i8** ({ i1, i8*, i8*, i32 }*)**
  store i8** ({ i1, i8*, i8*, i32 }*)* @execute_exec, i8** ({ i1, i8*, i8*, i32 }*)** %"function def4", align 8
  %malloccall5 = tail call i8* @malloc(i32 ptrtoint ({ i8*, i8*, i32 }* (i8** ({ i1, i8*, i8*, i32 }*)*, { i8*, i8*, i32 }*)** getelementptr ({ i8*, i8*, i32 }* (i8** ({ i1, i8*, i8*, i32 }*)*, { i8*, i8*, i32 }*)*, { i8*, i8*, i32 }* (i8** ({ i1, i8*, i8*, i32 }*)*, { i8*, i8*, i32 }*)** null, i32 1) to i32))
  %"function def6" = bitcast i8* %malloccall5 to { i8*, i8*, i32 }* (i8** ({ i1, i8*, i8*, i32 }*)*, { i8*, i8*, i32 }*)**
```

```
173   store { i8*, i8*, i32 }* (i8** ({ i1, i8*, i8*, i32 }*)*, {
       i8*, i8*, i32 }*)* @map_exec_to_string, { i8*, i8*, i32 }*
       (i8** ({ i1, i8*, i8*, i32 }*)*, { i8*, i8*, i32 }*)**
       %"function def6", align 8
174   %malloccall7 = tail call i8* @malloc(i32 ptrtoint ({ i8*,
       i8*, i32 }* ({ i1, i8*, i8*, i32 }* (i8**)*, { i8*, i8*,
       i32 }*)** getelementptr ({ i8*, i8*, i32 }* ({ i1, i8*,
       i8*, i32 }* (i8**)*, { i8*, i8*, i32 }*)*, { i8*, i8*, i32
       }* ({ i1, i8*, i8*, i32 }* (i8**)*, { i8*, i8*, i32 }*)**
       null, i32 1) to i32))
175   %"function def8" = bitcast i8* %malloccall7 to { i8*, i8*,
       i32 }* ({ i1, i8*, i8*, i32 }* (i8**)*, { i8*, i8*, i32
       }*)**
176   store { i8*, i8*, i32 }* ({ i1, i8*, i8*, i32 }* (i8**)*, {
       i8*, i8*, i32 }*)* @map_string_to_exec, { i8*, i8*, i32 }*
       ({ i1, i8*, i8*, i32 }* (i8**)*, { i8*, i8*, i32 }*)**
       %"function def8", align 8
177   %malloccall9 = tail call i8* @malloc(i32 ptrtoint (i8** ({
       i1, i8*, i8*, i32 }*)** getelementptr (i8** ({ i1, i8*,
       i8*, i32 }*)*, i8** ({ i1, i8*, i8*, i32 }*)** null, i32 1)
       to i32))
178   %func = bitcast i8* %malloccall9 to i8** ({ i1, i8*, i8*,
       i32 }*)**
179   store i8** ({ i1, i8*, i8*, i32 }*)* %0, i8** ({ i1, i8*,
       i8*, i32 }*)** %func, align 8
180   %malloccall10 = tail call i8* @malloc(i32 ptrtoint ({ i8*,
       i8*, i32 }** getelementptr ({ i8*, i8*, i32 }*, { i8*, i8*,
       i32 }** null, i32 1) to i32))
181   %execs = bitcast i8* %malloccall10 to { i8*, i8*, i32 }**
182   store { i8*, i8*, i32 }* %1, { i8*, i8*, i32 }** %execs,
       align 8
183   %execs11 = load { i8*, i8*, i32 }*, { i8*, i8*, i32 }**
       %execs, align 8
184   %malloccall12 = tail call i8* @malloc(i32 ptrtoint ({ i8*,
       i8*, i32 }** getelementptr ({ i8*, i8*, i32 }*, { i8*, i8*,
       i32 }** null, i32 1) to i32))
185   %"e1 pointer" = bitcast i8* %malloccall12 to { i8*, i8*, i32
       }**
186   store { i8*, i8*, i32 }* %execs11, { i8*, i8*, i32 }** %"e1
       pointer", align 8
187   %malloccall13 = tail call i8* @malloc(i32 ptrtoint (i32*
       getelementptr (i32, i32* null, i32 1) to i32))
188   %counter_ptr = bitcast i8* %malloccall13 to i32*
189   store i32 0, i32* %counter_ptr, align 4
190   br label %length
191
192 length:                                           ; preds =
       %index_body, %entry
193   %malloccall14 = tail call i8* @malloc(i32 ptrtoint (i1*
       getelementptr (i1, i1* null, i32 1) to i32))
194   %bool_mem = bitcast i8* %malloccall14 to i1*
```

```
195   %2 = load { i8*, i8*, i32 }*, { i8*, i8*, i32 }** %"e1
       pointer", align 8
196   %3 = icmp ne { i8*, i8*, i32 }* %2, null
197   store i1 %3, i1* %bool_mem, align 1
198   %bool_mem15 = load i1, i1* %bool_mem, align 1
199   br i1 %bool_mem15, label %index_body, label %"length merge"
200
201 index_body:                                       ; preds =
       %length
202   %counter = load i32, i32* %counter_ptr, align 4
203   %"increment counter" = add i32 %counter, 1
204   store i32 %"increment counter", i32* %counter_ptr, align 4
205   %"get struct" = load { i8*, i8*, i32 }*, { i8*, i8*, i32 }**
       %"e1 pointer", align 8
206   %next_struct_ptr = getelementptr inbounds { i8*, i8*, i32 },
       { i8*, i8*, i32 }* %"get struct", i32 0, i32 1
207   %"e1' in while loop" = load i8*, i8** %next_struct_ptr,
       align 8
208   %"temp'" = bitcast i8* %"e1' in while loop" to { i8*, i8*,
       i32 }*
209   store { i8*, i8*, i32 }* %"temp'", { i8*, i8*, i32 }** %"e1
       pointer", align 8
210   %casted_ptr_ptr = bitcast i8* %"e1' in while loop" to { i8*,
       i8*, i32 }*
211   store { i8*, i8*, i32 }* %casted_ptr_ptr, { i8*, i8*, i32
       }** %"e1 pointer", align 8
212   br label %length
213
214 "length merge":                                   ; preds =
       %length
215   %malloccall16 = tail call i8* @malloc(i32 ptrtoint (i32**
       getelementptr (i32*, i32** null, i32 1) to i32))
216   %"variable ptr" = bitcast i8* %malloccall16 to i32**
217   store i32* %counter_ptr, i32** %"variable ptr", align 8
218   %malloccall17 = tail call i8* @malloc(i32 ptrtoint ({ i8*,
       i8*, i32 }** getelementptr ({ i8*, i8*, i32 }*, { i8*, i8*,
       i32 }** null, i32 1) to i32))
219   %"variable ptr18" = bitcast i8* %malloccall17 to { i8*, i8*,
       i32 }**
220   store { i8*, i8*, i32 }* null, { i8*, i8*, i32 }**
       %"variable ptr18", align 8
221   %malloccall19 = tail call i8* @malloc(i32 ptrtoint (i32*
       getelementptr (i32, i32* null, i32 1) to i32))
222   %int_mem = bitcast i8* %malloccall19 to i32*
223   store i32 1, i32* %int_mem, align 4
224   %l = load i32*, i32** %"variable ptr", align 8
225   %malloccall20 = tail call i8* @malloc(i32 ptrtoint (i32*
       getelementptr (i32, i32* null, i32 1) to i32))
226   %int_mem21 = bitcast i8* %malloccall20 to i32*
227   %"right side of sub" = load i32, i32* %int_mem, align 4
228   %"left side of sub" = load i32, i32* %l, align 4
229   %tmp = sub i32 %"left side of sub", %"right side of sub"
```

```
230   store i32 %tmp, i32* %int_mem21, align 4
231   %malloccall22 = tail call i8* @malloc(i32 ptrtoint (i32**
        getelementptr (i32*, i32** null, i32 1) to i32))
232   %"variable ptr23" = bitcast i8* %malloccall22 to i32**
233   store i32* %int_mem21, i32** %"variable ptr23", align 8
234   br label %while
235
236 while:                                            ; preds =
        %index_merge, %"length merge"
237   %malloccall51 = tail call i8* @malloc(i32 ptrtoint (i32*
        getelementptr (i32, i32* null, i32 1) to i32))
238   %int_mem52 = bitcast i8* %malloccall51 to i32*
239   store i32 0, i32* %int_mem52, align 4
240   %i53 = load i32*, i32** %"variable ptr23", align 8
241   %malloccall54 = tail call i8* @malloc(i32 ptrtoint (i1*
        getelementptr (i1, i1* null, i32 1) to i32))
242   %bool_mem55 = bitcast i8* %malloccall54 to i1*
243   %"right side of geq" = load i32, i32* %int_mem52, align 4
244   %"left side of geq" = load i32, i32* %i53, align 4
245   %tmp56 = icmp sge i32 %"left side of geq", %"right side of
        geq"
246   store i1 %tmp56, i1* %bool_mem55, align 1
247   %bool = load i1, i1* %bool_mem55, align 1
248   br i1 %bool, label %while_body, label %"while merge"
249
250 while_body:                                       ; preds =
        %while
251   %new_strings = load { i8*, i8*, i32 }*, { i8*, i8*, i32 }**
        %"variable ptr18", align 8
252   %fval = load i8** ({ i1, i8*, i8*, i32 }*)*, i8** ({ i1,
        i8*, i8*, i32 }*)** %func, align 8
253   %execs24 = load { i8*, i8*, i32 }*, { i8*, i8*, i32 }**
        %execs, align 8
254   %i = load i32*, i32** %"variable ptr23", align 8
255   %index_val = load i32, i32* %i, align 4
256   %malloccall25 = tail call i8* @malloc(i32 ptrtoint ({ i8*,
        i8*, i32 }** getelementptr ({ i8*, i8*, i32 }*, { i8*, i8*,
        i32 }** null, i32 1) to i32))
257   %"e1 pointer26" = bitcast i8* %malloccall25 to { i8*, i8*,
        i32 }**
258   store { i8*, i8*, i32 }* %execs24, { i8*, i8*, i32 }** %"e1
        pointer26", align 8
259   %malloccall27 = tail call i8* @malloc(i32 ptrtoint (i32*
        getelementptr (i32, i32* null, i32 1) to i32))
260   %counter_ptr28 = bitcast i8* %malloccall27 to i32*
261   store i32 0, i32* %counter_ptr28, align 4
262   br label %index
263
264 index:                                            ; preds =
        %index_body30, %while_body
265   %counter29 = load i32, i32* %counter_ptr28, align 4
266   %"index pred" = icmp ne i32 %index_val, %counter29
```

```
267   br i1 %"index pred", label %index_body30, label %index_merge
268
269 index_body30:                                     ; preds =
      %index
270   %counter31 = load i32, i32* %counter_ptr28, align 4
271   %"increment counter32" = add i32 %counter31, 1
272   store i32 %"increment counter32", i32* %counter_ptr28, align
      4
273   %"get struct33" = load { i8*, i8*, i32 }*, { i8*, i8*, i32
      }** %"e1 pointer26", align 8
274   %next_struct_ptr34 = getelementptr inbounds { i8*, i8*, i32
      }, { i8*, i8*, i32 }* %"get struct33", i32 0, i32 1
275   %"e1' in while loop35" = load i8*, i8** %next_struct_ptr34,
      align 8
276   %"temp'36" = bitcast i8* %"e1' in while loop35" to { i8*,
      i8*, i32 }*
277   store { i8*, i8*, i32 }* %"temp'36", { i8*, i8*, i32 }**
      %"e1 pointer26", align 8
278   %casted_ptr_ptr37 = bitcast i8* %"e1' in while loop35" to {
      i8*, i8*, i32 }*
279   store { i8*, i8*, i32 }* %casted_ptr_ptr37, { i8*, i8*, i32
      }** %"e1 pointer26", align 8
280   br label %index
281
282 index_merge:                                      ; preds =
      %index
283   %"get struct38" = load { i8*, i8*, i32 }*, { i8*, i8*, i32
      }** %"e1 pointer26", align 8
284   %elem_ptr_ptr = getelementptr inbounds { i8*, i8*, i32 }, {
      i8*, i8*, i32 }* %"get struct38", i32 0, i32 0
285   %casted = bitcast i8** %elem_ptr_ptr to { i1, i8*, i8*, i32
      }***
286   %elem_to_return = load { i1, i8*, i8*, i32 }**, { i1, i8*,
      i8*, i32 }*** %casted, align 8
287   %elem_to_return39 = load { i1, i8*, i8*, i32 }*, { i1, i8*,
      i8*, i32 }** %elem_to_return, align 8
288   %func_result = call i8** %fval({ i1, i8*, i8*, i32 }*
      %elem_to_return39)
289   %malloccall40 = tail call i8* @malloc(i32 ptrtoint (i8***
      getelementptr (i8**, i8*** null, i32 1) to i32))
290   %value_ptr = bitcast i8* %malloccall40 to i8***
291   store i8** %func_result, i8*** %value_ptr, align 8
292   %malloccall41 = tail call i8* @malloc(i32 ptrtoint ({ i8*,
      i8*, i32 }* getelementptr ({ i8*, i8*, i32 }, { i8*, i8*,
      i32 }* null, i32 1) to i32))
293   %list_node = bitcast i8* %malloccall41 to { i8*, i8*, i32 }*
294   %struct_val_ptr = getelementptr inbounds { i8*, i8*, i32 },
      { i8*, i8*, i32 }* %list_node, i32 0, i32 0
295   %struct_ptr_ptr = getelementptr inbounds { i8*, i8*, i32 },
      { i8*, i8*, i32 }* %list_node, i32 0, i32 1
296   %struct_ty_ptr = getelementptr inbounds { i8*, i8*, i32 }, {
      i8*, i8*, i32 }* %list_node, i32 0, i32 2
```

```
297   %casted_ptr_ptr42 = bitcast i8** %struct_ptr_ptr to { i8*,
       i8*, i32 }**
298   store { i8*, i8*, i32 }* %new_strings, { i8*, i8*, i32 }**
       %casted_ptr_ptr42, align 8
299   %casted_val = bitcast i8*** %value_ptr to i8*
300   store i8* %casted_val, i8** %struct_val_ptr, align 8
301   store i32 4, i32* %struct_ty_ptr, align 4
302   store { i8*, i8*, i32 }* %list_node, { i8*, i8*, i32 }**
       %"variable ptr18", align 8
303   %malloccall43 = tail call i8* @malloc(i32 ptrtoint (i32*
       getelementptr (i32, i32* null, i32 1) to i32))
304   %int_mem44 = bitcast i8* %malloccall43 to i32*
305   store i32 1, i32* %int_mem44, align 4
306   %i45 = load i32*, i32** %"variable ptr23", align 8
307   %malloccall46 = tail call i8* @malloc(i32 ptrtoint (i32*
       getelementptr (i32, i32* null, i32 1) to i32))
308   %int_mem47 = bitcast i8* %malloccall46 to i32*
309   %"right side of sub48" = load i32, i32* %int_mem44, align 4
310   %"left side of sub49" = load i32, i32* %i45, align 4
311   %tmp50 = sub i32 %"left side of sub49", %"right side of
       sub48"
312   store i32 %tmp50, i32* %int_mem47, align 4
313   store i32* %int_mem47, i32** %"variable ptr23", align 8
314   br label %while
315
316 "while merge":                                    ; preds =
       %while
317   %malloccall57 = tail call i8* @malloc(i32 ptrtoint ({ i8*,
       i8*, i32 }* getelementptr ({ i8*, i8*, i32 }, { i8*, i8*,
       i32 }* null, i32 1) to i32))
318   %"return malloc" = bitcast i8* %malloccall57 to { i8*, i8*,
       i32 }*
319   %new_strings58 = load { i8*, i8*, i32 }*, { i8*, i8*, i32
       }** %"variable ptr18", align 8
320   %"return load" = load { i8*, i8*, i32 }, { i8*, i8*, i32 }*
       %new_strings58, align 8
321   store { i8*, i8*, i32 } %"return load", { i8*, i8*, i32 }*
       %"return malloc", align 8
322   ret { i8*, i8*, i32 }* %"return malloc"
323 }
324
325 define { i8*, i8*, i32 }* @map_string_to_exec({ i1, i8*, i8*,
       i32 }* (i8**)* %0, { i8*, i8*, i32 }* %1) {
326 entry:
327   %malloccall = tail call i8* @malloc(i32 ptrtoint ({ i1, i8*,
       i8*, i32 }* (i8**)** getelementptr ({ i1, i8*, i8*, i32 }*
       (i8**)*, { i1, i8*, i8*, i32 }* (i8**)** null, i32 1) to
       i32))
328   %"function def" = bitcast i8* %malloccall to { i1, i8*, i8*,
       i32 }* (i8**)**
329   store { i1, i8*, i8*, i32 }* (i8**)* @create_compile_execs,
       { i1, i8*, i8*, i32 }* (i8**)** %"function def", align 8
```

```
330   %malloccall1 = tail call i8* @malloc(i32 ptrtoint ({ i1,
       i8*, i8*, i32 }* (i8**)** getelementptr ({ i1, i8*, i8*,
       i32 }* (i8**)*, { i1, i8*, i8*, i32 }* (i8**)** null, i32
       1) to i32))
331   %"function def2" = bitcast i8* %malloccall1 to { i1, i8*,
       i8*, i32 }* (i8**)**
332   store { i1, i8*, i8*, i32 }* (i8**)* @create_execs, { i1,
       i8*, i8*, i32 }* (i8**)** %"function def2", align 8
333   %malloccall3 = tail call i8* @malloc(i32 ptrtoint (i8** ({
       i1, i8*, i8*, i32 }*)** getelementptr (i8** ({ i1, i8*,
       i8*, i32 }*)*, i8** ({ i1, i8*, i8*, i32 }*)** null, i32 1)
       to i32))
334   %"function def4" = bitcast i8* %malloccall3 to i8** ({ i1,
       i8*, i8*, i32 }*)**
335   store i8** ({ i1, i8*, i8*, i32 }*)* @execute_exec, i8** ({
       i1, i8*, i8*, i32 }*)** %"function def4", align 8
336   %malloccall5 = tail call i8* @malloc(i32 ptrtoint ({ i8*,
       i8*, i32 }* (i8** ({ i1, i8*, i8*, i32 }*)*, { i8*, i8*,
       i32 }*)** getelementptr ({ i8*, i8*, i32 }* (i8** ({ i1,
       i8*, i8*, i32 }*)*, { i8*, i8*, i32 }*)*, { i8*, i8*, i32
       }* (i8** ({ i1, i8*, i8*, i32 }*)*, { i8*, i8*, i32 }*)**
       null, i32 1) to i32))
337   %"function def6" = bitcast i8* %malloccall5 to { i8*, i8*,
       i32 }* (i8** ({ i1, i8*, i8*, i32 }*)*, { i8*, i8*, i32
       }*)**
338   store { i8*, i8*, i32 }* (i8** ({ i1, i8*, i8*, i32 }*)*, {
       i8*, i8*, i32 }*)* @map_exec_to_string, { i8*, i8*, i32 }*
       (i8** ({ i1, i8*, i8*, i32 }*)*, { i8*, i8*, i32 }*)**
       %"function def6", align 8
339   %malloccall7 = tail call i8* @malloc(i32 ptrtoint ({ i8*,
       i8*, i32 }* ({ i1, i8*, i8*, i32 }* (i8**)*, { i8*, i8*,
       i32 }*)** getelementptr ({ i8*, i8*, i32 }* ({ i1, i8*,
       i8*, i32 }* (i8**)*, { i8*, i8*, i32 }*)*, { i8*, i8*, i32
       }* ({ i1, i8*, i8*, i32 }* (i8**)*, { i8*, i8*, i32 }*)**
       null, i32 1) to i32))
340   %"function def8" = bitcast i8* %malloccall7 to { i8*, i8*,
       i32 }* ({ i1, i8*, i8*, i32 }* (i8**)*, { i8*, i8*, i32
       }*)**
341   store { i8*, i8*, i32 }* ({ i1, i8*, i8*, i32 }* (i8**)*, {
       i8*, i8*, i32 }*)* @map_string_to_exec, { i8*, i8*, i32 }*
       ({ i1, i8*, i8*, i32 }* (i8**)*, { i8*, i8*, i32 }*)**
       %"function def8", align 8
342   %malloccall9 = tail call i8* @malloc(i32 ptrtoint ({ i1,
       i8*, i8*, i32 }* (i8**)** getelementptr ({ i1, i8*, i8*,
       i32 }* (i8**)*, { i1, i8*, i8*, i32 }* (i8**)** null, i32
       1) to i32))
343   %func = bitcast i8* %malloccall9 to { i1, i8*, i8*, i32 }*
       (i8**)**
344   store { i1, i8*, i8*, i32 }* (i8**)* %0, { i1, i8*, i8*, i32
       }* (i8**)** %func, align 8
345   %malloccall10 = tail call i8* @malloc(i32 ptrtoint ({ i8*,
       i8*, i32 }** getelementptr ({ i8*, i8*, i32 }*, { i8*, i8*,
```

```
    i32 }** null, i32 1) to i32))
  %strings = bitcast i8* %malloccall10 to { i8*, i8*, i32 }**
  store { i8*, i8*, i32 }* %1, { i8*, i8*, i32 }** %strings,
    align 8
  %strings11 = load { i8*, i8*, i32 }*, { i8*, i8*, i32 }**
    %strings, align 8
  %malloccall12 = tail call i8* @malloc(i32 ptrtoint ({ i8*,
    i8*, i32 }** getelementptr ({ i8*, i8*, i32 }*, { i8*, i8*,
    i32 }** null, i32 1) to i32))
  %"e1 pointer" = bitcast i8* %malloccall12 to { i8*, i8*, i32
    }**
  store { i8*, i8*, i32 }* %strings11, { i8*, i8*, i32 }**
    %"e1 pointer", align 8
  %malloccall13 = tail call i8* @malloc(i32 ptrtoint (i32*
    getelementptr (i32, i32* null, i32 1) to i32))
  %counter_ptr = bitcast i8* %malloccall13 to i32*
  store i32 0, i32* %counter_ptr, align 4
  br label %length

length:                                           ; preds =
    %index_body, %entry
  %malloccall14 = tail call i8* @malloc(i32 ptrtoint (i1*
    getelementptr (i1, i1* null, i32 1) to i32))
  %bool_mem = bitcast i8* %malloccall14 to i1*
  %2 = load { i8*, i8*, i32 }*, { i8*, i8*, i32 }** %"e1
    pointer", align 8
  %3 = icmp ne { i8*, i8*, i32 }* %2, null
  store i1 %3, i1* %bool_mem, align 1
  %bool_mem15 = load i1, i1* %bool_mem, align 1
  br i1 %bool_mem15, label %index_body, label %"length merge"

index_body:                                       ; preds =
    %length
  %counter = load i32, i32* %counter_ptr, align 4
  %"increment counter" = add i32 %counter, 1
  store i32 %"increment counter", i32* %counter_ptr, align 4
  %"get struct" = load { i8*, i8*, i32 }*, { i8*, i8*, i32 }**
    %"e1 pointer", align 8
  %next_struct_ptr = getelementptr inbounds { i8*, i8*, i32 },
    { i8*, i8*, i32 }* %"get struct", i32 0, i32 1
  %"e1' in while loop" = load i8*, i8** %next_struct_ptr,
    align 8
  %"temp'" = bitcast i8* %"e1' in while loop" to { i8*, i8*,
    i32 }*
  store { i8*, i8*, i32 }* %"temp'", { i8*, i8*, i32 }** %"e1
    pointer", align 8
  %casted_ptr_ptr = bitcast i8* %"e1' in while loop" to { i8*,
    i8*, i32 }*
  store { i8*, i8*, i32 }* %casted_ptr_ptr, { i8*, i8*, i32
    }** %"e1 pointer", align 8
  br label %length

```

```
"length merge":                                   ; preds =
    %length
  %malloccall16 = tail call i8* @malloc(i32 ptrtoint (i32**
    getelementptr (i32*, i32** null, i32 1) to i32))
  %"variable ptr" = bitcast i8* %malloccall16 to i32**
  store i32* %counter_ptr, i32** %"variable ptr", align 8
  %malloccall17 = tail call i8* @malloc(i32 ptrtoint ({ i8*,
    i8*, i32 }** getelementptr ({ i8*, i8*, i32 }*, { i8*, i8*,
    i32 }** null, i32 1) to i32))
  %"variable ptr18" = bitcast i8* %malloccall17 to { i8*, i8*,
    i32 }**
  store { i8*, i8*, i32 }* null, { i8*, i8*, i32 }**
    %"variable ptr18", align 8
  %malloccall19 = tail call i8* @malloc(i32 ptrtoint (i32*
    getelementptr (i32, i32* null, i32 1) to i32))
  %int_mem = bitcast i8* %malloccall19 to i32*
  store i32 1, i32* %int_mem, align 4
  %l = load i32*, i32** %"variable ptr", align 8
  %malloccall20 = tail call i8* @malloc(i32 ptrtoint (i32*
    getelementptr (i32, i32* null, i32 1) to i32))
  %int_mem21 = bitcast i8* %malloccall20 to i32*
  %"right side of sub" = load i32, i32* %int_mem, align 4
  %"left side of sub" = load i32, i32* %l, align 4
  %tmp = sub i32 %"left side of sub", %"right side of sub"
  store i32 %tmp, i32* %int_mem21, align 4
  %malloccall22 = tail call i8* @malloc(i32 ptrtoint (i32**
    getelementptr (i32*, i32** null, i32 1) to i32))
  %"variable ptr23" = bitcast i8* %malloccall22 to i32**
  store i32* %int_mem21, i32** %"variable ptr23", align 8
  br label %while

while:                                            ; preds =
    %index_merge, %"length merge"
  %malloccall51 = tail call i8* @malloc(i32 ptrtoint (i32*
    getelementptr (i32, i32* null, i32 1) to i32))
  %int_mem52 = bitcast i8* %malloccall51 to i32*
  store i32 0, i32* %int_mem52, align 4
  %i53 = load i32*, i32** %"variable ptr23", align 8
  %malloccall54 = tail call i8* @malloc(i32 ptrtoint (i1*
    getelementptr (i1, i1* null, i32 1) to i32))
  %bool_mem55 = bitcast i8* %malloccall54 to i1*
  %"right side of geq" = load i32, i32* %int_mem52, align 4
  %"left side of geq" = load i32, i32* %i53, align 4
  %tmp56 = icmp sge i32 %"left side of geq", %"right side of
    geq"
  store i1 %tmp56, i1* %bool_mem55, align 1
  %bool = load i1, i1* %bool_mem55, align 1
  br i1 %bool, label %while_body, label %"while merge"

while_body:                                       ; preds =
    %while
```

```
416   %new_execs = load { i8*, i8*, i32 }*, { i8*, i8*, i32 }**
       %"variable ptr18", align 8
417   %fval = load { i1, i8*, i8*, i32 }* (i8**)*, { i1, i8*, i8*,
       i32 }* (i8**)** %func, align 8
418   %strings24 = load { i8*, i8*, i32 }*, { i8*, i8*, i32 }**
       %strings, align 8
419   %i = load i32*, i32** %"variable ptr23", align 8
420   %index_val = load i32, i32* %i, align 4
421   %malloccall25 = tail call i8* @malloc(i32 ptrtoint ({ i8*,
       i8*, i32 }** getelementptr ({ i8*, i8*, i32 }*, { i8*, i8*,
       i32 }** null, i32 1) to i32))
422   %"e1 pointer26" = bitcast i8* %malloccall25 to { i8*, i8*,
       i32 }**
423   store { i8*, i8*, i32 }* %strings24, { i8*, i8*, i32 }**
       %"e1 pointer26", align 8
424   %malloccall27 = tail call i8* @malloc(i32 ptrtoint (i32*
       getelementptr (i32, i32* null, i32 1) to i32))
425   %counter_ptr28 = bitcast i8* %malloccall27 to i32*
426   store i32 0, i32* %counter_ptr28, align 4
427   br label %index
428
429 index:                                            ; preds =
       %index_body30, %while_body
430   %counter29 = load i32, i32* %counter_ptr28, align 4
431   %"index pred" = icmp ne i32 %index_val, %counter29
432   br i1 %"index pred", label %index_body30, label %index_merge
433
434 index_body30:                                     ; preds =
       %index
435   %counter31 = load i32, i32* %counter_ptr28, align 4
436   %"increment counter32" = add i32 %counter31, 1
437   store i32 %"increment counter32", i32* %counter_ptr28, align
       4
438   %"get struct33" = load { i8*, i8*, i32 }*, { i8*, i8*, i32
       }** %"e1 pointer26", align 8
439   %next_struct_ptr34 = getelementptr inbounds { i8*, i8*, i32
       }, { i8*, i8*, i32 }* %"get struct33", i32 0, i32 1
440   %"e1' in while loop35" = load i8*, i8** %next_struct_ptr34,
       align 8
441   %"temp'36" = bitcast i8* %"e1' in while loop35" to { i8*,
       i8*, i32 }*
442   store { i8*, i8*, i32 }* %"temp'36", { i8*, i8*, i32 }**
       %"e1 pointer26", align 8
443   %casted_ptr_ptr37 = bitcast i8* %"e1' in while loop35" to {
       i8*, i8*, i32 }*
444   store { i8*, i8*, i32 }* %casted_ptr_ptr37, { i8*, i8*, i32
       }** %"e1 pointer26", align 8
445   br label %index
446
447 index_merge:                                      ; preds =
       %index
```

```
448   %"get struct38" = load { i8*, i8*, i32 }*, { i8*, i8*, i32
       }** %"e1 pointer26", align 8
449   %elem_ptr_ptr = getelementptr inbounds { i8*, i8*, i32 }, {
       i8*, i8*, i32 }* %"get struct38", i32 0, i32 0
450   %casted = bitcast i8** %elem_ptr_ptr to i8****
451   %elem_to_return = load i8***, i8**** %casted, align 8
452   %elem_to_return39 = load i8**, i8*** %elem_to_return, align 8
453   %func_result = call { i1, i8*, i8*, i32 }* %fval(i8**
       %elem_to_return39)
454   %malloccall40 = tail call i8* @malloc(i32 ptrtoint ({ i1,
       i8*, i8*, i32 }** getelementptr ({ i1, i8*, i8*, i32 }*, {
       i1, i8*, i8*, i32 }** null, i32 1) to i32))
455   %value_ptr = bitcast i8* %malloccall40 to { i1, i8*, i8*,
       i32 }**
456   store { i1, i8*, i8*, i32 }* %func_result, { i1, i8*, i8*,
       i32 }** %value_ptr, align 8
457   %malloccall41 = tail call i8* @malloc(i32 ptrtoint ({ i8*,
       i8*, i32 }* getelementptr ({ i8*, i8*, i32 }, { i8*, i8*,
       i32 }* null, i32 1) to i32))
458   %list_node = bitcast i8* %malloccall41 to { i8*, i8*, i32 }*
459   %struct_val_ptr = getelementptr inbounds { i8*, i8*, i32 },
       { i8*, i8*, i32 }* %list_node, i32 0, i32 0
460   %struct_ptr_ptr = getelementptr inbounds { i8*, i8*, i32 },
       { i8*, i8*, i32 }* %list_node, i32 0, i32 1
461   %struct_ty_ptr = getelementptr inbounds { i8*, i8*, i32 }, {
       i8*, i8*, i32 }* %list_node, i32 0, i32 2
462   %casted_ptr_ptr42 = bitcast i8** %struct_ptr_ptr to { i8*,
       i8*, i32 }**
463   store { i8*, i8*, i32 }* %new_execs, { i8*, i8*, i32 }**
       %casted_ptr_ptr42, align 8
464   %casted_val = bitcast { i1, i8*, i8*, i32 }** %value_ptr to
       i8*
465   store i8* %casted_val, i8** %struct_val_ptr, align 8
466   store i32 5, i32* %struct_ty_ptr, align 4
467   store { i8*, i8*, i32 }* %list_node, { i8*, i8*, i32 }**
       %"variable ptr18", align 8
468   %malloccall43 = tail call i8* @malloc(i32 ptrtoint (i32*
       getelementptr (i32, i32* null, i32 1) to i32))
469   %int_mem44 = bitcast i8* %malloccall43 to i32*
470   store i32 1, i32* %int_mem44, align 4
471   %i45 = load i32*, i32** %"variable ptr23", align 8
472   %malloccall46 = tail call i8* @malloc(i32 ptrtoint (i32*
       getelementptr (i32, i32* null, i32 1) to i32))
473   %int_mem47 = bitcast i8* %malloccall46 to i32*
474   %"right side of sub48" = load i32, i32* %int_mem44, align 4
475   %"left side of sub49" = load i32, i32* %i45, align 4
476   %tmp50 = sub i32 %"left side of sub49", %"right side of
       sub48"
477   store i32 %tmp50, i32* %int_mem47, align 4
478   store i32* %int_mem47, i32** %"variable ptr23", align 8
479   br label %while
480
```

```
"while merge":                                    ; preds =
    %while
  %malloccall57 = tail call i8* @malloc(i32 ptrtoint ({ i8*,
    i8*, i32 }* getelementptr ({ i8*, i8*, i32 }, { i8*, i8*,
    i32 }* null, i32 1) to i32))
  %"return malloc" = bitcast i8* %malloccall57 to { i8*, i8*,
    i32 }*
  %new_execs58 = load { i8*, i8*, i32 }*, { i8*, i8*, i32 }**
    %"variable ptr18", align 8
  %"return load" = load { i8*, i8*, i32 }, { i8*, i8*, i32 }*
    %new_execs58, align 8
  store { i8*, i8*, i32 } %"return load", { i8*, i8*, i32 }*
    %"return malloc", align 8
  ret { i8*, i8*, i32 }* %"return malloc"
}

define { i1, i8*, i8*, i32 }* @create_compile_execs(i8** %0) {
entry:
  %malloccall = tail call i8* @malloc(i32 ptrtoint ({ i1, i8*,
    i8*, i32 }* (i8**)** getelementptr ({ i1, i8*, i8*, i32 }*
    (i8**)*, { i1, i8*, i8*, i32 }* (i8**)** null, i32 1) to
    i32))
  %"function def" = bitcast i8* %malloccall to { i1, i8*, i8*,
    i32 }* (i8**)**
  store { i1, i8*, i8*, i32 }* (i8**)* @create_compile_execs,
    { i1, i8*, i8*, i32 }* (i8**)** %"function def", align 8
  %malloccall1 = tail call i8* @malloc(i32 ptrtoint ({ i1,
    i8*, i8*, i32 }* (i8**)** getelementptr ({ i1, i8*, i8*,
    i32 }* (i8**)*, { i1, i8*, i8*, i32 }* (i8**)** null, i32
    1) to i32))
  %"function def2" = bitcast i8* %malloccall1 to { i1, i8*,
    i8*, i32 }* (i8**)**
  store { i1, i8*, i8*, i32 }* (i8**)* @create_execs, { i1,
    i8*, i8*, i32 }* (i8**)** %"function def2", align 8
  %malloccall3 = tail call i8* @malloc(i32 ptrtoint (i8** ({
    i1, i8*, i8*, i32 }*)** getelementptr (i8** ({ i1, i8*,
    i8*, i32 }*)*, i8** ({ i1, i8*, i8*, i32 }*)** null, i32 1)
    to i32))
  %"function def4" = bitcast i8* %malloccall3 to i8** ({ i1,
    i8*, i8*, i32 }*)**
  store i8** ({ i1, i8*, i8*, i32 }*)* @execute_exec, i8** ({
    i1, i8*, i8*, i32 }*)** %"function def4", align 8
  %malloccall5 = tail call i8* @malloc(i32 ptrtoint ({ i8*,
    i8*, i32 }* (i8** ({ i1, i8*, i8*, i32 }*)*, { i8*, i8*,
    i32 }*)** getelementptr ({ i8*, i8*, i32 }* (i8** ({ i1,
    i8*, i8*, i32 }*)*, { i8*, i8*, i32 }*)*, { i8*, i8*, i32
    }* (i8** ({ i1, i8*, i8*, i32 }*)*, { i8*, i8*, i32 }*)**
    null, i32 1) to i32))
  %"function def6" = bitcast i8* %malloccall5 to { i8*, i8*,
    i32 }* (i8** ({ i1, i8*, i8*, i32 }*)*, { i8*, i8*, i32
    }*)**
```

```
503   store { i8*, i8*, i32 }* (i8** ({ i1, i8*, i8*, i32 }*)*, {
       i8*, i8*, i32 }*)* @map_exec_to_string, { i8*, i8*, i32 }*
       (i8** ({ i1, i8*, i8*, i32 }*)*, { i8*, i8*, i32 }*)**
       %"function def6", align 8
504   %malloccall7 = tail call i8* @malloc(i32 ptrtoint ({ i8*,
       i8*, i32 }* ({ i1, i8*, i8*, i32 }* (i8**)*, { i8*, i8*,
       i32 }*)** getelementptr ({ i8*, i8*, i32 }* ({ i1, i8*,
       i8*, i32 }* (i8**)*, { i8*, i8*, i32 }*)*, { i8*, i8*, i32
       }* ({ i1, i8*, i8*, i32 }* (i8**)*, { i8*, i8*, i32 }*)**
       null, i32 1) to i32))
505   %"function def8" = bitcast i8* %malloccall7 to { i8*, i8*,
       i32 }* ({ i1, i8*, i8*, i32 }* (i8**)*, { i8*, i8*, i32
       }*)**
506   store { i8*, i8*, i32 }* ({ i1, i8*, i8*, i32 }* (i8**)*, {
       i8*, i8*, i32 }*)* @map_string_to_exec, { i8*, i8*, i32 }*
       ({ i1, i8*, i8*, i32 }* (i8**)*, { i8*, i8*, i32 }*)**
       %"function def8", align 8
507   %malloccall9 = tail call i8* @malloc(i32 ptrtoint (i8***
       getelementptr (i8**, i8*** null, i32 1) to i32))
508   %s = bitcast i8* %malloccall9 to i8***
509   store i8** %0, i8*** %s, align 8
510   %malloccall10 = tail call i8* @malloc(i32 ptrtoint ({ i1,
       i8*, i8*, i32 }* getelementptr ({ i1, i8*, i8*, i32 }, {
       i1, i8*, i8*, i32 }* null, i32 1) to i32))
511   %"return malloc" = bitcast i8* %malloccall10 to { i1, i8*,
       i8*, i32 }*
512   %malloccall11 = tail call i8* @malloc(i32 ptrtoint ({ i8**,
       { i8*, i8*, i32 }* }* getelementptr ({ i8**, { i8*, i8*,
       i32 }* }, { i8**, { i8*, i8*, i32 }* }* null, i32 1) to
       i32))
513   %struct_space = bitcast i8* %malloccall11 to { i8**, { i8*,
       i8*, i32 }* }*
514   %path_ptr = getelementptr inbounds { i8**, { i8*, i8*, i32
       }* }, { i8**, { i8*, i8*, i32 }* }* %struct_space, i32 0,
       i32 0
515   %malloccall12 = tail call i8* @malloc(i32 ptrtoint (i8**
       getelementptr (i8*, i8** null, i32 1) to i32))
516   %double_string_ptr = bitcast i8* %malloccall12 to i8**
517   store i8* getelementptr inbounds ([13 x i8], [13 x i8]*
       @string, i32 0, i32 0), i8** %double_string_ptr, align 8
518   store i8** %double_string_ptr, i8*** %path_ptr, align 8
519   %args_ptr = getelementptr inbounds { i8**, { i8*, i8*, i32
       }* }, { i8**, { i8*, i8*, i32 }* }* %struct_space, i32 0,
       i32 1
520   %s13 = load i8**, i8*** %s, align 8
521   %malloccall14 = tail call i8* @malloc(i32 ptrtoint (i8***
       getelementptr (i8**, i8*** null, i32 1) to i32))
522   %value_ptr = bitcast i8* %malloccall14 to i8***
523   store i8** %s13, i8*** %value_ptr, align 8
524   %malloccall15 = tail call i8* @malloc(i32 ptrtoint ({ i8*,
       i8*, i32 }* getelementptr ({ i8*, i8*, i32 }, { i8*, i8*,
       i32 }* null, i32 1) to i32))
```

```
525   %list_node = bitcast i8* %malloccall15 to { i8*, i8*, i32 }*
526   %struct_val_ptr = getelementptr inbounds { i8*, i8*, i32 },
       { i8*, i8*, i32 }* %list_node, i32 0, i32 0
527   %struct_ptr_ptr = getelementptr inbounds { i8*, i8*, i32 },
       { i8*, i8*, i32 }* %list_node, i32 0, i32 1
528   %struct_ty_ptr = getelementptr inbounds { i8*, i8*, i32 }, {
       i8*, i8*, i32 }* %list_node, i32 0, i32 2
529   %casted_ptr_ptr = bitcast i8** %struct_ptr_ptr to { i8*,
       i8*, i32 }**
530   store { i8*, i8*, i32 }* null, { i8*, i8*, i32 }**
       %casted_ptr_ptr, align 8
531   %casted_val = bitcast i8*** %value_ptr to i8*
532   store i8* %casted_val, i8** %struct_val_ptr, align 8
533   store i32 4, i32* %struct_ty_ptr, align 4
534   store { i8*, i8*, i32 }* %list_node, { i8*, i8*, i32 }**
       %args_ptr, align 8
535   %malloccall16 = tail call i8* @malloc(i32 ptrtoint ({ i1,
       i8*, i8*, i32 }* getelementptr ({ i1, i8*, i8*, i32 }, {
       i1, i8*, i8*, i32 }* null, i32 1) to i32))
536   %"complex exec struct" = bitcast i8* %malloccall16 to { i1,
       i8*, i8*, i32 }*
537   %"complex bool" = getelementptr inbounds { i1, i8*, i8*, i32
       }, { i1, i8*, i8*, i32 }* %"complex exec struct", i32 0,
       i32 0
538   %"complex e1" = getelementptr inbounds { i1, i8*, i8*, i32
       }, { i1, i8*, i8*, i32 }* %"complex exec struct", i32 0,
       i32 1
539   store i1 true, i1* %"complex bool", align 1
540   %casted_malloc = bitcast { i8**, { i8*, i8*, i32 }* }*
       %struct_space to i8*
541   store i8* %casted_malloc, i8** %"complex e1", align 8
542   %"return load" = load { i1, i8*, i8*, i32 }, { i1, i8*, i8*,
       i32 }* %"complex exec struct", align 8
543   store { i1, i8*, i8*, i32 } %"return load", { i1, i8*, i8*,
       i32 }* %"return malloc", align 8
544   ret { i1, i8*, i8*, i32 }* %"return malloc"
545 }
546
547 define { i1, i8*, i8*, i32 }* @create_execs(i8** %0) {
548 entry:
549   %malloccall = tail call i8* @malloc(i32 ptrtoint ({ i1, i8*,
       i8*, i32 }* (i8**)** getelementptr ({ i1, i8*, i8*, i32 }*
       (i8**)*, { i1, i8*, i8*, i32 }* (i8**)** null, i32 1) to
       i32))
550   %"function def" = bitcast i8* %malloccall to { i1, i8*, i8*,
       i32 }* (i8**)**
551   store { i1, i8*, i8*, i32 }* (i8**)* @create_compile_execs,
       { i1, i8*, i8*, i32 }* (i8**)** %"function def", align 8
552   %malloccall1 = tail call i8* @malloc(i32 ptrtoint ({ i1,
       i8*, i8*, i32 }* (i8**)** getelementptr ({ i1, i8*, i8*,
       i32 }* (i8**)*, { i1, i8*, i8*, i32 }* (i8**)** null, i32
       1) to i32))
```

```
553   %"function def2" = bitcast i8* %malloccall1 to { i1, i8*,
       i8*, i32 }* (i8**)**
554   store { i1, i8*, i8*, i32 }* (i8**)* @create_execs, { i1,
       i8*, i8*, i32 }* (i8**)** %"function def2", align 8
555   %malloccall3 = tail call i8* @malloc(i32 ptrtoint (i8** ({
       i1, i8*, i8*, i32 }*)** getelementptr (i8** ({ i1, i8*,
       i8*, i32 }*)*, i8** ({ i1, i8*, i8*, i32 }*)** null, i32 1)
       to i32))
556   %"function def4" = bitcast i8* %malloccall3 to i8** ({ i1,
       i8*, i8*, i32 }*)**
557   store i8** ({ i1, i8*, i8*, i32 }*)* @execute_exec, i8** ({
       i1, i8*, i8*, i32 }*)** %"function def4", align 8
558   %malloccall5 = tail call i8* @malloc(i32 ptrtoint ({ i8*,
       i8*, i32 }* (i8** ({ i1, i8*, i8*, i32 }*)*, { i8*, i8*,
       i32 }*)** getelementptr ({ i8*, i8*, i32 }* (i8** ({ i1,
       i8*, i8*, i32 }*)*, { i8*, i8*, i32 }*)*, { i8*, i8*, i32
       }* (i8** ({ i1, i8*, i8*, i32 }*)*, { i8*, i8*, i32 }*)**
       null, i32 1) to i32))
559   %"function def6" = bitcast i8* %malloccall5 to { i8*, i8*,
       i32 }* (i8** ({ i1, i8*, i8*, i32 }*)*, { i8*, i8*, i32
       }*)**
560   store { i8*, i8*, i32 }* (i8** ({ i1, i8*, i8*, i32 }*)*, {
       i8*, i8*, i32 }*)* @map_exec_to_string, { i8*, i8*, i32 }*
       (i8** ({ i1, i8*, i8*, i32 }*)*, { i8*, i8*, i32 }*)**
       %"function def6", align 8
561   %malloccall7 = tail call i8* @malloc(i32 ptrtoint ({ i8*,
       i8*, i32 }* ({ i1, i8*, i8*, i32 }* (i8**)*, { i8*, i8*,
       i32 }*)** getelementptr ({ i8*, i8*, i32 }* ({ i1, i8*,
       i8*, i32 }* (i8**)*, { i8*, i8*, i32 }*)*, { i8*, i8*, i32
       }* ({ i1, i8*, i8*, i32 }* (i8**)*, { i8*, i8*, i32 }*)**
       null, i32 1) to i32))
562   %"function def8" = bitcast i8* %malloccall7 to { i8*, i8*,
       i32 }* ({ i1, i8*, i8*, i32 }* (i8**)*, { i8*, i8*, i32
       }*)**
563   store { i8*, i8*, i32 }* ({ i1, i8*, i8*, i32 }* (i8**)*, {
       i8*, i8*, i32 }*)* @map_string_to_exec, { i8*, i8*, i32 }*
       ({ i1, i8*, i8*, i32 }* (i8**)*, { i8*, i8*, i32 }*)**
       %"function def8", align 8
564   %malloccall9 = tail call i8* @malloc(i32 ptrtoint (i8***
       getelementptr (i8**, i8*** null, i32 1) to i32))
565   %s = bitcast i8* %malloccall9 to i8***
566   store i8** %0, i8*** %s, align 8
567   %malloccall10 = tail call i8* @malloc(i32 ptrtoint ({ i1,
       i8*, i8*, i32 }* getelementptr ({ i1, i8*, i8*, i32 }, {
       i1, i8*, i8*, i32 }* null, i32 1) to i32))
568   %"return malloc" = bitcast i8* %malloccall10 to { i1, i8*,
       i8*, i32 }*
569   %malloccall11 = tail call i8* @malloc(i32 ptrtoint ({ i8**,
       { i8*, i8*, i32 }* }* getelementptr ({ i8**, { i8*, i8*,
       i32 }* }, { i8**, { i8*, i8*, i32 }* }* null, i32 1) to
       i32))
```

```
570   %struct_space = bitcast i8* %malloccall11 to { i8**, { i8*,
        i8*, i32 }* }*
571   %path_ptr = getelementptr inbounds { i8**, { i8*, i8*, i32
        }* }, { i8**, { i8*, i8*, i32 }* }* %struct_space, i32 0,
        i32 0
572   %s12 = load i8**, i8*** %s, align 8
573   store i8** %s12, i8*** %path_ptr, align 8
574   %args_ptr = getelementptr inbounds { i8**, { i8*, i8*, i32
        }* }, { i8**, { i8*, i8*, i32 }* }* %struct_space, i32 0,
        i32 1
575   store { i8*, i8*, i32 }* null, { i8*, i8*, i32 }**
        %args_ptr, align 8
576   %malloccall13 = tail call i8* @malloc(i32 ptrtoint ({ i1,
        i8*, i8*, i32 }* getelementptr ({ i1, i8*, i8*, i32 }, {
        i1, i8*, i8*, i32 }* null, i32 1) to i32))
577   %"complex exec struct" = bitcast i8* %malloccall13 to { i1,
        i8*, i8*, i32 }*
578   %"complex bool" = getelementptr inbounds { i1, i8*, i8*, i32
        }, { i1, i8*, i8*, i32 }* %"complex exec struct", i32 0,
        i32 0
579   %"complex e1" = getelementptr inbounds { i1, i8*, i8*, i32
        }, { i1, i8*, i8*, i32 }* %"complex exec struct", i32 0,
        i32 1
580   store i1 true, i1* %"complex bool", align 1
581   %casted_malloc = bitcast { i8**, { i8*, i8*, i32 }* }*
        %struct_space to i8*
582   store i8* %casted_malloc, i8** %"complex e1", align 8
583   %"return load" = load { i1, i8*, i8*, i32 }, { i1, i8*, i8*,
        i32 }* %"complex exec struct", align 8
584   store { i1, i8*, i8*, i32 } %"return load", { i1, i8*, i8*,
        i32 }* %"return malloc", align 8
585   ret { i1, i8*, i8*, i32 }* %"return malloc"
586 }
587
588 define i8** @execute_exec({ i1, i8*, i8*, i32 }* %0) {
589 entry:
590   %malloccall = tail call i8* @malloc(i32 ptrtoint ({ i1, i8*,
        i8*, i32 }* (i8**)** getelementptr ({ i1, i8*, i8*, i32 }*
        (i8**)*, { i1, i8*, i8*, i32 }* (i8**)** null, i32 1) to
        i32))
591   %"function def" = bitcast i8* %malloccall to { i1, i8*, i8*,
        i32 }* (i8**)**
592   store { i1, i8*, i8*, i32 }* (i8**)* @create_compile_execs,
        { i1, i8*, i8*, i32 }* (i8**)** %"function def", align 8
593   %malloccall1 = tail call i8* @malloc(i32 ptrtoint ({ i1,
        i8*, i8*, i32 }* (i8**)** getelementptr ({ i1, i8*, i8*,
        i32 }* (i8**)*, { i1, i8*, i8*, i32 }* (i8**)** null, i32
        1) to i32))
594   %"function def2" = bitcast i8* %malloccall1 to { i1, i8*,
        i8*, i32 }* (i8**)**
595   store { i1, i8*, i8*, i32 }* (i8**)* @create_execs, { i1,
        i8*, i8*, i32 }* (i8**)** %"function def2", align 8
```

```
596   %malloccall3 = tail call i8* @malloc(i32 ptrtoint (i8** ({
       i1, i8*, i8*, i32 }*)** getelementptr (i8** ({ i1, i8*,
       i8*, i32 }*)*, i8** ({ i1, i8*, i8*, i32 }*)** null, i32 1)
       to i32))
597   %"function def4" = bitcast i8* %malloccall3 to i8** ({ i1,
       i8*, i8*, i32 }*)**
598   store i8** ({ i1, i8*, i8*, i32 }*)* @execute_exec, i8** ({
       i1, i8*, i8*, i32 }*)** %"function def4", align 8
599   %malloccall5 = tail call i8* @malloc(i32 ptrtoint ({ i8*,
       i8*, i32 }* (i8** ({ i1, i8*, i8*, i32 }*)*, { i8*, i8*,
       i32 }*)** getelementptr ({ i8*, i8*, i32 }* (i8** ({ i1,
       i8*, i8*, i32 }*)*, { i8*, i8*, i32 }*)*, { i8*, i8*, i32
       }* (i8** ({ i1, i8*, i8*, i32 }*)*, { i8*, i8*, i32 }*)**
       null, i32 1) to i32))
600   %"function def6" = bitcast i8* %malloccall5 to { i8*, i8*,
       i32 }* (i8** ({ i1, i8*, i8*, i32 }*)*, { i8*, i8*, i32
       }*)**
601   store { i8*, i8*, i32 }* (i8** ({ i1, i8*, i8*, i32 }*)*, {
       i8*, i8*, i32 }*)* @map_exec_to_string, { i8*, i8*, i32 }*
       (i8** ({ i1, i8*, i8*, i32 }*)*, { i8*, i8*, i32 }*)**
       %"function def6", align 8
602   %malloccall7 = tail call i8* @malloc(i32 ptrtoint ({ i8*,
       i8*, i32 }* ({ i1, i8*, i8*, i32 }* (i8**)*, { i8*, i8*,
       i32 }*)** getelementptr ({ i8*, i8*, i32 }* ({ i1, i8*,
       i8*, i32 }* (i8**)*, { i8*, i8*, i32 }*)*, { i8*, i8*, i32
       }* ({ i1, i8*, i8*, i32 }* (i8**)*, { i8*, i8*, i32 }*)**
       null, i32 1) to i32))
603   %"function def8" = bitcast i8* %malloccall7 to { i8*, i8*,
       i32 }* ({ i1, i8*, i8*, i32 }* (i8**)*, { i8*, i8*, i32
       }*)**
604   store { i8*, i8*, i32 }* ({ i1, i8*, i8*, i32 }* (i8**)*, {
       i8*, i8*, i32 }*)* @map_string_to_exec, { i8*, i8*, i32 }*
       ({ i1, i8*, i8*, i32 }* (i8**)*, { i8*, i8*, i32 }*)**
       %"function def8", align 8
605   %malloccall9 = tail call i8* @malloc(i32 ptrtoint ({ i1,
       i8*, i8*, i32 }** getelementptr ({ i1, i8*, i8*, i32 }*, {
       i1, i8*, i8*, i32 }** null, i32 1) to i32))
606   %e = bitcast i8* %malloccall9 to { i1, i8*, i8*, i32 }**
607   store { i1, i8*, i8*, i32 }* %0, { i1, i8*, i8*, i32 }** %e,
       align 8
608   %malloccall10 = tail call i8* @malloc(i32 ptrtoint (i8**
       getelementptr (i8*, i8** null, i32 1) to i32))
609   %"return malloc" = bitcast i8* %malloccall10 to i8**
610   %e11 = load { i1, i8*, i8*, i32 }*, { i1, i8*, i8*, i32 }**
       %e, align 8
611   %complex_bool_ptr = getelementptr inbounds { i1, i8*, i8*,
       i32 }, { i1, i8*, i8*, i32 }* %e11, i32 0, i32 0
612   %complex_bool = load i1, i1* %complex_bool_ptr, align 1
613   %malloccall12 = tail call i8* @malloc(i32 ptrtoint (i8**
       getelementptr (i8*, i8** null, i32 1) to i32))
614   %return_str_ptr = bitcast i8* %malloccall12 to i8**
615   br i1 %complex_bool, label %then, label %else
```

```
616
617 "run merge":                                          ; preds =
      %else, %then
618   %"return load" = load i8*, i8** %return_str_ptr, align 8
619   store i8* %"return load", i8** %"return malloc", align 8
620   ret i8** %"return malloc"
621
622 then:                                                 ; preds =
      %entry
623   %exec_ptr = getelementptr inbounds { i1, i8*, i8*, i32 }, {
       i1, i8*, i8*, i32 }* %e11, i32 0, i32 1
624   %cast_run = bitcast i8** %exec_ptr to { i8**, { i8*, i8*,
       i32 }* }**
625   %exec = load { i8**, { i8*, i8*, i32 }* }*, { i8**, { i8*,
       i8*, i32 }* }** %cast_run, align 8
626   %dbl_path_ptr = getelementptr inbounds { i8**, { i8*, i8*,
       i32 }* }, { i8**, { i8*, i8*, i32 }* }* %exec, i32 0, i32 0
627   %path_ptr = load i8**, i8*** %dbl_path_ptr, align 8
628   %path = load i8*, i8** %path_ptr, align 8
629   %args_ptr = getelementptr inbounds { i8**, { i8*, i8*, i32
       }* }, { i8**, { i8*, i8*, i32 }* }* %exec, i32 0, i32 1
630   %args = load { i8*, i8*, i32 }*, { i8*, i8*, i32 }**
       %args_ptr, align 8
631   %execvp = call i8* (i8*, { i8*, i8*, i32 }*, ...)
       @execvp_helper(i8* %path, { i8*, i8*, i32 }* %args)
632   store i8* %execvp, i8** %return_str_ptr, align 8
633   br label %"run merge"
634
635 else:                                                 ; preds =
      %entry
636   %recurse_exec = call i8* ({ i1, i8*, i8*, i32 }*, ...)
       @recurse_exec({ i1, i8*, i8*, i32 }* %e11)
637   store i8* %recurse_exec, i8** %return_str_ptr, align 8
638   br label %"run merge"
639 }
640
641 declare noalias i8* @malloc(i32)
```

6.4 Example Program 2: misc-ops.bs

BlueShell Code

```
1 // test combinations of executable operators
2 // expected results are printed when this program is run
3
4 exec e1 = <"cat" withargs ["sample-files/test_file.txt"]>;
5 exec e2 = <"cat" withargs ["sample-files/test_file2.txt"]>;
6
7 exec e3 = <"grep" withargs ["-a", "zip"]>;
8 exec e4 = <"grep" withargs ["-a", "BlueShell"]>;
9
```

```
exec e5 = (e1 | e3) + (e2 | e4);
./<"echo" withargs ["TEST 1 OUTPUT: ZIPS FROM FIRST FILE AND
    BLUESHELLS FROM SECOND"]>;
string s = ./e5;



./<"echo" withargs ["_______________"]>;
exec e5 = (e1 | e3) * (e2 | e4);
./<"echo" withargs ["\n\nTEST 2 OUTPUT: BLUESHELLS FROM
    SECOND"]>;
string s = ./e5;




./<"echo" withargs ["_______________"]>;
exec e5 = (e1 + e2) | e3;
./<"echo" withargs ["\n\nTEST 3 OUTPUT: ZIPS FROM BOTH"]>;
string s = ./e5;




./<"echo" withargs ["_______________"]>;
exec e5 = (e1 * e2) | e4;
./<"echo" withargs ["\n\nTEST 4 OUTPUT: BLUESHELLS FROM
    SECOND"]>;
string s = ./e5;




./<"echo" withargs ["_______________"]>;
exec e6 = <"echo" withargs ["hello world"]>;
exec e5 = (e2 * e1) | (e3 + (e6 + e6)) | <"grep" withargs
    ["-a", "hello"]> ;
./<"echo" withargs ["\n\nTEST 5 OUTPUT: ZIPS FROM FIRST PLUS
    HELLO WORLDS"]>;
string s = ./e5;




exec e5 = (e2 * e1) | (e3 + (e6 + e6) | <"grep" withargs
    ["-a", "hello"]>);
./<"echo" withargs ["\n\nTEST 6 OUTPUT: GETS OUTPUT OF FIRST
    FILE, GREPS FOR ZIP WHICH GETS CONCATENATED WITH HELLO
    WORLD AND WE GREP FOR HELLO WORLD"]>;
string s = ./e5;
```

LLVM Code

```
; ModuleID = 'BlueShell'
source_filename = "BlueShell"

@string = private unnamed_addr constant [4 x i8] c"cat\00", align 1
@string.1 = private unnamed_addr constant [27 x i8] c"sample-files/test_file.txt\00", align 1
@string.2 = private unnamed_addr constant [4 x i8] c"cat\00", align 1
@string.3 = private unnamed_addr constant [28 x i8] c"sample-files/test_file2.txt\00", align 1
@string.4 = private unnamed_addr constant [5 x i8] c"grep\00", align 1
@string.5 = private unnamed_addr constant [3 x i8] c"-a\00", align 1
@string.6 = private unnamed_addr constant [4 x i8] c"zip\00", align 1
@string.7 = private unnamed_addr constant [5 x i8] c"grep\00", align 1
@string.8 = private unnamed_addr constant [3 x i8] c"-a\00", align 1
@string.9 = private unnamed_addr constant [10 x i8] c"BlueShell\00", align 1
@string.10 = private unnamed_addr constant [5 x i8] c"echo\00", align 1
@string.11 = private unnamed_addr constant [63 x i8] c"TEST 1 OUTPUT: ZIPS FROM FIRST FILE AND BLUESHELLS FROM SECOND\00", align 1
@string.12 = private unnamed_addr constant [5 x i8] c"echo\00", align 1
@string.13 = private unnamed_addr constant [15 x i8] c"--------------\00", align 1
@string.14 = private unnamed_addr constant [5 x i8] c"echo\00", align 1
@string.15 = private unnamed_addr constant [40 x i8] c"\0A\0ATEST 2 OUTPUT: BLUESHELLS FROM SECOND\00", align 1
@string.16 = private unnamed_addr constant [5 x i8] c"echo\00", align 1
@string.17 = private unnamed_addr constant [15 x i8] c"--------------\00", align 1
@string.18 = private unnamed_addr constant [5 x i8] c"echo\00", align 1
@string.19 = private unnamed_addr constant [32 x i8] c"\0A\0ATEST 3 OUTPUT: ZIPS FROM BOTH\00", align 1
@string.20 = private unnamed_addr constant [5 x i8] c"echo\00", align 1
@string.21 = private unnamed_addr constant [15 x i8] c"--------------\00", align 1
@string.22 = private unnamed_addr constant [5 x i8] c"echo\00", align 1
```

```
@string.23 = private unnamed_addr constant [40 x i8] c"\0A\0ATEST 4 OUTPUT: BLUESHELLS FROM SECOND\00", align 1
@string.24 = private unnamed_addr constant [5 x i8] c"echo\00", align 1
@string.25 = private unnamed_addr constant [15 x i8] c"_____________\00", align 1
@string.26 = private unnamed_addr constant [5 x i8] c"echo\00", align 1
@string.27 = private unnamed_addr constant [12 x i8] c"hello world\00", align 1
@string.28 = private unnamed_addr constant [5 x i8] c"grep\00", align 1
@string.29 = private unnamed_addr constant [3 x i8] c"-a\00", align 1
@string.30 = private unnamed_addr constant [6 x i8] c"hello\00", align 1
@string.31 = private unnamed_addr constant [5 x i8] c"echo\00", align 1
@string.32 = private unnamed_addr constant [51 x i8] c"\0A\0ATEST 5 OUTPUT: ZIPS FROM FIRST PLUS HELLO WORLDS\00", align 1
@string.33 = private unnamed_addr constant [5 x i8] c"grep\00", align 1
@string.34 = private unnamed_addr constant [3 x i8] c"-a\00", align 1
@string.35 = private unnamed_addr constant [6 x i8] c"hello\00", align 1
@string.36 = private unnamed_addr constant [5 x i8] c"echo\00", align 1
@string.37 = private unnamed_addr constant [127 x i8] c"\0A\0ATEST 6 OUTPUT: GETS OUTPUT OF FIRST FILE, GREPS FOR ZIP WHICH GETS CONCATENATED WITH HELLO WORLD AND WE GREP FOR HELLO WORLD\00", align 1

declare i8* @execvp_helper(i8*, { i8*, i8*, i32 }*, ...)

declare i8* @recurse_exec({ i1, i8*, i8*, i32 }*, ...)

define i32 @main() {
entry:
  %malloccall = tail call i8* @malloc(i32 ptrtoint ({ i8**, { i8*, i8*, i32 }* }* getelementptr ({ i8**, { i8*, i8*, i32 }* }, { i8**, { i8*, i8*, i32 }* }* null, i32 1) to i32))
  %struct_space = bitcast i8* %malloccall to { i8**, { i8*, i8*, i32 }* }*
  %path_ptr = getelementptr inbounds { i8**, { i8*, i8*, i32 }* }, { i8**, { i8*, i8*, i32 }* }* %struct_space, i32 0, i32 0
  %malloccall1 = tail call i8* @malloc(i32 ptrtoint (i8** getelementptr (i8*, i8** null, i32 1) to i32))
  %double_string_ptr = bitcast i8* %malloccall1 to i8**
```

```
54  store i8* getelementptr inbounds ([4 x i8], [4 x i8]*
     @string, i32 0, i32 0), i8** %double_string_ptr, align 8
55  store i8** %double_string_ptr, i8*** %path_ptr, align 8
56  %args_ptr = getelementptr inbounds { i8**, { i8*, i8*, i32
     }* }, { i8**, { i8*, i8*, i32 }* }* %struct_space, i32 0,
     i32 1
57  %malloccall2 = tail call i8* @malloc(i32 ptrtoint (i8**
     getelementptr (i8*, i8** null, i32 1) to i32))
58  %double_string_ptr3 = bitcast i8* %malloccall2 to i8**
59  store i8* getelementptr inbounds ([27 x i8], [27 x i8]*
     @string.1, i32 0, i32 0), i8** %double_string_ptr3, align 8
60  %malloccall4 = tail call i8* @malloc(i32 ptrtoint (i8***
     getelementptr (i8**, i8*** null, i32 1) to i32))
61  %value_ptr = bitcast i8* %malloccall4 to i8***
62  store i8** %double_string_ptr3, i8*** %value_ptr, align 8
63  %malloccall5 = tail call i8* @malloc(i32 ptrtoint ({ i8*,
     i8*, i32 }* getelementptr ({ i8*, i8*, i32 }, { i8*, i8*,
     i32 }* null, i32 1) to i32))
64  %list_node = bitcast i8* %malloccall5 to { i8*, i8*, i32 }*
65  %struct_val_ptr = getelementptr inbounds { i8*, i8*, i32 },
     { i8*, i8*, i32 }* %list_node, i32 0, i32 0
66  %struct_ptr_ptr = getelementptr inbounds { i8*, i8*, i32 },
     { i8*, i8*, i32 }* %list_node, i32 0, i32 1
67  %struct_ty_ptr = getelementptr inbounds { i8*, i8*, i32 }, {
     i8*, i8*, i32 }* %list_node, i32 0, i32 2
68  %casted_ptr_ptr = bitcast i8** %struct_ptr_ptr to { i8*,
     i8*, i32 }**
69  store { i8*, i8*, i32 }* null, { i8*, i8*, i32 }**
     %casted_ptr_ptr, align 8
70  %casted_val = bitcast i8*** %value_ptr to i8*
71  store i8* %casted_val, i8** %struct_val_ptr, align 8
72  store i32 4, i32* %struct_ty_ptr, align 4
73  store { i8*, i8*, i32 }* %list_node, { i8*, i8*, i32 }**
     %args_ptr, align 8
74  %malloccall6 = tail call i8* @malloc(i32 ptrtoint ({ i1,
     i8*, i8*, i32 }* getelementptr ({ i1, i8*, i8*, i32 }, {
     i1, i8*, i8*, i32 }* null, i32 1) to i32))
75  %"complex exec struct" = bitcast i8* %malloccall6 to { i1,
     i8*, i8*, i32 }*
76  %"complex bool" = getelementptr inbounds { i1, i8*, i8*, i32
     }, { i1, i8*, i8*, i32 }* %"complex exec struct", i32 0,
     i32 0
77  %"complex e1" = getelementptr inbounds { i1, i8*, i8*, i32
     }, { i1, i8*, i8*, i32 }* %"complex exec struct", i32 0,
     i32 1
78  store i1 true, i1* %"complex bool", align 1
79  %casted_malloc = bitcast { i8**, { i8*, i8*, i32 }* }*
     %struct_space to i8*
80  store i8* %casted_malloc, i8** %"complex e1", align 8
81  %malloccall7 = tail call i8* @malloc(i32 ptrtoint ({ i1,
     i8*, i8*, i32 }** getelementptr ({ i1, i8*, i8*, i32 }*, {
     i1, i8*, i8*, i32 }** null, i32 1) to i32))
```

```
  %"variable ptr" = bitcast i8* %malloccall7 to { i1, i8*,
   i8*, i32 }**
  store { i1, i8*, i8*, i32 }* %"complex exec struct", { i1,
   i8*, i8*, i32 }** %"variable ptr", align 8
  %malloccall8 = tail call i8* @malloc(i32 ptrtoint ({ i8**, {
   i8*, i8*, i32 }* }* getelementptr ({ i8**, { i8*, i8*, i32
   }* }, { i8**, { i8*, i8*, i32 }* }* null, i32 1) to i32))
  %struct_space9 = bitcast i8* %malloccall8 to { i8**, { i8*,
   i8*, i32 }* }*
  %path_ptr10 = getelementptr inbounds { i8**, { i8*, i8*, i32
   }* }, { i8**, { i8*, i8*, i32 }* }* %struct_space9, i32 0,
   i32 0
  %malloccall11 = tail call i8* @malloc(i32 ptrtoint (i8**
   getelementptr (i8*, i8** null, i32 1) to i32))
  %double_string_ptr12 = bitcast i8* %malloccall11 to i8**
  store i8* getelementptr inbounds ([4 x i8], [4 x i8]*
   @string.2, i32 0, i32 0), i8** %double_string_ptr12, align 8
  store i8** %double_string_ptr12, i8*** %path_ptr10, align 8
  %args_ptr13 = getelementptr inbounds { i8**, { i8*, i8*, i32
   }* }, { i8**, { i8*, i8*, i32 }* }* %struct_space9, i32 0,
   i32 1
  %malloccall14 = tail call i8* @malloc(i32 ptrtoint (i8**
   getelementptr (i8*, i8** null, i32 1) to i32))
  %double_string_ptr15 = bitcast i8* %malloccall14 to i8**
  store i8* getelementptr inbounds ([28 x i8], [28 x i8]*
   @string.3, i32 0, i32 0), i8** %double_string_ptr15, align 8
  %malloccall16 = tail call i8* @malloc(i32 ptrtoint (i8***
   getelementptr (i8**, i8*** null, i32 1) to i32))
  %value_ptr17 = bitcast i8* %malloccall16 to i8***
  store i8** %double_string_ptr15, i8*** %value_ptr17, align 8
  %malloccall18 = tail call i8* @malloc(i32 ptrtoint ({ i8*,
   i8*, i32 }* getelementptr ({ i8*, i8*, i32 }, { i8*, i8*,
   i32 }* null, i32 1) to i32))
  %list_node19 = bitcast i8* %malloccall18 to { i8*, i8*, i32
   }*
  %struct_val_ptr20 = getelementptr inbounds { i8*, i8*, i32
   }, { i8*, i8*, i32 }* %list_node19, i32 0, i32 0
  %struct_ptr_ptr21 = getelementptr inbounds { i8*, i8*, i32
   }, { i8*, i8*, i32 }* %list_node19, i32 0, i32 1
  %struct_ty_ptr22 = getelementptr inbounds { i8*, i8*, i32 },
   { i8*, i8*, i32 }* %list_node19, i32 0, i32 2
  %casted_ptr_ptr23 = bitcast i8** %struct_ptr_ptr21 to { i8*,
   i8*, i32 }**
  store { i8*, i8*, i32 }* null, { i8*, i8*, i32 }**
   %casted_ptr_ptr23, align 8
  %casted_val24 = bitcast i8*** %value_ptr17 to i8*
  store i8* %casted_val24, i8** %struct_val_ptr20, align 8
  store i32 4, i32* %struct_ty_ptr22, align 4
  store { i8*, i8*, i32 }* %list_node19, { i8*, i8*, i32 }**
   %args_ptr13, align 8
  %malloccall25 = tail call i8* @malloc(i32 ptrtoint ({ i1,
   i8*, i8*, i32 }* getelementptr ({ i1, i8*, i8*, i32 }, {
```

```
    i1, i8*, i8*, i32 }* null, i32 1) to i32))
110 %"complex exec struct26" = bitcast i8* %malloccall25 to {
    i1, i8*, i8*, i32 }*
111 %"complex bool27" = getelementptr inbounds { i1, i8*, i8*,
    i32 }, { i1, i8*, i8*, i32 }* %"complex exec struct26", i32
    0, i32 0
112 %"complex e128" = getelementptr inbounds { i1, i8*, i8*, i32
    }, { i1, i8*, i8*, i32 }* %"complex exec struct26", i32 0,
    i32 1
113 store i1 true, i1* %"complex bool27", align 1
114 %casted_malloc29 = bitcast { i8**, { i8*, i8*, i32 }* }*
    %struct_space9 to i8*
115 store i8* %casted_malloc29, i8** %"complex e128", align 8
116 %malloccall30 = tail call i8* @malloc(i32 ptrtoint ({ i1,
    i8*, i8*, i32 }** getelementptr ({ i1, i8*, i8*, i32 }*, {
    i1, i8*, i8*, i32 }** null, i32 1) to i32))
117 %"variable ptr31" = bitcast i8* %malloccall30 to { i1, i8*,
    i8*, i32 }**
118 store { i1, i8*, i8*, i32 }* %"complex exec struct26", { i1,
    i8*, i8*, i32 }** %"variable ptr31", align 8
119 %malloccall32 = tail call i8* @malloc(i32 ptrtoint ({ i8**,
    { i8*, i8*, i32 }* }* getelementptr ({ i8**, { i8*, i8*,
    i32 }* }, { i8**, { i8*, i8*, i32 }* }* null, i32 1) to
    i32))
120 %struct_space33 = bitcast i8* %malloccall32 to { i8**, {
    i8*, i8*, i32 }* }*
121 %path_ptr34 = getelementptr inbounds { i8**, { i8*, i8*, i32
    }* }, { i8**, { i8*, i8*, i32 }* }* %struct_space33, i32 0,
    i32 0
122 %malloccall35 = tail call i8* @malloc(i32 ptrtoint (i8**
    getelementptr (i8*, i8** null, i32 1) to i32))
123 %double_string_ptr36 = bitcast i8* %malloccall35 to i8**
124 store i8* getelementptr inbounds ([5 x i8], [5 x i8]*
    @string.4, i32 0, i32 0), i8** %double_string_ptr36, align 8
125 store i8** %double_string_ptr36, i8*** %path_ptr34, align 8
126 %args_ptr37 = getelementptr inbounds { i8**, { i8*, i8*, i32
    }* }, { i8**, { i8*, i8*, i32 }* }* %struct_space33, i32 0,
    i32 1
127 %malloccall38 = tail call i8* @malloc(i32 ptrtoint (i8**
    getelementptr (i8*, i8** null, i32 1) to i32))
128 %double_string_ptr39 = bitcast i8* %malloccall38 to i8**
129 store i8* getelementptr inbounds ([3 x i8], [3 x i8]*
    @string.5, i32 0, i32 0), i8** %double_string_ptr39, align 8
130 %malloccall40 = tail call i8* @malloc(i32 ptrtoint (i8***
    getelementptr (i8**, i8*** null, i32 1) to i32))
131 %value_ptr41 = bitcast i8* %malloccall40 to i8***
132 store i8** %double_string_ptr39, i8*** %value_ptr41, align 8
133 %malloccall42 = tail call i8* @malloc(i32 ptrtoint ({ i8*,
    i8*, i32 }* getelementptr ({ i8*, i8*, i32 }, { i8*, i8*,
    i32 }* null, i32 1) to i32))
134 %list_node43 = bitcast i8* %malloccall42 to { i8*, i8*, i32
    }*
```

```
  %struct_val_ptr44 = getelementptr inbounds { i8*, i8*, i32 }, { i8*, i8*, i32 }* %list_node43, i32 0, i32 0
  %struct_ptr_ptr45 = getelementptr inbounds { i8*, i8*, i32 }, { i8*, i8*, i32 }* %list_node43, i32 0, i32 1
  %struct_ty_ptr46 = getelementptr inbounds { i8*, i8*, i32 }, { i8*, i8*, i32 }* %list_node43, i32 0, i32 2
  %malloccall47 = tail call i8* @malloc(i32 ptrtoint (i8** getelementptr (i8*, i8** null, i32 1) to i32))
  %double_string_ptr48 = bitcast i8* %malloccall47 to i8**
  store i8* getelementptr inbounds ([4 x i8], [4 x i8]* @string.6, i32 0, i32 0), i8** %double_string_ptr48, align 8
  %malloccall49 = tail call i8* @malloc(i32 ptrtoint (i8*** getelementptr (i8**, i8*** null, i32 1) to i32))
  %value_ptr50 = bitcast i8* %malloccall49 to i8***
  store i8** %double_string_ptr48, i8*** %value_ptr50, align 8
  %malloccall51 = tail call i8* @malloc(i32 ptrtoint ({ i8*, i8*, i32 }* getelementptr ({ i8*, i8*, i32 }, { i8*, i8*, i32 }* null, i32 1) to i32))
  %list_node52 = bitcast i8* %malloccall51 to { i8*, i8*, i32 }*
  %struct_val_ptr53 = getelementptr inbounds { i8*, i8*, i32 }, { i8*, i8*, i32 }* %list_node52, i32 0, i32 0
  %struct_ptr_ptr54 = getelementptr inbounds { i8*, i8*, i32 }, { i8*, i8*, i32 }* %list_node52, i32 0, i32 1
  %struct_ty_ptr55 = getelementptr inbounds { i8*, i8*, i32 }, { i8*, i8*, i32 }* %list_node52, i32 0, i32 2
  %casted_ptr_ptr56 = bitcast i8** %struct_ptr_ptr54 to { i8*, i8*, i32 }**
  store { i8*, i8*, i32 }* null, { i8*, i8*, i32 }** %casted_ptr_ptr56, align 8
  %casted_val57 = bitcast i8*** %value_ptr50 to i8*
  store i8* %casted_val57, i8** %struct_val_ptr53, align 8
  store i32 4, i32* %struct_ty_ptr55, align 4
  %casted_ptr_ptr58 = bitcast i8** %struct_ptr_ptr45 to { i8*, i8*, i32 }**
  store { i8*, i8*, i32 }* %list_node52, { i8*, i8*, i32 }** %casted_ptr_ptr58, align 8
  %casted_val59 = bitcast i8*** %value_ptr41 to i8*
  store i8* %casted_val59, i8** %struct_val_ptr44, align 8
  store i32 4, i32* %struct_ty_ptr46, align 4
  store { i8*, i8*, i32 }* %list_node43, { i8*, i8*, i32 }** %args_ptr37, align 8
  %malloccall60 = tail call i8* @malloc(i32 ptrtoint ({ i1, i8*, i8*, i32 }* getelementptr ({ i1, i8*, i8*, i32 }, { i1, i8*, i8*, i32 }* null, i32 1) to i32))
  %"complex exec struct61" = bitcast i8* %malloccall60 to { i1, i8*, i8*, i32 }*
  %"complex bool62" = getelementptr inbounds { i1, i8*, i8*, i32 }, { i1, i8*, i8*, i32 }* %"complex exec struct61", i32 0, i32 0
  %"complex e163" = getelementptr inbounds { i1, i8*, i8*, i32 }, { i1, i8*, i8*, i32 }* %"complex exec struct61", i32 0,
```

```
  i32 1
  store i1 true, i1* %"complex bool62", align 1
  %casted_malloc64 = bitcast { i8**, { i8*, i8*, i32 }* }*
   %struct_space33 to i8*
  store i8* %casted_malloc64, i8** %"complex e163", align 8
  %malloccall65 = tail call i8* @malloc(i32 ptrtoint ({ i1,
   i8*, i8*, i32 }** getelementptr ({ i1, i8*, i8*, i32 }*, {
   i1, i8*, i8*, i32 }** null, i32 1) to i32))
  %"variable ptr66" = bitcast i8* %malloccall65 to { i1, i8*,
   i8*, i32 }**
  store { i1, i8*, i8*, i32 }* %"complex exec struct61", { i1,
   i8*, i8*, i32 }** %"variable ptr66", align 8
  %malloccall67 = tail call i8* @malloc(i32 ptrtoint ({ i8**,
   { i8*, i8*, i32 }* }* getelementptr ({ i8**, { i8*, i8*,
   i32 }* }, { i8**, { i8*, i8*, i32 }* }* null, i32 1) to
   i32))
  %struct_space68 = bitcast i8* %malloccall67 to { i8**, {
   i8*, i8*, i32 }* }*
  %path_ptr69 = getelementptr inbounds { i8**, { i8*, i8*, i32
   }* }, { i8**, { i8*, i8*, i32 }* }* %struct_space68, i32 0,
   i32 0
  %malloccall70 = tail call i8* @malloc(i32 ptrtoint (i8**
   getelementptr (i8*, i8** null, i32 1) to i32))
  %double_string_ptr71 = bitcast i8* %malloccall70 to i8**
  store i8* getelementptr inbounds ([5 x i8], [5 x i8]*
   @string.7, i32 0, i32 0), i8** %double_string_ptr71, align 8
  store i8** %double_string_ptr71, i8*** %path_ptr69, align 8
  %args_ptr72 = getelementptr inbounds { i8**, { i8*, i8*, i32
   }* }, { i8**, { i8*, i8*, i32 }* }* %struct_space68, i32 0,
   i32 1
  %malloccall73 = tail call i8* @malloc(i32 ptrtoint (i8**
   getelementptr (i8*, i8** null, i32 1) to i32))
  %double_string_ptr74 = bitcast i8* %malloccall73 to i8**
  store i8* getelementptr inbounds ([3 x i8], [3 x i8]*
   @string.8, i32 0, i32 0), i8** %double_string_ptr74, align 8
  %malloccall75 = tail call i8* @malloc(i32 ptrtoint (i8***
   getelementptr (i8**, i8*** null, i32 1) to i32))
  %value_ptr76 = bitcast i8* %malloccall75 to i8***
  store i8** %double_string_ptr74, i8*** %value_ptr76, align 8
  %malloccall77 = tail call i8* @malloc(i32 ptrtoint ({ i8*,
   i8*, i32 }* getelementptr ({ i8*, i8*, i32 }, { i8*, i8*,
   i32 }* null, i32 1) to i32))
  %list_node78 = bitcast i8* %malloccall77 to { i8*, i8*, i32
   }*
  %struct_val_ptr79 = getelementptr inbounds { i8*, i8*, i32
   }, { i8*, i8*, i32 }* %list_node78, i32 0, i32 0
  %struct_ptr_ptr80 = getelementptr inbounds { i8*, i8*, i32
   }, { i8*, i8*, i32 }* %list_node78, i32 0, i32 1
  %struct_ty_ptr81 = getelementptr inbounds { i8*, i8*, i32 },
   { i8*, i8*, i32 }* %list_node78, i32 0, i32 2
  %malloccall82 = tail call i8* @malloc(i32 ptrtoint (i8**
   getelementptr (i8*, i8** null, i32 1) to i32))
```

```
190   %double_string_ptr83 = bitcast i8* %malloccall82 to i8**
191   store i8* getelementptr inbounds ([10 x i8], [10 x i8]*
       @string.9, i32 0, i32 0), i8** %double_string_ptr83, align 8
192   %malloccall84 = tail call i8* @malloc(i32 ptrtoint (i8***
       getelementptr (i8**, i8*** null, i32 1) to i32))
193   %value_ptr85 = bitcast i8* %malloccall84 to i8***
194   store i8** %double_string_ptr83, i8*** %value_ptr85, align 8
195   %malloccall86 = tail call i8* @malloc(i32 ptrtoint ({ i8*,
       i8*, i32 }* getelementptr ({ i8*, i8*, i32 }, { i8*, i8*,
       i32 }* null, i32 1) to i32))
196   %list_node87 = bitcast i8* %malloccall86 to { i8*, i8*, i32
       }*
197   %struct_val_ptr88 = getelementptr inbounds { i8*, i8*, i32
       }, { i8*, i8*, i32 }* %list_node87, i32 0, i32 0
198   %struct_ptr_ptr89 = getelementptr inbounds { i8*, i8*, i32
       }, { i8*, i8*, i32 }* %list_node87, i32 0, i32 1
199   %struct_ty_ptr90 = getelementptr inbounds { i8*, i8*, i32 },
       { i8*, i8*, i32 }* %list_node87, i32 0, i32 2
200   %casted_ptr_ptr91 = bitcast i8** %struct_ptr_ptr89 to { i8*,
       i8*, i32 }**
201   store { i8*, i8*, i32 }* null, { i8*, i8*, i32 }**
       %casted_ptr_ptr91, align 8
202   %casted_val92 = bitcast i8*** %value_ptr85 to i8*
203   store i8* %casted_val92, i8** %struct_val_ptr88, align 8
204   store i32 4, i32* %struct_ty_ptr90, align 4
205   %casted_ptr_ptr93 = bitcast i8** %struct_ptr_ptr80 to { i8*,
       i8*, i32 }**
206   store { i8*, i8*, i32 }* %list_node87, { i8*, i8*, i32 }**
       %casted_ptr_ptr93, align 8
207   %casted_val94 = bitcast i8*** %value_ptr76 to i8*
208   store i8* %casted_val94, i8** %struct_val_ptr79, align 8
209   store i32 4, i32* %struct_ty_ptr81, align 4
210   store { i8*, i8*, i32 }* %list_node78, { i8*, i8*, i32 }**
       %args_ptr72, align 8
211   %malloccall95 = tail call i8* @malloc(i32 ptrtoint ({ i1,
       i8*, i8*, i32 }* getelementptr ({ i1, i8*, i8*, i32 }, {
       i1, i8*, i8*, i32 }* null, i32 1) to i32))
212   %"complex exec struct96" = bitcast i8* %malloccall95 to {
       i1, i8*, i8*, i32 }*
213   %"complex bool97" = getelementptr inbounds { i1, i8*, i8*,
       i32 }, { i1, i8*, i8*, i32 }* %"complex exec struct96", i32
       0, i32 0
214   %"complex e198" = getelementptr inbounds { i1, i8*, i8*, i32
       }, { i1, i8*, i8*, i32 }* %"complex exec struct96", i32 0,
       i32 1
215   store i1 true, i1* %"complex bool97", align 1
216   %casted_malloc99 = bitcast { i8**, { i8*, i8*, i32 }* }*
       %struct_space68 to i8*
217   store i8* %casted_malloc99, i8** %"complex e198", align 8
218   %malloccall100 = tail call i8* @malloc(i32 ptrtoint ({ i1,
       i8*, i8*, i32 }** getelementptr ({ i1, i8*, i8*, i32 }*, {
       i1, i8*, i8*, i32 }** null, i32 1) to i32))
```

```
219   %"variable ptr101" = bitcast i8* %malloccall100 to { i1,
       i8*, i8*, i32 }**
220   store { i1, i8*, i8*, i32 }* %"complex exec struct96", { i1,
       i8*, i8*, i32 }** %"variable ptr101", align 8
221   %e4 = load { i1, i8*, i8*, i32 }*, { i1, i8*, i8*, i32 }**
       %"variable ptr101", align 8
222   %e2 = load { i1, i8*, i8*, i32 }*, { i1, i8*, i8*, i32 }**
       %"variable ptr31", align 8
223   %malloccall102 = tail call i8* @malloc(i32 ptrtoint ({ i1,
       i8*, i8*, i32 }* getelementptr ({ i1, i8*, i8*, i32 }, {
       i1, i8*, i8*, i32 }* null, i32 1) to i32))
224   %"complex exec struct103" = bitcast i8* %malloccall102 to {
       i1, i8*, i8*, i32 }*
225   %"complex bool104" = getelementptr inbounds { i1, i8*, i8*,
       i32 }, { i1, i8*, i8*, i32 }* %"complex exec struct103",
       i32 0, i32 0
226   %"complex e1105" = getelementptr inbounds { i1, i8*, i8*,
       i32 }, { i1, i8*, i8*, i32 }* %"complex exec struct103",
       i32 0, i32 1
227   %"complex e2" = getelementptr inbounds { i1, i8*, i8*, i32
       }, { i1, i8*, i8*, i32 }* %"complex exec struct103", i32 0,
       i32 2
228   %"complex op" = getelementptr inbounds { i1, i8*, i8*, i32
       }, { i1, i8*, i8*, i32 }* %"complex exec struct103", i32 0,
       i32 3
229   store i1 false, i1* %"complex bool104", align 1
230   %casted_e1 = bitcast { i1, i8*, i8*, i32 }* %e2 to i8*
231   store i8* %casted_e1, i8** %"complex e1105", align 8
232   %casted_e2 = bitcast { i1, i8*, i8*, i32 }* %e4 to i8*
233   store i8* %casted_e2, i8** %"complex e2", align 8
234   store i32 2, i32* %"complex op", align 4
235   %e3 = load { i1, i8*, i8*, i32 }*, { i1, i8*, i8*, i32 }**
       %"variable ptr66", align 8
236   %e1 = load { i1, i8*, i8*, i32 }*, { i1, i8*, i8*, i32 }**
       %"variable ptr", align 8
237   %malloccall106 = tail call i8* @malloc(i32 ptrtoint ({ i1,
       i8*, i8*, i32 }* getelementptr ({ i1, i8*, i8*, i32 }, {
       i1, i8*, i8*, i32 }* null, i32 1) to i32))
238   %"complex exec struct107" = bitcast i8* %malloccall106 to {
       i1, i8*, i8*, i32 }*
239   %"complex bool108" = getelementptr inbounds { i1, i8*, i8*,
       i32 }, { i1, i8*, i8*, i32 }* %"complex exec struct107",
       i32 0, i32 0
240   %"complex e1109" = getelementptr inbounds { i1, i8*, i8*,
       i32 }, { i1, i8*, i8*, i32 }* %"complex exec struct107",
       i32 0, i32 1
241   %"complex e2110" = getelementptr inbounds { i1, i8*, i8*,
       i32 }, { i1, i8*, i8*, i32 }* %"complex exec struct107",
       i32 0, i32 2
242   %"complex op111" = getelementptr inbounds { i1, i8*, i8*,
       i32 }, { i1, i8*, i8*, i32 }* %"complex exec struct107",
       i32 0, i32 3
```

```
243   store i1 false, i1* %"complex bool108", align 1
244   %casted_e1112 = bitcast { i1, i8*, i8*, i32 }* %e1 to i8*
245   store i8* %casted_e1112, i8** %"complex e1109", align 8
246   %casted_e2113 = bitcast { i1, i8*, i8*, i32 }* %e3 to i8*
247   store i8* %casted_e2113, i8** %"complex e2110", align 8
248   store i32 2, i32* %"complex op111", align 4
249   %malloccall114 = tail call i8* @malloc(i32 ptrtoint ({ i1,
       i8*, i8*, i32 }* getelementptr ({ i1, i8*, i8*, i32 }, {
       i1, i8*, i8*, i32 }* null, i32 1) to i32))
250   %"complex exec struct115" = bitcast i8* %malloccall114 to {
       i1, i8*, i8*, i32 }*
251   %"complex bool116" = getelementptr inbounds { i1, i8*, i8*,
       i32 }, { i1, i8*, i8*, i32 }* %"complex exec struct115",
       i32 0, i32 0
252   %"complex e1117" = getelementptr inbounds { i1, i8*, i8*,
       i32 }, { i1, i8*, i8*, i32 }* %"complex exec struct115",
       i32 0, i32 1
253   %"complex e2118" = getelementptr inbounds { i1, i8*, i8*,
       i32 }, { i1, i8*, i8*, i32 }* %"complex exec struct115",
       i32 0, i32 2
254   %"complex op119" = getelementptr inbounds { i1, i8*, i8*,
       i32 }, { i1, i8*, i8*, i32 }* %"complex exec struct115",
       i32 0, i32 3
255   store i1 false, i1* %"complex bool116", align 1
256   %casted_e1120 = bitcast { i1, i8*, i8*, i32 }* %"complex
       exec struct107" to i8*
257   store i8* %casted_e1120, i8** %"complex e1117", align 8
258   %casted_e2121 = bitcast { i1, i8*, i8*, i32 }* %"complex
       exec struct103" to i8*
259   store i8* %casted_e2121, i8** %"complex e2118", align 8
260   store i32 0, i32* %"complex op119", align 4
261   %malloccall122 = tail call i8* @malloc(i32 ptrtoint ({ i1,
       i8*, i8*, i32 }** getelementptr ({ i1, i8*, i8*, i32 }*, {
       i1, i8*, i8*, i32 }** null, i32 1) to i32))
262   %"variable ptr123" = bitcast i8* %malloccall122 to { i1,
       i8*, i8*, i32 }**
263   store { i1, i8*, i8*, i32 }* %"complex exec struct115", {
       i1, i8*, i8*, i32 }** %"variable ptr123", align 8
264   %malloccall124 = tail call i8* @malloc(i32 ptrtoint ({ i8**,
       { i8*, i8*, i32 }* }* getelementptr ({ i8**, { i8*, i8*,
       i32 }* }, { i8**, { i8*, i8*, i32 }* }* null, i32 1) to
       i32))
265   %struct_space125 = bitcast i8* %malloccall124 to { i8**, {
       i8*, i8*, i32 }* }*
266   %path_ptr126 = getelementptr inbounds { i8**, { i8*, i8*,
       i32 }* }, { i8**, { i8*, i8*, i32 }* }* %struct_space125,
       i32 0, i32 0
267   %malloccall127 = tail call i8* @malloc(i32 ptrtoint (i8**
       getelementptr (i8*, i8** null, i32 1) to i32))
268   %double_string_ptr128 = bitcast i8* %malloccall127 to i8**
269   store i8* getelementptr inbounds ([5 x i8], [5 x i8]*
       @string.10, i32 0, i32 0), i8** %double_string_ptr128,
```

```
    align 8
  store i8** %double_string_ptr128, i8*** %path_ptr126, align 8
  %args_ptr129 = getelementptr inbounds { i8**, { i8*, i8*,
    i32 }* }, { i8**, { i8*, i8*, i32 }* }* %struct_space125,
    i32 0, i32 1
  %malloccall130 = tail call i8* @malloc(i32 ptrtoint (i8**
    getelementptr (i8*, i8** null, i32 1) to i32))
  %double_string_ptr131 = bitcast i8* %malloccall130 to i8**
  store i8* getelementptr inbounds ([63 x i8], [63 x i8]*
    @string.11, i32 0, i32 0), i8** %double_string_ptr131,
    align 8
  %malloccall132 = tail call i8* @malloc(i32 ptrtoint (i8***
    getelementptr (i8**, i8*** null, i32 1) to i32))
  %value_ptr133 = bitcast i8* %malloccall132 to i8***
  store i8** %double_string_ptr131, i8*** %value_ptr133, align
    8
  %malloccall134 = tail call i8* @malloc(i32 ptrtoint ({ i8*,
    i8*, i32 }* getelementptr ({ i8*, i8*, i32 }, { i8*, i8*,
    i32 }* null, i32 1) to i32))
  %list_node135 = bitcast i8* %malloccall134 to { i8*, i8*,
    i32 }*
  %struct_val_ptr136 = getelementptr inbounds { i8*, i8*, i32
    }, { i8*, i8*, i32 }* %list_node135, i32 0, i32 0
  %struct_ptr_ptr137 = getelementptr inbounds { i8*, i8*, i32
    }, { i8*, i8*, i32 }* %list_node135, i32 0, i32 1
  %struct_ty_ptr138 = getelementptr inbounds { i8*, i8*, i32
    }, { i8*, i8*, i32 }* %list_node135, i32 0, i32 2
  %casted_ptr_ptr139 = bitcast i8** %struct_ptr_ptr137 to {
    i8*, i8*, i32 }**
  store { i8*, i8*, i32 }* null, { i8*, i8*, i32 }**
    %casted_ptr_ptr139, align 8
  %casted_val140 = bitcast i8*** %value_ptr133 to i8*
  store i8* %casted_val140, i8** %struct_val_ptr136, align 8
  store i32 4, i32* %struct_ty_ptr138, align 4
  store { i8*, i8*, i32 }* %list_node135, { i8*, i8*, i32 }**
    %args_ptr129, align 8
  %malloccall141 = tail call i8* @malloc(i32 ptrtoint ({ i1,
    i8*, i8*, i32 }* getelementptr ({ i1, i8*, i8*, i32 }, {
    i1, i8*, i8*, i32 }* null, i32 1) to i32))
  %"complex exec struct142" = bitcast i8* %malloccall141 to {
    i1, i8*, i8*, i32 }*
  %"complex bool143" = getelementptr inbounds { i1, i8*, i8*,
    i32 }, { i1, i8*, i8*, i32 }* %"complex exec struct142",
    i32 0, i32 0
  %"complex e1144" = getelementptr inbounds { i1, i8*, i8*,
    i32 }, { i1, i8*, i8*, i32 }* %"complex exec struct142",
    i32 0, i32 1
  store i1 true, i1* %"complex bool143", align 1
  %casted_malloc145 = bitcast { i8**, { i8*, i8*, i32 }* }*
    %struct_space125 to i8*
  store i8* %casted_malloc145, i8** %"complex e1144", align 8
```

```
296   %complex_bool_ptr = getelementptr inbounds { i1, i8*, i8*,
       i32 }, { i1, i8*, i8*, i32 }* %"complex exec struct142",
       i32 0, i32 0
297   %complex_bool = load i1, i1* %complex_bool_ptr, align 1
298   %malloccall146 = tail call i8* @malloc(i32 ptrtoint (i8**
       getelementptr (i8*, i8** null, i32 1) to i32))
299   %return_str_ptr = bitcast i8* %malloccall146 to i8**
300   br i1 %complex_bool, label %then, label %else
301
302 "run merge":                                      ; preds =
       %else, %then
303   %e5 = load { i1, i8*, i8*, i32 }*, { i1, i8*, i8*, i32 }**
       %"variable ptr123", align 8
304   %complex_bool_ptr149 = getelementptr inbounds { i1, i8*,
       i8*, i32 }, { i1, i8*, i8*, i32 }* %e5, i32 0, i32 0
305   %complex_bool150 = load i1, i1* %complex_bool_ptr149, align 1
306   %malloccall151 = tail call i8* @malloc(i32 ptrtoint (i8**
       getelementptr (i8*, i8** null, i32 1) to i32))
307   %return_str_ptr152 = bitcast i8* %malloccall151 to i8**
308   br i1 %complex_bool150, label %then154, label %else164
309
310 then:                                             ; preds =
       %entry
311   %exec_ptr = getelementptr inbounds { i1, i8*, i8*, i32 }, {
       i1, i8*, i8*, i32 }* %"complex exec struct142", i32 0, i32 1
312   %cast_run = bitcast i8** %exec_ptr to { i8**, { i8*, i8*,
       i32 }* }**
313   %exec = load { i8**, { i8*, i8*, i32 }* }*, { i8**, { i8*,
       i8*, i32 }* }** %cast_run, align 8
314   %dbl_path_ptr = getelementptr inbounds { i8**, { i8*, i8*,
       i32 }* }, { i8**, { i8*, i8*, i32 }* }* %exec, i32 0, i32 0
315   %path_ptr147 = load i8**, i8*** %dbl_path_ptr, align 8
316   %path = load i8*, i8** %path_ptr147, align 8
317   %args_ptr148 = getelementptr inbounds { i8**, { i8*, i8*,
       i32 }* }, { i8**, { i8*, i8*, i32 }* }* %exec, i32 0, i32 1
318   %args = load { i8*, i8*, i32 }*, { i8*, i8*, i32 }**
       %args_ptr148, align 8
319   %execvp = call i8* (i8*, { i8*, i8*, i32 }*, ...)
       @execvp_helper(i8* %path, { i8*, i8*, i32 }* %args)
320   store i8* %execvp, i8** %return_str_ptr, align 8
321   br label %"run merge"
322
323 else:                                             ; preds =
       %entry
324   %recurse_exec = call i8* ({ i1, i8*, i8*, i32 }*, ...)
       @recurse_exec({ i1, i8*, i8*, i32 }* %"complex exec
       struct142")
325   store i8* %recurse_exec, i8** %return_str_ptr, align 8
326   br label %"run merge"
327
328 "run merge153":                                   ; preds =
       %else164, %then154
```

```
329   %malloccall166 = tail call i8* @malloc(i32 ptrtoint (i8***
       getelementptr (i8**, i8*** null, i32 1) to i32))
330   %"variable ptr167" = bitcast i8* %malloccall166 to i8***
331   store i8** %return_str_ptr152, i8*** %"variable ptr167",
       align 8
332   %malloccall168 = tail call i8* @malloc(i32 ptrtoint ({ i8**,
       { i8*, i8*, i32 }* }* getelementptr ({ i8**, { i8*, i8*,
       i32 }* }, { i8**, { i8*, i8*, i32 }* }* null, i32 1) to
       i32))
333   %struct_space169 = bitcast i8* %malloccall168 to { i8**, {
       i8*, i8*, i32 }* }*
334   %path_ptr170 = getelementptr inbounds { i8**, { i8*, i8*,
       i32 }* }, { i8**, { i8*, i8*, i32 }* }* %struct_space169,
       i32 0, i32 0
335   %malloccall171 = tail call i8* @malloc(i32 ptrtoint (i8**
       getelementptr (i8*, i8** null, i32 1) to i32))
336   %double_string_ptr172 = bitcast i8* %malloccall171 to i8**
337   store i8* getelementptr inbounds ([5 x i8], [5 x i8]*
       @string.12, i32 0, i32 0), i8** %double_string_ptr172,
       align 8
338   store i8** %double_string_ptr172, i8*** %path_ptr170, align 8
339   %args_ptr173 = getelementptr inbounds { i8**, { i8*, i8*,
       i32 }* }, { i8**, { i8*, i8*, i32 }* }* %struct_space169,
       i32 0, i32 1
340   %malloccall174 = tail call i8* @malloc(i32 ptrtoint (i8**
       getelementptr (i8*, i8** null, i32 1) to i32))
341   %double_string_ptr175 = bitcast i8* %malloccall174 to i8**
342   store i8* getelementptr inbounds ([15 x i8], [15 x i8]*
       @string.13, i32 0, i32 0), i8** %double_string_ptr175,
       align 8
343   %malloccall176 = tail call i8* @malloc(i32 ptrtoint (i8***
       getelementptr (i8**, i8*** null, i32 1) to i32))
344   %value_ptr177 = bitcast i8* %malloccall176 to i8***
345   store i8** %double_string_ptr175, i8*** %value_ptr177, align
       8
346   %malloccall178 = tail call i8* @malloc(i32 ptrtoint ({ i8*,
       i8*, i32 }* getelementptr ({ i8*, i8*, i32 }, { i8*, i8*,
       i32 }* null, i32 1) to i32))
347   %list_node179 = bitcast i8* %malloccall178 to { i8*, i8*,
       i32 }*
348   %struct_val_ptr180 = getelementptr inbounds { i8*, i8*, i32
       }, { i8*, i8*, i32 }* %list_node179, i32 0, i32 0
349   %struct_ptr_ptr181 = getelementptr inbounds { i8*, i8*, i32
       }, { i8*, i8*, i32 }* %list_node179, i32 0, i32 1
350   %struct_ty_ptr182 = getelementptr inbounds { i8*, i8*, i32
       }, { i8*, i8*, i32 }* %list_node179, i32 0, i32 2
351   %casted_ptr_ptr183 = bitcast i8** %struct_ptr_ptr181 to {
       i8*, i8*, i32 }**
352   store { i8*, i8*, i32 }* null, { i8*, i8*, i32 }**
       %casted_ptr_ptr183, align 8
353   %casted_val184 = bitcast i8*** %value_ptr177 to i8*
354   store i8* %casted_val184, i8** %struct_val_ptr180, align 8
```

```
355   store i32 4, i32* %struct_ty_ptr182, align 4
356   store { i8*, i8*, i32 }* %list_node179, { i8*, i8*, i32 }**
       %args_ptr173, align 8
357   %malloccall185 = tail call i8* @malloc(i32 ptrtoint ({ i1,
       i8*, i8*, i32 }* getelementptr ({ i1, i8*, i8*, i32 }, {
       i1, i8*, i8*, i32 }* null, i32 1) to i32))
358   %"complex exec struct186" = bitcast i8* %malloccall185 to {
       i1, i8*, i8*, i32 }*
359   %"complex bool187" = getelementptr inbounds { i1, i8*, i8*,
       i32 }, { i1, i8*, i8*, i32 }* %"complex exec struct186",
       i32 0, i32 0
360   %"complex e1188" = getelementptr inbounds { i1, i8*, i8*,
       i32 }, { i1, i8*, i8*, i32 }* %"complex exec struct186",
       i32 0, i32 1
361   store i1 true, i1* %"complex bool187", align 1
362   %casted_malloc189 = bitcast { i8**, { i8*, i8*, i32 }* }*
       %struct_space169 to i8*
363   store i8* %casted_malloc189, i8** %"complex e1188", align 8
364   %complex_bool_ptr190 = getelementptr inbounds { i1, i8*,
       i8*, i32 }, { i1, i8*, i8*, i32 }* %"complex exec
       struct186", i32 0, i32 0
365   %complex_bool191 = load i1, i1* %complex_bool_ptr190, align 1
366   %malloccall192 = tail call i8* @malloc(i32 ptrtoint (i8**
       getelementptr (i8*, i8** null, i32 1) to i32))
367   %return_str_ptr193 = bitcast i8* %malloccall192 to i8**
368   br i1 %complex_bool191, label %then195, label %else205
369
370 then154:                                          ; preds =
       %"run merge"
371   %exec_ptr155 = getelementptr inbounds { i1, i8*, i8*, i32 },
       { i1, i8*, i8*, i32 }* %e5, i32 0, i32 1
372   %cast_run156 = bitcast i8** %exec_ptr155 to { i8**, { i8*,
       i8*, i32 }* }**
373   %exec157 = load { i8**, { i8*, i8*, i32 }* }*, { i8**, {
       i8*, i8*, i32 }* }** %cast_run156, align 8
374   %dbl_path_ptr158 = getelementptr inbounds { i8**, { i8*,
       i8*, i32 }* }, { i8**, { i8*, i8*, i32 }* }* %exec157, i32
       0, i32 0
375   %path_ptr159 = load i8**, i8*** %dbl_path_ptr158, align 8
376   %path160 = load i8*, i8** %path_ptr159, align 8
377   %args_ptr161 = getelementptr inbounds { i8**, { i8*, i8*,
       i32 }* }, { i8**, { i8*, i8*, i32 }* }* %exec157, i32 0,
       i32 1
378   %args162 = load { i8*, i8*, i32 }*, { i8*, i8*, i32 }**
       %args_ptr161, align 8
379   %execvp163 = call i8* (i8*, { i8*, i8*, i32 }*, ...)
       @execvp_helper(i8* %path160, { i8*, i8*, i32 }* %args162)
380   store i8* %execvp163, i8** %return_str_ptr152, align 8
381   br label %"run merge153"
382
383 else164:                                          ; preds =
       %"run merge"
```

```
384   %recurse_exec165 = call i8* ({ i1, i8*, i8*, i32 }*, ...)
       @recurse_exec({ i1, i8*, i8*, i32 }* %e5)
385   store i8* %recurse_exec165, i8** %return_str_ptr152, align 8
386   br label %"run merge153"
387
388 "run merge194":                                   ; preds =
       %else205, %then195
389   %e4207 = load { i1, i8*, i8*, i32 }*, { i1, i8*, i8*, i32
       }** %"variable ptr101", align 8
390   %e2208 = load { i1, i8*, i8*, i32 }*, { i1, i8*, i8*, i32
       }** %"variable ptr31", align 8
391   %malloccall209 = tail call i8* @malloc(i32 ptrtoint ({ i1,
       i8*, i8*, i32 }* getelementptr ({ i1, i8*, i8*, i32 }, {
       i1, i8*, i8*, i32 }* null, i32 1) to i32))
392   %"complex exec struct210" = bitcast i8* %malloccall209 to {
       i1, i8*, i8*, i32 }*
393   %"complex bool211" = getelementptr inbounds { i1, i8*, i8*,
       i32 }, { i1, i8*, i8*, i32 }* %"complex exec struct210",
       i32 0, i32 0
394   %"complex e1212" = getelementptr inbounds { i1, i8*, i8*,
       i32 }, { i1, i8*, i8*, i32 }* %"complex exec struct210",
       i32 0, i32 1
395   %"complex e2213" = getelementptr inbounds { i1, i8*, i8*,
       i32 }, { i1, i8*, i8*, i32 }* %"complex exec struct210",
       i32 0, i32 2
396   %"complex op214" = getelementptr inbounds { i1, i8*, i8*,
       i32 }, { i1, i8*, i8*, i32 }* %"complex exec struct210",
       i32 0, i32 3
397   store i1 false, i1* %"complex bool211", align 1
398   %casted_e1215 = bitcast { i1, i8*, i8*, i32 }* %e2208 to i8*
399   store i8* %casted_e1215, i8** %"complex e1212", align 8
400   %casted_e2216 = bitcast { i1, i8*, i8*, i32 }* %e4207 to i8*
401   store i8* %casted_e2216, i8** %"complex e2213", align 8
402   store i32 2, i32* %"complex op214", align 4
403   %e3217 = load { i1, i8*, i8*, i32 }*, { i1, i8*, i8*, i32
       }** %"variable ptr66", align 8
404   %e1218 = load { i1, i8*, i8*, i32 }*, { i1, i8*, i8*, i32
       }** %"variable ptr", align 8
405   %malloccall219 = tail call i8* @malloc(i32 ptrtoint ({ i1,
       i8*, i8*, i32 }* getelementptr ({ i1, i8*, i8*, i32 }, {
       i1, i8*, i8*, i32 }* null, i32 1) to i32))
406   %"complex exec struct220" = bitcast i8* %malloccall219 to {
       i1, i8*, i8*, i32 }*
407   %"complex bool221" = getelementptr inbounds { i1, i8*, i8*,
       i32 }, { i1, i8*, i8*, i32 }* %"complex exec struct220",
       i32 0, i32 0
408   %"complex e1222" = getelementptr inbounds { i1, i8*, i8*,
       i32 }, { i1, i8*, i8*, i32 }* %"complex exec struct220",
       i32 0, i32 1
409   %"complex e2223" = getelementptr inbounds { i1, i8*, i8*,
       i32 }, { i1, i8*, i8*, i32 }* %"complex exec struct220",
       i32 0, i32 2
```

```
410   %"complex op224" = getelementptr inbounds { i1, i8*, i8*,
       i32 }, { i1, i8*, i8*, i32 }* %"complex exec struct220",
       i32 0, i32 3
411   store i1 false, i1* %"complex bool221", align 1
412   %casted_e1225 = bitcast { i1, i8*, i8*, i32 }* %e1218 to i8*
413   store i8* %casted_e1225, i8** %"complex e1222", align 8
414   %casted_e2226 = bitcast { i1, i8*, i8*, i32 }* %e3217 to i8*
415   store i8* %casted_e2226, i8** %"complex e2223", align 8
416   store i32 2, i32* %"complex op224", align 4
417   %malloccall227 = tail call i8* @malloc(i32 ptrtoint ({ i1,
       i8*, i8*, i32 }* getelementptr ({ i1, i8*, i8*, i32 }, {
       i1, i8*, i8*, i32 }* null, i32 1) to i32))
418   %"complex exec struct228" = bitcast i8* %malloccall227 to {
       i1, i8*, i8*, i32 }*
419   %"complex bool229" = getelementptr inbounds { i1, i8*, i8*,
       i32 }, { i1, i8*, i8*, i32 }* %"complex exec struct228",
       i32 0, i32 0
420   %"complex e1230" = getelementptr inbounds { i1, i8*, i8*,
       i32 }, { i1, i8*, i8*, i32 }* %"complex exec struct228",
       i32 0, i32 1
421   %"complex e2231" = getelementptr inbounds { i1, i8*, i8*,
       i32 }, { i1, i8*, i8*, i32 }* %"complex exec struct228",
       i32 0, i32 2
422   %"complex op232" = getelementptr inbounds { i1, i8*, i8*,
       i32 }, { i1, i8*, i8*, i32 }* %"complex exec struct228",
       i32 0, i32 3
423   store i1 false, i1* %"complex bool229", align 1
424   %casted_e1233 = bitcast { i1, i8*, i8*, i32 }* %"complex
       exec struct220" to i8*
425   store i8* %casted_e1233, i8** %"complex e1230", align 8
426   %casted_e2234 = bitcast { i1, i8*, i8*, i32 }* %"complex
       exec struct210" to i8*
427   store i8* %casted_e2234, i8** %"complex e2231", align 8
428   store i32 1, i32* %"complex op232", align 4
429   %malloccall235 = tail call i8* @malloc(i32 ptrtoint ({ i1,
       i8*, i8*, i32 }** getelementptr ({ i1, i8*, i8*, i32 }*, {
       i1, i8*, i8*, i32 }** null, i32 1) to i32))
430   %"variable ptr236" = bitcast i8* %malloccall235 to { i1,
       i8*, i8*, i32 }**
431   store { i1, i8*, i8*, i32 }* %"complex exec struct228", {
       i1, i8*, i8*, i32 }** %"variable ptr236", align 8
432   %malloccall237 = tail call i8* @malloc(i32 ptrtoint ({ i8**,
       { i8*, i8*, i32 }* }* getelementptr ({ i8**, { i8*, i8*,
       i32 }* }, { i8**, { i8*, i8*, i32 }* }* null, i32 1) to
       i32))
433   %struct_space238 = bitcast i8* %malloccall237 to { i8**, {
       i8*, i8*, i32 }* }*
434   %path_ptr239 = getelementptr inbounds { i8**, { i8*, i8*,
       i32 }* }, { i8**, { i8*, i8*, i32 }* }* %struct_space238,
       i32 0, i32 0
435   %malloccall240 = tail call i8* @malloc(i32 ptrtoint (i8**
       getelementptr (i8*, i8** null, i32 1) to i32))
```

```
436   %double_string_ptr241 = bitcast i8* %malloccall240 to i8**
437   store i8* getelementptr inbounds ([5 x i8], [5 x i8]*
       @string.14, i32 0, i32 0), i8** %double_string_ptr241,
       align 8
438   store i8** %double_string_ptr241, i8*** %path_ptr239, align 8
439   %args_ptr242 = getelementptr inbounds { i8**, { i8*, i8*,
       i32 }* }, { i8**, { i8*, i8*, i32 }* }* %struct_space238,
       i32 0, i32 1
440   %malloccall243 = tail call i8* @malloc(i32 ptrtoint (i8**
       getelementptr (i8*, i8** null, i32 1) to i32))
441   %double_string_ptr244 = bitcast i8* %malloccall243 to i8**
442   store i8* getelementptr inbounds ([40 x i8], [40 x i8]*
       @string.15, i32 0, i32 0), i8** %double_string_ptr244,
       align 8
443   %malloccall245 = tail call i8* @malloc(i32 ptrtoint (i8***
       getelementptr (i8**, i8*** null, i32 1) to i32))
444   %value_ptr246 = bitcast i8* %malloccall245 to i8***
445   store i8** %double_string_ptr244, i8*** %value_ptr246, align
       8
446   %malloccall247 = tail call i8* @malloc(i32 ptrtoint ({ i8*,
       i8*, i32 }* getelementptr ({ i8*, i8*, i32 }, { i8*, i8*,
       i32 }* null, i32 1) to i32))
447   %list_node248 = bitcast i8* %malloccall247 to { i8*, i8*,
       i32 }*
448   %struct_val_ptr249 = getelementptr inbounds { i8*, i8*, i32
       }, { i8*, i8*, i32 }* %list_node248, i32 0, i32 0
449   %struct_ptr_ptr250 = getelementptr inbounds { i8*, i8*, i32
       }, { i8*, i8*, i32 }* %list_node248, i32 0, i32 1
450   %struct_ty_ptr251 = getelementptr inbounds { i8*, i8*, i32
       }, { i8*, i8*, i32 }* %list_node248, i32 0, i32 2
451   %casted_ptr_ptr252 = bitcast i8** %struct_ptr_ptr250 to {
       i8*, i8*, i32 }**
452   store { i8*, i8*, i32 }* null, { i8*, i8*, i32 }**
       %casted_ptr_ptr252, align 8
453   %casted_val253 = bitcast i8*** %value_ptr246 to i8*
454   store i8* %casted_val253, i8** %struct_val_ptr249, align 8
455   store i32 4, i32* %struct_ty_ptr251, align 4
456   store { i8*, i8*, i32 }* %list_node248, { i8*, i8*, i32 }**
       %args_ptr242, align 8
457   %malloccall254 = tail call i8* @malloc(i32 ptrtoint ({ i1,
       i8*, i8*, i32 }* getelementptr ({ i1, i8*, i8*, i32 }, {
       i1, i8*, i8*, i32 }* null, i32 1) to i32))
458   %"complex exec struct255" = bitcast i8* %malloccall254 to {
       i1, i8*, i8*, i32 }*
459   %"complex bool256" = getelementptr inbounds { i1, i8*, i8*,
       i32 }, { i1, i8*, i8*, i32 }* %"complex exec struct255",
       i32 0, i32 0
460   %"complex e1257" = getelementptr inbounds { i1, i8*, i8*,
       i32 }, { i1, i8*, i8*, i32 }* %"complex exec struct255",
       i32 0, i32 1
461   store i1 true, i1* %"complex bool256", align 1
```

```
462   %casted_malloc258 = bitcast { i8**, { i8*, i8*, i32 }* }*
       %struct_space238 to i8*
463   store i8* %casted_malloc258, i8** %"complex e1257", align 8
464   %complex_bool_ptr259 = getelementptr inbounds { i1, i8*,
       i8*, i32 }, { i1, i8*, i8*, i32 }* %"complex exec
       struct255", i32 0, i32 0
465   %complex_bool260 = load i1, i1* %complex_bool_ptr259, align 1
466   %malloccall261 = tail call i8* @malloc(i32 ptrtoint (i8**
       getelementptr (i8*, i8** null, i32 1) to i32))
467   %return_str_ptr262 = bitcast i8* %malloccall261 to i8**
468   br i1 %complex_bool260, label %then264, label %else274
469
470 then195:                                          ; preds =
       %"run merge153"
471   %exec_ptr196 = getelementptr inbounds { i1, i8*, i8*, i32 },
       { i1, i8*, i8*, i32 }* %"complex exec struct186", i32 0,
       i32 1
472   %cast_run197 = bitcast i8** %exec_ptr196 to { i8**, { i8*,
       i8*, i32 }* }**
473   %exec198 = load { i8**, { i8*, i8*, i32 }* }*, { i8**, {
       i8*, i8*, i32 }* }** %cast_run197, align 8
474   %dbl_path_ptr199 = getelementptr inbounds { i8**, { i8*,
       i8*, i32 }* }, { i8**, { i8*, i8*, i32 }* }* %exec198, i32
       0, i32 0
475   %path_ptr200 = load i8**, i8*** %dbl_path_ptr199, align 8
476   %path201 = load i8*, i8** %path_ptr200, align 8
477   %args_ptr202 = getelementptr inbounds { i8**, { i8*, i8*,
       i32 }* }, { i8**, { i8*, i8*, i32 }* }* %exec198, i32 0,
       i32 1
478   %args203 = load { i8*, i8*, i32 }*, { i8*, i8*, i32 }**
       %args_ptr202, align 8
479   %execvp204 = call i8* (i8*, { i8*, i8*, i32 }*, ...)
       @execvp_helper(i8* %path201, { i8*, i8*, i32 }* %args203)
480   store i8* %execvp204, i8** %return_str_ptr193, align 8
481   br label %"run merge194"
482
483 else205:                                          ; preds =
       %"run merge153"
484   %recurse_exec206 = call i8* ({ i1, i8*, i8*, i32 }*, ...)
       @recurse_exec({ i1, i8*, i8*, i32 }* %"complex exec
       struct186")
485   store i8* %recurse_exec206, i8** %return_str_ptr193, align 8
486   br label %"run merge194"
487
488 "run merge263":                                   ; preds =
       %else274, %then264
489   %e5276 = load { i1, i8*, i8*, i32 }*, { i1, i8*, i8*, i32
       }** %"variable ptr236", align 8
490   %complex_bool_ptr277 = getelementptr inbounds { i1, i8*,
       i8*, i32 }, { i1, i8*, i8*, i32 }* %e5276, i32 0, i32 0
491   %complex_bool278 = load i1, i1* %complex_bool_ptr277, align 1
```

```
492   %malloccall279 = tail call i8* @malloc(i32 ptrtoint (i8**
        getelementptr (i8*, i8** null, i32 1) to i32))
493   %return_str_ptr280 = bitcast i8* %malloccall279 to i8**
494   br i1 %complex_bool278, label %then282, label %else292
495
496 then264:                                          ; preds =
        %"run merge194"
497   %exec_ptr265 = getelementptr inbounds { i1, i8*, i8*, i32 },
        { i1, i8*, i8*, i32 }* %"complex exec struct255", i32 0,
        i32 1
498   %cast_run266 = bitcast i8** %exec_ptr265 to { i8**, { i8*,
        i8*, i32 }* }**
499   %exec267 = load { i8**, { i8*, i8*, i32 }* }*, { i8**, {
        i8*, i8*, i32 }* }** %cast_run266, align 8
500   %dbl_path_ptr268 = getelementptr inbounds { i8**, { i8*,
        i8*, i32 }* }, { i8**, { i8*, i8*, i32 }* }* %exec267, i32
        0, i32 0
501   %path_ptr269 = load i8**, i8*** %dbl_path_ptr268, align 8
502   %path270 = load i8*, i8** %path_ptr269, align 8
503   %args_ptr271 = getelementptr inbounds { i8**, { i8*, i8*,
        i32 }* }, { i8**, { i8*, i8*, i32 }* }* %exec267, i32 0,
        i32 1
504   %args272 = load { i8*, i8*, i32 }*, { i8*, i8*, i32 }**
        %args_ptr271, align 8
505   %execvp273 = call i8* (i8*, { i8*, i8*, i32 }*, ...)
        @execvp_helper(i8* %path270, { i8*, i8*, i32 }* %args272)
506   store i8* %execvp273, i8** %return_str_ptr262, align 8
507   br label %"run merge263"
508
509 else274:                                          ; preds =
        %"run merge194"
510   %recurse_exec275 = call i8* ({ i1, i8*, i8*, i32 }*, ...)
        @recurse_exec({ i1, i8*, i8*, i32 }* %"complex exec
        struct255")
511   store i8* %recurse_exec275, i8** %return_str_ptr262, align 8
512   br label %"run merge263"
513
514 "run merge281":                                   ; preds =
        %else292, %then282
515   %malloccall294 = tail call i8* @malloc(i32 ptrtoint (i8***
        getelementptr (i8**, i8*** null, i32 1) to i32))
516   %"variable ptr295" = bitcast i8* %malloccall294 to i8***
517   store i8** %return_str_ptr280, i8*** %"variable ptr295",
        align 8
518   %malloccall296 = tail call i8* @malloc(i32 ptrtoint ({ i8**,
        { i8*, i8*, i32 }* }* getelementptr ({ i8**, { i8*, i8*,
        i32 }* }, { i8**, { i8*, i8*, i32 }* }* null, i32 1) to
        i32))
519   %struct_space297 = bitcast i8* %malloccall296 to { i8**, {
        i8*, i8*, i32 }* }*
520   %path_ptr298 = getelementptr inbounds { i8**, { i8*, i8*,
        i32 }* }, { i8**, { i8*, i8*, i32 }* }* %struct_space297,
```

```
  i32 0, i32 0
%malloccall299 = tail call i8* @malloc(i32 ptrtoint (i8**
  getelementptr (i8*, i8** null, i32 1) to i32))
%double_string_ptr300 = bitcast i8* %malloccall299 to i8**
store i8* getelementptr inbounds ([5 x i8], [5 x i8]*
  @string.16, i32 0, i32 0), i8** %double_string_ptr300,
  align 8
store i8** %double_string_ptr300, i8*** %path_ptr298, align 8
%args_ptr301 = getelementptr inbounds { i8**, { i8*, i8*,
  i32 }* }, { i8**, { i8*, i8*, i32 }* }* %struct_space297,
  i32 0, i32 1
%malloccall302 = tail call i8* @malloc(i32 ptrtoint (i8**
  getelementptr (i8*, i8** null, i32 1) to i32))
%double_string_ptr303 = bitcast i8* %malloccall302 to i8**
store i8* getelementptr inbounds ([15 x i8], [15 x i8]*
  @string.17, i32 0, i32 0), i8** %double_string_ptr303,
  align 8
%malloccall304 = tail call i8* @malloc(i32 ptrtoint (i8***
  getelementptr (i8**, i8*** null, i32 1) to i32))
%value_ptr305 = bitcast i8* %malloccall304 to i8***
store i8** %double_string_ptr303, i8*** %value_ptr305, align
  8
%malloccall306 = tail call i8* @malloc(i32 ptrtoint ({ i8*,
  i8*, i32 }* getelementptr ({ i8*, i8*, i32 }, { i8*, i8*,
  i32 }* null, i32 1) to i32))
%list_node307 = bitcast i8* %malloccall306 to { i8*, i8*,
  i32 }*
%struct_val_ptr308 = getelementptr inbounds { i8*, i8*, i32
  }, { i8*, i8*, i32 }* %list_node307, i32 0, i32 0
%struct_ptr_ptr309 = getelementptr inbounds { i8*, i8*, i32
  }, { i8*, i8*, i32 }* %list_node307, i32 0, i32 1
%struct_ty_ptr310 = getelementptr inbounds { i8*, i8*, i32
  }, { i8*, i8*, i32 }* %list_node307, i32 0, i32 2
%casted_ptr_ptr311 = bitcast i8** %struct_ptr_ptr309 to {
  i8*, i8*, i32 }**
store { i8*, i8*, i32 }* null, { i8*, i8*, i32 }**
  %casted_ptr_ptr311, align 8
%casted_val312 = bitcast i8*** %value_ptr305 to i8*
store i8* %casted_val312, i8** %struct_val_ptr308, align 8
store i32 4, i32* %struct_ty_ptr310, align 4
store { i8*, i8*, i32 }* %list_node307, { i8*, i8*, i32 }**
  %args_ptr301, align 8
%malloccall313 = tail call i8* @malloc(i32 ptrtoint ({ i1,
  i8*, i8*, i32 }* getelementptr ({ i1, i8*, i8*, i32 }, {
  i1, i8*, i8*, i32 }* null, i32 1) to i32))
%"complex exec struct314" = bitcast i8* %malloccall313 to {
  i1, i8*, i8*, i32 }*
%"complex bool315" = getelementptr inbounds { i1, i8*, i8*,
  i32 }, { i1, i8*, i8*, i32 }* %"complex exec struct314",
  i32 0, i32 0
%"complex e1316" = getelementptr inbounds { i1, i8*, i8*,
  i32 }, { i1, i8*, i8*, i32 }* %"complex exec struct314",
```

```
    i32 0, i32 1
  store i1 true, i1* %"complex bool315", align 1
  %casted_malloc317 = bitcast { i8**, { i8*, i8*, i32 }* }*
    %struct_space297 to i8*
  store i8* %casted_malloc317, i8** %"complex e1316", align 8
  %complex_bool_ptr318 = getelementptr inbounds { i1, i8*,
    i8*, i32 }, { i1, i8*, i8*, i32 }* %"complex exec
    struct314", i32 0, i32 0
  %complex_bool319 = load i1, i1* %complex_bool_ptr318, align 1
  %malloccall320 = tail call i8* @malloc(i32 ptrtoint (i8**
    getelementptr (i8*, i8** null, i32 1) to i32))
  %return_str_ptr321 = bitcast i8* %malloccall320 to i8**
  br i1 %complex_bool319, label %then323, label %else333

then282:                                          ; preds =
    %"run merge263"
  %exec_ptr283 = getelementptr inbounds { i1, i8*, i8*, i32 },
    { i1, i8*, i8*, i32 }* %e5276, i32 0, i32 1
  %cast_run284 = bitcast i8** %exec_ptr283 to { i8**, { i8*,
    i8*, i32 }* }**
  %exec285 = load { i8**, { i8*, i8*, i32 }* }*, { i8**, {
    i8*, i8*, i32 }* }** %cast_run284, align 8
  %dbl_path_ptr286 = getelementptr inbounds { i8**, { i8*,
    i8*, i32 }* }, { i8**, { i8*, i8*, i32 }* }* %exec285, i32
    0, i32 0
  %path_ptr287 = load i8**, i8*** %dbl_path_ptr286, align 8
  %path288 = load i8*, i8** %path_ptr287, align 8
  %args_ptr289 = getelementptr inbounds { i8**, { i8*, i8*,
    i32 }* }, { i8**, { i8*, i8*, i32 }* }* %exec285, i32 0,
    i32 1
  %args290 = load { i8*, i8*, i32 }*, { i8*, i8*, i32 }**
    %args_ptr289, align 8
  %execvp291 = call i8* (i8*, { i8*, i8*, i32 }*, ...)
    @execvp_helper(i8* %path288, { i8*, i8*, i32 }* %args290)
  store i8* %execvp291, i8** %return_str_ptr280, align 8
  br label %"run merge281"

else292:                                          ; preds =
    %"run merge263"
  %recurse_exec293 = call i8* ({ i1, i8*, i8*, i32 }*, ...)
    @recurse_exec({ i1, i8*, i8*, i32 }* %e5276)
  store i8* %recurse_exec293, i8** %return_str_ptr280, align 8
  br label %"run merge281"

"run merge322":                                   ; preds =
    %else333, %then323
  %e3335 = load { i1, i8*, i8*, i32 }*, { i1, i8*, i8*, i32
    }** %"variable ptr66", align 8
  %e2336 = load { i1, i8*, i8*, i32 }*, { i1, i8*, i8*, i32
    }** %"variable ptr31", align 8
  %e1337 = load { i1, i8*, i8*, i32 }*, { i1, i8*, i8*, i32
    }** %"variable ptr", align 8
```

```
578   %malloccall338 = tail call i8* @malloc(i32 ptrtoint ({ i1,
       i8*, i8*, i32 }* getelementptr ({ i1, i8*, i8*, i32 }, {
       i1, i8*, i8*, i32 }* null, i32 1) to i32))
579   %"complex exec struct339" = bitcast i8* %malloccall338 to {
       i1, i8*, i8*, i32 }*
580   %"complex bool340" = getelementptr inbounds { i1, i8*, i8*,
       i32 }, { i1, i8*, i8*, i32 }* %"complex exec struct339",
       i32 0, i32 0
581   %"complex e1341" = getelementptr inbounds { i1, i8*, i8*,
       i32 }, { i1, i8*, i8*, i32 }* %"complex exec struct339",
       i32 0, i32 1
582   %"complex e2342" = getelementptr inbounds { i1, i8*, i8*,
       i32 }, { i1, i8*, i8*, i32 }* %"complex exec struct339",
       i32 0, i32 2
583   %"complex op343" = getelementptr inbounds { i1, i8*, i8*,
       i32 }, { i1, i8*, i8*, i32 }* %"complex exec struct339",
       i32 0, i32 3
584   store i1 false, i1* %"complex bool340", align 1
585   %casted_e1344 = bitcast { i1, i8*, i8*, i32 }* %e1337 to i8*
586   store i8* %casted_e1344, i8** %"complex e1341", align 8
587   %casted_e2345 = bitcast { i1, i8*, i8*, i32 }* %e2336 to i8*
588   store i8* %casted_e2345, i8** %"complex e2342", align 8
589   store i32 0, i32* %"complex op343", align 4
590   %malloccall346 = tail call i8* @malloc(i32 ptrtoint ({ i1,
       i8*, i8*, i32 }* getelementptr ({ i1, i8*, i8*, i32 }, {
       i1, i8*, i8*, i32 }* null, i32 1) to i32))
591   %"complex exec struct347" = bitcast i8* %malloccall346 to {
       i1, i8*, i8*, i32 }*
592   %"complex bool348" = getelementptr inbounds { i1, i8*, i8*,
       i32 }, { i1, i8*, i8*, i32 }* %"complex exec struct347",
       i32 0, i32 0
593   %"complex e1349" = getelementptr inbounds { i1, i8*, i8*,
       i32 }, { i1, i8*, i8*, i32 }* %"complex exec struct347",
       i32 0, i32 1
594   %"complex e2350" = getelementptr inbounds { i1, i8*, i8*,
       i32 }, { i1, i8*, i8*, i32 }* %"complex exec struct347",
       i32 0, i32 2
595   %"complex op351" = getelementptr inbounds { i1, i8*, i8*,
       i32 }, { i1, i8*, i8*, i32 }* %"complex exec struct347",
       i32 0, i32 3
596   store i1 false, i1* %"complex bool348", align 1
597   %casted_e1352 = bitcast { i1, i8*, i8*, i32 }* %"complex
       exec struct339" to i8*
598   store i8* %casted_e1352, i8** %"complex e1349", align 8
599   %casted_e2353 = bitcast { i1, i8*, i8*, i32 }* %e3335 to i8*
600   store i8* %casted_e2353, i8** %"complex e2350", align 8
601   store i32 2, i32* %"complex op351", align 4
602   %malloccall354 = tail call i8* @malloc(i32 ptrtoint ({ i1,
       i8*, i8*, i32 }** getelementptr ({ i1, i8*, i8*, i32 }*, {
       i1, i8*, i8*, i32 }** null, i32 1) to i32))
603   %"variable ptr355" = bitcast i8* %malloccall354 to { i1,
       i8*, i8*, i32 }**
```

```
  store { i1, i8*, i8*, i32 }* %"complex exec struct347", {
   i1, i8*, i8*, i32 }** %"variable ptr355", align 8
  %malloccall356 = tail call i8* @malloc(i32 ptrtoint ({ i8**,
   { i8*, i8*, i32 }* }* getelementptr ({ i8**, { i8*, i8*,
   i32 }* }, { i8**, { i8*, i8*, i32 }* }* null, i32 1) to
   i32))
  %struct_space357 = bitcast i8* %malloccall356 to { i8**, {
   i8*, i8*, i32 }* }*
  %path_ptr358 = getelementptr inbounds { i8**, { i8*, i8*,
   i32 }* }, { i8**, { i8*, i8*, i32 }* }* %struct_space357,
   i32 0, i32 0
  %malloccall359 = tail call i8* @malloc(i32 ptrtoint (i8**
   getelementptr (i8*, i8** null, i32 1) to i32))
  %double_string_ptr360 = bitcast i8* %malloccall359 to i8**
  store i8* getelementptr inbounds ([5 x i8], [5 x i8]*
   @string.18, i32 0, i32 0), i8** %double_string_ptr360,
   align 8
  store i8** %double_string_ptr360, i8*** %path_ptr358, align 8
  %args_ptr361 = getelementptr inbounds { i8**, { i8*, i8*,
   i32 }* }, { i8**, { i8*, i8*, i32 }* }* %struct_space357,
   i32 0, i32 1
  %malloccall362 = tail call i8* @malloc(i32 ptrtoint (i8**
   getelementptr (i8*, i8** null, i32 1) to i32))
  %double_string_ptr363 = bitcast i8* %malloccall362 to i8**
  store i8* getelementptr inbounds ([32 x i8], [32 x i8]*
   @string.19, i32 0, i32 0), i8** %double_string_ptr363,
   align 8
  %malloccall364 = tail call i8* @malloc(i32 ptrtoint (i8***
   getelementptr (i8**, i8*** null, i32 1) to i32))
  %value_ptr365 = bitcast i8* %malloccall364 to i8***
  store i8** %double_string_ptr363, i8*** %value_ptr365, align
   8
  %malloccall366 = tail call i8* @malloc(i32 ptrtoint ({ i8*,
   i8*, i32 }* getelementptr ({ i8*, i8*, i32 }, { i8*, i8*,
   i32 }* null, i32 1) to i32))
  %list_node367 = bitcast i8* %malloccall366 to { i8*, i8*,
   i32 }*
  %struct_val_ptr368 = getelementptr inbounds { i8*, i8*, i32
   }, { i8*, i8*, i32 }* %list_node367, i32 0, i32 0
  %struct_ptr_ptr369 = getelementptr inbounds { i8*, i8*, i32
   }, { i8*, i8*, i32 }* %list_node367, i32 0, i32 1
  %struct_ty_ptr370 = getelementptr inbounds { i8*, i8*, i32
   }, { i8*, i8*, i32 }* %list_node367, i32 0, i32 2
  %casted_ptr_ptr371 = bitcast i8** %struct_ptr_ptr369 to {
   i8*, i8*, i32 }**
  store { i8*, i8*, i32 }* null, { i8*, i8*, i32 }**
   %casted_ptr_ptr371, align 8
  %casted_val372 = bitcast i8*** %value_ptr365 to i8*
  store i8* %casted_val372, i8** %struct_val_ptr368, align 8
  store i32 4, i32* %struct_ty_ptr370, align 4
  store { i8*, i8*, i32 }* %list_node367, { i8*, i8*, i32 }**
   %args_ptr361, align 8
```

```
630   %malloccall373 = tail call i8* @malloc(i32 ptrtoint ({ i1,
       i8*, i8*, i32 }* getelementptr ({ i1, i8*, i8*, i32 }, {
       i1, i8*, i8*, i32 }* null, i32 1) to i32))
631   %"complex exec struct374" = bitcast i8* %malloccall373 to {
       i1, i8*, i8*, i32 }*
632   %"complex bool375" = getelementptr inbounds { i1, i8*, i8*,
       i32 }, { i1, i8*, i8*, i32 }* %"complex exec struct374",
       i32 0, i32 0
633   %"complex e1376" = getelementptr inbounds { i1, i8*, i8*,
       i32 }, { i1, i8*, i8*, i32 }* %"complex exec struct374",
       i32 0, i32 1
634   store i1 true, i1* %"complex bool375", align 1
635   %casted_malloc377 = bitcast { i8**, { i8*, i8*, i32 }* }*
       %struct_space357 to i8*
636   store i8* %casted_malloc377, i8** %"complex e1376", align 8
637   %complex_bool_ptr378 = getelementptr inbounds { i1, i8*,
       i8*, i32 }, { i1, i8*, i8*, i32 }* %"complex exec
       struct374", i32 0, i32 0
638   %complex_bool379 = load i1, i1* %complex_bool_ptr378, align 1
639   %malloccall380 = tail call i8* @malloc(i32 ptrtoint (i8**
       getelementptr (i8*, i8** null, i32 1) to i32))
640   %return_str_ptr381 = bitcast i8* %malloccall380 to i8**
641   br i1 %complex_bool379, label %then383, label %else393
642
643 then323:                                          ; preds =
       %"run merge281"
644   %exec_ptr324 = getelementptr inbounds { i1, i8*, i8*, i32 },
       { i1, i8*, i8*, i32 }* %"complex exec struct314", i32 0,
       i32 1
645   %cast_run325 = bitcast i8** %exec_ptr324 to { i8**, { i8*,
       i8*, i32 }* }**
646   %exec326 = load { i8**, { i8*, i8*, i32 }* }*, { i8**, {
       i8*, i8*, i32 }* }** %cast_run325, align 8
647   %dbl_path_ptr327 = getelementptr inbounds { i8**, { i8*,
       i8*, i32 }* }, { i8**, { i8*, i8*, i32 }* }* %exec326, i32
       0, i32 0
648   %path_ptr328 = load i8**, i8*** %dbl_path_ptr327, align 8
649   %path329 = load i8*, i8** %path_ptr328, align 8
650   %args_ptr330 = getelementptr inbounds { i8**, { i8*, i8*,
       i32 }* }, { i8**, { i8*, i8*, i32 }* }* %exec326, i32 0,
       i32 1
651   %args331 = load { i8*, i8*, i32 }*, { i8*, i8*, i32 }**
       %args_ptr330, align 8
652   %execvp332 = call i8* (i8*, { i8*, i8*, i32 }*, ...)
       @execvp_helper(i8* %path329, { i8*, i8*, i32 }* %args331)
653   store i8* %execvp332, i8** %return_str_ptr321, align 8
654   br label %"run merge322"
655
656 else333:                                          ; preds =
       %"run merge281"
657   %recurse_exec334 = call i8* ({ i1, i8*, i8*, i32 }*, ...)
       @recurse_exec({ i1, i8*, i8*, i32 }* %"complex exec
```

```
    struct314")
  store i8* %recurse_exec334, i8** %return_str_ptr321, align 8
  br label %"run merge322"

"run merge382":                                   ; preds =
    %else393, %then383
  %e5395 = load { i1, i8*, i8*, i32 }*, { i1, i8*, i8*, i32
    }** %"variable ptr355", align 8
  %complex_bool_ptr396 = getelementptr inbounds { i1, i8*,
    i8*, i32 }, { i1, i8*, i8*, i32 }* %e5395, i32 0, i32 0
  %complex_bool397 = load i1, i1* %complex_bool_ptr396, align 1
  %malloccall398 = tail call i8* @malloc(i32 ptrtoint (i8**
    getelementptr (i8*, i8** null, i32 1) to i32))
  %return_str_ptr399 = bitcast i8* %malloccall398 to i8**
  br i1 %complex_bool397, label %then401, label %else411

then383:                                          ; preds =
    %"run merge322"
  %exec_ptr384 = getelementptr inbounds { i1, i8*, i8*, i32 },
    { i1, i8*, i8*, i32 }* %"complex exec struct374", i32 0,
    i32 1
  %cast_run385 = bitcast i8** %exec_ptr384 to { i8**, { i8*,
    i8*, i32 }* }**
  %exec386 = load { i8**, { i8*, i8*, i32 }* }*, { i8**, {
    i8*, i8*, i32 }* }** %cast_run385, align 8
  %dbl_path_ptr387 = getelementptr inbounds { i8**, { i8*,
    i8*, i32 }* }, { i8**, { i8*, i8*, i32 }* }* %exec386, i32
    0, i32 0
  %path_ptr388 = load i8**, i8*** %dbl_path_ptr387, align 8
  %path389 = load i8*, i8** %path_ptr388, align 8
  %args_ptr390 = getelementptr inbounds { i8**, { i8*, i8*,
    i32 }* }, { i8**, { i8*, i8*, i32 }* }* %exec386, i32 0,
    i32 1
  %args391 = load { i8*, i8*, i32 }*, { i8*, i8*, i32 }**
    %args_ptr390, align 8
  %execvp392 = call i8* (i8*, { i8*, i8*, i32 }*, ...)
    @execvp_helper(i8* %path389, { i8*, i8*, i32 }* %args391)
  store i8* %execvp392, i8** %return_str_ptr381, align 8
  br label %"run merge382"

else393:                                          ; preds =
    %"run merge322"
  %recurse_exec394 = call i8* ({ i1, i8*, i8*, i32 }*, ...)
    @recurse_exec({ i1, i8*, i8*, i32 }* %"complex exec
    struct374")
  store i8* %recurse_exec394, i8** %return_str_ptr381, align 8
  br label %"run merge382"

"run merge400":                                   ; preds =
    %else411, %then401
  %malloccall413 = tail call i8* @malloc(i32 ptrtoint (i8***
    getelementptr (i8**, i8*** null, i32 1) to i32))
```

```
  %"variable ptr414" = bitcast i8* %malloccall413 to i8***
  store i8** %return_str_ptr399, i8*** %"variable ptr414",
   align 8
  %malloccall415 = tail call i8* @malloc(i32 ptrtoint ({ i8**,
   { i8*, i8*, i32 }* }* getelementptr ({ i8**, { i8*, i8*,
   i32 }* }, { i8**, { i8*, i8*, i32 }* }* null, i32 1) to
   i32))
  %struct_space416 = bitcast i8* %malloccall415 to { i8**, {
   i8*, i8*, i32 }* }*
  %path_ptr417 = getelementptr inbounds { i8**, { i8*, i8*,
   i32 }* }, { i8**, { i8*, i8*, i32 }* }* %struct_space416,
   i32 0, i32 0
  %malloccall418 = tail call i8* @malloc(i32 ptrtoint (i8**
   getelementptr (i8*, i8** null, i32 1) to i32))
  %double_string_ptr419 = bitcast i8* %malloccall418 to i8**
  store i8* getelementptr inbounds ([5 x i8], [5 x i8]*
   @string.20, i32 0, i32 0), i8** %double_string_ptr419,
   align 8
  store i8** %double_string_ptr419, i8*** %path_ptr417, align 8
  %args_ptr420 = getelementptr inbounds { i8**, { i8*, i8*,
   i32 }* }, { i8**, { i8*, i8*, i32 }* }* %struct_space416,
   i32 0, i32 1
  %malloccall421 = tail call i8* @malloc(i32 ptrtoint (i8**
   getelementptr (i8*, i8** null, i32 1) to i32))
  %double_string_ptr422 = bitcast i8* %malloccall421 to i8**
  store i8* getelementptr inbounds ([15 x i8], [15 x i8]*
   @string.21, i32 0, i32 0), i8** %double_string_ptr422,
   align 8
  %malloccall423 = tail call i8* @malloc(i32 ptrtoint (i8***
   getelementptr (i8**, i8*** null, i32 1) to i32))
  %value_ptr424 = bitcast i8* %malloccall423 to i8***
  store i8** %double_string_ptr422, i8*** %value_ptr424, align
   8
  %malloccall425 = tail call i8* @malloc(i32 ptrtoint ({ i8*,
   i8*, i32 }* getelementptr ({ i8*, i8*, i32 }, { i8*, i8*,
   i32 }* null, i32 1) to i32))
  %list_node426 = bitcast i8* %malloccall425 to { i8*, i8*,
   i32 }*
  %struct_val_ptr427 = getelementptr inbounds { i8*, i8*, i32
   }, { i8*, i8*, i32 }* %list_node426, i32 0, i32 0
  %struct_ptr_ptr428 = getelementptr inbounds { i8*, i8*, i32
   }, { i8*, i8*, i32 }* %list_node426, i32 0, i32 1
  %struct_ty_ptr429 = getelementptr inbounds { i8*, i8*, i32
   }, { i8*, i8*, i32 }* %list_node426, i32 0, i32 2
  %casted_ptr_ptr430 = bitcast i8** %struct_ptr_ptr428 to {
   i8*, i8*, i32 }**
  store { i8*, i8*, i32 }* null, { i8*, i8*, i32 }**
   %casted_ptr_ptr430, align 8
  %casted_val431 = bitcast i8*** %value_ptr424 to i8*
  store i8* %casted_val431, i8** %struct_val_ptr427, align 8
  store i32 4, i32* %struct_ty_ptr429, align 4
```

```
715   store { i8*, i8*, i32 }* %list_node426, { i8*, i8*, i32 }**
       %args_ptr420, align 8
716   %malloccall432 = tail call i8* @malloc(i32 ptrtoint ({ i1,
       i8*, i8*, i32 }* getelementptr ({ i1, i8*, i8*, i32 }, {
       i1, i8*, i8*, i32 }* null, i32 1) to i32))
717   %"complex exec struct433" = bitcast i8* %malloccall432 to {
       i1, i8*, i8*, i32 }*
718   %"complex bool434" = getelementptr inbounds { i1, i8*, i8*,
       i32 }, { i1, i8*, i8*, i32 }* %"complex exec struct433",
       i32 0, i32 0
719   %"complex e1435" = getelementptr inbounds { i1, i8*, i8*,
       i32 }, { i1, i8*, i8*, i32 }* %"complex exec struct433",
       i32 0, i32 1
720   store i1 true, i1* %"complex bool434", align 1
721   %casted_malloc436 = bitcast { i8**, { i8*, i8*, i32 }* }*
       %struct_space416 to i8*
722   store i8* %casted_malloc436, i8** %"complex e1435", align 8
723   %complex_bool_ptr437 = getelementptr inbounds { i1, i8*,
       i8*, i32 }, { i1, i8*, i8*, i32 }* %"complex exec
       struct433", i32 0, i32 0
724   %complex_bool438 = load i1, i1* %complex_bool_ptr437, align 1
725   %malloccall439 = tail call i8* @malloc(i32 ptrtoint (i8**
       getelementptr (i8*, i8** null, i32 1) to i32))
726   %return_str_ptr440 = bitcast i8* %malloccall439 to i8**
727   br i1 %complex_bool438, label %then442, label %else452
728
729 then401:                                          ; preds =
       %"run merge382"
730   %exec_ptr402 = getelementptr inbounds { i1, i8*, i8*, i32 },
       { i1, i8*, i8*, i32 }* %e5395, i32 0, i32 1
731   %cast_run403 = bitcast i8** %exec_ptr402 to { i8**, { i8*,
       i8*, i32 }* }**
732   %exec404 = load { i8**, { i8*, i8*, i32 }* }*, { i8**, {
       i8*, i8*, i32 }* }** %cast_run403, align 8
733   %dbl_path_ptr405 = getelementptr inbounds { i8**, { i8*,
       i8*, i32 }* }, { i8**, { i8*, i8*, i32 }* }* %exec404, i32
       0, i32 0
734   %path_ptr406 = load i8**, i8*** %dbl_path_ptr405, align 8
735   %path407 = load i8*, i8** %path_ptr406, align 8
736   %args_ptr408 = getelementptr inbounds { i8**, { i8*, i8*,
       i32 }* }, { i8**, { i8*, i8*, i32 }* }* %exec404, i32 0,
       i32 1
737   %args409 = load { i8*, i8*, i32 }*, { i8*, i8*, i32 }**
       %args_ptr408, align 8
738   %execvp410 = call i8* (i8*, { i8*, i8*, i32 }*, ...)
       @execvp_helper(i8* %path407, { i8*, i8*, i32 }* %args409)
739   store i8* %execvp410, i8** %return_str_ptr399, align 8
740   br label %"run merge400"
741
742 else411:                                          ; preds =
       %"run merge382"
```

```
743   %recurse_exec412 = call i8* ({ i1, i8*, i8*, i32 }*, ...)
        @recurse_exec({ i1, i8*, i8*, i32 }* %e5395)
744   store i8* %recurse_exec412, i8** %return_str_ptr399, align 8
745   br label %"run merge400"
746
747 "run merge441":                                   ; preds =
        %else452, %then442
748   %e4454 = load { i1, i8*, i8*, i32 }*, { i1, i8*, i8*, i32
        }** %"variable ptr101", align 8
749   %e2455 = load { i1, i8*, i8*, i32 }*, { i1, i8*, i8*, i32
        }** %"variable ptr31", align 8
750   %e1456 = load { i1, i8*, i8*, i32 }*, { i1, i8*, i8*, i32
        }** %"variable ptr", align 8
751   %malloccall457 = tail call i8* @malloc(i32 ptrtoint ({ i1,
        i8*, i8*, i32 }* getelementptr ({ i1, i8*, i8*, i32 }, {
        i1, i8*, i8*, i32 }* null, i32 1) to i32))
752   %"complex exec struct458" = bitcast i8* %malloccall457 to {
        i1, i8*, i8*, i32 }*
753   %"complex bool459" = getelementptr inbounds { i1, i8*, i8*,
        i32 }, { i1, i8*, i8*, i32 }* %"complex exec struct458",
        i32 0, i32 0
754   %"complex e1460" = getelementptr inbounds { i1, i8*, i8*,
        i32 }, { i1, i8*, i8*, i32 }* %"complex exec struct458",
        i32 0, i32 1
755   %"complex e2461" = getelementptr inbounds { i1, i8*, i8*,
        i32 }, { i1, i8*, i8*, i32 }* %"complex exec struct458",
        i32 0, i32 2
756   %"complex op462" = getelementptr inbounds { i1, i8*, i8*,
        i32 }, { i1, i8*, i8*, i32 }* %"complex exec struct458",
        i32 0, i32 3
757   store i1 false, i1* %"complex bool459", align 1
758   %casted_e1463 = bitcast { i1, i8*, i8*, i32 }* %e1456 to i8*
759   store i8* %casted_e1463, i8** %"complex e1460", align 8
760   %casted_e2464 = bitcast { i1, i8*, i8*, i32 }* %e2455 to i8*
761   store i8* %casted_e2464, i8** %"complex e2461", align 8
762   store i32 1, i32* %"complex op462", align 4
763   %malloccall465 = tail call i8* @malloc(i32 ptrtoint ({ i1,
        i8*, i8*, i32 }* getelementptr ({ i1, i8*, i8*, i32 }, {
        i1, i8*, i8*, i32 }* null, i32 1) to i32))
764   %"complex exec struct466" = bitcast i8* %malloccall465 to {
        i1, i8*, i8*, i32 }*
765   %"complex bool467" = getelementptr inbounds { i1, i8*, i8*,
        i32 }, { i1, i8*, i8*, i32 }* %"complex exec struct466",
        i32 0, i32 0
766   %"complex e1468" = getelementptr inbounds { i1, i8*, i8*,
        i32 }, { i1, i8*, i8*, i32 }* %"complex exec struct466",
        i32 0, i32 1
767   %"complex e2469" = getelementptr inbounds { i1, i8*, i8*,
        i32 }, { i1, i8*, i8*, i32 }* %"complex exec struct466",
        i32 0, i32 2
768   %"complex op470" = getelementptr inbounds { i1, i8*, i8*,
        i32 }, { i1, i8*, i8*, i32 }* %"complex exec struct466",
```

```
    i32 0, i32 3
  store i1 false, i1* %"complex bool467", align 1
  %casted_e1471 = bitcast { i1, i8*, i8*, i32 }* %"complex
    exec struct458" to i8*
  store i8* %casted_e1471, i8** %"complex e1468", align 8
  %casted_e2472 = bitcast { i1, i8*, i8*, i32 }* %e4454 to i8*
  store i8* %casted_e2472, i8** %"complex e2469", align 8
  store i32 2, i32* %"complex op470", align 4
  %malloccall473 = tail call i8* @malloc(i32 ptrtoint ({ i1,
    i8*, i8*, i32 }** getelementptr ({ i1, i8*, i8*, i32 }*, {
    i1, i8*, i8*, i32 }** null, i32 1) to i32))
  %"variable ptr474" = bitcast i8* %malloccall473 to { i1,
    i8*, i8*, i32 }**
  store { i1, i8*, i8*, i32 }* %"complex exec struct466", {
    i1, i8*, i8*, i32 }** %"variable ptr474", align 8
  %malloccall475 = tail call i8* @malloc(i32 ptrtoint ({ i8**,
    { i8*, i8*, i32 }* }* getelementptr ({ i8**, { i8*, i8*,
    i32 }* }, { i8**, { i8*, i8*, i32 }* }* null, i32 1) to
    i32))
  %struct_space476 = bitcast i8* %malloccall475 to { i8**, {
    i8*, i8*, i32 }* }*
  %path_ptr477 = getelementptr inbounds { i8**, { i8*, i8*,
    i32 }* }, { i8**, { i8*, i8*, i32 }* }* %struct_space476,
    i32 0, i32 0
  %malloccall478 = tail call i8* @malloc(i32 ptrtoint (i8**
    getelementptr (i8*, i8** null, i32 1) to i32))
  %double_string_ptr479 = bitcast i8* %malloccall478 to i8**
  store i8* getelementptr inbounds ([5 x i8], [5 x i8]*
    @string.22, i32 0, i32 0), i8** %double_string_ptr479,
    align 8
  store i8** %double_string_ptr479, i8*** %path_ptr477, align 8
  %args_ptr480 = getelementptr inbounds { i8**, { i8*, i8*,
    i32 }* }, { i8**, { i8*, i8*, i32 }* }* %struct_space476,
    i32 0, i32 1
  %malloccall481 = tail call i8* @malloc(i32 ptrtoint (i8**
    getelementptr (i8*, i8** null, i32 1) to i32))
  %double_string_ptr482 = bitcast i8* %malloccall481 to i8**
  store i8* getelementptr inbounds ([40 x i8], [40 x i8]*
    @string.23, i32 0, i32 0), i8** %double_string_ptr482,
    align 8
  %malloccall483 = tail call i8* @malloc(i32 ptrtoint (i8***
    getelementptr (i8**, i8*** null, i32 1) to i32))
  %value_ptr484 = bitcast i8* %malloccall483 to i8***
  store i8** %double_string_ptr482, i8*** %value_ptr484, align
    8
  %malloccall485 = tail call i8* @malloc(i32 ptrtoint ({ i8*,
    i8*, i32 }* getelementptr ({ i8*, i8*, i32 }, { i8*, i8*,
    i32 }* null, i32 1) to i32))
  %list_node486 = bitcast i8* %malloccall485 to { i8*, i8*,
    i32 }*
  %struct_val_ptr487 = getelementptr inbounds { i8*, i8*, i32
    }, { i8*, i8*, i32 }* %list_node486, i32 0, i32 0
```

```
795   %struct_ptr_ptr488 = getelementptr inbounds { i8*, i8*, i32
       }, { i8*, i8*, i32 }* %list_node486, i32 0, i32 1
796   %struct_ty_ptr489 = getelementptr inbounds { i8*, i8*, i32
       }, { i8*, i8*, i32 }* %list_node486, i32 0, i32 2
797   %casted_ptr_ptr490 = bitcast i8** %struct_ptr_ptr488 to {
       i8*, i8*, i32 }**
798   store { i8*, i8*, i32 }* null, { i8*, i8*, i32 }**
       %casted_ptr_ptr490, align 8
799   %casted_val491 = bitcast i8*** %value_ptr484 to i8*
800   store i8* %casted_val491, i8** %struct_val_ptr487, align 8
801   store i32 4, i32* %struct_ty_ptr489, align 4
802   store { i8*, i8*, i32 }* %list_node486, { i8*, i8*, i32 }**
       %args_ptr480, align 8
803   %malloccall492 = tail call i8* @malloc(i32 ptrtoint ({ i1,
       i8*, i8*, i32 }* getelementptr ({ i1, i8*, i8*, i32 }, {
       i1, i8*, i8*, i32 }* null, i32 1) to i32))
804   %"complex exec struct493" = bitcast i8* %malloccall492 to {
       i1, i8*, i8*, i32 }*
805   %"complex bool494" = getelementptr inbounds { i1, i8*, i8*,
       i32 }, { i1, i8*, i8*, i32 }* %"complex exec struct493",
       i32 0, i32 0
806   %"complex e1495" = getelementptr inbounds { i1, i8*, i8*,
       i32 }, { i1, i8*, i8*, i32 }* %"complex exec struct493",
       i32 0, i32 1
807   store i1 true, i1* %"complex bool494", align 1
808   %casted_malloc496 = bitcast { i8**, { i8*, i8*, i32 }* }*
       %struct_space476 to i8*
809   store i8* %casted_malloc496, i8** %"complex e1495", align 8
810   %complex_bool_ptr497 = getelementptr inbounds { i1, i8*,
       i8*, i32 }, { i1, i8*, i8*, i32 }* %"complex exec
       struct493", i32 0, i32 0
811   %complex_bool498 = load i1, i1* %complex_bool_ptr497, align 1
812   %malloccall499 = tail call i8* @malloc(i32 ptrtoint (i8**
       getelementptr (i8*, i8** null, i32 1) to i32))
813   %return_str_ptr500 = bitcast i8* %malloccall499 to i8**
814   br i1 %complex_bool498, label %then502, label %else512
815
816 then442:                                          ; preds =
       %"run merge400"
817   %exec_ptr443 = getelementptr inbounds { i1, i8*, i8*, i32 },
       { i1, i8*, i8*, i32 }* %"complex exec struct433", i32 0,
       i32 1
818   %cast_run444 = bitcast i8** %exec_ptr443 to { i8**, { i8*,
       i8*, i32 }* }**
819   %exec445 = load { i8**, { i8*, i8*, i32 }* }*, { i8**, {
       i8*, i8*, i32 }* }** %cast_run444, align 8
820   %dbl_path_ptr446 = getelementptr inbounds { i8**, { i8*,
       i8*, i32 }* }, { i8**, { i8*, i8*, i32 }* }* %exec445, i32
       0, i32 0
821   %path_ptr447 = load i8**, i8*** %dbl_path_ptr446, align 8
822   %path448 = load i8*, i8** %path_ptr447, align 8
```

```
823   %args_ptr449 = getelementptr inbounds { i8**, { i8*, i8*,
        i32 }* }, { i8**, { i8*, i8*, i32 }* }* %exec445, i32 0,
        i32 1
824   %args450 = load { i8*, i8*, i32 }*, { i8*, i8*, i32 }**
        %args_ptr449, align 8
825   %execvp451 = call i8* (i8*, { i8*, i8*, i32 }*, ...)
        @execvp_helper(i8* %path448, { i8*, i8*, i32 }* %args450)
826   store i8* %execvp451, i8** %return_str_ptr440, align 8
827   br label %"run merge441"
828
829 else452:                                          ; preds =
        %"run merge400"
830   %recurse_exec453 = call i8* ({ i1, i8*, i8*, i32 }*, ...)
        @recurse_exec({ i1, i8*, i8*, i32 }* %"complex exec
        struct433")
831   store i8* %recurse_exec453, i8** %return_str_ptr440, align 8
832   br label %"run merge441"
833
834 "run merge501":                                   ; preds =
        %else512, %then502
835   %e5514 = load { i1, i8*, i8*, i32 }*, { i1, i8*, i8*, i32
        }** %"variable ptr474", align 8
836   %complex_bool_ptr515 = getelementptr inbounds { i1, i8*,
        i8*, i32 }, { i1, i8*, i8*, i32 }* %e5514, i32 0, i32 0
837   %complex_bool516 = load i1, i1* %complex_bool_ptr515, align 1
838   %malloccall517 = tail call i8* @malloc(i32 ptrtoint (i8**
        getelementptr (i8*, i8** null, i32 1) to i32))
839   %return_str_ptr518 = bitcast i8* %malloccall517 to i8**
840   br i1 %complex_bool516, label %then520, label %else530
841
842 then502:                                          ; preds =
        %"run merge441"
843   %exec_ptr503 = getelementptr inbounds { i1, i8*, i8*, i32 },
        { i1, i8*, i8*, i32 }* %"complex exec struct493", i32 0,
        i32 1
844   %cast_run504 = bitcast i8** %exec_ptr503 to { i8**, { i8*,
        i8*, i32 }* }**
845   %exec505 = load { i8**, { i8*, i8*, i32 }* }*, { i8**, {
        i8*, i8*, i32 }* }** %cast_run504, align 8
846   %dbl_path_ptr506 = getelementptr inbounds { i8**, { i8*,
        i8*, i32 }* }, { i8**, { i8*, i8*, i32 }* }* %exec505, i32
        0, i32 0
847   %path_ptr507 = load i8**, i8*** %dbl_path_ptr506, align 8
848   %path508 = load i8*, i8** %path_ptr507, align 8
849   %args_ptr509 = getelementptr inbounds { i8**, { i8*, i8*,
        i32 }* }, { i8**, { i8*, i8*, i32 }* }* %exec505, i32 0,
        i32 1
850   %args510 = load { i8*, i8*, i32 }*, { i8*, i8*, i32 }**
        %args_ptr509, align 8
851   %execvp511 = call i8* (i8*, { i8*, i8*, i32 }*, ...)
        @execvp_helper(i8* %path508, { i8*, i8*, i32 }* %args510)
852   store i8* %execvp511, i8** %return_str_ptr500, align 8
```

```
853   br label %"run merge501"
854
855 else512:                                          ; preds =
      %"run merge441"
856   %recurse_exec513 = call i8* ({ i1, i8*, i8*, i32 }*, ...)
      @recurse_exec({ i1, i8*, i8*, i32 }* %"complex exec
      struct493")
857   store i8* %recurse_exec513, i8** %return_str_ptr500, align 8
858   br label %"run merge501"
859
860 "run merge519":                                   ; preds =
      %else530, %then520
861   %malloccall532 = tail call i8* @malloc(i32 ptrtoint (i8***
      getelementptr (i8**, i8*** null, i32 1) to i32))
862   %"variable ptr533" = bitcast i8* %malloccall532 to i8***
863   store i8** %return_str_ptr518, i8*** %"variable ptr533",
      align 8
864   %malloccall534 = tail call i8* @malloc(i32 ptrtoint ({ i8**,
      { i8*, i8*, i32 }* }* getelementptr ({ i8**, { i8*, i8*,
      i32 }* }, { i8**, { i8*, i8*, i32 }* }* null, i32 1) to
      i32))
865   %struct_space535 = bitcast i8* %malloccall534 to { i8**, {
      i8*, i8*, i32 }* }*
866   %path_ptr536 = getelementptr inbounds { i8**, { i8*, i8*,
      i32 }* }, { i8**, { i8*, i8*, i32 }* }* %struct_space535,
      i32 0, i32 0
867   %malloccall537 = tail call i8* @malloc(i32 ptrtoint (i8**
      getelementptr (i8*, i8** null, i32 1) to i32))
868   %double_string_ptr538 = bitcast i8* %malloccall537 to i8**
869   store i8* getelementptr inbounds ([5 x i8], [5 x i8]*
      @string.24, i32 0, i32 0), i8** %double_string_ptr538,
      align 8
870   store i8** %double_string_ptr538, i8*** %path_ptr536, align 8
871   %args_ptr539 = getelementptr inbounds { i8**, { i8*, i8*,
      i32 }* }, { i8**, { i8*, i8*, i32 }* }* %struct_space535,
      i32 0, i32 1
872   %malloccall540 = tail call i8* @malloc(i32 ptrtoint (i8**
      getelementptr (i8*, i8** null, i32 1) to i32))
873   %double_string_ptr541 = bitcast i8* %malloccall540 to i8**
874   store i8* getelementptr inbounds ([15 x i8], [15 x i8]*
      @string.25, i32 0, i32 0), i8** %double_string_ptr541,
      align 8
875   %malloccall542 = tail call i8* @malloc(i32 ptrtoint (i8***
      getelementptr (i8**, i8*** null, i32 1) to i32))
876   %value_ptr543 = bitcast i8* %malloccall542 to i8***
877   store i8** %double_string_ptr541, i8*** %value_ptr543, align
      8
878   %malloccall544 = tail call i8* @malloc(i32 ptrtoint ({ i8*,
      i8*, i32 }* getelementptr ({ i8*, i8*, i32 }, { i8*, i8*,
      i32 }* null, i32 1) to i32))
879   %list_node545 = bitcast i8* %malloccall544 to { i8*, i8*,
      i32 }*
```

```
880   %struct_val_ptr546 = getelementptr inbounds { i8*, i8*, i32
       }, { i8*, i8*, i32 }* %list_node545, i32 0, i32 0
881   %struct_ptr_ptr547 = getelementptr inbounds { i8*, i8*, i32
       }, { i8*, i8*, i32 }* %list_node545, i32 0, i32 1
882   %struct_ty_ptr548 = getelementptr inbounds { i8*, i8*, i32
       }, { i8*, i8*, i32 }* %list_node545, i32 0, i32 2
883   %casted_ptr_ptr549 = bitcast i8** %struct_ptr_ptr547 to {
       i8*, i8*, i32 }**
884   store { i8*, i8*, i32 }* null, { i8*, i8*, i32 }**
       %casted_ptr_ptr549, align 8
885   %casted_val550 = bitcast i8*** %value_ptr543 to i8*
886   store i8* %casted_val550, i8** %struct_val_ptr546, align 8
887   store i32 4, i32* %struct_ty_ptr548, align 4
888   store { i8*, i8*, i32 }* %list_node545, { i8*, i8*, i32 }**
       %args_ptr539, align 8
889   %malloccall551 = tail call i8* @malloc(i32 ptrtoint ({ i1,
       i8*, i8*, i32 }* getelementptr ({ i1, i8*, i8*, i32 }, {
       i1, i8*, i8*, i32 }* null, i32 1) to i32))
890   %"complex exec struct552" = bitcast i8* %malloccall551 to {
       i1, i8*, i8*, i32 }*
891   %"complex bool553" = getelementptr inbounds { i1, i8*, i8*,
       i32 }, { i1, i8*, i8*, i32 }* %"complex exec struct552",
       i32 0, i32 0
892   %"complex e1554" = getelementptr inbounds { i1, i8*, i8*,
       i32 }, { i1, i8*, i8*, i32 }* %"complex exec struct552",
       i32 0, i32 1
893   store i1 true, i1* %"complex bool553", align 1
894   %casted_malloc555 = bitcast { i8**, { i8*, i8*, i32 }* }*
       %struct_space535 to i8*
895   store i8* %casted_malloc555, i8** %"complex e1554", align 8
896   %complex_bool_ptr556 = getelementptr inbounds { i1, i8*,
       i8*, i32 }, { i1, i8*, i8*, i32 }* %"complex exec
       struct552", i32 0, i32 0
897   %complex_bool557 = load i1, i1* %complex_bool_ptr556, align 1
898   %malloccall558 = tail call i8* @malloc(i32 ptrtoint (i8**
       getelementptr (i8*, i8** null, i32 1) to i32))
899   %return_str_ptr559 = bitcast i8* %malloccall558 to i8**
900   br i1 %complex_bool557, label %then561, label %else571
901
902 then520:                                          ; preds =
       %"run merge501"
903   %exec_ptr521 = getelementptr inbounds { i1, i8*, i8*, i32 },
       { i1, i8*, i8*, i32 }* %e5514, i32 0, i32 1
904   %cast_run522 = bitcast i8** %exec_ptr521 to { i8**, { i8*,
       i8*, i32 }* }**
905   %exec523 = load { i8**, { i8*, i8*, i32 }* }*, { i8**, {
       i8*, i8*, i32 }* }** %cast_run522, align 8
906   %dbl_path_ptr524 = getelementptr inbounds { i8**, { i8*,
       i8*, i32 }* }, { i8**, { i8*, i8*, i32 }* }* %exec523, i32
       0, i32 0
907   %path_ptr525 = load i8**, i8*** %dbl_path_ptr524, align 8
908   %path526 = load i8*, i8** %path_ptr525, align 8
```

```
909   %args_ptr527 = getelementptr inbounds { i8**, { i8*, i8*,
       i32 }* }, { i8**, { i8*, i8*, i32 }* }* %exec523, i32 0,
       i32 1
910   %args528 = load { i8*, i8*, i32 }*, { i8*, i8*, i32 }**
       %args_ptr527, align 8
911   %execvp529 = call i8* (i8*, { i8*, i8*, i32 }*, ...)
       @execvp_helper(i8* %path526, { i8*, i8*, i32 }* %args528)
912   store i8* %execvp529, i8** %return_str_ptr518, align 8
913   br label %"run merge519"
914
915 else530:                                          ; preds =
       %"run merge501"
916   %recurse_exec531 = call i8* ({ i1, i8*, i8*, i32 }*, ...)
       @recurse_exec({ i1, i8*, i8*, i32 }* %e5514)
917   store i8* %recurse_exec531, i8** %return_str_ptr518, align 8
918   br label %"run merge519"
919
920 "run merge560":                                   ; preds =
       %else571, %then561
921   %malloccall573 = tail call i8* @malloc(i32 ptrtoint ({ i8**,
       { i8*, i8*, i32 }* }* getelementptr ({ i8**, { i8*, i8*,
       i32 }* }, { i8**, { i8*, i8*, i32 }* }* null, i32 1) to
       i32))
922   %struct_space574 = bitcast i8* %malloccall573 to { i8**, {
       i8*, i8*, i32 }* }*
923   %path_ptr575 = getelementptr inbounds { i8**, { i8*, i8*,
       i32 }* }, { i8**, { i8*, i8*, i32 }* }* %struct_space574,
       i32 0, i32 0
924   %malloccall576 = tail call i8* @malloc(i32 ptrtoint (i8**
       getelementptr (i8*, i8** null, i32 1) to i32))
925   %double_string_ptr577 = bitcast i8* %malloccall576 to i8**
926   store i8* getelementptr inbounds ([5 x i8], [5 x i8]*
       @string.26, i32 0, i32 0), i8** %double_string_ptr577,
       align 8
927   store i8** %double_string_ptr577, i8*** %path_ptr575, align 8
928   %args_ptr578 = getelementptr inbounds { i8**, { i8*, i8*,
       i32 }* }, { i8**, { i8*, i8*, i32 }* }* %struct_space574,
       i32 0, i32 1
929   %malloccall579 = tail call i8* @malloc(i32 ptrtoint (i8**
       getelementptr (i8*, i8** null, i32 1) to i32))
930   %double_string_ptr580 = bitcast i8* %malloccall579 to i8**
931   store i8* getelementptr inbounds ([12 x i8], [12 x i8]*
       @string.27, i32 0, i32 0), i8** %double_string_ptr580,
       align 8
932   %malloccall581 = tail call i8* @malloc(i32 ptrtoint (i8***
       getelementptr (i8**, i8*** null, i32 1) to i32))
933   %value_ptr582 = bitcast i8* %malloccall581 to i8***
934   store i8** %double_string_ptr580, i8*** %value_ptr582, align
       8
935   %malloccall583 = tail call i8* @malloc(i32 ptrtoint ({ i8*,
       i8*, i32 }* getelementptr ({ i8*, i8*, i32 }, { i8*, i8*,
       i32 }* null, i32 1) to i32))
```

```
936   %list_node584 = bitcast i8* %malloccall583 to { i8*, i8*,
       i32 }*
937   %struct_val_ptr585 = getelementptr inbounds { i8*, i8*, i32
       }, { i8*, i8*, i32 }* %list_node584, i32 0, i32 0
938   %struct_ptr_ptr586 = getelementptr inbounds { i8*, i8*, i32
       }, { i8*, i8*, i32 }* %list_node584, i32 0, i32 1
939   %struct_ty_ptr587 = getelementptr inbounds { i8*, i8*, i32
       }, { i8*, i8*, i32 }* %list_node584, i32 0, i32 2
940   %casted_ptr_ptr588 = bitcast i8** %struct_ptr_ptr586 to {
       i8*, i8*, i32 }**
941   store { i8*, i8*, i32 }* null, { i8*, i8*, i32 }**
       %casted_ptr_ptr588, align 8
942   %casted_val589 = bitcast i8*** %value_ptr582 to i8*
943   store i8* %casted_val589, i8** %struct_val_ptr585, align 8
944   store i32 4, i32* %struct_ty_ptr587, align 4
945   store { i8*, i8*, i32 }* %list_node584, { i8*, i8*, i32 }**
       %args_ptr578, align 8
946   %malloccall590 = tail call i8* @malloc(i32 ptrtoint ({ i1,
       i8*, i8*, i32 }* getelementptr ({ i1, i8*, i8*, i32 }, {
       i1, i8*, i8*, i32 }* null, i32 1) to i32))
947   %"complex exec struct591" = bitcast i8* %malloccall590 to {
       i1, i8*, i8*, i32 }*
948   %"complex bool592" = getelementptr inbounds { i1, i8*, i8*,
       i32 }, { i1, i8*, i8*, i32 }* %"complex exec struct591",
       i32 0, i32 0
949   %"complex e1593" = getelementptr inbounds { i1, i8*, i8*,
       i32 }, { i1, i8*, i8*, i32 }* %"complex exec struct591",
       i32 0, i32 1
950   store i1 true, i1* %"complex bool592", align 1
951   %casted_malloc594 = bitcast { i8**, { i8*, i8*, i32 }* }*
       %struct_space574 to i8*
952   store i8* %casted_malloc594, i8** %"complex e1593", align 8
953   %malloccall595 = tail call i8* @malloc(i32 ptrtoint ({ i1,
       i8*, i8*, i32 }** getelementptr ({ i1, i8*, i8*, i32 }*, {
       i1, i8*, i8*, i32 }** null, i32 1) to i32))
954   %"variable ptr596" = bitcast i8* %malloccall595 to { i1,
       i8*, i8*, i32 }**
955   store { i1, i8*, i8*, i32 }* %"complex exec struct591", {
       i1, i8*, i8*, i32 }** %"variable ptr596", align 8
956   %malloccall597 = tail call i8* @malloc(i32 ptrtoint ({ i8**,
       { i8*, i8*, i32 }* }* getelementptr ({ i8**, { i8*, i8*,
       i32 }* }, { i8**, { i8*, i8*, i32 }* }* null, i32 1) to
       i32))
957   %struct_space598 = bitcast i8* %malloccall597 to { i8**, {
       i8*, i8*, i32 }* }*
958   %path_ptr599 = getelementptr inbounds { i8**, { i8*, i8*,
       i32 }* }, { i8**, { i8*, i8*, i32 }* }* %struct_space598,
       i32 0, i32 0
959   %malloccall600 = tail call i8* @malloc(i32 ptrtoint (i8**
       getelementptr (i8*, i8** null, i32 1) to i32))
960   %double_string_ptr601 = bitcast i8* %malloccall600 to i8**
```

```
961   store i8* getelementptr inbounds ([5 x i8], [5 x i8]*
       @string.28, i32 0, i32 0), i8** %double_string_ptr601,
       align 8
962   store i8** %double_string_ptr601, i8*** %path_ptr599, align 8
963   %args_ptr602 = getelementptr inbounds { i8**, { i8*, i8*,
       i32 }* }, { i8**, { i8*, i8*, i32 }* }* %struct_space598,
       i32 0, i32 1
964   %malloccall603 = tail call i8* @malloc(i32 ptrtoint (i8**
       getelementptr (i8*, i8** null, i32 1) to i32))
965   %double_string_ptr604 = bitcast i8* %malloccall603 to i8**
966   store i8* getelementptr inbounds ([3 x i8], [3 x i8]*
       @string.29, i32 0, i32 0), i8** %double_string_ptr604,
       align 8
967   %malloccall605 = tail call i8* @malloc(i32 ptrtoint (i8***
       getelementptr (i8**, i8*** null, i32 1) to i32))
968   %value_ptr606 = bitcast i8* %malloccall605 to i8***
969   store i8** %double_string_ptr604, i8*** %value_ptr606, align
       8
970   %malloccall607 = tail call i8* @malloc(i32 ptrtoint ({ i8*,
       i8*, i32 }* getelementptr ({ i8*, i8*, i32 }, { i8*, i8*,
       i32 }* null, i32 1) to i32))
971   %list_node608 = bitcast i8* %malloccall607 to { i8*, i8*,
       i32 }*
972   %struct_val_ptr609 = getelementptr inbounds { i8*, i8*, i32
       }, { i8*, i8*, i32 }* %list_node608, i32 0, i32 0
973   %struct_ptr_ptr610 = getelementptr inbounds { i8*, i8*, i32
       }, { i8*, i8*, i32 }* %list_node608, i32 0, i32 1
974   %struct_ty_ptr611 = getelementptr inbounds { i8*, i8*, i32
       }, { i8*, i8*, i32 }* %list_node608, i32 0, i32 2
975   %malloccall612 = tail call i8* @malloc(i32 ptrtoint (i8**
       getelementptr (i8*, i8** null, i32 1) to i32))
976   %double_string_ptr613 = bitcast i8* %malloccall612 to i8**
977   store i8* getelementptr inbounds ([6 x i8], [6 x i8]*
       @string.30, i32 0, i32 0), i8** %double_string_ptr613,
       align 8
978   %malloccall614 = tail call i8* @malloc(i32 ptrtoint (i8***
       getelementptr (i8**, i8*** null, i32 1) to i32))
979   %value_ptr615 = bitcast i8* %malloccall614 to i8***
980   store i8** %double_string_ptr613, i8*** %value_ptr615, align
       8
981   %malloccall616 = tail call i8* @malloc(i32 ptrtoint ({ i8*,
       i8*, i32 }* getelementptr ({ i8*, i8*, i32 }, { i8*, i8*,
       i32 }* null, i32 1) to i32))
982   %list_node617 = bitcast i8* %malloccall616 to { i8*, i8*,
       i32 }*
983   %struct_val_ptr618 = getelementptr inbounds { i8*, i8*, i32
       }, { i8*, i8*, i32 }* %list_node617, i32 0, i32 0
984   %struct_ptr_ptr619 = getelementptr inbounds { i8*, i8*, i32
       }, { i8*, i8*, i32 }* %list_node617, i32 0, i32 1
985   %struct_ty_ptr620 = getelementptr inbounds { i8*, i8*, i32
       }, { i8*, i8*, i32 }* %list_node617, i32 0, i32 2
```

```
  %casted_ptr_ptr621 = bitcast i8** %struct_ptr_ptr619 to { i8*, i8*, i32 }**
  store { i8*, i8*, i32 }* null, { i8*, i8*, i32 }** %casted_ptr_ptr621, align 8
  %casted_val622 = bitcast i8*** %value_ptr615 to i8*
  store i8* %casted_val622, i8** %struct_val_ptr618, align 8
  store i32 4, i32* %struct_ty_ptr620, align 4
  %casted_ptr_ptr623 = bitcast i8** %struct_ptr_ptr610 to { i8*, i8*, i32 }**
  store { i8*, i8*, i32 }* %list_node617, { i8*, i8*, i32 }** %casted_ptr_ptr623, align 8
  %casted_val624 = bitcast i8*** %value_ptr606 to i8*
  store i8* %casted_val624, i8** %struct_val_ptr609, align 8
  store i32 4, i32* %struct_ty_ptr611, align 4
  store { i8*, i8*, i32 }* %list_node608, { i8*, i8*, i32 }** %args_ptr602, align 8
  %malloccall625 = tail call i8* @malloc(i32 ptrtoint ({ i1, i8*, i8*, i32 }* getelementptr ({ i1, i8*, i8*, i32 }, { i1, i8*, i8*, i32 }* null, i32 1) to i32))
  %"complex exec struct626" = bitcast i8* %malloccall625 to { i1, i8*, i8*, i32 }*
  %"complex bool627" = getelementptr inbounds { i1, i8*, i8*, i32 }, { i1, i8*, i8*, i32 }* %"complex exec struct626", i32 0, i32 0
  %"complex e1628" = getelementptr inbounds { i1, i8*, i8*, i32 }, { i1, i8*, i8*, i32 }* %"complex exec struct626", i32 0, i32 1
  store i1 true, i1* %"complex bool627", align 1
  %casted_malloc629 = bitcast { i8**, { i8*, i8*, i32 }* }* %struct_space598 to i8*
  store i8* %casted_malloc629, i8** %"complex e1628", align 8
  %e6 = load { i1, i8*, i8*, i32 }*, { i1, i8*, i8*, i32 }** %"variable ptr596", align 8
  %e6630 = load { i1, i8*, i8*, i32 }*, { i1, i8*, i8*, i32 }** %"variable ptr596", align 8
  %malloccall631 = tail call i8* @malloc(i32 ptrtoint ({ i1, i8*, i8*, i32 }* getelementptr ({ i1, i8*, i8*, i32 }, { i1, i8*, i8*, i32 }* null, i32 1) to i32))
  %"complex exec struct632" = bitcast i8* %malloccall631 to { i1, i8*, i8*, i32 }*
  %"complex bool633" = getelementptr inbounds { i1, i8*, i8*, i32 }, { i1, i8*, i8*, i32 }* %"complex exec struct632", i32 0, i32 0
  %"complex e1634" = getelementptr inbounds { i1, i8*, i8*, i32 }, { i1, i8*, i8*, i32 }* %"complex exec struct632", i32 0, i32 1
  %"complex e2635" = getelementptr inbounds { i1, i8*, i8*, i32 }, { i1, i8*, i8*, i32 }* %"complex exec struct632", i32 0, i32 2
  %"complex op636" = getelementptr inbounds { i1, i8*, i8*, i32 }, { i1, i8*, i8*, i32 }* %"complex exec struct632", i32 0, i32 3
```

```
  store i1 false, i1* %"complex bool633", align 1
  %casted_e1637 = bitcast { i1, i8*, i8*, i32 }* %e6630 to i8*
  store i8* %casted_e1637, i8** %"complex e1634", align 8
  %casted_e2638 = bitcast { i1, i8*, i8*, i32 }* %e6 to i8*
  store i8* %casted_e2638, i8** %"complex e2635", align 8
  store i32 0, i32* %"complex op636", align 4
  %e3639 = load { i1, i8*, i8*, i32 }*, { i1, i8*, i8*, i32
   }** %"variable ptr66", align 8
  %malloccall640 = tail call i8* @malloc(i32 ptrtoint ({ i1,
   i8*, i8*, i32 }* getelementptr ({ i1, i8*, i8*, i32 }, {
   i1, i8*, i8*, i32 }* null, i32 1) to i32))
  %"complex exec struct641" = bitcast i8* %malloccall640 to {
   i1, i8*, i8*, i32 }*
  %"complex bool642" = getelementptr inbounds { i1, i8*, i8*,
   i32 }, { i1, i8*, i8*, i32 }* %"complex exec struct641",
   i32 0, i32 0
  %"complex e1643" = getelementptr inbounds { i1, i8*, i8*,
   i32 }, { i1, i8*, i8*, i32 }* %"complex exec struct641",
   i32 0, i32 1
  %"complex e2644" = getelementptr inbounds { i1, i8*, i8*,
   i32 }, { i1, i8*, i8*, i32 }* %"complex exec struct641",
   i32 0, i32 2
  %"complex op645" = getelementptr inbounds { i1, i8*, i8*,
   i32 }, { i1, i8*, i8*, i32 }* %"complex exec struct641",
   i32 0, i32 3
  store i1 false, i1* %"complex bool642", align 1
  %casted_e1646 = bitcast { i1, i8*, i8*, i32 }* %e3639 to i8*
  store i8* %casted_e1646, i8** %"complex e1643", align 8
  %casted_e2647 = bitcast { i1, i8*, i8*, i32 }* %"complex
   exec struct632" to i8*
  store i8* %casted_e2647, i8** %"complex e2644", align 8
  store i32 0, i32* %"complex op645", align 4
  %e1648 = load { i1, i8*, i8*, i32 }*, { i1, i8*, i8*, i32
   }** %"variable ptr", align 8
  %e2649 = load { i1, i8*, i8*, i32 }*, { i1, i8*, i8*, i32
   }** %"variable ptr31", align 8
  %malloccall650 = tail call i8* @malloc(i32 ptrtoint ({ i1,
   i8*, i8*, i32 }* getelementptr ({ i1, i8*, i8*, i32 }, {
   i1, i8*, i8*, i32 }* null, i32 1) to i32))
  %"complex exec struct651" = bitcast i8* %malloccall650 to {
   i1, i8*, i8*, i32 }*
  %"complex bool652" = getelementptr inbounds { i1, i8*, i8*,
   i32 }, { i1, i8*, i8*, i32 }* %"complex exec struct651",
   i32 0, i32 0
  %"complex e1653" = getelementptr inbounds { i1, i8*, i8*,
   i32 }, { i1, i8*, i8*, i32 }* %"complex exec struct651",
   i32 0, i32 1
  %"complex e2654" = getelementptr inbounds { i1, i8*, i8*,
   i32 }, { i1, i8*, i8*, i32 }* %"complex exec struct651",
   i32 0, i32 2
  %"complex op655" = getelementptr inbounds { i1, i8*, i8*,
   i32 }, { i1, i8*, i8*, i32 }* %"complex exec struct651",
```

```
  i32 0, i32 3
  store i1 false, i1* %"complex bool652", align 1
  %casted_e1656 = bitcast { i1, i8*, i8*, i32 }* %e2649 to i8*
  store i8* %casted_e1656, i8** %"complex e1653", align 8
  %casted_e2657 = bitcast { i1, i8*, i8*, i32 }* %e1648 to i8*
  store i8* %casted_e2657, i8** %"complex e2654", align 8
  store i32 1, i32* %"complex op655", align 4
  %malloccall658 = tail call i8* @malloc(i32 ptrtoint ({ i1,
   i8*, i8*, i32 }* getelementptr ({ i1, i8*, i8*, i32 }, {
   i1, i8*, i8*, i32 }* null, i32 1) to i32))
  %"complex exec struct659" = bitcast i8* %malloccall658 to {
   i1, i8*, i8*, i32 }*
  %"complex bool660" = getelementptr inbounds { i1, i8*, i8*,
   i32 }, { i1, i8*, i8*, i32 }* %"complex exec struct659",
   i32 0, i32 0
  %"complex e1661" = getelementptr inbounds { i1, i8*, i8*,
   i32 }, { i1, i8*, i8*, i32 }* %"complex exec struct659",
   i32 0, i32 1
  %"complex e2662" = getelementptr inbounds { i1, i8*, i8*,
   i32 }, { i1, i8*, i8*, i32 }* %"complex exec struct659",
   i32 0, i32 2
  %"complex op663" = getelementptr inbounds { i1, i8*, i8*,
   i32 }, { i1, i8*, i8*, i32 }* %"complex exec struct659",
   i32 0, i32 3
  store i1 false, i1* %"complex bool660", align 1
  %casted_e1664 = bitcast { i1, i8*, i8*, i32 }* %"complex
   exec struct651" to i8*
  store i8* %casted_e1664, i8** %"complex e1661", align 8
  %casted_e2665 = bitcast { i1, i8*, i8*, i32 }* %"complex
   exec struct641" to i8*
  store i8* %casted_e2665, i8** %"complex e2662", align 8
  store i32 2, i32* %"complex op663", align 4
  %malloccall666 = tail call i8* @malloc(i32 ptrtoint ({ i1,
   i8*, i8*, i32 }* getelementptr ({ i1, i8*, i8*, i32 }, {
   i1, i8*, i8*, i32 }* null, i32 1) to i32))
  %"complex exec struct667" = bitcast i8* %malloccall666 to {
   i1, i8*, i8*, i32 }*
  %"complex bool668" = getelementptr inbounds { i1, i8*, i8*,
   i32 }, { i1, i8*, i8*, i32 }* %"complex exec struct667",
   i32 0, i32 0
  %"complex e1669" = getelementptr inbounds { i1, i8*, i8*,
   i32 }, { i1, i8*, i8*, i32 }* %"complex exec struct667",
   i32 0, i32 1
  %"complex e2670" = getelementptr inbounds { i1, i8*, i8*,
   i32 }, { i1, i8*, i8*, i32 }* %"complex exec struct667",
   i32 0, i32 2
  %"complex op671" = getelementptr inbounds { i1, i8*, i8*,
   i32 }, { i1, i8*, i8*, i32 }* %"complex exec struct667",
   i32 0, i32 3
  store i1 false, i1* %"complex bool668", align 1
  %casted_e1672 = bitcast { i1, i8*, i8*, i32 }* %"complex
   exec struct659" to i8*
```

```
  store i8* %casted_e1672, i8** %"complex e1669", align 8
  %casted_e2673 = bitcast { i1, i8*, i8*, i32 }* %"complex
   exec struct626" to i8*
  store i8* %casted_e2673, i8** %"complex e2670", align 8
  store i32 2, i32* %"complex op671", align 4
  %malloccall674 = tail call i8* @malloc(i32 ptrtoint ({ i1,
   i8*, i8*, i32 }** getelementptr ({ i1, i8*, i8*, i32 }*, {
   i1, i8*, i8*, i32 }** null, i32 1) to i32))
  %"variable ptr675" = bitcast i8* %malloccall674 to { i1,
   i8*, i8*, i32 }**
  store { i1, i8*, i8*, i32 }* %"complex exec struct667", {
   i1, i8*, i8*, i32 }** %"variable ptr675", align 8
  %malloccall676 = tail call i8* @malloc(i32 ptrtoint ({ i8**,
   { i8*, i8*, i32 }* }* getelementptr ({ i8**, { i8*, i8*,
   i32 }* }, { i8**, { i8*, i8*, i32 }* }* null, i32 1) to
   i32))
  %struct_space677 = bitcast i8* %malloccall676 to { i8**, {
   i8*, i8*, i32 }* }*
  %path_ptr678 = getelementptr inbounds { i8**, { i8*, i8*,
   i32 }* }, { i8**, { i8*, i8*, i32 }* }* %struct_space677,
   i32 0, i32 0
  %malloccall679 = tail call i8* @malloc(i32 ptrtoint (i8**
   getelementptr (i8*, i8** null, i32 1) to i32))
  %double_string_ptr680 = bitcast i8* %malloccall679 to i8**
  store i8* getelementptr inbounds ([5 x i8], [5 x i8]*
   @string.31, i32 0, i32 0), i8** %double_string_ptr680,
   align 8
  store i8** %double_string_ptr680, i8*** %path_ptr678, align 8
  %args_ptr681 = getelementptr inbounds { i8**, { i8*, i8*,
   i32 }* }, { i8**, { i8*, i8*, i32 }* }* %struct_space677,
   i32 0, i32 1
  %malloccall682 = tail call i8* @malloc(i32 ptrtoint (i8**
   getelementptr (i8*, i8** null, i32 1) to i32))
  %double_string_ptr683 = bitcast i8* %malloccall682 to i8**
  store i8* getelementptr inbounds ([51 x i8], [51 x i8]*
   @string.32, i32 0, i32 0), i8** %double_string_ptr683,
   align 8
  %malloccall684 = tail call i8* @malloc(i32 ptrtoint (i8***
   getelementptr (i8**, i8*** null, i32 1) to i32))
  %value_ptr685 = bitcast i8* %malloccall684 to i8***
  store i8** %double_string_ptr683, i8*** %value_ptr685, align
   8
  %malloccall686 = tail call i8* @malloc(i32 ptrtoint ({ i8*,
   i8*, i32 }* getelementptr ({ i8*, i8*, i32 }, { i8*, i8*,
   i32 }* null, i32 1) to i32))
  %list_node687 = bitcast i8* %malloccall686 to { i8*, i8*,
   i32 }*
  %struct_val_ptr688 = getelementptr inbounds { i8*, i8*, i32
   }, { i8*, i8*, i32 }* %list_node687, i32 0, i32 0
  %struct_ptr_ptr689 = getelementptr inbounds { i8*, i8*, i32
   }, { i8*, i8*, i32 }* %list_node687, i32 0, i32 1
```

```
  %struct_ty_ptr690 = getelementptr inbounds { i8*, i8*, i32 }, { i8*, i8*, i32 }* %list_node687, i32 0, i32 2
  %casted_ptr_ptr691 = bitcast i8** %struct_ptr_ptr689 to { i8*, i8*, i32 }**
  store { i8*, i8*, i32 }* null, { i8*, i8*, i32 }** %casted_ptr_ptr691, align 8
  %casted_val692 = bitcast i8*** %value_ptr685 to i8*
  store i8* %casted_val692, i8** %struct_val_ptr688, align 8
  store i32 4, i32* %struct_ty_ptr690, align 4
  store { i8*, i8*, i32 }* %list_node687, { i8*, i8*, i32 }** %args_ptr681, align 8
  %malloccall693 = tail call i8* @malloc(i32 ptrtoint ({ i1, i8*, i8*, i32 }* getelementptr ({ i1, i8*, i8*, i32 }, { i1, i8*, i8*, i32 }* null, i32 1) to i32))
  %"complex exec struct694" = bitcast i8* %malloccall693 to { i1, i8*, i8*, i32 }*
  %"complex bool695" = getelementptr inbounds { i1, i8*, i8*, i32 }, { i1, i8*, i8*, i32 }* %"complex exec struct694", i32 0, i32 0
  %"complex e1696" = getelementptr inbounds { i1, i8*, i8*, i32 }, { i1, i8*, i8*, i32 }* %"complex exec struct694", i32 0, i32 1
  store i1 true, i1* %"complex bool695", align 1
  %casted_malloc697 = bitcast { i8**, { i8*, i8*, i32 }* }* %struct_space677 to i8*
  store i8* %casted_malloc697, i8** %"complex e1696", align 8
  %complex_bool_ptr698 = getelementptr inbounds { i1, i8*, i8*, i32 }, { i1, i8*, i8*, i32 }* %"complex exec struct694", i32 0, i32 0
  %complex_bool699 = load i1, i1* %complex_bool_ptr698, align 1
  %malloccall700 = tail call i8* @malloc(i32 ptrtoint (i8** getelementptr (i8*, i8** null, i32 1) to i32))
  %return_str_ptr701 = bitcast i8* %malloccall700 to i8**
  br i1 %complex_bool699, label %then703, label %else713

then561:                                          ; preds = %"run merge519"
  %exec_ptr562 = getelementptr inbounds { i1, i8*, i8*, i32 }, { i1, i8*, i8*, i32 }* %"complex exec struct552", i32 0, i32 1
  %cast_run563 = bitcast i8** %exec_ptr562 to { i8**, { i8*, i8*, i32 }* }**
  %exec564 = load { i8**, { i8*, i8*, i32 }* }*, { i8**, { i8*, i8*, i32 }* }** %cast_run563, align 8
  %dbl_path_ptr565 = getelementptr inbounds { i8**, { i8*, i8*, i32 }* }, { i8**, { i8**, { i8*, i8*, i32 }* }* }* %exec564, i32 0, i32 0
  %path_ptr566 = load i8**, i8*** %dbl_path_ptr565, align 8
  %path567 = load i8*, i8** %path_ptr566, align 8
  %args_ptr568 = getelementptr inbounds { i8**, { i8*, i8*, i32 }* }, { i8**, { i8**, { i8*, i8*, i32 }* }* }* %exec564, i32 0, i32 1
```

```
1118   %args569 = load { i8*, i8*, i32 }*, { i8*, i8*, i32 }**
         %args_ptr568, align 8
1119   %execvp570 = call i8* (i8*, { i8*, i8*, i32 }*, ...)
         @execvp_helper(i8* %path567, { i8*, i8*, i32 }* %args569)
1120   store i8* %execvp570, i8** %return_str_ptr559, align 8
1121   br label %"run merge560"
1122
1123 else571:                                          ; preds =
         %"run merge519"
1124   %recurse_exec572 = call i8* ({ i1, i8*, i8*, i32 }*, ...)
         @recurse_exec({ i1, i8*, i8*, i32 }* %"complex exec
         struct552")
1125   store i8* %recurse_exec572, i8** %return_str_ptr559, align 8
1126   br label %"run merge560"
1127
1128 "run merge702":                                   ; preds =
         %else713, %then703
1129   %e5715 = load { i1, i8*, i8*, i32 }*, { i1, i8*, i8*, i32
         }** %"variable ptr675", align 8
1130   %complex_bool_ptr716 = getelementptr inbounds { i1, i8*,
         i8*, i32 }, { i1, i8*, i8*, i32 }* %e5715, i32 0, i32 0
1131   %complex_bool717 = load i1, i1* %complex_bool_ptr716, align 1
1132   %malloccall718 = tail call i8* @malloc(i32 ptrtoint (i8**
         getelementptr (i8*, i8** null, i32 1) to i32))
1133   %return_str_ptr719 = bitcast i8* %malloccall718 to i8**
1134   br i1 %complex_bool717, label %then721, label %else731
1135
1136 then703:                                          ; preds =
         %"run merge560"
1137   %exec_ptr704 = getelementptr inbounds { i1, i8*, i8*, i32 },
         { i1, i8*, i8*, i32 }* %"complex exec struct694", i32 0,
         i32 1
1138   %cast_run705 = bitcast i8** %exec_ptr704 to { i8**, { i8*,
         i8*, i32 }* }**
1139   %exec706 = load { i8**, { i8*, i8*, i32 }* }*, { i8**, {
         i8*, i8*, i32 }* }** %cast_run705, align 8
1140   %dbl_path_ptr707 = getelementptr inbounds { i8**, { i8*,
         i8*, i32 }* }, { i8**, { i8*, i8*, i32 }* }* %exec706, i32
         0, i32 0
1141   %path_ptr708 = load i8**, i8*** %dbl_path_ptr707, align 8
1142   %path709 = load i8*, i8** %path_ptr708, align 8
1143   %args_ptr710 = getelementptr inbounds { i8**, { i8*, i8*,
         i32 }* }, { i8**, { i8*, i8*, i32 }* }* %exec706, i32 0,
         i32 1
1144   %args711 = load { i8*, i8*, i32 }*, { i8*, i8*, i32 }**
         %args_ptr710, align 8
1145   %execvp712 = call i8* (i8*, { i8*, i8*, i32 }*, ...)
         @execvp_helper(i8* %path709, { i8*, i8*, i32 }* %args711)
1146   store i8* %execvp712, i8** %return_str_ptr701, align 8
1147   br label %"run merge702"
1148
```

```
1149 else713:                                          ; preds =
       %"run merge560"
1150   %recurse_exec714 = call i8* ({ i1, i8*, i8*, i32 }*, ...)
       @recurse_exec({ i1, i8*, i8*, i32 }* %"complex exec
       struct694")
1151   store i8* %recurse_exec714, i8** %return_str_ptr701, align 8
1152   br label %"run merge702"
1153
1154 "run merge720":                                   ; preds =
       %else731, %then721
1155   %malloccall733 = tail call i8* @malloc(i32 ptrtoint (i8***
       getelementptr (i8**, i8*** null, i32 1) to i32))
1156   %"variable ptr734" = bitcast i8* %malloccall733 to i8***
1157   store i8** %return_str_ptr719, i8*** %"variable ptr734",
       align 8
1158   %malloccall735 = tail call i8* @malloc(i32 ptrtoint ({ i8**,
       { i8*, i8*, i32 }* }* getelementptr ({ i8**, { i8*, i8*,
       i32 }* }, { i8**, { i8*, i8*, i32 }* }* null, i32 1) to
       i32))
1159   %struct_space736 = bitcast i8* %malloccall735 to { i8**, {
       i8*, i8*, i32 }* }*
1160   %path_ptr737 = getelementptr inbounds { i8**, { i8*, i8*,
       i32 }* }, { i8**, { i8*, i8*, i32 }* }* %struct_space736,
       i32 0, i32 0
1161   %malloccall738 = tail call i8* @malloc(i32 ptrtoint (i8**
       getelementptr (i8*, i8** null, i32 1) to i32))
1162   %double_string_ptr739 = bitcast i8* %malloccall738 to i8**
1163   store i8* getelementptr inbounds ([5 x i8], [5 x i8]*
       @string.33, i32 0, i32 0), i8** %double_string_ptr739,
       align 8
1164   store i8** %double_string_ptr739, i8*** %path_ptr737, align 8
1165   %args_ptr740 = getelementptr inbounds { i8**, { i8*, i8*,
       i32 }* }, { i8**, { i8*, i8*, i32 }* }* %struct_space736,
       i32 0, i32 1
1166   %malloccall741 = tail call i8* @malloc(i32 ptrtoint (i8**
       getelementptr (i8*, i8** null, i32 1) to i32))
1167   %double_string_ptr742 = bitcast i8* %malloccall741 to i8**
1168   store i8* getelementptr inbounds ([3 x i8], [3 x i8]*
       @string.34, i32 0, i32 0), i8** %double_string_ptr742,
       align 8
1169   %malloccall743 = tail call i8* @malloc(i32 ptrtoint (i8***
       getelementptr (i8**, i8*** null, i32 1) to i32))
1170   %value_ptr744 = bitcast i8* %malloccall743 to i8***
1171   store i8** %double_string_ptr742, i8*** %value_ptr744, align
       8
1172   %malloccall745 = tail call i8* @malloc(i32 ptrtoint ({ i8*,
       i8*, i32 }* getelementptr ({ i8*, i8*, i32 }, { i8*, i8*,
       i32 }* null, i32 1) to i32))
1173   %list_node746 = bitcast i8* %malloccall745 to { i8*, i8*,
       i32 }*
1174   %struct_val_ptr747 = getelementptr inbounds { i8*, i8*, i32
       }, { i8*, i8*, i32 }* %list_node746, i32 0, i32 0
```

```
  %struct_ptr_ptr748 = getelementptr inbounds { i8*, i8*, i32 }, { i8*, i8*, i32 }* %list_node746, i32 0, i32 1
  %struct_ty_ptr749 = getelementptr inbounds { i8*, i8*, i32 }, { i8*, i8*, i32 }* %list_node746, i32 0, i32 2
  %malloccall750 = tail call i8* @malloc(i32 ptrtoint (i8** getelementptr (i8*, i8** null, i32 1) to i32))
  %double_string_ptr751 = bitcast i8* %malloccall750 to i8**
  store i8* getelementptr inbounds ([6 x i8], [6 x i8]* @string.35, i32 0, i32 0), i8** %double_string_ptr751, align 8
  %malloccall752 = tail call i8* @malloc(i32 ptrtoint (i8*** getelementptr (i8**, i8*** null, i32 1) to i32))
  %value_ptr753 = bitcast i8* %malloccall752 to i8***
  store i8** %double_string_ptr751, i8*** %value_ptr753, align 8
  %malloccall754 = tail call i8* @malloc(i32 ptrtoint ({ i8*, i8*, i32 }* getelementptr ({ i8*, i8*, i32 }, { i8*, i8*, i32 }* null, i32 1) to i32))
  %list_node755 = bitcast i8* %malloccall754 to { i8*, i8*, i32 }*
  %struct_val_ptr756 = getelementptr inbounds { i8*, i8*, i32 }, { i8*, i8*, i32 }* %list_node755, i32 0, i32 0
  %struct_ptr_ptr757 = getelementptr inbounds { i8*, i8*, i32 }, { i8*, i8*, i32 }* %list_node755, i32 0, i32 1
  %struct_ty_ptr758 = getelementptr inbounds { i8*, i8*, i32 }, { i8*, i8*, i32 }* %list_node755, i32 0, i32 2
  %casted_ptr_ptr759 = bitcast i8** %struct_ptr_ptr757 to { i8*, i8*, i32 }**
  store { i8*, i8*, i32 }* null, { i8*, i8*, i32 }** %casted_ptr_ptr759, align 8
  %casted_val760 = bitcast i8*** %value_ptr753 to i8*
  store i8* %casted_val760, i8** %struct_val_ptr756, align 8
  store i32 4, i32* %struct_ty_ptr758, align 4
  %casted_ptr_ptr761 = bitcast i8** %struct_ptr_ptr748 to { i8*, i8*, i32 }**
  store { i8*, i8*, i32 }* %list_node755, { i8*, i8*, i32 }** %casted_ptr_ptr761, align 8
  %casted_val762 = bitcast i8*** %value_ptr744 to i8*
  store i8* %casted_val762, i8** %struct_val_ptr747, align 8
  store i32 4, i32* %struct_ty_ptr749, align 4
  store { i8*, i8*, i32 }* %list_node746, { i8*, i8*, i32 }** %args_ptr740, align 8
  %malloccall763 = tail call i8* @malloc(i32 ptrtoint ({ i1, i8*, i8*, i32 }* getelementptr ({ i1, i8*, i8*, i32 }, { i1, i8*, i8*, i32 }* null, i32 1) to i32))
  %"complex exec struct764" = bitcast i8* %malloccall763 to { i1, i8*, i8*, i32 }*
  %"complex bool765" = getelementptr inbounds { i1, i8*, i8*, i32 }, { i1, i8*, i8*, i32 }* %"complex exec struct764", i32 0, i32 0
  %"complex e1766" = getelementptr inbounds { i1, i8*, i8*, i32 }, { i1, i8*, i8*, i32 }* %"complex exec struct764",
```

```
  i32 0, i32 1
  store i1 true, i1* %"complex bool765", align 1
  %casted_malloc767 = bitcast { i8**, { i8*, i8*, i32 }* }*
   %struct_space736 to i8*
  store i8* %casted_malloc767, i8** %"complex e1766", align 8
  %e6768 = load { i1, i8*, i8*, i32 }*, { i1, i8*, i8*, i32
   }** %"variable ptr596", align 8
  %e6769 = load { i1, i8*, i8*, i32 }*, { i1, i8*, i8*, i32
   }** %"variable ptr596", align 8
  %malloccall770 = tail call i8* @malloc(i32 ptrtoint ({ i1,
   i8*, i8*, i32 }* getelementptr ({ i1, i8*, i8*, i32 }, {
   i1, i8*, i8*, i32 }* null, i32 1) to i32))
  %"complex exec struct771" = bitcast i8* %malloccall770 to {
   i1, i8*, i8*, i32 }*
  %"complex bool772" = getelementptr inbounds { i1, i8*, i8*,
   i32 }, { i1, i8*, i8*, i32 }* %"complex exec struct771",
   i32 0, i32 0
  %"complex e1773" = getelementptr inbounds { i1, i8*, i8*,
   i32 }, { i1, i8*, i8*, i32 }* %"complex exec struct771",
   i32 0, i32 1
  %"complex e2774" = getelementptr inbounds { i1, i8*, i8*,
   i32 }, { i1, i8*, i8*, i32 }* %"complex exec struct771",
   i32 0, i32 2
  %"complex op775" = getelementptr inbounds { i1, i8*, i8*,
   i32 }, { i1, i8*, i8*, i32 }* %"complex exec struct771",
   i32 0, i32 3
  store i1 false, i1* %"complex bool772", align 1
  %casted_e1776 = bitcast { i1, i8*, i8*, i32 }* %e6769 to i8*
  store i8* %casted_e1776, i8** %"complex e1773", align 8
  %casted_e2777 = bitcast { i1, i8*, i8*, i32 }* %e6768 to i8*
  store i8* %casted_e2777, i8** %"complex e2774", align 8
  store i32 0, i32* %"complex op775", align 4
  %malloccall778 = tail call i8* @malloc(i32 ptrtoint ({ i1,
   i8*, i8*, i32 }* getelementptr ({ i1, i8*, i8*, i32 }, {
   i1, i8*, i8*, i32 }* null, i32 1) to i32))
  %"complex exec struct779" = bitcast i8* %malloccall778 to {
   i1, i8*, i8*, i32 }*
  %"complex bool780" = getelementptr inbounds { i1, i8*, i8*,
   i32 }, { i1, i8*, i8*, i32 }* %"complex exec struct779",
   i32 0, i32 0
  %"complex e1781" = getelementptr inbounds { i1, i8*, i8*,
   i32 }, { i1, i8*, i8*, i32 }* %"complex exec struct779",
   i32 0, i32 1
  %"complex e2782" = getelementptr inbounds { i1, i8*, i8*,
   i32 }, { i1, i8*, i8*, i32 }* %"complex exec struct779",
   i32 0, i32 2
  %"complex op783" = getelementptr inbounds { i1, i8*, i8*,
   i32 }, { i1, i8*, i8*, i32 }* %"complex exec struct779",
   i32 0, i32 3
  store i1 false, i1* %"complex bool780", align 1
  %casted_e1784 = bitcast { i1, i8*, i8*, i32 }* %"complex
   exec struct771" to i8*
```

```
  store i8* %casted_e1784, i8** %"complex e1781", align 8
  %casted_e2785 = bitcast { i1, i8*, i8*, i32 }* %"complex
   exec struct764" to i8*
  store i8* %casted_e2785, i8** %"complex e2782", align 8
  store i32 2, i32* %"complex op783", align 4
  %e3786 = load { i1, i8*, i8*, i32 }*, { i1, i8*, i8*, i32
   }** %"variable ptr66", align 8
  %malloccall787 = tail call i8* @malloc(i32 ptrtoint ({ i1,
   i8*, i8*, i32 }* getelementptr ({ i1, i8*, i8*, i32 }, {
   i1, i8*, i8*, i32 }* null, i32 1) to i32))
  %"complex exec struct788" = bitcast i8* %malloccall787 to {
   i1, i8*, i8*, i32 }*
  %"complex bool789" = getelementptr inbounds { i1, i8*, i8*,
   i32 }, { i1, i8*, i8*, i32 }* %"complex exec struct788",
   i32 0, i32 0
  %"complex e1790" = getelementptr inbounds { i1, i8*, i8*,
   i32 }, { i1, i8*, i8*, i32 }* %"complex exec struct788",
   i32 0, i32 1
  %"complex e2791" = getelementptr inbounds { i1, i8*, i8*,
   i32 }, { i1, i8*, i8*, i32 }* %"complex exec struct788",
   i32 0, i32 2
  %"complex op792" = getelementptr inbounds { i1, i8*, i8*,
   i32 }, { i1, i8*, i8*, i32 }* %"complex exec struct788",
   i32 0, i32 3
  store i1 false, i1* %"complex bool789", align 1
  %casted_e1793 = bitcast { i1, i8*, i8*, i32 }* %e3786 to i8*
  store i8* %casted_e1793, i8** %"complex e1790", align 8
  %casted_e2794 = bitcast { i1, i8*, i8*, i32 }* %"complex
   exec struct779" to i8*
  store i8* %casted_e2794, i8** %"complex e2791", align 8
  store i32 0, i32* %"complex op792", align 4
  %e1795 = load { i1, i8*, i8*, i32 }*, { i1, i8*, i8*, i32
   }** %"variable ptr", align 8
  %e2796 = load { i1, i8*, i8*, i32 }*, { i1, i8*, i8*, i32
   }** %"variable ptr31", align 8
  %malloccall797 = tail call i8* @malloc(i32 ptrtoint ({ i1,
   i8*, i8*, i32 }* getelementptr ({ i1, i8*, i8*, i32 }, {
   i1, i8*, i8*, i32 }* null, i32 1) to i32))
  %"complex exec struct798" = bitcast i8* %malloccall797 to {
   i1, i8*, i8*, i32 }*
  %"complex bool799" = getelementptr inbounds { i1, i8*, i8*,
   i32 }, { i1, i8*, i8*, i32 }* %"complex exec struct798",
   i32 0, i32 0
  %"complex e1800" = getelementptr inbounds { i1, i8*, i8*,
   i32 }, { i1, i8*, i8*, i32 }* %"complex exec struct798",
   i32 0, i32 1
  %"complex e2801" = getelementptr inbounds { i1, i8*, i8*,
   i32 }, { i1, i8*, i8*, i32 }* %"complex exec struct798",
   i32 0, i32 2
  %"complex op802" = getelementptr inbounds { i1, i8*, i8*,
   i32 }, { i1, i8*, i8*, i32 }* %"complex exec struct798",
   i32 0, i32 3
```

```
  store i1 false, i1* %"complex bool799", align 1
  %casted_e1803 = bitcast { i1, i8*, i8*, i32 }* %e2796 to i8*
  store i8* %casted_e1803, i8** %"complex e1800", align 8
  %casted_e2804 = bitcast { i1, i8*, i8*, i32 }* %e1795 to i8*
  store i8* %casted_e2804, i8** %"complex e2801", align 8
  store i32 1, i32* %"complex op802", align 4
  %malloccall805 = tail call i8* @malloc(i32 ptrtoint ({ i1,
   i8*, i8*, i32 }* getelementptr ({ i1, i8*, i8*, i32 }, {
   i1, i8*, i8*, i32 }* null, i32 1) to i32))
  %"complex exec struct806" = bitcast i8* %malloccall805 to {
   i1, i8*, i8*, i32 }*
  %"complex bool807" = getelementptr inbounds { i1, i8*, i8*,
   i32 }, { i1, i8*, i8*, i32 }* %"complex exec struct806",
   i32 0, i32 0
  %"complex e1808" = getelementptr inbounds { i1, i8*, i8*,
   i32 }, { i1, i8*, i8*, i32 }* %"complex exec struct806",
   i32 0, i32 1
  %"complex e2809" = getelementptr inbounds { i1, i8*, i8*,
   i32 }, { i1, i8*, i8*, i32 }* %"complex exec struct806",
   i32 0, i32 2
  %"complex op810" = getelementptr inbounds { i1, i8*, i8*,
   i32 }, { i1, i8*, i8*, i32 }* %"complex exec struct806",
   i32 0, i32 3
  store i1 false, i1* %"complex bool807", align 1
  %casted_e1811 = bitcast { i1, i8*, i8*, i32 }* %"complex
   exec struct798" to i8*
  store i8* %casted_e1811, i8** %"complex e1808", align 8
  %casted_e2812 = bitcast { i1, i8*, i8*, i32 }* %"complex
   exec struct788" to i8*
  store i8* %casted_e2812, i8** %"complex e2809", align 8
  store i32 2, i32* %"complex op810", align 4
  %malloccall813 = tail call i8* @malloc(i32 ptrtoint ({ i1,
   i8*, i8*, i32 }** getelementptr ({ i1, i8*, i8*, i32 }*, {
   i1, i8*, i8*, i32 }** null, i32 1) to i32))
  %"variable ptr814" = bitcast i8* %malloccall813 to { i1,
   i8*, i8*, i32 }**
  store { i1, i8*, i8*, i32 }* %"complex exec struct806", {
   i1, i8*, i8*, i32 }** %"variable ptr814", align 8
  %malloccall815 = tail call i8* @malloc(i32 ptrtoint ({ i8**,
   { i8*, i8*, i32 }* }* getelementptr ({ i8**, { i8*, i8*,
   i32 }* }, { i8**, { i8*, i8*, i32 }* }* null, i32 1) to
   i32))
  %struct_space816 = bitcast i8* %malloccall815 to { i8**, {
   i8*, i8*, i32 }* }*
  %path_ptr817 = getelementptr inbounds { i8**, { i8*, i8*,
   i32 }* }, { i8**, { i8*, i8*, i32 }* }* %struct_space816,
   i32 0, i32 0
  %malloccall818 = tail call i8* @malloc(i32 ptrtoint (i8**
   getelementptr (i8*, i8** null, i32 1) to i32))
  %double_string_ptr819 = bitcast i8* %malloccall818 to i8**
  store i8* getelementptr inbounds ([5 x i8], [5 x i8]*
   @string.36, i32 0, i32 0), i8** %double_string_ptr819,
```

```
  align 8
  store i8** %double_string_ptr819, i8*** %path_ptr817, align 8
  %args_ptr820 = getelementptr inbounds { i8**, { i8*, i8*, i32 }* }, { i8**, { i8*, i8*, i32 }* }* %struct_space816, i32 0, i32 1
  %malloccall821 = tail call i8* @malloc(i32 ptrtoint (i8** getelementptr (i8*, i8** null, i32 1) to i32))
  %double_string_ptr822 = bitcast i8* %malloccall821 to i8**
  store i8* getelementptr inbounds ([127 x i8], [127 x i8]* @string.37, i32 0, i32 0), i8** %double_string_ptr822, align 8
  %malloccall823 = tail call i8* @malloc(i32 ptrtoint (i8*** getelementptr (i8**, i8*** null, i32 1) to i32))
  %value_ptr824 = bitcast i8* %malloccall823 to i8***
  store i8** %double_string_ptr822, i8*** %value_ptr824, align 8
  %malloccall825 = tail call i8* @malloc(i32 ptrtoint ({ i8*, i8*, i32 }* getelementptr ({ i8*, i8*, i32 }, { i8*, i8*, i32 }* null, i32 1) to i32))
  %list_node826 = bitcast i8* %malloccall825 to { i8*, i8*, i32 }*
  %struct_val_ptr827 = getelementptr inbounds { i8*, i8*, i32 }, { i8*, i8*, i32 }* %list_node826, i32 0, i32 0
  %struct_ptr_ptr828 = getelementptr inbounds { i8*, i8*, i32 }, { i8*, i8*, i32 }* %list_node826, i32 0, i32 1
  %struct_ty_ptr829 = getelementptr inbounds { i8*, i8*, i32 }, { i8*, i8*, i32 }* %list_node826, i32 0, i32 2
  %casted_ptr_ptr830 = bitcast i8** %struct_ptr_ptr828 to { i8*, i8*, i32 }**
  store { i8*, i8*, i32 }* null, { i8*, i8*, i32 }** %casted_ptr_ptr830, align 8
  %casted_val831 = bitcast i8*** %value_ptr824 to i8*
  store i8* %casted_val831, i8** %struct_val_ptr827, align 8
  store i32 4, i32* %struct_ty_ptr829, align 4
  store { i8*, i8*, i32 }* %list_node826, { i8*, i8*, i32 }** %args_ptr820, align 8
  %malloccall832 = tail call i8* @malloc(i32 ptrtoint ({ i1, i8*, i8*, i32 }* getelementptr ({ i1, i8*, i8*, i32 }, { i1, i8*, i8*, i32 }* null, i32 1) to i32))
  %"complex exec struct833" = bitcast i8* %malloccall832 to { i1, i8*, i8*, i32 }*
  %"complex bool834" = getelementptr inbounds { i1, i8*, i8*, i32 }, { i1, i8*, i8*, i32 }* %"complex exec struct833", i32 0, i32 0
  %"complex e1835" = getelementptr inbounds { i1, i8*, i8*, i32 }, { i1, i8*, i8*, i32 }* %"complex exec struct833", i32 0, i32 1
  store i1 true, i1* %"complex bool834", align 1
  %casted_malloc836 = bitcast { i8**, { i8*, i8*, i32 }* }* %struct_space816 to i8*
  store i8* %casted_malloc836, i8** %"complex e1835", align 8
```

```
1306   %complex_bool_ptr837 = getelementptr inbounds { i1, i8*,
        i8*, i32 }, { i1, i8*, i8*, i32 }* %"complex exec
        struct833", i32 0, i32 0
1307   %complex_bool838 = load i1, i1* %complex_bool_ptr837, align 1
1308   %malloccall839 = tail call i8* @malloc(i32 ptrtoint (i8**
        getelementptr (i8*, i8** null, i32 1) to i32))
1309   %return_str_ptr840 = bitcast i8* %malloccall839 to i8**
1310   br i1 %complex_bool838, label %then842, label %else852
1311
1312 then721:                                          ; preds =
        %"run merge702"
1313   %exec_ptr722 = getelementptr inbounds { i1, i8*, i8*, i32 },
        { i1, i8*, i8*, i32 }* %e5715, i32 0, i32 1
1314   %cast_run723 = bitcast i8** %exec_ptr722 to { i8**, { i8*,
        i8*, i32 }* }**
1315   %exec724 = load { i8**, { i8*, i8*, i32 }* }*, { i8**, {
        i8*, i8*, i32 }* }** %cast_run723, align 8
1316   %dbl_path_ptr725 = getelementptr inbounds { i8**, { i8*,
        i8*, i32 }* }, { i8**, { i8*, i8*, i32 }* }* %exec724, i32
        0, i32 0
1317   %path_ptr726 = load i8**, i8*** %dbl_path_ptr725, align 8
1318   %path727 = load i8*, i8** %path_ptr726, align 8
1319   %args_ptr728 = getelementptr inbounds { i8**, { i8*, i8*,
        i32 }* }, { i8**, { i8*, i8*, i32 }* }* %exec724, i32 0,
        i32 1
1320   %args729 = load { i8*, i8*, i32 }*, { i8*, i8*, i32 }**
        %args_ptr728, align 8
1321   %execvp730 = call i8* (i8*, { i8*, i8*, i32 }*, ...)
        @execvp_helper(i8* %path727, { i8*, i8*, i32 }* %args729)
1322   store i8* %execvp730, i8** %return_str_ptr719, align 8
1323   br label %"run merge720"
1324
1325 else731:                                          ; preds =
        %"run merge702"
1326   %recurse_exec732 = call i8* ({ i1, i8*, i8*, i32 }*, ...)
        @recurse_exec({ i1, i8*, i8*, i32 }* %e5715)
1327   store i8* %recurse_exec732, i8** %return_str_ptr719, align 8
1328   br label %"run merge720"
1329
1330 "run merge841":                                   ; preds =
        %else852, %then842
1331   %e5854 = load { i1, i8*, i8*, i32 }*, { i1, i8*, i8*, i32
        }** %"variable ptr814", align 8
1332   %complex_bool_ptr855 = getelementptr inbounds { i1, i8*,
        i8*, i32 }, { i1, i8*, i8*, i32 }* %e5854, i32 0, i32 0
1333   %complex_bool856 = load i1, i1* %complex_bool_ptr855, align 1
1334   %malloccall857 = tail call i8* @malloc(i32 ptrtoint (i8**
        getelementptr (i8*, i8** null, i32 1) to i32))
1335   %return_str_ptr858 = bitcast i8* %malloccall857 to i8**
1336   br i1 %complex_bool856, label %then860, label %else870
1337
```

```
then842:                                          ; preds =
    %"run merge720"
  %exec_ptr843 = getelementptr inbounds { i1, i8*, i8*, i32 },
    { i1, i8*, i8*, i32 }* %"complex exec struct833", i32 0,
    i32 1
  %cast_run844 = bitcast i8** %exec_ptr843 to { i8**, { i8*,
    i8*, i32 }* }**
  %exec845 = load { i8**, { i8*, i8*, i32 }* }*, { i8**, {
    i8*, i8*, i32 }* }** %cast_run844, align 8
  %dbl_path_ptr846 = getelementptr inbounds { i8**, { i8*,
    i8*, i32 }* }, { i8**, { i8*, i8*, i32 }* }* %exec845, i32
    0, i32 0
  %path_ptr847 = load i8**, i8*** %dbl_path_ptr846, align 8
  %path848 = load i8*, i8** %path_ptr847, align 8
  %args_ptr849 = getelementptr inbounds { i8**, { i8*, i8*,
    i32 }* }, { i8**, { i8*, i8*, i32 }* }* %exec845, i32 0,
    i32 1
  %args850 = load { i8*, i8*, i32 }*, { i8*, i8*, i32 }**
    %args_ptr849, align 8
  %execvp851 = call i8* (i8*, { i8*, i8*, i32 }*, ...)
    @execvp_helper(i8* %path848, { i8*, i8*, i32 }* %args850)
  store i8* %execvp851, i8** %return_str_ptr840, align 8
  br label %"run merge841"

else852:                                          ; preds =
    %"run merge720"
  %recurse_exec853 = call i8* ({ i1, i8*, i8*, i32 }*, ...)
    @recurse_exec({ i1, i8*, i8*, i32 }* %"complex exec
    struct833")
  store i8* %recurse_exec853, i8** %return_str_ptr840, align 8
  br label %"run merge841"

"run merge859":                                   ; preds =
    %else870, %then860
  %malloccall872 = tail call i8* @malloc(i32 ptrtoint (i8***
    getelementptr (i8**, i8*** null, i32 1) to i32))
  %"variable ptr873" = bitcast i8* %malloccall872 to i8***
  store i8** %return_str_ptr858, i8*** %"variable ptr873",
    align 8
  ret i32 0

then860:                                          ; preds =
    %"run merge841"
  %exec_ptr861 = getelementptr inbounds { i1, i8*, i8*, i32 },
    { i1, i8*, i8*, i32 }* %e5854, i32 0, i32 1
  %cast_run862 = bitcast i8** %exec_ptr861 to { i8**, { i8*,
    i8*, i32 }* }**
  %exec863 = load { i8**, { i8*, i8*, i32 }* }*, { i8**, {
    i8*, i8*, i32 }* }** %cast_run862, align 8
  %dbl_path_ptr864 = getelementptr inbounds { i8**, { i8*,
    i8*, i32 }* }, { i8**, { i8*, i8*, i32 }* }* %exec863, i32
    0, i32 0
```

```
  %path_ptr865 = load i8**, i8*** %dbl_path_ptr864, align 8
  %path866 = load i8*, i8** %path_ptr865, align 8
  %args_ptr867 = getelementptr inbounds { i8**, { i8*, i8*, i32 }* }, { i8**, { i8*, i8*, i32 }* }* %exec863, i32 0, i32 1
  %args868 = load { i8*, i8*, i32 }*, { i8*, i8*, i32 }** %args_ptr867, align 8
  %execvp869 = call i8* (i8*, { i8*, i8*, i32 }*, ...) @execvp_helper(i8* %path866, { i8*, i8*, i32 }* %args868)
  store i8* %execvp869, i8** %return_str_ptr858, align 8
  br label %"run merge859"

else870:                                          ; preds = %"run merge841"
  %recurse_exec871 = call i8* ({ i1, i8*, i8*, i32 }*, ...) @recurse_exec({ i1, i8*, i8*, i32 }* %e5854)
  store i8* %recurse_exec871, i8** %return_str_ptr858, align 8
  br label %"run merge859"
}

declare noalias i8* @malloc(i32)
```

7 Lessons Learned

7.1 Kenny

Having the opportunity to design and implement our own coding language was an extremely fun and rewarding experience. This project allowed me to combine skills I've learned through the course of my computer science degree to make something really interesting. One of my biggest takeaways was that language design is an extremely iterative process. For example, we spent weeks changing and improving the syntax and functionality of executables. Even after we settled on a plan, I would constantly come up with new ideas or features which I wish we could have added to the language.

My first piece of advice for future groups is to not underestimate anything. Even simple features can cause unexpected headaches. My second piece of advice is to spend a lot of time brainstorming. Language design is hard. When you're coming up with your language, think of how the user will interact with it and what they can do to break it. Finally, my last piece of advice is to be flexible. The language you create probably won't be perfect and that's okay!

7.2 Alan

This class brought me some of my favorite moments at Tufts. There have been countless "Eureka!" moments, from designing the language to implementing codegen. At every stage of the project, I was constantly inspired about things to do to improve our language.

One of the most important things I learned during this project was the importance of team coordination. Especially with the creation and implementation of a language, its extremely important that everyone stays on the same page about what the language should look like and do. Moments of uncoordination often led to mistakes in our deliverables, which certainly could have been avoided. On a similar note, keeping good documentation was also another key takeaway. The code being written throughout this project was generally quite complex, and it was in a language none of us had worked in before. Because of that, it was especially difficult to understand code that someone else had written unless it was well commented.

For future classes: stay organized and resilient! You will most definitely run into roadblocks throughout this project, and staying organized will help you bounce back from them. If it feels overwhelming to you, keep in mind that it was overwhelming for everyone else as well. By staying committed to the project, you will get to see the fruits of your labor when things begin to click.

7.3 Tina

I think I learned a lot about language design and not taking things too seriously. There were a lot of times during feature brainstorming where I felt frustrated that the features we were brainstorming weren't the most pratical, useful, and necessary. I was taking it very seriously and actively considering features that would have maximized usability for users. However, we all talked through our expectations of the language as a group and with their help, eventually came to understand that the point of the class was more just to design a language for sake of designing a language, not to make it the most usability and helpful one that existed. I also enjoyed learning more about how compilers worked which gave me more insight to the machine behaviors I see while TA-ing COMP40.

7.4 Mary-Joy

First of all, I want to express that this class has been one of my most enjoyable and rewarding at Tufts. Seeing a project that we had brainstormed a few months ago come to fruition has been a surreal process.

Probably the biggest take away from this course for me is that the small design choices you think won't matter at the beginning of the process may actually matter more than you think. For example, our decision to make function pointers was one that we thought would be fairly straightforward to implement. However, we ran into various problems with this. In general, at the beginning of this process, we underestimated the time and effort that would have to go into each feature. And in the end, even though we got most of our functionality correct, there were a few side effects from previous design choices that didn't allow our features to be as flexible as we would have liked. Language design is truly a tedious process that requires a lot of care and consideration.

If I were to give one piece of advice for future groups, its that common language design choices that you see in common languages may not be as easy as it seems. At the beginning of the project it's good to be ambitious; however, don't be discouraged if some features may be more difficult than you originally anticipated. Additionally, try to enjoy the process as much as possible. For me, this was one of the very few classes where I didn't care as much about the grade as much as I did about the final product we were creating.

8 Appendix

8.1 Translator

scanner.mll

```
(* scanner.mll *)
(* BlueShell *)
(* Kenny Lin, Alan Luc, Tina Ma, Mary-Joy Sidhom *)

{ open Parser } (* Header which opens Parser file *)

let digit = ['0' - '9']
let digits = digit+

(* general token rule *)
rule tokenize = parse
  [' ' '\t' '\r' '\n'] { tokenize lexbuf }
(* comments *)
| "/*"       { multiline_comment lexbuf }
| "//"       { singleline_comment lexbuf }
(* structural tokens *)
| '('        { LPAREN }
| ')'        { RPAREN }
| '{'        { LBRACE }
| '}'        { RBRACE }
| ';'        { SEMI }
| ','        { COMMA }
| '\''       { character_of lexbuf }
| '"'        { string_of (Buffer.create 10) lexbuf }
| "&"        { AMPERSAND }
| "<"        { LANGLE }
| ">"        { RANGLE }
(* arithmetic symbols *)
| '+'        { PLUS }
| '-'        { MINUS }
| '*'        { TIMES }
| '/'        { DIVIDE }
| '='        { ASSIGN }
(* boolean operators *)
| "and"      { AND }
| "&&"       { AND }
| "or"       { OR }
| "||"       { OR }
| "not"      { NOT }
| "!"        { NOT }
| "=="       { EQ }
| "!="       { NEQ }
| ">="       { GEQ }
| "<="       { LEQ }
(* stmts *)
| "if"       { IF }
```

```
| "else"     { ELSE }
| "for"      { FOR }
| "while"    { WHILE }
| "return"   { RETURN }
(* types *)
| "int"      { INT }
| "bool"     { BOOL }
| "float"    { FLOAT }
| "void"     { VOID }
| "exec"     { EXEC }
| "char"     { CHR }
| "string"   { STR }
| "list"     { LIST }
| "true"     { BLIT(true) }
| "false"    { BLIT(false) }
| "function" { FUNCTION }
(* executable operators *)
| "|"        { PIPE }
| "./"       { RUN }
| "?"        { EXITCODE }
| "$"        { PATH }
| "withargs" { WITHARGS }
(* list operators *)
| '['        { LBRACKET }
| ']'        { RBRACKET }
| "::"       { CONS }
| "len"      { LEN }
| "of"       { OF }
(* first-class function operators *)
| "->"       { ARROW }
| digits as lxm { LITERAL(int_of_string lxm) }
| digits '.'  digit* as lxm { FLIT(lxm) }
| ['A'-'Z' 'a'-'z']['A'-'Z' 'a'-'z' '0'-'9' '_']* + as lit {
   ID(lit) }
| eof { EOF }
| _ as char { raise (Failure("illegal character " ^
   Char.escaped char)) }

(* multiline comment rule *)
and multiline_comment = parse
  "*/"  { tokenize lexbuf }
  | _   { multiline_comment lexbuf }
  | eof { raise (Failure("did not close multiline comment")) }

(* single line comment rule *)
and singleline_comment = parse
  "\n"  { tokenize lexbuf }
  | eof { EOF }
  | _   { singleline_comment lexbuf }

(* character of rule *)
and character_of = parse
```

```
97    '\''                { CHAR("")   }
98    | '\\' '\'' '\''    { CHAR("'")  }
99    | '\\' 'n' '\''     { CHAR("\n") }
100   | '\\' 'r' '\''     { CHAR("\r") }
101   | '\\' 't' '\''     { CHAR("\t") }
102   | '\\' '\\' '\''    { CHAR("\\") }
103   | [^ '"' '\\'] as single_char '\'' { CHAR(String.make 1
      single_char) }
104   | _ { raise (Failure("invalid char")) }
105   | eof { raise (Failure("read EOF with char open")) }
106
107 (* string of rule *)
108 and string_of buf = parse
109   '"'           {  STRING(Buffer.contents buf) }
110   | '\\' 'n'  { Buffer.add_char buf '\n'; string_of buf lexbuf
      }
111   | '\\' '"'  { Buffer.add_char buf '"'; string_of buf lexbuf
      }
112   | '\\' 'r'  { Buffer.add_char buf '\r'; string_of buf lexbuf
      }
113   | '\\' 't'  { Buffer.add_char buf '\t'; string_of buf lexbuf
      }
114   | '\\' '\\' { Buffer.add_char buf '\\'; string_of buf lexbuf
      }
115   | [^ '"' '\\']+ as cont_string { Buffer.add_string buf
      cont_string; string_of buf lexbuf }
116   | _ as char { raise (Failure("illegal character in string" ^
      Char.escaped char)) }
117   | eof       { raise (Failure("read EOF with string open"))
      }
```

parser.mly

```
/* parser.mly */
/* BlueShell */
/* Kenny Lin, Alan Luc, Tina Ma, Mary-Joy Sidhom */

%{ open Ast %}

%token SEMI LPAREN RPAREN LBRACE RBRACE COMMA LBRACKET
   RBRACKET AMPERSAND LANGLE
RANGLE OF ARROW/* structural tokens */
%token PLUS MINUS TIMES DIVIDE ASSIGN /* type operators */
%token AND OR NOT /* logical operators */
%token EQ GEQ LEQ NEQ /* comparisons */
%token IF ELSE WHILE FOR RETURN /* statements */
%token INT BOOL FLOAT VOID EXEC CHR STR LIST FUNCTION /* types
   */
%token <int> LITERAL
%token <bool> BLIT
%token <string> ID FLIT CHAR STRING
%token EOF
%token PIPE RUN EXITCODE PATH WITHARGS /* executable operators
   */
%token CONS LEN /* list operators */

%start program
%type <Ast.program> program

/* precedence */
%nonassoc ID
%nonassoc NOELSE
%nonassoc ELSE
%left EXITCODE
%left RUN
%right ASSIGN
%left OR
%left AND
%left EQ NEQ
%left LANGLE RANGLE
%left LEQ GEQ
%left PLUS MINUS
%left TIMES DIVIDE
%left PIPE
%right NOT
%right LEN
%left CONS
%right LBRACKET LPAREN
%right PATH

%%

program:
```

```
  decls EOF          { $1 }

decls:
   /* nothing */     { ([], []) }
 | decls stmt        { let (sdecls, fdecls) = $1 in (($2 ::
   sdecls), fdecls) }
 | decls fdecl       { let (sdecls, fdecls) = $1 in (sdecls,
   ($2 :: fdecls)) }

typ:
    INT              { Int      }
  | BOOL             { Bool     }
  | FLOAT            { Float    }
  | VOID             { Void     }
  | EXEC             { Exec     }
  | CHR              { Char     }
  | STR              { String   }
  | LIST OF typ      { List_type($3) }
  | FUNCTION formals_type      { Function($2) }

formals_type:
  LPAREN cont_formal_list { $2 }

cont_formal_list:
  typ RPAREN {([], $1)}
| typ ARROW cont_formal_list {( $1 :: (fst $3), snd $3)}
/* Executables */
exec:
  LANGLE expr WITHARGS expr RANGLE  { Exec($2, $4) }
| LANGLE expr RANGLE  { Exec($2, List([])) }

/* Lists */
list:
  LBRACKET cont_list       { List($2) }
  | LBRACKET RBRACKET      { List([]) }

cont_list:
  expr COMMA cont_list     { $1 :: $3 }
  | expr RBRACKET          { [$1] }

index:
  expr LBRACKET expr RBRACKET { Index($1, $3) }

list_cons:
  expr CONS expr           { Binop($1, Cons, $3) }

list_length:
  LEN expr                 { PreUnop(Length, $2) }

/* Functions */
fdecl:
  typ ID LPAREN formals_opt RPAREN LBRACE stmt_list RBRACE
```

```
98      { {  typ = $1;
99             fname = $2;
100            formals = List.rev $4;
101            body = List.rev $7; } }
102
103 formals_opt:
104     /* nothing */             { [] }
105   | formal_list               { $1 }
106
107 formal_list:
108     typ ID                    { [($1,$2)]     }
109   | formal_list COMMA typ ID  { ($3,$4) :: $1 }
110
111 vdecl:
112   typ ID                      { ($1, $2) }
113
114 stmt_list:
115      /* nothing */            { [] }
116   | stmt_list stmt            { $2 :: $1 }
117
118 stmt:
119     expr SEMI                                 { Expr $1
            }
120   | RETURN expr_opt SEMI                      { Return $2
            }
121   | LBRACE stmt_list RBRACE                   { Block(List.rev
     $2)    }
122   | IF LPAREN expr RPAREN stmt %prec NOELSE { If($3, $5,
     Block([])) }
123   | IF LPAREN expr RPAREN stmt ELSE stmt    { If($3, $5, $7)
            }
124   | FOR LPAREN expr_opt SEMI expr SEMI expr_opt RPAREN stmt
125                                             { For($3, $5, $7,
     $9)   }
126   | WHILE LPAREN expr RPAREN stmt           { While($3, $5)
            }
127
128 expr_opt:
129     /* nothing */ { Noexpr }
130   | expr          { $1 }
131
132 /* Expressions */
133 expr:
134     LITERAL                   { Literal($1)            }
135   | FLIT                        { Fliteral($1)           }
136   | BLIT                      { BoolLit($1)            }
137   | ID                        { Id($1)                 }
138   | CHAR                      { Char($1)               }
139   | STRING                    { String($1)             }
140   | expr PLUS   expr          { Binop($1, Add,   $3)   }
141   | expr MINUS  expr          { Binop($1, Sub,   $3)   }
142   | expr TIMES  expr          { Binop($1, Mult,  $3)   }
```

```
143   | expr DIVIDE expr          { Binop($1, Div,   $3)   }
144   | expr EQ     expr          { Binop($1, Equal, $3)   }
145   | expr NEQ    expr          { Binop($1, Neq,   $3)   }
146   | expr LANGLE     expr          { Binop($1, Less,  $3)   }
147   | expr LEQ    expr          { Binop($1, Leq,   $3)   }
148   | expr RANGLE     expr          { Binop($1, Greater, $3) }
149   | expr GEQ    expr          { Binop($1, Geq,   $3)   }
150   | expr AND    expr          { Binop($1, And,   $3)   }
151   | expr OR     expr          { Binop($1, Or,    $3)   }
152   | expr PIPE   expr          { Binop($1, Pipe, $3)    }
153   | MINUS expr %prec NOT      { PreUnop(Neg, $2)       }
154   | NOT expr                  { PreUnop(Not, $2)       }
155   | expr ASSIGN expr          { Binop($1, ExprAssign, $3) }
156   | ID ASSIGN expr            { Assign($1, $3)         }
157   | vdecl                     { Bind($1)               }
158   | ID LPAREN args_opt RPAREN { Call($1, $3)           }
159   | LPAREN expr RPAREN        { $2                     }
160   | expr EXITCODE             { PostUnop($1, ExitCode) }
161   | index                     { $1                     }
162   | expr PATH                 { PreUnop(Path, $1)      }
163   | RUN expr                  { PreUnop(Run, $2)       }
164   | exec                      { $1                     }
165   | list                      { $1                     }
166   | list_cons                 { $1                     }
167   | list_length               { $1                     }
168
169 args_opt:
170     /* nothing */             { []                     }
171   | args_list                 { List.rev $1            }
172
173 args_list:
174     expr                      { [$1]                   }
175   | args_list COMMA expr      { $3 :: $1               }
```

ast.ml

```
(* ast.ml *)
(* BlueShell *)
(* Kenny Lin, Alan Luc, Tina Ma, Mary-Joy Sidhom *)

(* general binary operators *)
type op = Add | Sub | Mult | Div | Equal | Neq | Less | Leq |
   Greater | Geq |
          And | Or | Pipe | Cons | ExprAssign

(* index has a special type because of its formatting *)
type index = Index

(* general unary operators *)
type uop = Neg | Not | ExitCode | Run | Path | Length

(* types *)
type typ = Int | Bool | Float | Void | Exec | Char | String |
   List_type of typ |
           EmptyList | Function of (typ list * typ) |
   ComplexExec

(* bind sets a variable name to a type *)
type bind = typ * string

(* expression *)
type expr =
    Literal of int
  | Fliteral of string
  | BoolLit of bool
  | Id of string
  | Char of string
  | String of string
  | Exec of expr * expr
  | Index of expr * expr
  | Binop of expr * op * expr
  | PreUnop of uop * expr
  | PostUnop of expr * uop
  | Assign of string * expr
  | Call of string * expr list
  | List of expr list
  | Bind of bind
  | Noexpr

(* statement *)
type stmt =
    Block of stmt list
  | Expr of expr
  | Return of expr
  | If of expr * stmt * stmt
  | For of expr * expr * expr * stmt
```

```
  | While of expr * stmt

(* function declaration *)
type func_decl = {
    typ : typ;
    fname : string;
    formals : bind list;
    body : stmt list;
}

(* program *)
type program = stmt list * func_decl list

(* Pretty-printing functions *)

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

let string_of_uop = function
    Neg ->         "-"
  | ExitCode ->    "?"
  | Run ->         "./"
  | Path ->        "$"
  | Not ->         "!"
  | Length ->      "len "

let string_of_path = function
  Id(s) ->         s
  | String(s) ->   "\"" ^ s ^ "\""
  | _ ->           "Error: not a viable path type"

let rec string_of_typ = function
    Int ->         "int"
  | Bool ->        "bool"
  | Float ->       "float"
  | Void ->        "void"
  | Exec ->        "exec"
  | ComplexExec -> "exec"
```

```
100   | Char ->       "char"
101   | String ->     "string"
102   | List_type(t) ->    "list of " ^ string_of_typ t
103   | Function(args, ret) ->   "function ("  ^ String.concat "
      -> " (List.map
104   string_of_typ (List.rev (ret :: (List.rev args)))) ^ ")"
105   | EmptyList ->  "emptylist"
106
107 let string_of_vdecl (t, id) = string_of_typ t ^ " " ^ id
108
109 let rec string_of_expr = function
110     Literal(l) ->     string_of_int l
111   | Fliteral(l) ->    l
112   | BoolLit(true) ->  "true"
113   | BoolLit(false) -> "false"
114   | Id(s) ->          s
115   | Char(c) ->        "'" ^ c ^ "'"
116   | String(s) ->      "\"" ^ s ^ "\""
117   | Exec(e1, e2) ->
118       "<" ^ string_of_expr e1 ^ " withargs " ^ string_of_expr
      e2 ^ ">"
119   | Binop(e1, o, e2) ->
120       string_of_expr e1 ^ " " ^ string_of_op o ^ " " ^
      string_of_expr e2
121   | PreUnop(o, e) ->  string_of_uop o ^ string_of_expr e
122   | PostUnop(e, o) -> string_of_expr e ^ string_of_uop o
123   | Assign(v, e) ->   v ^ " = " ^ string_of_expr e
124   | Call(f, el) ->
125       f ^ "(" ^ String.concat ", " (List.map string_of_expr
      el) ^ ")"
126   | List(l) ->        "[" ^ (String.concat ", " (List.map
      string_of_expr l)) ^ "]"
127   | Index(list, index) ->
128       string_of_expr list ^ "[" ^ string_of_expr index ^ "]"
129   | Bind(var) ->      string_of_vdecl var
130   | Noexpr ->         ""
131
132 let rec string_of_stmt = function
133     Block(stmts) ->
134       "{\n" ^ String.concat "" (List.map string_of_stmt stmts)
      ^ "}\n"
135   | Expr(expr) ->     string_of_expr expr ^ ";\n";
136   | Return(expr) ->   "return " ^ string_of_expr expr ^ ";\n";
137   | If(e, s, Block([])) ->
138       "if (" ^ string_of_expr e ^ ")\n" ^ string_of_stmt s
139   | If(e, s1, s2) ->
140       "if (" ^ string_of_expr e ^ ")\n" ^
141       string_of_stmt s1 ^ "else\n" ^ string_of_stmt s2
142   | For(e1, e2, e3, s) ->
143       "for (" ^ string_of_expr e1  ^ " ; " ^ string_of_expr e2
      ^ " ; " ^
144       string_of_expr e3  ^ ") " ^ string_of_stmt s
```

```
145   | While(e, s) ->    "while (" ^ string_of_expr e ^ ") " ^
       string_of_stmt s
146
147 let string_of_args args =
148   string_of_typ (fst args) ^ " " ^ (snd args)
149
150 let string_of_fdecl fdecl =
151   string_of_typ fdecl.typ ^ " " ^
152   fdecl.fname ^ "(" ^ String.concat ", " (List.map
       string_of_args fdecl.formals) ^
153   ")\n{\n" ^ "" ^ String.concat "" (List.map string_of_stmt
       fdecl.body) ^ "}\n"
154
155 let string_of_program (stmts, funcs) =
156   String.concat "" (List.rev (List.map string_of_stmt stmts))
       ^ "\n" ^
157   String.concat "\n" (List.rev (List.map string_of_fdecl
       funcs))
```

semant.ml

```
(* Semantic checking for the Blue Shell compiler *)

open Ast
open Sast

module StringMap = Map.Make(String)

type symbol_table = {
  (* Variables bound in current block *)
  variables : typ StringMap.t;
  (* Enclosing scope *)
  parent : symbol_table option;
}

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

   Check each global variable, then check each function *)

let check (stmts, functions) =

  (* carry a boolean when checking all the arg types of a
   function *)
  let check_arg_types is_valid args1 args2  = is_valid &&
   (args1 = args2)
  in

  (* Raise an exception if the given rvalue type cannot be
   assigned to
      the given lvalue type *)
  let check_assign lvaluet rvaluet err =
    (match (lvaluet, rvaluet) with
      (List_type _, EmptyList) -> lvaluet
      | (List_type ty1, List_type ty2) -> (match ty1 = ty2 with
          true -> List_type ty1
          | false -> raise (Failure err))
      | (Function (args1, ret1), Function (args2, ret2)) ->
   (match ret1 = ret2 with
          true -> (match (List.fold_left2 check_arg_types true
   args1 args2 ) with
                  true -> Function (args1, ret1)
                  | false -> raise (Failure err))
          | false -> raise (Failure err))
      | (ComplexExec, Exec) | (Exec, ComplexExec) -> rvaluet
      | _ -> (match lvaluet = rvaluet with
              true -> lvaluet
            | false -> raise (Failure err)))
  in

  let rec type_of_identifier (scope : symbol_table) name =
```

```
    try
      (* Try to find binding in nearest block *)
      StringMap.find name scope.variables
    with Not_found -> (* Try looking in outer blocks *)
      match scope.parent with
        Some(parent) -> type_of_identifier parent name
      | _ -> raise (Failure ("semant identifier not found"))
  in

  let add_bind (scope : symbol_table) (typ, name) =
    let map = scope.variables in
    let new_map = StringMap.add name typ map in
    { variables = new_map; parent = scope.parent }

  in

  let add_func_symbol_table map fd =
    let n = fd.fname (* Name of the function *)
    and ty = Function(List.map fst fd.formals, fd.typ) in
    add_bind map (ty, n)
  in

  (* Collect all other function names into one symbol table *)
  (* Start with empty environment and map over statements,
   carrying updated
  environment as you go *)
  let empty_env = { variables = StringMap.empty ; parent =
   None }
  in
  let env_with_functions = List.fold_left
   add_func_symbol_table empty_env functions
  in

  (* check function types *)
  let same_func ((ty1 : typ), (ty2 : typ)) =
    (match ty1 = ty2 with
      true -> true
      | _ -> (match (ty1, ty2) with
        (Exec, ComplexExec) | (ComplexExec, Exec) -> true
        | _ -> false))
  in

  let rec expr (curr_symbol_table : symbol_table) expression =
  match expression with
    (* all literals evaluate to their own type *)
    Literal l     -> (curr_symbol_table, (Int, SLiteral l))
  | Fliteral l    -> (curr_symbol_table, (Float, SFliteral l))
  | BoolLit l     -> (curr_symbol_table, (Bool, SBoolLit l))
    (* search symbol table for identifiers *)
  | Id var        -> (curr_symbol_table, (type_of_identifier
   curr_symbol_table var, SId var))
  | Char s        -> (curr_symbol_table, (Char, SChar s))
```

```
94    | String s       -> (curr_symbol_table, (String, SString s))
95    | Exec(e1, e2)  ->
96      let (_, (ty1, e1')) = expr curr_symbol_table e1 in
97      let (_, (ty2, e2')) = expr curr_symbol_table e2 in
98      (* path must be a string, args must be a list *)
99      (match ty1 with
100       String ->
101         (match ty2 with
102           List_type ty ->(match ty with
103             Int | Float | Bool | String | Char ->
      (curr_symbol_table, (Exec, (SExec ((ty1, e1'), (ty2,
      e2'))))))
104           | _ -> raise (Failure ("exec args cannot be of
      type function, list, or exec")))
105
106
107         | EmptyList -> (curr_symbol_table, (Exec, (SExec
      ((ty1, e1'), (List_type String, SList [])))))
108         | _ -> raise (Failure ("args must be a list of
      string")))
109     | _ -> raise (Failure ("path must be a string")))
110   | Index(e1, e2) ->
111     let (_, (ty1, e1')) = expr curr_symbol_table e1 in
112     let (_, (ty2, e2')) = expr curr_symbol_table e2 in
113     (* can only index into lists *)
114     (match (ty1, ty2) with
115       (List_type ty, Int) -> (curr_symbol_table, (ty, (SIndex
      ((ty1, e1'), (ty2, e2'))))))
116     | _ -> raise (Failure ("Indexing takes a list and
      integer")))
117   | Binop(e1, op, e2) ->
118     let (symbol_table', (ty2, e2')) = expr curr_symbol_table e2
119     in let (symbol_table'', (ty1, e1')) = expr symbol_table'
      e1 in
120     (match e2 with Bind _ -> raise (Failure "Bind cannot
      happen on left side of binops")
121     | _ ->
122     (match (e1, op) with
123       (* only certain expressions can appear on the left side
      of an ExprAssign *)
124       (* bind allows for "int x = 1;" *)
125       (* index allows for assignment of list elements *)
126       (* path is valid on the left side to change the path of
      an executable *)
127       (Bind b, ExprAssign) ->
128                     let same = same_func(ty1, ty2) in
129                     let ty = (match e1' with
130                       | SBind b when same -> fst b
131                       | _ -> (match ty1 with
132                         List_type _ -> (match ty2 with
133                           EmptyList -> ty1
```

```
                          | _ -> raise (Failure ("invalid assignment")))
                        | _ -> raise (Failure ("invalid assignment"))))
                  in (symbol_table'', (ty, SBinop((ty1, e1'), op, (ty2,
                  e2'))))
    | (Index _, ExprAssign) ->
                  let same = same_func(ty1, ty2) in
                  (match same with
                    true -> (symbol_table'', (ty1, SBinop((ty1, e1'), op, (ty2,
                    e2'))))
                    | false -> raise (Failure ("index exprassign with
                    incompatible types")))
    | (PreUnop (op1, _), ExprAssign) ->
                  (match op1 with
                    Path -> let same = same_func(ty1, ty2) in
                    (match same with
                      true -> (symbol_table'', (ty1, SBinop((ty1, e1'), op, (ty2,
                      e2'))))
                      | false -> raise (Failure ("index exprassign with
                      incompatible types")))
                    | _ -> raise (Failure ("invalid preunop with exprassign")))
    | (_, ExprAssign) -> raise (Failure "expression assignment needs a bind")
    | (Bind _, _) -> raise (Failure "bind needs an expression assignment")
    (* arithmetic and boolean operations require 2 of the same types *)
    (* executable operations can work on any combination of simple and complex executables *)
    | (_, Add) | (_, Mult) ->
                  let same = same_func(ty1, ty2) in
                  (match ty1 with
                    Int | Float when same -> (symbol_table'', (ty1, SBinop((ty1, e1'), op,
                    (ty2, e2'))))
                    | Exec | ComplexExec when same -> (symbol_table'', (ComplexExec, SBinop((ty1, e1'), op,
                    (ty2, e2'))))
                  | _ -> raise (Failure ("+ and * take two integers,
                  floats, or executables")))
    | (_, Sub) | (_, Div) ->
                  let same = same_func (ty1, ty2) in
                  (match ty1 with
```

```
                      Int | Float when same ->
(symbol_table'', (ty1, SBinop((ty1, e1'), op,
                      (ty2, e2'))))
                    | _ -> raise (Failure ("Operator expected
int or float")))
   | (_, Less) | (_, Leq) | (_, Greater) | (_, Geq) ->
                    let same = same_func (ty1, ty2) in
                    (match ty1 with
                      Int | Float when same ->
(symbol_table'', (Bool, SBinop((ty1, e1'), op,
                      (ty2, e2'))))
                    | _ -> raise (Failure ("Operator expected
int or float")))
   | (_, And) | (_, Or) ->
                    let same = same_func (ty1, ty2) in
                    (match ty1 with
                      Bool when same  -> (symbol_table'',
(Bool, SBinop((ty1, e1'), op,
                      (ty2, e2'))))
                    | _ -> raise (Failure ("Boolean operators
must take two booleans")))
   | (_, Equal) | (_, Neq) ->
                    let same = same_func (ty1, ty2) in
                    (match ty1 with
                       | Float | Int when same ->
(symbol_table'', (Bool, SBinop((ty1, e1'), op,
                               (ty2, e2'))))
                    | _ -> raise (Failure ("operator expected
int, or float")))
   (* cons requires the element being appended to match the
type of the list *)
   (* any element can be cons'd to an empty list *)
   | (_, Cons) ->
                    (match ty2 with
                      EmptyList -> (symbol_table'', (List_type
ty1, SBinop((ty1, e1'), op,
                        (ty2, e2'))))
                     | List_type ty  -> let same = ty = ty1 in
                        (match same with
                        true -> (symbol_table'', (ty2,
SBinop((ty1, e1'), op,
                        (ty2, e2'))))
                        | false -> raise (Failure ("lists are
monomorphic")))
                    | _ -> raise (Failure ("Cons takes a list
and primitive type")))
   | (_, Pipe) ->
                    let same = same_func (ty1, ty2) in
                    (match ty1 with
                      ComplexExec | Exec when same ->
(symbol_table'', (ComplexExec, SBinop((ty1, e1'), op,
                      (ty2, e2'))))
```

```
208                     | _ -> raise (Failure ("Pipe expects two
    executables")))))
209   | PreUnop(op, e)    ->
210     let (_, (ty1, e1)) = expr curr_symbol_table e in
211     (match e1 with
212     SBind _ -> raise (Failure "No bind can occur in a larger
    expression")
213     | _ ->
214     (match op with
215       Run ->
216               (match ty1 with
217                 Exec | ComplexExec -> (curr_symbol_table,
    (String, SPreUnop (Run, (ty1, e1))))
218               | _ -> raise (Failure ("Run takes type
    executable")))
219     | Neg ->
220               (match ty1 with
221                 Int | Float -> (curr_symbol_table, (ty1,
    SPreUnop (Neg, (ty1, e1))))
222               | _ -> raise (Failure ("Negation takes an
    interger, float, or list")))
223     | Length ->
224               (match ty1 with
225                 EmptyList | List_type _ -> (curr_symbol_table,
    (Int, SPreUnop (Length,(ty1,e1))))
226               | _ -> raise (Failure ("Length takes a list")))
227     | Path ->
228               (match ty1 with
229                 Exec -> (curr_symbol_table, (String, SPreUnop
    (Path, (ty1, e1))))
230               | _ -> raise (Failure ("Run takes type
    executable")))
231     | Not ->
232               (match ty1 with
233                 Bool -> (curr_symbol_table, (Bool, SPreUnop
    (Not, (ty1, e1))))
234               | _ -> raise (Failure ("Boolean negation takes a
    boolean")))))
235   | PostUnop(e, op)   ->
236     let (_, (ty1, e1)) = expr curr_symbol_table e in
237     (match e1 with
238       SBind _ -> raise (Failure "No bind can occur in a larger
    expression")
239       | _ ->
240     (match op with
241       | _ -> raise (Failure ("invalid postunop"))))
242   | Assign(var, e) as ex ->
243     (* ensure that type of variable matches type of expression
    being assigned to
244     it *)
245     (match e with
```

```
246     Bind _ -> raise (Failure "No bind can occur in a larger
    expression")
247     | _ ->
248     let lt = type_of_identifier curr_symbol_table var
249     and (_, (rt, e')) = expr curr_symbol_table e in
250     let err = "illegal assignment " ^ string_of_typ lt ^ " = "
    ^
251       string_of_typ rt ^ " in " ^ string_of_expr ex
252     in (curr_symbol_table, (check_assign lt rt err,
    SAssign(var, (rt, e'))))))
253   | Call(fname, args) as call ->
254     (* ensure that calling a function is done with the correct
    parameter types *)
255     (* also checks that the return value of the function isn't
    improperly assigned *)
256       (match type_of_identifier curr_symbol_table fname with
257           Function (args_typs, ret_typ) ->
258             let param_length = List.length args_typs in
259             if List.length args != param_length then
260               raise (Failure ("expecting " ^ string_of_int
    param_length ^
261                               " arguments in " ^
    string_of_expr call))
262             else let check_call ft e =
263               let (_, (et, e')) = expr curr_symbol_table e in
264               let err = "illegal argument found " ^
    string_of_typ et ^
265                 " expected " ^ string_of_typ ft ^ " in " ^
    string_of_expr e
266               in (check_assign ft et err, e')
267             in
268             let args' = List.map2 check_call args_typs args
269             in (curr_symbol_table, (ret_typ, SCall(fname,
    args'))))
270         | _ -> raise (Failure ("Not a function")))
271   | List expr_list ->
272     (* assert that all elements of a list are the same type *)
273     let rec check_list (exprs : expr list) curr_symbol_table =
274       match exprs with
275       fst_elem :: snd_elem :: rest -> let (_, (ty1, e1)) =
    expr curr_symbol_table fst_elem in let
276                                 (_, (ty2, e2)) = expr
    curr_symbol_table snd_elem
277                                 in (match (e1, e2) with
278                                 (SBind _, _) -> raise (Failure
    ("can't bind in a list"))
279                                 | (_, SBind _) -> raise
    (Failure ("can't bind in a list"))
280                                 | _ -> (ty1 = ty2 &&
    check_list (snd_elem :: rest) curr_symbol_table))
281       | fst_elem :: [] -> let (_, (_, e1)) = expr
    curr_symbol_table fst_elem
```

```
282       in (match e1 with
283           SBind _ -> raise (Failure ("No binds in list"))
284           |_ -> true)
285       | [] -> true
286     in
287     let expr_to_sexpr elem =
288       let (_, elem') = expr curr_symbol_table elem in
289       elem'
290     in
291     (match expr_list with
292       [] -> (curr_symbol_table, (EmptyList, SList([])))
293     | elem :: elems -> let (_, elem') = expr curr_symbol_table
    elem in
294       match (check_list elems curr_symbol_table) with
295         true -> (curr_symbol_table, (List_type (fst elem'),
    SList(List.map expr_to_sexpr expr_list)))
296       | false -> raise (Failure ("list must be monotype"))
297     )
298   | Bind bind -> (add_bind curr_symbol_table bind, (fst bind,
    SBind bind))
299   | Noexpr      -> (curr_symbol_table, (Void, SNoexpr))
300
301   in
302   let rec check_stmt (curr_symbol_table : symbol_table)
    statement =
303   match statement with
304     Block stmts ->
305       (* recurse on statements in a block *)
306       let rec check_stmt_list (curr_symbol_table' :
    symbol_table) sl =
307         (match sl with
308               [Return _ as s] -> [snd (check_stmt
    curr_symbol_table' s)]
309           | Return _ :: _   -> raise (Failure "nothing may
    follow a return")
310           | Block sl :: ss  ->
311             let temp = { variables = StringMap.empty ;
    parent =
312             Some curr_symbol_table' } in
313             let checked_sl = (check_stmt_list temp sl)
314             in
315             SBlock(checked_sl) :: (check_stmt_list
    curr_symbol_table' ss) (* Flatten blocks *)
316           | s :: ss         ->
317             let checked_first = (check_stmt
    curr_symbol_table' s) in
318             (snd checked_first) :: (check_stmt_list (fst
    checked_first) ss)
319           | []              -> [])
320         in (curr_symbol_table, SBlock(check_stmt_list {
    variables = StringMap.empty ; parent = Some
    curr_symbol_table} stmts))
```

```
321   | Expr e ->
322     let (new_symbol_table, e') = expr curr_symbol_table e in
323     (new_symbol_table, SExpr e')
324   | If(e, s1, s2) ->
325     let (curr_symbol_table', (ty, e')) = expr
      curr_symbol_table e in
326     (* condition in if statement must be a boolean *)
327     (match ty with
328       Bool -> (curr_symbol_table, SIf((ty, e'), snd
      (check_stmt curr_symbol_table' s1), snd (check_stmt
329       curr_symbol_table' s2)))
330       | _ -> raise (Failure ("if needs a boolean predicate")))
331   | For(e1, e2, e3, s) ->
332     let (curr_symbol_table', (ty1, e1')) = expr
      curr_symbol_table e1 in
333     let (curr_symbol_table'', (ty2, e2')) = expr
      curr_symbol_table' e2 in
334     let (curr_symbol_table''', (ty3, e3')) = expr
      curr_symbol_table'' e3 in
335     (* second expression in for loop must be a boolean *)
336     (match ty2 with
337       Bool -> (curr_symbol_table''', SFor((ty1, e1'), (ty2,
      e2'), (ty3, e3'),
338       snd (check_stmt curr_symbol_table''' s)))
339       | _ -> raise (Failure ("for needs a boolean as the
      second expression")))
340   | While(e, s) ->
341     let (curr_symbol_table', (ty, e')) = expr
      curr_symbol_table e in
342     (* condition in while loop must be a boolean *)
343       (match ty with
344       Bool -> (curr_symbol_table, SWhile((ty, e'), snd
      (check_stmt curr_symbol_table' s)))
345       | _ -> raise (Failure ("if needs a boolean predicate")))
346   | Return e -> raise (Failure ("cannot return from not a
      function"))
347   (* Final checked program to return *)
348   in
349
350   (* check statements in a function, slightly different from
      check_stmt because
351   return statements can appear in a function but not in
      top-level statements *)
352   let check_func symbol_table func =
353     let add_formals formal_map name typ =
354       StringMap.add name typ formal_map
355     in
356     let formals_map = List.fold_left2 add_formals
      symbol_table.variables (List.map snd func.formals) (List.map
357     fst func.formals) in
358     let formals_env = { variables = formals_map ; parent =
      None } in
```

```
359     let rec check_stmt_wrap (curr_symbol_table : symbol_table)
    statement =
360       (match statement with
361       Return ret ->
362         let (curr_symbol_table', (ty_ret, ret')) = expr
    curr_symbol_table ret in
363         let same = ty_ret = func.typ in
364         (match same with
365           true -> (curr_symbol_table', SReturn (ty_ret, ret'))
366           | false -> raise (Failure ("return type invalid")))
367       | Block stmts ->
368         let rec check_stmt_list (curr_symbol_table' :
    symbol_table) sl =
369           (match sl with
370                 [Return _ as s] -> [snd (check_stmt_wrap
    curr_symbol_table' s)]
371               | Return _ :: _   -> raise (Failure "nothing may
    follow a return")
372               | Block sl :: ss  ->
373                 let temp = { variables = StringMap.empty ;
    parent =
374                 Some curr_symbol_table' } in
375                 let checked_sl = (check_stmt_list temp sl)
376                 in
377                 SBlock(checked_sl) :: (check_stmt_list
    curr_symbol_table' ss) (* Flatten blocks *)
378               | s :: ss         ->
379                 let checked_first = (check_stmt_wrap
    curr_symbol_table' s) in
380                 (snd checked_first) :: (check_stmt_list (fst
    checked_first) ss)
381               | []              -> [])
382           in (curr_symbol_table, SBlock(check_stmt_list {
    variables = StringMap.empty ; parent = Some
    curr_symbol_table} stmts))
383     | Expr e ->
384       let (new_symbol_table, e') = expr curr_symbol_table e in
385       (new_symbol_table, SExpr e')
386     | If(e, s1, s2) ->
387       let (curr_symbol_table', (ty, e')) = expr
    curr_symbol_table e in
388       (match ty with
389         Bool -> (curr_symbol_table, SIf((ty, e'), snd
    (check_stmt_wrap curr_symbol_table' s1), snd
    (check_stmt_wrap
390         curr_symbol_table' s2)))
391         | _ -> raise (Failure ("if needs a boolean
    predicate")))
392     | For(e1, e2, e3, s) ->
393       let (curr_symbol_table', (ty1, e1')) = expr
    curr_symbol_table e1 in
```

```
394       let (curr_symbol_table'', (ty2, e2')) = expr
    curr_symbol_table' e2 in
395       let (curr_symbol_table''', (ty3, e3')) = expr
    curr_symbol_table'' e3 in
396       (match ty2 with
397         Bool -> (curr_symbol_table''', SFor((ty1, e1'), (ty2,
    e2'), (ty3, e3'),
398         snd (check_stmt_wrap curr_symbol_table''' s)))
399         | _ -> raise (Failure ("for needs a boolean as the
    second expression")))
400     | While(e, s) ->
401       let (curr_symbol_table', (ty, e')) = expr
    curr_symbol_table e in
402         (match ty with
403         Bool -> (curr_symbol_table, SWhile((ty, e'), snd
    (check_stmt_wrap curr_symbol_table' s)))
404         | _ -> raise (Failure ("if needs a boolean
    predicate"))))
405      in
406      let (_, checked_statements) = List.fold_left_map
    check_stmt_wrap formals_env func.body in
407      (symbol_table, {
408        styp = func.typ;
409        sfname = func.fname;
410        sformals = func.formals;
411        sbody = checked_statements; (* add new symbol table for
    body, but references parent st*)
412      })
413
414   in
415   let (env_with_checked_funcs, checked_functions) =
      List.fold_left_map check_func env_with_functions (List.rev
      functions)
416   in
417
418   let (_, checked_statements) = List.fold_left_map check_stmt
      env_with_checked_funcs (List.rev stmts)
419   (* go through map called function_decls, put all the fdecls
      into sfdecls and
420   gather  into a list *)
421
422   in (List.rev checked_statements, List.rev checked_functions)
```

sast.ml

```
(* Semantically-checked Abstract Syntax Tree and functions for
   printing it *)

open Ast

type sexpr = typ * sx
and sx =
    SLiteral of int
  | SFliteral of string
  | SBoolLit of bool
  | SId of string
  | SChar of string
  | SString of string
  | SExec of sexpr * sexpr
  | SIndex of sexpr * sexpr
  | SBinop of sexpr * op * sexpr
  | SPreUnop of uop * sexpr
  | SPostUnop of sexpr * uop
  | SAssign of string * sexpr
  | SCall of string * sexpr list
  | SList of sexpr list
  | SBind of bind
  | SNoexpr

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

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

type sprogram = sstmt list * sfunc_decl list

(* Pretty-printing functions *)

let rec string_of_sexpr (t, e) =
  "(" ^ string_of_typ t ^ " : " ^ (match e with
    SLiteral(l) ->     string_of_int l
  | SFliteral(l) ->    l
  | SBoolLit(true) ->  "true"
  | SBoolLit(false) -> "false"
  | SId(s) ->          s
```

```
50    | SChar(c) ->         "'" ^ c ^ "'"
51    | SString(s) ->       "\"" ^ s ^ "\""
52    | SExec(e1, e2) ->
53        "<" ^ string_of_sexpr e1 ^ " withargs " ^
      string_of_sexpr e2 ^ ">"
54    | SBinop(e1, o, e2) ->
55        string_of_sexpr e1 ^ " " ^ string_of_op o ^ " " ^
      string_of_sexpr e2
56    | SPreUnop(o, e) ->  string_of_uop o ^ string_of_sexpr e
57    | SPostUnop(e, o) -> string_of_sexpr e ^ string_of_uop o
58    | SAssign(v, e) ->   v ^ " = " ^ string_of_sexpr e
59    | SCall(f, el) ->
60        f ^ "(" ^ String.concat ", " (List.map string_of_sexpr
      el) ^ ")"
61    | SList(l) ->        "[" ^ (String.concat ", " (List.map
      string_of_sexpr l)) ^ "]"
62    | SIndex(list, index) ->
63        string_of_sexpr list ^ "[" ^ string_of_sexpr index ^ "]"
64    | SBind(var) ->      string_of_vdecl var
65    | SNoexpr ->         ""
66          ) ^ ")"
67
68  let rec string_of_sstmt = function
69      SBlock(stmts) ->
70        "{\n" ^ String.concat "" (List.map string_of_sstmt
      stmts) ^ "}\n"
71    | SExpr(expr) ->     string_of_sexpr expr ^ ";\n";
72    | SReturn(expr) ->   "return " ^ string_of_sexpr expr ^
      ";\n";
73    | SIf(e, s, SBlock([])) ->
74        "if (" ^ string_of_sexpr e ^ ")\n" ^ string_of_sstmt s
75    | SIf(e, s1, s2) ->
76        "if (" ^ string_of_sexpr e ^ ")\n" ^
77        string_of_sstmt s1 ^ "else\n" ^ string_of_sstmt s2
78    | SFor(e1, e2, e3, s) ->
79        "for (" ^ string_of_sexpr e1  ^ " ; " ^ string_of_sexpr
      e2 ^ " ; " ^
80        string_of_sexpr e3  ^ ") " ^ string_of_sstmt s
81    | SWhile(e, s) ->    "while (" ^ string_of_sexpr e ^ ") " ^
      string_of_sstmt s
82
83  let string_of_sfdecl fdecl =
84    string_of_typ fdecl.styp ^ " " ^
85    fdecl.sfname ^ "(" ^ String.concat ", " (List.map
      string_of_args fdecl.sformals) ^
86    ")\n{\n" ^ "" ^ String.concat "" (List.map string_of_sstmt
      fdecl.sbody) ^ "}\n"
87
88  let string_of_sprogram (stmts, funcs) =
89    String.concat "" (List.rev (List.map string_of_sstmt stmts))
      ^ "\n" ^
```

```
90   String.concat "\n" (List.rev (List.map string_of_sfdecl
      funcs))
```

codegen.ml

```
module L = Llvm
module A = Ast
open Sast

module StringMap = Map.Make(String)

type symbol_table = {
  (* Variables bound in current block *)
  variables : L.llvalue StringMap.t;
  (* Enclosing scope *)
  parent : symbol_table option;
}

let translate (stmts, functions) =
  let context = L.global_context () in
  (* Add types to the context so we can use them in our LLVM
   code *)
  let i32_t      = L.i32_type    context
  and i8_t       = L.i8_type     context
  and i1_t       = L.i1_type     context
  and float_t    = L.double_type context
  and void_t     = L.void_type context
  in
  let string_t   = L.pointer_type i8_t
  in
  let list_t     = L.struct_type context [| L.pointer_type
   i8_t (* value *) ; L.pointer_type i8_t (* next*) ; i32_t |]
  in
  let exec_t     = L.struct_type context [| L.pointer_type
   string_t (* path *) ; L.pointer_type list_t (* args *) |]
  in
  let complex_exec_t = L.struct_type context [| i1_t (* 0 for
   simple, 1 for complex *) ; L.pointer_type i8_t (* left
   operand *) ; L.pointer_type i8_t (* right operand *) ;
   i32_t (* op *) |]

  (* Create an LLVM module -- this is a "container" into which
   we'll
     generate actual code *)
  and the_module = L.create_module context "BlueShell" in

  (* Convert BlueShell types to LLVM types *)
  let rec ltype_of_typ = function
      A.Int     -> i32_t
    | A.Bool    -> i1_t
    | A.Float   -> float_t
    | A.Void    -> void_t
    | A.Char    -> string_t
    | A.String  -> string_t
    | A.Exec    -> complex_exec_t
```

```
44      | A.ComplexExec -> complex_exec_t
45      | A.List_type ty    -> list_t
46      | A.Function (ty_list, ty) -> let ret_type =
        L.pointer_type (ltype_of_typ ty) in
47                                    let ltype_helper ty1 =
        (L.pointer_type (ltype_of_typ ty1)) in
48                                    let args_type =
        Array.of_list (List.map ltype_helper ty_list ) in
49       L.function_type ret_type args_type
50      | _ -> raise (Failure "ltype_of_typ fail")
51    in
52
53    (* Define and link execvp helper *)
54    let execvp_t : L.lltype =
55        L.var_arg_function_type (L.pointer_type i8_t) [|
        L.pointer_type i8_t;  L.pointer_type list_t |] in
56    let execvp_func : L.llvalue =
57       L.declare_function "execvp_helper" execvp_t the_module in
58    let recurse_exec_t : L.lltype =
59        L.var_arg_function_type (L.pointer_type i8_t) [|
        L.pointer_type complex_exec_t |] in
60    let recurse_exec_func : L.llvalue =
61       L.declare_function "recurse_exec" recurse_exec_t
        the_module in
62
63    (* Helper function, since the llvalue being returned from
        expr is the 4th elem of a tuple *)
64    let fourth x =
65      (match x with
66      (_, _, _, y) -> y)
67    in
68
69    (* Make a fake "main" that contains our toplevel statements
        *)
70    let main_func = L.define_function "main" (L.function_type
        i32_t [||]) the_module in
71    let main_builder = L.builder_at_end context (L.entry_block
        main_func) in
72
73    (* Helper function to find a name in a symbol table *)
74    let rec lookup (curr_symbol_table : symbol_table) s =
75      try
76        (* Try to find binding in nearest block *)
77        StringMap.find s curr_symbol_table.variables
78      with Not_found -> (* Try looking in outer blocks *)
79        match curr_symbol_table.parent with
80          Some(parent) -> lookup parent s
81        | _ -> raise Not_found
82    in
83
84    let rec expr (curr_symbol_table : symbol_table)
        function_decls builder (func_llvalue : L.llvalue) ((_, e) :
```

```
    sexpr) =
85   (* All literals are allocated on the stack, with pointers
    to them being returned *)
86   match e with
87     SLiteral x -> let int_val = L.const_int i32_t x in
88       let int_mem = L.build_malloc i32_t "int_mem" builder in
89       let _ = L.build_store int_val int_mem builder in
90       (curr_symbol_table, function_decls, builder, int_mem)
91   | SFliteral l -> let float_val = L.const_float_of_string
    float_t l in
92       let float_mem = L.build_malloc float_t "float_mem"
    builder in
93       let _ = L.build_store float_val float_mem builder in
94       (curr_symbol_table, function_decls, builder, float_mem)
95   | SBoolLit b -> let bool_val = L.const_int i1_t (if b then
    1 else 0) in
96       let bool_mem = L.build_malloc i1_t "bool_mem" builder
    in
97       let _ = L.build_store bool_val bool_mem builder in
98       (curr_symbol_table, function_decls, builder, bool_mem)
99   | SId s ->
100      (* Dereference a pointer to the variable's memory
    location *)
101      let address = lookup curr_symbol_table s in
102      (curr_symbol_table, function_decls, builder,
    L.build_load address s builder)
103  | SChar c ->
104    let char_ptr = L.build_global_stringptr c "char" builder
    in
105    let dbl_char_ptr = L.build_malloc string_t
    "double_char_ptr" builder in
106    let _ = L.build_store char_ptr dbl_char_ptr builder in
107    (curr_symbol_table, function_decls, builder,
    dbl_char_ptr)
108  | SString s ->
109    let string_ptr = L.build_global_stringptr s "string"
    builder in
110    let dbl_string_ptr = L.build_malloc string_t
    "double_string_ptr" builder in
111    let _ = L.build_store string_ptr dbl_string_ptr builder
    in
112    (curr_symbol_table, function_decls, builder,
    dbl_string_ptr)
113  | SNoexpr -> (curr_symbol_table, function_decls, builder,
    L.const_int i32_t 0)
114  | SExec (e1, e2) ->
115    (* Create space for an exec struct, populate it, and
    return the pointer *)
116    let struct_space = L.build_malloc exec_t "struct_space"
    builder in
117    let path_ptr = L.build_struct_gep struct_space 0
    "path_ptr" builder in
```

```
118       let (_, _, builder, new_value) = (expr curr_symbol_table
    function_decls builder func_llvalue e1) in
119       let _ = L.build_store new_value path_ptr builder in
120       let args_ptr = L.build_struct_gep struct_space 1
    "args_ptr" builder in
121       let casted_args_ptr = L.build_pointercast args_ptr
    (L.pointer_type (L.pointer_type list_t)) "casted_args_ptr"
    builder in
122       let (_, _, builder, new_value') = (expr
    curr_symbol_table function_decls builder func_llvalue e2) in
123       let _ = L.build_store new_value' casted_args_ptr builder
    in
124       let complex_exec_space = L.build_malloc complex_exec_t
    "complex exec struct" builder in
125       let bool_ptr = L.build_struct_gep complex_exec_space 0
    "complex bool" builder in
126       let exec_ptr = L.build_struct_gep complex_exec_space 1
    "complex e1" builder in
127       let _ = L.build_store (L.const_int i1_t 1) bool_ptr
    builder in
128       let casted_struct_space = L.build_pointercast
    struct_space (L.pointer_type i8_t) "casted_malloc" builder
    in
129       let _ = L.build_store casted_struct_space exec_ptr
    builder in
130       (curr_symbol_table, function_decls, builder,
    complex_exec_space)
131     | SIndex (e1, e2) ->
132       (* Get the list pointer and the index value *)
133       let (curr_symbol_table', new_function_decls, builder,
    e1') = expr curr_symbol_table function_decls builder
    func_llvalue e1 in
134       let (curr_symbol_table'', new_function_decls', builder,
    e2') = expr curr_symbol_table' new_function_decls builder
    func_llvalue e2 in
135       let index_val = L.build_load e2' "index_val" builder in
136       let e1_pointer = L.build_malloc (L.pointer_type list_t)
    "e1 pointer" builder in
137       let _ = L.build_store e1' e1_pointer builder in
138
139       (* Basically have a while loop that goes until counter
    == index *)
140       let counter_ptr = L.build_malloc i32_t "counter_ptr"
    builder in
141       let _ = L.build_store (L.const_int i32_t 0) counter_ptr
    builder in
142       let pred_bb = L.append_block context "index"
    func_llvalue in
143       let _ = L.build_br pred_bb builder in
144       let pred_builder = L.builder_at_end context pred_bb in
145       let bool_val = L.build_icmp L.Icmp.Ne index_val
    (L.build_load counter_ptr "counter" pred_builder) "index
```

```
    pred" pred_builder in

      (* In body of this loop, index to next node *)
      let index_body_bb = L.append_block context "index_body"
    func_llvalue in
      let index_body_builder = L.builder_at_end context
    index_body_bb in
      let counter = L.build_add (L.build_load counter_ptr
    "counter" index_body_builder) (L.const_int i32_t 1)
    "increment counter" index_body_builder in
      let _ = L.build_store counter counter_ptr
    index_body_builder in
      let next_ptr_ptr = L.build_struct_gep (L.build_load
    e1_pointer "get struct" index_body_builder) 1
    "next_struct_ptr" index_body_builder in
      let temp = L.build_load next_ptr_ptr "e1' in while loop"
    index_body_builder in
      let temp' = L.build_pointercast temp ((L.pointer_type
    list_t)) "temp'" index_body_builder in
      let _ = L.build_store temp' e1_pointer
    index_body_builder in
      let casted_ptr_ptr = L.build_pointercast temp
    (L.pointer_type list_t) "casted_ptr_ptr" index_body_builder
    in

      let _ = L.build_store casted_ptr_ptr e1_pointer
    index_body_builder in
      let _ = L.build_br pred_bb index_body_builder in

      (* Once loop is done, dereference ptr to get element *)
      let merge_bb = L.append_block context "index_merge"
    func_llvalue in
      let _ = L.build_cond_br bool_val index_body_bb merge_bb
    pred_builder in
      let merge_body_builder = L.builder_at_end context
    merge_bb in
      let elem_ptr_ptr = L.build_struct_gep (L.build_load
    e1_pointer "get struct" merge_body_builder) 0
    "elem_ptr_ptr" merge_body_builder in

      (* Cast pointer to the type of the list element *)
      let ty = (match (fst e1) with
              List_type typ  -> typ
              | _ -> raise (Failure "should have been caught
    in semant"))
      in
      let casted_ptr = L.build_pointercast elem_ptr_ptr
    (L.pointer_type (L.pointer_type (L.pointer_type
    (ltype_of_typ ty)) )) "casted" merge_body_builder in
      let loaded_temp = L.build_load casted_ptr
    "elem_to_return" merge_body_builder in
```

```
174       let elem_to_return = L.build_load loaded_temp
    "elem_to_return" merge_body_builder in
175       (curr_symbol_table'', new_function_decls',
    merge_body_builder, elem_to_return)
176     | SBinop (e1, op, e2) ->
177       let (t, _) = e1
178       in let (curr_symbol_table', new_function_decls, builder,
    e2') = expr curr_symbol_table function_decls builder
    func_llvalue e2
179       in let (curr_symbol_table'', new_function_decls',
    builder, e1') = expr curr_symbol_table' new_function_decls
    builder func_llvalue e1 in
180       (match op with
181         ExprAssign ->
182           let e2' = (match (snd e1) with
183             (* Special cases for index and path because those
    need to dereference the value being assigned to them *)
184             SIndex _ -> L.build_load e2' "true_value" builder
185             | SPreUnop (Path, _ ) -> L.build_load e2'
    "true_value" builder
186             | _ -> e2')
187           in
188           let _ = L.build_store e2' e1' builder in
189           let new_function_decls'' =
190           (* If it's a function variable, update the
    function_decls *)
191           (match (fst e2) with
192               Function _ -> (match (snd e2) with
193                               SId s1 -> (match (snd e1) with
194                                 SBind (ty, n) ->
195                                 let mapping = StringMap.find
    s1 new_function_decls'
196                                             in StringMap.add n
    mapping new_function_decls'
197                                   | _  -> raise (Failure "Only
    binds can be assigned"))
198                              | _ -> raise (Failure "Only ids
    can be assigned"))
199             | _ -> new_function_decls')
200           in
201           (curr_symbol_table'', new_function_decls'', builder,
    e2')
202         (* For operations, need to dereference both sides and
    store the result back to the memory location *)
203         | Add -> (match t with
204           Float ->
205             let float_mem = L.build_malloc float_t "int_mem"
    builder in
206             let new_float = L.build_fadd (L.build_load e1'
    "left side of fadd" builder ) (L.build_load e2' "right side
    of fadd" builder) "tmp" builder in
```

```
207             let _ = L.build_store new_float float_mem builder
    in
208             (curr_symbol_table'', function_decls, builder,
    float_mem)
209           | Int ->
210             let int_mem = L.build_malloc i32_t "int_mem"
    builder in
211             let new_int = L.build_add (L.build_load e1' "left
    side of add" builder) (L.build_load e2' "right side of add"
    builder) "tmp" builder in
212             let _ = L.build_store new_int int_mem builder in
213             (curr_symbol_table'', function_decls, builder,
    int_mem)
214           | Exec | ComplexExec ->
215             let complex_exec_space = L.build_malloc
    complex_exec_t "complex exec struct" builder in
216             let bool_ptr = L.build_struct_gep
    complex_exec_space 0 "complex bool" builder in
217             let exec1_ptr = L.build_struct_gep
    complex_exec_space 1 "complex e1" builder in
218             let exec2_ptr = L.build_struct_gep
    complex_exec_space 2 "complex e2" builder in
219             let op_ptr = L.build_struct_gep complex_exec_space
    3 "complex op" builder in
220             let _ = L.build_store (L.const_int i1_t 0)
    bool_ptr builder in
221             let casted_e1 = L.build_pointercast e1'
    (L.pointer_type i8_t) "casted_e1" builder in
222             let _ = L.build_store casted_e1 exec1_ptr builder
    in
223             let casted_e2 = L.build_pointercast e2'
    (L.pointer_type i8_t) "casted_e2" builder in
224             let _ = L.build_store casted_e2 exec2_ptr builder
    in
225             let _ = L.build_store (L.const_int i32_t 0) op_ptr
    builder in
226             (curr_symbol_table'', function_decls, builder,
    complex_exec_space)
227
228           | _ -> raise (Failure "semant should have caught add
    with invalid types")
229         )
230         | Sub -> (match t with
231           Float ->
232             let float_mem = L.build_malloc float_t "int_mem"
    builder in
233             let new_float = L.build_fsub (L.build_load e1'
    "left side of fsub" builder ) (L.build_load e2' "right side
    of fsub" builder) "tmp" builder in
234             let _ = L.build_store new_float float_mem builder
    in
```

```
235             (curr_symbol_table'', function_decls, builder,
    float_mem)
236           | Int ->
237             let int_mem = L.build_malloc i32_t "int_mem"
    builder in
238             let new_int = L.build_sub (L.build_load e1' "left
    side of sub" builder) (L.build_load e2' "right side of sub"
    builder) "tmp" builder in
239             let _ = L.build_store new_int int_mem builder in
240             (curr_symbol_table'', function_decls, builder,
    int_mem)
241           | _ -> raise (Failure "semant should have caught sub
    with invalid types")
242         )
243         | Mult -> (match t with
244           Float ->
245             let float_mem = L.build_malloc float_t "int_mem"
    builder in
246             let new_float = L.build_fmul (L.build_load e1'
    "left side of fmult" builder ) (L.build_load e2' "right
    side of fmult" builder) "tmp" builder in
247             let _ = L.build_store new_float float_mem builder
    in
248             (curr_symbol_table'', function_decls, builder,
    float_mem)
249           | Int ->
250             let int_mem = L.build_malloc i32_t "int_mem"
    builder in
251             let new_int = L.build_mul (L.build_load e1' "left
    side of mult" builder) (L.build_load e2' "right side of
    mult" builder) "tmp" builder in
252             let _ = L.build_store new_int int_mem builder in
253             (curr_symbol_table'', function_decls, builder,
    int_mem)
254           | Exec | ComplexExec ->
255             let complex_exec_space = L.build_malloc
    complex_exec_t "complex exec struct" builder in
256             let bool_ptr = L.build_struct_gep
    complex_exec_space 0 "complex bool" builder in
257             let exec1_ptr = L.build_struct_gep
    complex_exec_space 1 "complex e1" builder in
258             let exec2_ptr = L.build_struct_gep
    complex_exec_space 2 "complex e2" builder in
259             let op_ptr = L.build_struct_gep complex_exec_space
    3 "complex op" builder in
260             let _ = L.build_store (L.const_int i1_t 0)
    bool_ptr builder in
261             let casted_e1 = L.build_pointercast e1'
    (L.pointer_type i8_t) "casted_e1" builder in
262             let _ = L.build_store casted_e1 exec1_ptr builder
    in
```

```
263             let casted_e2 = L.build_pointercast e2'
    (L.pointer_type i8_t) "casted_e2" builder in
264             let _ = L.build_store casted_e2 exec2_ptr builder
    in
265             let _ = L.build_store (L.const_int i32_t 1) op_ptr
    builder in
266             (curr_symbol_table'', function_decls, builder,
    complex_exec_space)
267           | _ -> raise (Failure "semant should have caught mul
    with invalid types")
268         )
269         | Div -> (match t with
270           Float ->
271             let float_mem = L.build_malloc float_t "int_mem"
    builder in
272             let new_float = L.build_fdiv (L.build_load e1'
    "left side of fdiv" builder ) (L.build_load e2' "right side
    of fdiv" builder) "tmp" builder in
273             let _ = L.build_store new_float float_mem builder
    in
274             (curr_symbol_table'', function_decls, builder,
    float_mem)
275           | Int ->
276             let int_mem = L.build_malloc i32_t "int_mem"
    builder in
277             let new_int = L.build_sdiv (L.build_load e1' "left
    side of div" builder) (L.build_load e2' "right side of div"
    builder) "tmp" builder in
278             let _ = L.build_store new_int int_mem builder in
279             (curr_symbol_table'', function_decls, builder,
    int_mem)
280           | _ -> raise (Failure "semant should have caught div
    with invalid types")
281         )
282         | Less -> (match t with
283           Float ->
284             let bool_mem = L.build_malloc i1_t "int_mem"
    builder in
285             let new_bool = L.build_fcmp L.Fcmp.Olt
    (L.build_load e1' "left side of flt" builder) (L.build_load
    e2' "right side of flt" builder) "tmp" builder in
286             let _ = L.build_store new_bool bool_mem builder in
287             (curr_symbol_table'', function_decls, builder,
    bool_mem)
288           | Int ->
289             let bool_mem = L.build_malloc i1_t "bool_mem"
    builder in
290             let new_bool = L.build_icmp L.Icmp.Slt
    (L.build_load e1' "left side of lt" builder) (L.build_load
    e2' "right side of lt" builder) "tmp" builder in
291             let _ = L.build_store new_bool bool_mem builder in
```

```
292             (curr_symbol_table'', function_decls, builder,
    bool_mem)
293           | _ -> raise (Failure "semant should have caught
    less with invalid types")
294         )
295         | Leq -> (match t with
296           Float ->
297             let bool_mem = L.build_malloc i1_t "bool_mem"
    builder in
298             let new_bool = L.build_fcmp L.Fcmp.Ole
    (L.build_load e1' "left side of fleq" builder)
    (L.build_load e2' "right side of fle" builder) "tmp"
    builder in
299             let _ = L.build_store new_bool bool_mem builder in
300             (curr_symbol_table'', function_decls, builder,
    bool_mem)
301           | Int ->
302             let bool_mem = L.build_malloc i1_t "bool_mem"
    builder in
303             let new_bool = L.build_icmp L.Icmp.Sle
    (L.build_load e1' "left side of leq" builder) (L.build_load
    e2' "right side of le" builder) "tmp" builder in
304             let _ = L.build_store new_bool bool_mem builder in
305             (curr_symbol_table'', function_decls, builder,
    bool_mem)
306           | _ -> raise (Failure "semant should have caught leq
    with invalid types")
307         )
308         | Greater -> (match t with
309           Float ->
310             let bool_mem = L.build_malloc i1_t "bool_mem"
    builder in
311             let new_bool = L.build_fcmp L.Fcmp.Ogt
    (L.build_load e1' "left side of fgt" builder) (L.build_load
    e2' "right side of fgt" builder) "tmp" builder in
312             let _ = L.build_store new_bool bool_mem builder in
313             (curr_symbol_table'', function_decls, builder,
    bool_mem)
314           | Int ->
315             let bool_mem = L.build_malloc i1_t "bool_mem"
    builder in
316             let new_bool = L.build_icmp L.Icmp.Sgt
    (L.build_load e1' "left side of gt" builder) (L.build_load
    e2' "right side of gt" builder) "tmp" builder in
317             let _ = L.build_store new_bool bool_mem builder in
318             (curr_symbol_table'', function_decls, builder,
    bool_mem)
319           | _ -> raise (Failure "semant should have caught gt
    with invalid types")
320         )
321         | Geq -> (match t with
322           Float ->
```

```
323             let bool_mem = L.build_malloc i1_t "bool_mem"
    builder in
324             let new_bool = L.build_fcmp L.Fcmp.Oge
    (L.build_load e1' "left side of fgeq" builder)
    (L.build_load e2' "right side of fgeq" builder) "tmp"
    builder in
325             let _ = L.build_store new_bool bool_mem builder in
326             (curr_symbol_table'', function_decls, builder,
    bool_mem)
327           | Int ->
328             let bool_mem = L.build_malloc i1_t "bool_mem"
    builder in
329             let new_bool = L.build_icmp L.Icmp.Sge
    (L.build_load e1' "left side of geq" builder) (L.build_load
    e2' "right side of geq" builder) "tmp" builder in
330             let _ = L.build_store new_bool bool_mem builder in
331             (curr_symbol_table'', function_decls, builder,
    bool_mem)
332           | _ -> raise (Failure "semant should have caught geq
    with invalid types")
333         )
334         | And -> (match t with
335           Bool ->
336             let bool_mem = L.build_malloc i1_t "bool_mem"
    builder in
337             let new_bool = L.build_and (L.build_load e1' "left
    side of and" builder) (L.build_load e2' "right side of and"
    builder) "tmp" builder in
338             let _ = L.build_store new_bool bool_mem builder in
339             (curr_symbol_table'', function_decls, builder,
    bool_mem)
340           | _ -> raise (Failure "semant should have caught and
    with invalid types")
341         )
342         | Or -> (match t with
343           Bool ->
344             let bool_mem = L.build_malloc i1_t "bool_mem"
    builder in
345             let new_bool = L.build_or (L.build_load e1' "left
    side of or" builder) (L.build_load e2' "right side of or"
    builder) "tmp" builder in
346             let _ = L.build_store new_bool bool_mem builder in
347             (curr_symbol_table'', function_decls, builder,
    bool_mem)
348           | _ -> raise (Failure "semant should have caught or
    with invalid types")
349         )
350         | Equal -> (match t with
351            Float ->
352             let bool_mem = L.build_malloc i1_t "bool_mem"
    builder in
```

```
353             let new_bool = L.build_fcmp L.Fcmp.Oeq
    (L.build_load e1' "left side of feq" builder) (L.build_load
    e2' "right side of feq" builder) "tmp" builder in
354             let _ = L.build_store new_bool bool_mem builder in
355             (curr_symbol_table'', function_decls, builder,
    bool_mem)
356           | Int ->
357             let bool_mem = L.build_malloc i1_t "bool_mem"
    builder in
358             let new_bool = L.build_icmp L.Icmp.Eq
    (L.build_load e1' "left side of eq" builder) (L.build_load
    e2' "right side of eq" builder) "tmp" builder in
359             let _ = L.build_store new_bool bool_mem builder in
360             (curr_symbol_table'', function_decls, builder,
    bool_mem)
361           | _ -> raise (Failure "semant should have caught eq
    with invalid types")
362         )
363         | Neq -> (match t with
364           Float ->
365             let bool_mem = L.build_malloc i1_t "bool_mem"
    builder in
366             let new_bool = L.build_fcmp L.Fcmp.One
    (L.build_load e1' "left side of fneq" builder)
    (L.build_load e2' "right side of nfeq" builder) "tmp"
    builder in
367             let _ = L.build_store new_bool bool_mem builder in
368             (curr_symbol_table'', function_decls,
    builder,bool_mem)
369           | Int ->
370             let bool_mem = L.build_malloc i1_t "bool_mem"
    builder in
371             let new_bool = L.build_icmp L.Icmp.Ne
    (L.build_load e1' "left side of neq" builder) (L.build_load
    e2' "right side of neq" builder) "tmp" builder in
372             let _ = L.build_store new_bool bool_mem builder in
373             (curr_symbol_table'', function_decls, builder,
    bool_mem)
374           | _ -> raise (Failure "semant should have caught neq
    with invalid types")
375         )
376         | Cons -> let (t2, _) = e2 in
377           (match t2 with
378             (* Cons to an empty list means we need to make a
    new list *)
379             EmptyList -> expr curr_symbol_table function_decls
    builder func_llvalue (List_type t, (SList [e1]))
380             (* Cons to an existing list means that we need to
    append a new node to an already existing head and move the
    head to the new node *)
381             | List_type _ ->
382               let value = e1' in
```

```
383                 let enum_type = match (fst e1) with
384                           Int -> L.const_int i32_t 0
385                           | Float -> L.const_int i32_t 1
386                           | Bool -> L.const_int i32_t 2
387                           | Char -> L.const_int i32_t 3
388                           | String -> L.const_int i32_t 4
389                           | _  -> L.const_int i32_t 5
390                           in
391                 (* allocate space for the element and store *)
392                 let value_ptr = L.build_malloc (L.pointer_type
    (ltype_of_typ (fst
393                 e1))) "value_ptr" builder in
394                 let _ = L.build_store value value_ptr builder in
395                 (* allocate and fill a list node *)
396
397                 let struct_space = L.build_malloc list_t
    "list_node" builder in
398                 let struct_val_ptr = L.build_struct_gep
    struct_space 0
399                 "struct_val_ptr" builder in
400                 let struct_ptr_ptr = L.build_struct_gep
    struct_space 1
401                 "struct_ptr_ptr" builder in
402                 let struct_ty_ptr = L.build_struct_gep
    struct_space 2
403                 "struct_ty_ptr" builder in
404                 let list_ptr = e2' in
405
406                 let casted_ptr_ptr = L.build_pointercast
    struct_ptr_ptr (L.pointer_type (L.pointer_type list_t))
    "casted_ptr_ptr" builder in
407                 let _ = L.build_store list_ptr casted_ptr_ptr
    builder in
408                 let casted_val_ptr = L.build_pointercast
    struct_val_ptr (L.pointer_type (L.pointer_type i8_t))
    "casted_val_ptr" builder in
409                 let casted_val = L.build_pointercast value_ptr
    (L.pointer_type i8_t) "casted_val" builder in
410                 let casted_ty_ptr = L.build_pointercast
    struct_ty_ptr (L.pointer_type i32_t) "casted_ty_ptr"
    builder in
411                 let casted_ty = L.build_pointercast enum_type
    i32_t "casted_ty" builder in
412                 let _ = L.build_store casted_val casted_val_ptr
    builder in
413                 let _ = L.build_store casted_ty casted_ty_ptr
    builder in
414                 (* put value of element into the allocated space
    *)
415                 (curr_symbol_table, function_decls, builder,
    struct_space)
416               | _ -> raise (Failure "incorrect type in cons"))
```

```
417         | Pipe ->
418             let complex_exec_space = L.build_malloc
    complex_exec_t "complex exec struct" builder in
419             let bool_ptr = L.build_struct_gep
    complex_exec_space 0 "complex bool" builder in
420             let exec1_ptr = L.build_struct_gep
    complex_exec_space 1 "complex e1" builder in
421             let exec2_ptr = L.build_struct_gep
    complex_exec_space 2 "complex e2" builder in
422             let op_ptr = L.build_struct_gep complex_exec_space
    3 "complex op" builder in
423             let _ = L.build_store (L.const_int i1_t 0)
    bool_ptr builder in
424             let casted_e1 = L.build_pointercast e1'
    (L.pointer_type i8_t) "casted_e1" builder in
425             let _ = L.build_store casted_e1 exec1_ptr builder
    in
426             let casted_e2 = L.build_pointercast e2'
    (L.pointer_type i8_t) "casted_e2" builder in
427             let _ = L.build_store casted_e2 exec2_ptr builder
    in
428             let _ = L.build_store (L.const_int i32_t 2) op_ptr
    builder in
429             (curr_symbol_table'', function_decls, builder,
    complex_exec_space)
430      )
431      | SPreUnop(op, e) -> (match op with
432          Run ->
433            (* Grab path and args from exec struct and pass to
    execvp *)
434            let (_, _, builder, exec) = expr curr_symbol_table
    function_decls builder func_llvalue e in
435
436            (* Determine whether to recurse or not *)
437            let complex_bool_ptr = L.build_struct_gep exec 0
    "complex_bool_ptr" builder in
438            let complex_bool = L.build_load complex_bool_ptr
    "complex_bool" builder in
439            let return_str_ptr = L.build_malloc (L.pointer_type
    i8_t) "return_str_ptr" builder in
440
441            (* Connect then block for simple executables *)
442            let merge_bb = L.append_block context "run merge"
    func_llvalue in
443            let then_bb = L.append_block context "then"
    func_llvalue in
444            let then_builder = L.builder_at_end context then_bb
    in
445
446            (* Build then block for simple executables *)
447            let simple_exec_ptr = L.build_struct_gep exec 1
    "exec_ptr" then_builder in
```

```
448         let casted_ptr = L.build_pointercast simple_exec_ptr
    (L.pointer_type (L.pointer_type exec_t)) "cast_run"
    then_builder in
449         let simple_exec = L.build_load casted_ptr "exec"
    then_builder in
450
451         let dbl_path_ptr = L.build_struct_gep simple_exec 0
    "dbl_path_ptr" then_builder in
452         let path_ptr = L.build_load dbl_path_ptr "path_ptr"
    then_builder in
453         let path = L.build_load path_ptr "path" then_builder
    in
454         let args_ptr = L.build_struct_gep simple_exec 1
    "args_ptr" then_builder in
455         let args = L.build_load args_ptr "args" then_builder
    in
456
457         (* Execvp will convert from our list representation
    to the array needed *)
458         let return_str = L.build_call execvp_func [| path ;
    args |] "execvp" then_builder in
459         let _ = L.build_store return_str return_str_ptr
    then_builder in
460         let _ = L.build_br merge_bb then_builder in
461         (* End of then block *)
462
463         (* Build else block for complex executables *)
464         let else_bb = L.append_block context "else"
    func_llvalue in
465         let else_builder = L.builder_at_end context else_bb
    in
466
467         let return_str = L.build_call recurse_exec_func [|
    exec |] "recurse_exec" else_builder in
468         let _ = L.build_store return_str return_str_ptr
    else_builder in
469
470         (* After switch statement, finish getting the
    resulting string *)
471         let _ = L.build_br merge_bb else_builder in
472         (* End of else block *)
473
474         (* Execute correct code depending on whether the
    executable is complex or not *)
475         let _ = L.build_cond_br complex_bool then_bb else_bb
    builder in
476
477         (curr_symbol_table, function_decls,
    (L.builder_at_end context merge_bb), return_str_ptr)
478       | Neg ->
479         let (curr_symbol_table'', function_decls', builder,
    e') = expr curr_symbol_table function_decls builder
```

```
    func_llvalue e in
480         let (t,_) = e in
481         (match t with
482         Float ->
483           let float_mem = L.build_malloc float_t "float_mem"
    builder in
484           let new_float =  L.build_fneg (L.build_load e'
    "neg float" builder ) "tmp" builder in
485           let _ = L.build_store new_float float_mem builder
    in
486           (curr_symbol_table'', function_decls', builder,
    float_mem)
487         | Int ->
488           let int_mem = L.build_malloc i32_t "int_mem"
    builder in
489           let new_int =L.build_neg (L.build_load e' "neg
    int" builder) "tmp" builder in
490           let _ = L.build_store new_int int_mem builder in
491           (curr_symbol_table'', function_decls', builder,
    int_mem)
492         | List_type _ ->  raise (Failure "List remove not
    implemented")
493         | _ -> raise (Failure "semant should have caught neg
    with invalid types"))
494       | Not ->
495           let (curr_symbol_table'', function_decls', builder,
    e') = expr curr_symbol_table function_decls builder
    func_llvalue e in
496         let (t,_) = e in
497         (match t with
498           Bool ->
499           let bool_mem = L.build_malloc i1_t "bool_mem"
    builder in
500           let new_bool = L.build_not (L.build_load e' "not
    bool" builder)  "tmp" builder in
501           let _ = L.build_store new_bool bool_mem builder in
502           (curr_symbol_table'', function_decls', builder,
    bool_mem)
503           | _ -> raise (Failure "semant should have caught
    not invalid type"))
504       | Path ->
505           (* Get a pointer to the path of a list *)
506         let (curr_symbol_table', function_decls', builder,
    comp_exec) = expr curr_symbol_table function_decls builder
    func_llvalue e in
507         let simple_exec_ptr = L.build_struct_gep comp_exec 1
    "exec_ptr" builder in
508         let casted_ptr = L.build_pointercast simple_exec_ptr
    (L.pointer_type (L.pointer_type exec_t)) "cast_run" builder
    in
509         let simple_exec = L.build_load casted_ptr "exec"
    builder in
```

```
510         let dbl_path_ptr = L.build_struct_gep simple_exec 0
    "dbl_path_ptr" builder in
511         let path_ptr = L.build_load dbl_path_ptr "path_ptr"
    builder in
512         (curr_symbol_table', function_decls', builder,
    path_ptr)
513       | Length ->
514         (* Get the list pointer and the index value *)
515         let (curr_symbol_table', new_function_decls,
    builder, e1') = expr curr_symbol_table function_decls
    builder func_llvalue e in
516
517         let e1_pointer = L.build_malloc (L.pointer_type
    list_t) "e1 pointer" builder in
518         let _ = L.build_store e1' e1_pointer builder in
519
520         (* Basically have a while loop that goes until
    counter == index *)
521         let counter_ptr = L.build_malloc i32_t "counter_ptr"
    builder in
522         let _ = L.build_store (L.const_int i32_t 0)
    counter_ptr builder in
523         let pred_bb = L.append_block context "length"
    func_llvalue in
524         let _ = L.build_br pred_bb builder in
525         let pred_builder = L.builder_at_end context pred_bb
    in
526
527         let bool_mem = L.build_malloc i1_t "bool_mem"
    pred_builder in
528         let _ = L.build_store (L.build_is_not_null
    (L.build_load e1_pointer "" pred_builder) "" pred_builder)
    bool_mem pred_builder in
529
530         (* In body of this loop, traverse to next node *)
531         let index_body_bb = L.append_block context
    "index_body" func_llvalue in
532         let index_body_builder = L.builder_at_end context
    index_body_bb in
533         let counter = L.build_add (L.build_load counter_ptr
    "counter" index_body_builder) (L.const_int i32_t 1)
    "increment counter" index_body_builder in
534         let _ = L.build_store counter counter_ptr
    index_body_builder in
535         let next_ptr_ptr = L.build_struct_gep (L.build_load
    e1_pointer "get struct" index_body_builder) 1
    "next_struct_ptr" index_body_builder in
536         let temp = L.build_load next_ptr_ptr "e1' in while
    loop" index_body_builder in
537         let temp' = L.build_pointercast temp
    ((L.pointer_type list_t)) "temp'" index_body_builder in
```

```
538         let _ = L.build_store temp' e1_pointer
    index_body_builder in
539         let casted_ptr_ptr = L.build_pointercast temp
    (L.pointer_type list_t) "casted_ptr_ptr" index_body_builder
    in
540
541         let _ = L.build_store casted_ptr_ptr e1_pointer
    index_body_builder in
542         let _ = L.build_br pred_bb index_body_builder in
543
544         (* Once loop is done, return counter *)
545         let merge_bb = L.append_block context "length merge"
    func_llvalue in
546         let _ = L.build_cond_br (L.build_load bool_mem
    "bool_mem" pred_builder) index_body_bb merge_bb
    pred_builder in
547         let merge_body_builder = L.builder_at_end context
    merge_bb in
548
549         (curr_symbol_table', new_function_decls,
    merge_body_builder, counter_ptr)
550    | _    -> raise (Failure "preuop not implemented"))
551  | SList l -> (* Returns a pointer to the first node in the
    list *)
552    (match l with
553    [] -> (curr_symbol_table, function_decls, builder,
    L.const_pointer_null (L.pointer_type list_t))
554    | first :: rest ->
555          let enum_type = match (fst first) with
556          Int -> L.const_int i32_t 0
557          | Float -> L.const_int i32_t 1
558          | Bool -> L.const_int i32_t 2
559          | Char -> L.const_int i32_t 3
560          | String -> L.const_int i32_t 4
561          | _  -> L.const_int i32_t 5
562          in
563          let (_, function_decls', builder, value) = expr
    curr_symbol_table function_decls builder func_llvalue first
564          in
565
566          (* allocate space for the element and store *)
567          let value_ptr = L.build_malloc (L.pointer_type
    (ltype_of_typ (fst
568          first))) "value_ptr" builder in
569            (* to do: strings are pointers but other things
    are
570            not *)
571          let _ = L.build_store value value_ptr builder in
572          (* allocate and fill a list node *)
573
574          let struct_space = L.build_malloc list_t
    "list_node" builder in
```

```
575             let struct_val_ptr = L.build_struct_gep
    struct_space 0
576             "struct_val_ptr" builder in
577             let struct_ptr_ptr = L.build_struct_gep
    struct_space 1
578             "struct_ptr_ptr" builder in
579             let struct_ty_ptr = L.build_struct_gep
    struct_space 2
580             "struct_ty_ptr" builder in
581
582             let (_, function_decls'', builder, list_ptr) =
    expr curr_symbol_table function_decls' builder func_llvalue
    (List_type (fst first), SList(rest))
583             in
584
585             let casted_ptr_ptr = L.build_pointercast
    struct_ptr_ptr (L.pointer_type (L.pointer_type list_t))
    "casted_ptr_ptr" builder in
586             let _ = L.build_store list_ptr casted_ptr_ptr
    builder in
587             let casted_val_ptr = L.build_pointercast
    struct_val_ptr (L.pointer_type (L.pointer_type i8_t))
    "casted_val_ptr" builder in
588             let casted_val = L.build_pointercast value_ptr
    (L.pointer_type i8_t) "casted_val" builder in
589             let casted_ty_ptr = L.build_pointercast
    struct_ty_ptr (L.pointer_type i32_t) "casted_ty_ptr"
    builder in
590             let casted_ty = L.build_pointercast enum_type
    i32_t "casted_ty" builder in
591             let _ = L.build_store casted_val casted_val_ptr
    builder in
592             let _ = L.build_store casted_ty casted_ty_ptr
    builder in
593             (* put value of element into the allocated space *)
594             (curr_symbol_table, function_decls'', builder,
    struct_space ))
595        | SAssign (s, e) ->
596             (* Get memory associated with a variable and update
    it *)
597            let address = lookup curr_symbol_table s in
598            let (_, function_decls', builder, e') = expr
    curr_symbol_table function_decls builder func_llvalue e in
599            let _  = L.build_store e' address builder in
600            let new_function_decls = (match (fst e) with
601              Function _ -> (match (snd e) with
602                               SId s1 -> let mapping =
    StringMap.find s1 function_decls'
603                                         in StringMap.add s
    mapping function_decls'
604                             | _  -> raise (Failure "Only
    ids can be assigned"))
```

```
605             | _ -> function_decls')
606           in (curr_symbol_table, new_function_decls, builder,
    e')
607       | SBind (ty, n)  ->
608           (* Bind is the only case where we need to allocate
    memory corresponding to a variable *)
609           let ptr = L.build_malloc ( L.pointer_type
    (ltype_of_typ ty)) "variable ptr" builder in
610           let new_sym_table = StringMap.add n ptr
    curr_symbol_table.variables in
611           ({ variables = new_sym_table ; parent =
    curr_symbol_table.parent }, function_decls, builder, ptr)
612       | SCall (f, args) ->
613         (* Get all the variables for args, pass them to the
    function *)
614         let (_, fdecl) = StringMap.find f function_decls in
615         let fptr = lookup curr_symbol_table f in
616
617         (* All functions are held as pointers to the address
    of the function, so dereference *)
618         let fval = L.build_load fptr "fval" builder in
619         let evaluate_args = (List.map (expr curr_symbol_table
    function_decls builder func_llvalue) (List.rev args)) in
620         let (curr_symbol_table, function_decls, builder, _) =
    (match evaluate_args with
621           [] -> (curr_symbol_table, function_decls, builder,
    (L.const_null i8_t))
622           | elem :: elems -> elem)
623         in
624         let llargs = List.map fourth (List.rev evaluate_args)
    in
625         let result = (match fdecl.styp with
626                         A.Void -> ""
627                       | _ -> f ^ "_result") in
628         (curr_symbol_table, function_decls, builder,
    L.build_call fval (Array.of_list llargs) result builder)
629         | _ -> raise(Failure "Calling a non function")
630   in
631   let rec stmt ((curr_symbol_table : symbol_table),
    (function_decls : (L.llvalue * sfunc_decl) StringMap.t),
    builder, (fdecl_option: sfunc_decl option), (func_llvalue :
    L.llvalue)) (statement : sstmt) =
632     match statement with
633       SReturn e -> (match fdecl_option with
634         Some(fdecl) ->
635           (* Get the function return type and build the right
    return *)
636           (match fdecl.styp with
637             Void -> let _ = L.build_ret_void builder in
638               (curr_symbol_table, function_decls, builder,
    fdecl_option, func_llvalue)
```

```
639             | _ -> let ret_mem = L.build_malloc (ltype_of_typ
    fdecl.styp) "return malloc" builder in
640                  let (curr_symbol_table, function_decls,
    builder, evaluated_expr) = expr curr_symbol_table
    function_decls builder func_llvalue e in
641                  let ret = L.build_load evaluated_expr
    "return load" builder in
642                  let _ = L.build_store ret ret_mem builder
    in
643                  let _ = L.build_ret ret_mem builder in
644            (curr_symbol_table, function_decls, builder,
    fdecl_option, func_llvalue))
645       | None -> raise (Failure "semant should have caught
    return outside of a function"))
646     | SBlock sl ->
647       (* Fold stmt over a block *)
648       let new_symbol_table = { variables = StringMap.empty ;
    parent = Some curr_symbol_table} in
649       List.fold_left stmt (new_symbol_table, function_decls,
    builder, fdecl_option, func_llvalue) sl
650     | SExpr e ->
651       (* Evaluate an expression, but may possible lead to
    changes in the function_decls or builder *)
652       let (new_symbol_table, new_function_decls, builder,
    expr_val) = expr curr_symbol_table function_decls builder
    func_llvalue e in (new_symbol_table, new_function_decls,
    builder, fdecl_option, func_llvalue)
653     | SIf (predicate, then_stmt, else_stmt) ->
654       (* Branch and return new builder to continue building
    from *)
655       let (curr_symbol_table', new_function_decls, builder,
    bool_val) = expr curr_symbol_table function_decls builder
    func_llvalue predicate in
656       let merge_bb = L.append_block context "if merge"
    func_llvalue in
657       let then_bb = L.append_block context "then" func_llvalue
    in
658       let (_, _, then_builder, _, _) = stmt
    (curr_symbol_table', new_function_decls, (L.builder_at_end
    context then_bb), fdecl_option, func_llvalue) then_stmt in
659       let _ = L.build_br merge_bb then_builder in
660       let else_bb = L.append_block context "else" func_llvalue
    in
661       let (_, _, else_builder, _, _) = stmt
    (curr_symbol_table', new_function_decls, (L.builder_at_end
    context else_bb), fdecl_option, func_llvalue) else_stmt in
662       let _ = L.build_br merge_bb else_builder in
663       let dereferenced_bool = L.build_load bool_val "bool"
    builder in
664       let _ = L.build_cond_br dereferenced_bool then_bb
    else_bb builder in
```

```
665       (curr_symbol_table', new_function_decls,
    (L.builder_at_end context merge_bb), fdecl_option,
    func_llvalue)
666     | SWhile (predicate, body) ->
667       (* Branch and return new builder to continue building
    from *)
668       let pred_bb = L.append_block context "while"
    func_llvalue in
669       let _ = L.build_br pred_bb builder in
670       let body_bb = L.append_block context "while_body"
    func_llvalue in
671       let (_, _, while_builder, _, _) = stmt
    (curr_symbol_table, function_decls, (L.builder_at_end
    context body_bb), fdecl_option, func_llvalue) body in
672       let _ = L.build_br pred_bb while_builder in
673       let pred_builder = L.builder_at_end context pred_bb in
674       let (curr_symbol_table', new_function_decls, builder,
    bool_val) = expr curr_symbol_table function_decls
    pred_builder func_llvalue predicate in
675       let dereferenced_bool = L.build_load bool_val "bool"
    pred_builder in
676       let merge_bb = L.append_block context "while merge"
    func_llvalue in
677       let _ = L.build_cond_br dereferenced_bool body_bb
    merge_bb pred_builder in
678       (curr_symbol_table, new_function_decls,
    (L.builder_at_end context merge_bb), fdecl_option,
    func_llvalue)
679     | SFor (e1, e2, e3, body) ->
680       (* Branch and return new builder to continue building
    from *)
681       stmt (curr_symbol_table, function_decls, builder,
    fdecl_option, func_llvalue) (SBlock [SExpr e1 ; SWhile (e2,
    SBlock [body ; SExpr e3])])
682   in
683
684   (* Define addresses for the body of each function to go *)
685   let function_decls : (L.llvalue * sfunc_decl) StringMap.t =
686     let function_decl m fdecl =
687       let name = fdecl.sfname
688       and formal_types =
689   Array.of_list (List.map (fun (t,_) -> (L.pointer_type
    (ltype_of_typ t))) fdecl.sformals)
690       in let return_type =
691         (match fdecl.styp with
692         Void -> void_t
693         | _ -> (L.pointer_type (ltype_of_typ fdecl.styp)))
694       in let ftype = L.function_type return_type formal_types
    in
695       StringMap.add name (L.define_function name ftype
    the_module, fdecl) m in
696     List.fold_left function_decl StringMap.empty functions
```

```
697   in
698
699   (* Helper function to add pointers to each global function
       to a map *)
700   let func_def name (fdef, fdecl) (m, builder) =
701     let formal_types =
702       Array.of_list (List.map (fun (t,_) -> L.pointer_type
       (ltype_of_typ t)) fdecl.sformals) in
703     let return_type =
704       (match fdecl.styp with
705       Void -> void_t
706       | _ -> (L.pointer_type (ltype_of_typ fdecl.styp))) in
707     let ftype = L.function_type return_type formal_types in
708     let variable = L.build_malloc (L.pointer_type ftype)
       "function def" builder in
709     let _ = L.build_store fdef variable builder in
710     (StringMap.add name variable m, builder)
711   in
712
713   (* Add every function to the main scope *)
714   let curr_symbol_table = { variables = (fst (StringMap.fold
       func_def function_decls (StringMap.empty, main_builder))) ;
       parent = None }
715   in
716
717   (* Build the body of each function using the addresses
       corresponding to each one *)
718   let build_function_body fdecl =
719     let (the_function, _) = StringMap.find fdecl.sfname
       function_decls in
720     let func_builder = L.builder_at_end context (L.entry_block
       the_function) in
721
722     (* Use a symbol table that contains all the globally
       defined function *)
723     let curr_symbol_table = { variables = (fst (StringMap.fold
       func_def function_decls (StringMap.empty, func_builder))) ;
       parent = None } in
724
725     (* Create space for parameters *)
726     let add_formal ((curr_symbol_table : symbol_table),
       function_decls) ((t : A.typ), n) p =
727       let new_map =
728           let old_map = curr_symbol_table.variables in
729           let variable = L.build_malloc (L.pointer_type
       (ltype_of_typ t)) n func_builder in
730           let _ = L.build_store p variable func_builder in
731           StringMap.add n variable old_map
732       in
733       let new_function_decls =
734           let function_decl m (t , name) =
735             (match t with
```

```
736                 A.Function ( _, ty_ret) -> StringMap.add name
    (L.const_int i32_t 32, { styp = ty_ret; sbody = [];
    sformals = []; sfname = name }) m
737               | _ -> m)
738           in List.fold_left function_decl function_decls
    fdecl.sformals
739       in ( { variables = new_map; parent = None },
    new_function_decls )
740     in
741     let ( formals_table, new_function_decls ) =
    List.fold_left2 add_formal (curr_symbol_table,
    function_decls) fdecl.sformals
742         (Array.to_list (L.params the_function))
743
744     (* Go through the statements in each function body *)
745     in let _ = (List.fold_left stmt (formals_table,
    new_function_decls, func_builder, Some fdecl, fst
    (StringMap.find fdecl.sfname new_function_decls))
    (fdecl.sbody))
746     in ()
747   in
748
749   (* Build all functions *)
750   let _ = List.iter build_function_body functions in
751
752   (* Build all toplevel statements *)
753   let (_, _, curr_builder, _, _) = (List.fold_left stmt
     (curr_symbol_table, function_decls, main_builder, None,
     main_func) (List.rev stmts)) in
754
755   (* Wherever the program finishes, make that basic block
     return 0 *)
756   let _ = L.build_ret (L.const_int i32_t 0) curr_builder in
757   the_module
```

exec.c

```c
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <sys/wait.h>
#include "string.h"
#include "assert.h"
#include <fcntl.h>

// amount of output allowed by an executable
const int BUF_SIZE = 16384;

// all temp files to store executable outputs for pipes will start with "temp"
// for example, if a line in BlueShell has 1 pipe, then a file named "temp1.txt" will be used
char* TEMP_FILE = "temp";

// permissions for temp files
const int PERMISSIONS = 0644;

// count for number of temps (to avoid nested pipes colliding with the same temp file)
int num_temps = 0;

// struct to represent a list node (linked list under the hood)
struct list {
    void **val;
    struct list* next;

    // type to cast arguments to (see Type enum below)
    int typ;
};

// struct to represent a simple executable
struct simple_exec {
    char **path;
    struct list* args;
};

// struct to represent a complex executable
struct complex_exec {
    // 1 means simple, 0 means complex
    int is_simple;

    // void pointers may point to simple OR complex executables
    void *e1;
    void *e2;

    // operation (see Opcode enum below)
    int op;
```

```
};

char* recurse_exec(struct complex_exec *e);
char* pipe_helper(struct complex_exec *e, int is_left);
char* recurse_helper(struct complex_exec *e);
char* execvp_helper(char *path, struct list *orig_args);
char* execvp_execute(char *path, struct list *orig_args);
char** organize_args(char *path, struct list *orig_args);

enum Type { INT = 0, FLOAT = 1, BOOL = 2, CHAR = 3, STRING =
    4, OTHER = 5 };
enum Opcode { CONCAT = 0, SEQ = 1, PIPE = 2 };

// this is called directly by the LLVM code
char* recurse_exec(struct complex_exec *e) {
    char *final_str = recurse_helper(e);
    fprintf(stdout, "%s", final_str); // DON'T REMOVE THIS,
    IT'S TO OUTPUT
    return final_str;
}

// this is called once we've seen a pipe
char* pipe_helper(struct complex_exec *e, int is_left) {
    char *final_str;

    // handle simple executables
    if (e->is_simple == 1) {
      struct simple_exec* simple = (struct
    simple_exec*)(e->e1);
      char *simple_path = *(char **)simple->path;

      struct list *orig_args = simple->args;
      // if it's the leftmost simple executable, read from the
    pipe
      if (is_left) {
        char **args = organize_args(simple_path, orig_args);

        // pipe to read the output of execvp back to this
    program
        int get_output_fds[2];
        pipe(get_output_fds);

        // open file containing cached result of left side of
    pipe
        char *int_string = calloc(32, 1);
        sprintf(int_string, "%d", num_temps);
        char * temp = calloc(8, 1);
        strcpy(temp, TEMP_FILE);
        char * file = strcat(temp, int_string);
        file = strcat(file, ".txt");

```

```
93          int file_fd = open(file, O_RDWR | O_CREAT,
    PERMISSIONS);
94
95          // fork
96          int exec_rc = fork();
97          int status = 0;
98          if (exec_rc == 0) {
99            // attach the file to the next executable as stdin
100           close(0);
101           dup2(file_fd, 0);
102
103           // attach the pipe back to this program as stdout
104           close(get_output_fds[0]);
105           dup2(get_output_fds[1], 1);
106           close(get_output_fds[1]);
107           int err = execvp(simple_path, args);
108           exit(1);
109         }
110
111         // wait until the forked process finishes
112         int still_waiting = wait(&status);
113         while (still_waiting > 0) {
114             still_waiting = wait(&status);
115         }
116
117         // read the output of the process and save it
118         char *buf = calloc(BUF_SIZE, 1);
119
120         read(get_output_fds[0], buf, BUF_SIZE);
121         close(file_fd);
122         remove(file);
123         return buf;
124       }
125
126       return execvp_execute(simple_path, orig_args);
127     }
128     else {
129       struct complex_exec* complex1 = (struct
    complex_exec*)(e->e1);
130       struct complex_exec* complex2 = (struct
    complex_exec*)(e->e2);
131       char *result1;
132       char *result2;
133       switch (e->op) {
134         case CONCAT:
135           if (complex1->is_simple == 1) {
136             result1 = pipe_helper(complex1, 1);
137           }
138           else {
139             result1 = pipe_helper(complex1, 0);
140           }
141           result2 = recurse_helper(complex2);
```

```
142
143           // concatenates the results of the two executables
144           final_str = strcat(result1, result2);
145           break;
146
147         case SEQ:
148           if (complex1->is_simple == 1) {
149             result1 = pipe_helper(complex1, 1);
150           }
151           else {
152             result1 = pipe_helper(complex1, 0);
153           }
154           result2 = recurse_helper(complex2);
155
156           // only returns the right executable
157           final_str = result2;
158           break;
159
160         case PIPE:
161           num_temps++;
162           if (complex1->is_simple == 1) {
163             result1 = pipe_helper(complex1, 1);
164           }
165           else {
166             result1 = pipe_helper(complex1, 0);
167           }
168
169           // create a file to cache the result of the left
    executable
170           char *int_string = calloc(32, 1);
171           sprintf(int_string, "%d", num_temps);
172           char* temp = calloc(8, 1);
173           strcpy(temp, TEMP_FILE);
174           char* file = strcat(temp, int_string);
175           file = strcat(file, ".txt");
176           int file_fd = open(file, O_RDWR | O_CREAT,
    PERMISSIONS);
177           write(file_fd, result1, BUF_SIZE);
178
179           // checks if there is a need to recurse further on
    the right executable
180           if (complex2->is_simple == 1) {
181             result2 = pipe_helper(complex2, 1);
182           }
183           else {
184             result2 = pipe_helper(complex2, 0);
185           }
186           final_str = result2;
187
188           // delete the cached file
189           close(file_fd);
190           remove(file);
```

```
191             break;
192         }
193     }
194     return final_str;
195 }
196 
197 // regular recursive case (doesn't handle the stdin end of a
    pipe)
198 char* recurse_helper(struct complex_exec *e) {
199     char *final_str;
200 
201     // if simple, just execute normally
202     if (e->is_simple == 1) {
203       struct simple_exec* simple = (struct
    simple_exec*)(e->e1);
204       char *simple_path = *(char **)simple->path;
205 
206       struct list *orig_args = simple->args;
207       char *final_str = execvp_execute(simple_path, orig_args);
208       return final_str;
209     }
210     else {
211       struct complex_exec* complex1 = (struct
    complex_exec*)(e->e1);
212       struct complex_exec* complex2 = (struct
    complex_exec*)(e->e2);
213       char *result1;
214       char *result2;
215       switch (e->op) {
216         // concat executes both ends and returns the
    concatenated result
217         case CONCAT:
218           result1 = recurse_helper(complex1);
219           result2 = recurse_helper(complex2);
220           final_str = strcat(result1, result2);
221           break;
222 
223         // sequence executes both ends and only returns the
    right result
224         case SEQ:
225           result1 = recurse_helper(complex1);
226           result2 = recurse_helper(complex2);
227           final_str = result2;
228           break;
229 
230         // pipe caches the result of the left side in a file
    and calls pipe_helper
231         // pipe_helper looks for the appropriate executable to
    read the cached file as stdin
232         case PIPE:
233           num_temps++;
234           result1 = recurse_helper(complex1);
```

```
235
236             char *int_string = calloc(32, 1);
237             sprintf(int_string, "%d", num_temps);
238             char *temp = calloc(8, 1);
239             strcpy(temp, TEMP_FILE);
240             char* file = strcat(temp, int_string);
241             file = strcat(file, ".txt");
242             int file_fd = open(file, O_RDWR | O_CREAT,
    PERMISSIONS);
243             write(file_fd, result1, BUF_SIZE);
244
245             int fds[2];
246             pipe(fds);
247             if (complex2->is_simple == 1) {
248               result2 = pipe_helper(complex2, 1);
249             }
250             else {
251               result2 = pipe_helper(complex2, 0);
252             }
253
254             final_str = result2;
255             break;
256           }
257         }
258         return final_str;
259 }
260
261 /* execvp_helper
262 Purpose: Forks and calls execvp on the path and arguments,
    interfacing with the Blue Shell codegen.
263 Arguments: char* representing path, char* array representing
    arguments
264 */
265 char* execvp_helper(char *path, struct list *orig_args) {
266     char *return_string = execvp_execute(path, orig_args);
267     fprintf(stdout, "%s", return_string); // DON'T REMOVE
    THIS, IT'S NOT A DEBUG STATEMENT
268
269     return return_string;
270 }
271
272 // move args from linked list into array for execvp to use
273 char **organize_args(char* path, struct list *orig_args) {
274         int i = 0;
275         struct list *args_copy = orig_args;
276
277         char* str;
278         char** temp;
279
280         // count the number of args
281         while (args_copy != NULL) {
282             char **temp1 = *(char***)(args_copy->val);
```

```
283             i += 1;
284             args_copy = args_copy->next;
285         }
286 
287         // move args from linked list into array for execvp to
    use
288         char **args = malloc(sizeof(char*) * (i + 2));
289         args_copy = orig_args;
290         args[0] = path;
291         for (int j = 0; j < i; j++) {
292           str = calloc(BUF_SIZE, 1);
293           int typ = orig_args->typ;
294 
295           // cast differently depending on type
296           switch (typ) {
297             case INT:
298               sprintf(str, "%d", **(int **)(args_copy->val));
299               break;
300             case FLOAT:
301               sprintf(str, "%lf", **(double
    **)(args_copy->val));
302               break;
303             case BOOL:
304               if ((**(int **)(args_copy->val) & 1) == 0) {
305                 strcpy(str, "false");
306               } else {
307                 strcpy(str, "true");
308               }
309               break;
310             case CHAR:
311               temp = *(char ***)(args_copy->val);
312               strcpy(str, *temp);
313               break;
314             case STRING:
315               temp = *(char ***)(args_copy->val);
316               strcpy(str, *temp);
317               break;
318             case OTHER:
319                 fprintf(stderr, "Can only have lists of ints,
    bools, floats, chars, or string in executable");
320                 exit(1);
321             }
322 
323           args[j + 1] = str;
324           args_copy = args_copy->next;
325         }
326 
327         // last argument to execvp must be NULL
328         args[i + 1] = NULL;
329         return args;
330 }
331 
```

```
332 // fork and run the executable, saving the result in this
    program
333 char *execvp_execute(char *path, struct list *orig_args) {
334         char **args = organize_args(path, orig_args);
335 
336         // fork and run the executable
337         int fds[2];
338         pipe(fds);
339 
340         int rc = fork();
341         int status = 0;
342         if (rc == 0) {
343             // pipe stdout of the executable back to this
    program
344             close(fds[0]);
345             dup2(fds[1], 1);
346             close(fds[1]);
347             int err = execvp(path, args);
348             exit(1);
349         }
350         int still_waiting = wait(&status);
351         while (still_waiting > 0) {
352             still_waiting = wait(&status);
353         }
354 
355         char *buf = calloc(BUF_SIZE, 1);
356         read(fds[0], buf, BUF_SIZE);
357         return buf;
358 }
```

toplevel.ml

```
(* toplevel.ml *)
(* BlueShell *)
(* Kenny Lin, Alan Luc, Tina Ma, Mary-Joy Sidhom *)

open Ast
open Sast

type action = Ast | Sast | LLVM_IR | Compile

let () =
  let action = ref Compile in
  let set_action a () = action := a in
  let speclist = [
    ("-a", Arg.Unit (set_action Ast), "Print the AST");
    ("-s", Arg.Unit (set_action Sast), "Print the SAST");
    ("-l", Arg.Unit (set_action LLVM_IR), "Print the generated
     LLVM IR");
    ("-c", Arg.Unit (set_action Compile),
      "Check and print the generated LLVM IR (default)");
  ] in
  let usage_msg = "usage: ./microc.native [-a|-s|-l|-c]
   [file.mc]" in
  let channel = ref stdin in
  Arg.parse speclist (fun filename -> channel := open_in
   filename) usage_msg;

  let lexbuf = Lexing.from_channel !channel in
  let ast = Parser.program Scanner.tokenize lexbuf in
  match !action with
    Ast -> print_string (Ast.string_of_program ast)
  | _ -> let sast = Semant.check ast in
    match !action with
      Ast     -> ()
    | Sast    -> print_string (Sast.string_of_sprogram sast)
    | LLVM_IR -> print_string (Llvm.string_of_llmodule
   (Codegen.translate sast))
    | Compile -> let m = Codegen.translate sast in
    Llvm_analysis.assert_valid_module m;
    print_string (Llvm.string_of_llmodule m)
```

8.2 Testing Scripts

Makefile

```
1
2  # "make all" builds the executable
3  .PHONY : all
4  all : toplevel.native exec.o
5
6  # "make test" compiles everything and runs the regression tests
7  .PHONY : test
8  test : all testall.sh
9      ./testall.sh
10
11 # "make test-sast" compiles everything and runs the regression
       tests for
12 # semantic checking
13 .PHONY : test-sast
14 test-sast : all testsast.sh
15     ./testsast.sh
16
17 # 'make gold' compiles everything and re-runs all the tests to
       produce new
18 # gold standards. DO NOT RUN UNLESS YOU WANT ALL GOLD
       STANDARDS TO BE REMADE
19 .PHONY : gold
20 gold : all make-gsts.sh
21     ./make-gsts.sh
22
23 # "make toplevel.native" builds the scanner, parser, and
       toplevel for testing
24 toplevel.native :parser.mly scanner.mll codegen.ml semant.ml
       toplevel.ml
25     opam exec -- \
26     ocamlbuild -use-ocamlfind toplevel.native
27
28
29 # "make clean" removes all generated files
30 .PHONY : clean
31 clean :
32     ocamlbuild -clean
33     rm -rf testall.log ocamllllvm *.diff *.tsout *.llvm *.o
       *.llvm.s *.out *.exe
34
35 # compiles the helper C file which executes shell commands
36 exec : exec.c
37     cc -o exec
38
39 # Filling the ziploc
40
41 # SP passing and failing tests for the scanner and parser
```

```
SPTESTS = $(shell find sp-tests -type f -name 'test*.bs' -exec basename {} \;)

SPFAILS = $(shell find sp-tests -type f -name 'fail*.bs' -exec basename {} \;)

SUCCSP_NAMES = $(SPTESTS:%.bs=%)
FAILSP_NAMES = $(SPFAILS:%.bs=%)

print_succsp:
	@echo $(SUCCSP_NAMES)

print_failsp:
	@echo $(FAILSP_NAMES)

# sast tests
SAST_TESTS = $(shell find sast-tests -type f -name 'test*.bs' -exec basename {} \;)

SAST_FAILS = $(shell find sast-tests -type f -name 'fail*.bs' -exec basename {} \;)


SUCCSAST_NAMES = $(SAST_TESTS:%.bs=%)
FAILSAST_NAMES = $(SAST_FAILS:%.bs=%)

print_succsast:
	@echo $(SUCCSAST_NAMES)

print_failsast:
	@echo $(FAILSAST_NAMES)


# tests for codegen
TESTS = $(shell find tests -type f -name 'test*.bs' -exec basename {} \;)

FAILS = $(shell find tests -type f -name 'fail*.bs' -exec basename {} \;)

TESTFILES = $(TESTS) $(TESTS:%.bs=gsts/%.gst) \
            $(FAILS) $(FAILS:%.bs=gsts/%.gst)

SAST_TESTFILES = $(SAST_TESTS) $(SAST_TESTS:%.bs=gsts/%.gst) \
                 $(SAST_FAILS) $(SAST_FAILS:%.bs=gsts/%.gst) \

ZIPFILES = ast.ml scanner.mll toplevel.ml parser.mly sast.ml semant.ml \
           codegen.ml _tags exec.c testall.sh compile.sh README Makefile \
            tests sast-tests sp-tests make-gsts.sh demo-programs sample-files
```

```
85
86  # zips files and tests together
87  bostonbitpackers.zip : $(ZIPFILES)
88      mkdir blueshell && cp -r $(ZIPFILES) blueshell && \
89      zip -r bostonbitpackers.zip blueshell && rm -r blueshell
90
91  # prints the list of tests which should pass
92
93  TESTNAMES = $(TESTS:%.bs=%)
94
95  FAILTESTNAMES = $(FAILS:%.bs=%)
96
97  print_succtests:
98      @echo $(TESTNAMES)
99
100 # prints the list of tests which should fail
101 print_failtests:
102     @echo $(FAILTESTNAMES)
103
104 print_files:
105     @echo $(TESTFILES)
106
107 #removes .out and .diff files produced by the testing script
108 clean_tests:
109     rm -rf tests/diff/*.diff tests/out/*.out
110
111 # removes .exes produced
112 clean_exes:
113     rm -rf *.exe
114
115 clean_intermediates:
116     rm -rf *.s *.llvm
```

testall.sh

```
#!/bin/sh
# testall runs all tests in the corresponding test directory and compares them to the
# gold standard. The gold standards must have already been created for this
# script to work.

tests=$(make print_succtests)
fail_tests=$(make print_failtests)
test_dir="tests/"

sast_tests=$(make print_succsast)
sast_fail=$(make print_failsast)
sast_dir="sast-tests"

sp_tests=$(make print_succsp)
sp_fail=$(make print_failsp)
sp_dir="sp-tests"

Usage() {
    echo "Usage: ./testall.sh [-sast | -sp | [-a | -s <testname>] [-keept] [-keepc]]  \n
          Flags:
            -sast: Run sast tests
            -sp: Run Scanner-parser tests
            -a : run all regular executable tests specified in /tests directories
                 and test against expected output
            -s <testname>:
                Run a single executable test located in test/<fail|tests>-<testname>.bs
                and tests against expected output
            -keept: Optional: Keep intermediate output files produced during testing
                    (.out and .diff files)
            -keepc: Keep intermediate output files produced during
                   compilation (.s and .llvm files)"
    exit
}


# runs all sast tests
run_sast_tests()
{
    echo "************************RUNNING SAST TESTS************************\n"
    echo "************************RUNNING SAST SUCCESS TESTS************************\n"
    for test in $sast_tests
```

```
42      do
43          echo "Running test $test........"
44          file_name="${sast_dir}/${test}.bs"
45          gold_standard="${sast_dir}/${test}.gst"
46          ./toplevel.native -s < $file_name >
    "${sast_dir}/out/$test.out"
47          diff "${sast_dir}/out/$test.out" $gold_standard >
    "${sast_dir}/diff/$test.diff"
48          if [ -s "${sast_dir}/diff/$test.diff" ]; then
49              echo "\nERROR: SAST FOR ${test} DOES NOT MATCH
    GOLD STANDARD\n"
50              echo "The difference: \n"
51              cat ${sast_dir}/diff/$test.diff
52          else
53              echo "PASSED \n"
54          fi
55
56      done
57
58      echo "\n"
59      echo "\n"
60      echo "\n"
61      echo "**********************RUNNING SAST FAILURE
    TESTS**********************\n"
62
63      # cringe fail test compilation
64      for ftest in $sast_fail
65      do
66          echo "Running failure test $ftest............"
67          file_name="${sast_dir}/${ftest}.bs"
68          fail_standard="${sast_dir}/${ftest}.gst"
69          ./toplevel.native -a < $file_name 2>
    "${sast_dir}/out/$ftest.out"
70          diff "${sast_dir}/out/$ftest.out" $fail_standard >
    "${sast_dir}/diff/$ftest.diff"
71          if [ -s "${sast_dir}/diff/$ftest.diff" ]; then
72              echo "ERROR: OUTPUT FOR ${ftest} DOES NOT MATCH
    EXPECTED OUTPUT \n "
73              echo "The difference: \n"
74              cat "${sast_dir}/diff/$ftest.diff"
75          else
76              echo "PASSED \n"
77          fi
78
79      done
80
81      echo "removing .out and .diff files created:"
82
83      diffpath="sast-tests/diff/*"
84      rm -f $path
85
86      outpath="sast-tests/gsts/*"
```

```
87
88      rm -f $path
89
90      echo "\n"
91
92      echo "bye"
93      # bye
94  }
95
96
97
98  # runs all scanner parser tests
99  run_sp_tests()
100 {
101     echo "************************RUNNING SP
    TESTS************************\n"
102     echo "************************RUNNING SUCCESS
    TESTS************************\n"
103     for test in $sp_tests
104     do
105         echo "Running test $test........"
106         file_name="${sp_dir}/${test}.bs"
107         gold_standard="${sp_dir}/${test}.gst"
108         ./toplevel.native -a < $file_name >
    "${sp_dir}/out/$test.out"
109         diff "${sp_dir}/out/${test}.out" $gold_standard >
    "${sp_dir}/diff/$test.diff"
110         if [ -s "${sp_dir}/diff/$test.diff" ]; then
111             echo "\nERROR: AST FOR ${test} DOES NOT MATCH
    GSAST\n\n"
112             cat "${sp_dir}/diff/$test.diff"
113         else
114             echo "PASSED \n"
115         fi
116
117     done
118
119     echo "\n"
120     echo "\n"
121     echo "\n"
122     echo "************************RUNNING FAILURE
    TESTS************************\n"
123
124     # cringe fail test compilation
125     for ftest in $sp_fail
126     do
127         echo "Running failure test $ftest............"
128         file_name="${sp_dir}/${ftest}.bs"
129         fail_standard="${sp_dir}/${ftest}.gst"
130         ./toplevel.native < $file_name 2>
    "${sp_dir}/out/$ftest.out"
```

```
131         diff "${sp_dir}/out/${ftest}.out" $fail_standard >
    "${sp_dir}/diff/$ftest.diff"
132         if [ -s "${sp_dir}/diff/$ftest.diff" ]; then
133             echo "ERROR: OUTPUT FOR ${ftest} DOES NOT MATCH
    EXPECTED OUTPUT \n "
134             cat "${sp_dir}/diff/$ftest.diff"
135         else
136             echo "PASSED \n"
137         fi
138
139     done
140
141     echo "removing .out and .diff files created:"
142
143     make clean_tests
144     echo "\n"
145
146     echo "bye"
147     exit
148     # bye
149
150 }
151
152
153 # cecks if a file exists
154 check_success() {
155     if [ $1 -ne 0 ]
156         # last command failed
157         then
158             echo "Previous command failed with exit code
    {$1}\n"
159             echo  "Exiting script now...."
160             exit
161     fi
162 }
163
164
165 # compiles one .bs file into an executable
166 compile_one_test() {
167     echo $1
168     ./toplevel.native < "tests/$1.bs" > $1.llvm
169     check_success $?
170     llc "-relocation-model=pic" $1.llvm
171     check_success $?
172     cc -c exec.c # links with our c file
173     cc $1.llvm.s exec.o -o $1.exe
174     check_success $?
175 }
176
177 # runs one single executable tests and compares it to its gold
    standard
178 run_single_test() {
```

```
    testname=$1
    testpath="tests/$testname.bs"
    if [ ! -f $testpath ]
    then
        echo "File $testpath doesn't exist"
        return
    fi
    # we're in top dir

    compile_one_test $testname
    # we're in tests/
    # copy exe into testing directory
    output="$testname.exe"
        echo "RUNNING TEST ${testname}......"
    ./$output > "tests/out/$testname.out"

    #compare with gst
    gst="tests/gsts/$testname.gst"
    if [ ! -f $gst ]
    then
        echo "File $gst does not exist"
        return
    else
        diff "tests/out/$testname.out" $gst >
   "tests/diff/$testname.diff"
        if [ -s "tests/diff/$testname.diff" ]; then
                echo "ERROR: OUTPUT FOR $testname DOES NOT
   MATCH EXPECTED OUTPUT \n "
                cat "tests/diff/$testname.diff"
            else
                echo "PASSED \n"
        fi

    fi
}

# runs one failure tests; should semantically fail
run_fail_test() {
    ftest=$1
    echo "Running failure test $1..........."
    file_name="tests/${ftest}.bs"
    fail_standard="tests/gsts/${ftest}.gst"
    ./toplevel.native -s < $file_name 2> "tests/out/$ftest.out"
    diff "tests/out/$ftest.out" $fail_standard >
   "tests/diff/$ftest.diff"
    if [ -s $ftest.diff ]; then
        echo "ERROR: OUTPUT FOR ${ftest} DOES NOT MATCH
   EXPECTED OUTPUT \n "
        cat $ftest.diff
    else
        echo "PASSED \n"
    fi
```

```
}


# runs all tests
run_all_tests() {
    # get all test names

    echo "ABOUT TO RUN ${#tests[@]} TESTS....."
    for test in $tests
    do
      name=${test%.*}
      run_single_test $name
    done
    for test in $fail_tests
    do
      #name=${test::-3}
      name=${test%.*}
      run_fail_test $name
    done
    # for each test name call run single test on it
}

# Given a dummy string, a file name, and potentially two flags,
# clean up the intermediate files and keep ones specified by the flags.
# see Usage function to get flag specifications.
clean_up() {
    if [ $# -eq 2 ]
    then # just file name was passed in
        echo "Removing all intermediate files created by $2 (.s, .llvm, .exe, .out, and .diff)..."
        rm $2.llvm
        rm $2.llvm.s
        rm $2.exe
        rm $2.out
        rm $2.diff
    else
        keepc=false
        keept=false

        if [[ $3 = "-keept" ]] || [[ $4 = "-keept" ]]
        then
            keept=true
        fi

        if [[ $4 = "-keepc" ]] || [[ $3 = "-keepc" ]]
        then
            keepc=true
        fi

        if [ "$keepc" = true ]
        then
```

```
277             echo "Keeping intermediate compiler files..."
278         else
279             echo "Removing all .s, .llvm, and .exe files
    created by $2..."
280             rm $2.llvm
281             rm $2.llvm.s
282             rm $2.exe
283         fi
284 
285         if [ "$keept" = true ]
286         then
287             echo "Keeping intermediate testing files..."
288         else
289             echo "Removing all .out and .diff files created by
    $2..."
290             rm $2.out
291             rm $2.diff
292         fi
293     fi
294     echo "Done. Bye!"
295 }
296 
297 
298 # THE PROGRAM IS STARTING
299 
300 
301 # wrong number of arguments
302 if [ $# -lt 1 ]
303     then
304     Usage
305 fi
306 
307 make clean
308 echo "Making...."
309 make
310 
311 if [ $1 = "-sast" ]
312     then
313     run_sast_tests
314     exit
315 fi
316 
317 
318 if [ $1 = "-sp" ]
319     then
320     run_sp_tests
321     exit
322 fi
323 
324 
325 # -a runs all tests
326 if [ $1 = "-a" ]
```

```
    then
    run_all_tests
    if [ $# -eq 1 ]
    then
        echo "Removing all intermediate outputs (.s, .llvm,
   .exes, .out, and .diff) files..."
        make clean_intermediates
        make clean_tests
        make clean_exes
        echo "Done. Bye!"
    else
        keepc=false
        keept=false

        if [[ $2 = "-keept" ]] || [[ $3 = "-keept" ]]
        then
            keept=true
        fi

        if [[ $2 = "-keepc" ]] || [[ $3 = "-keepc" ]]
        then
            keepc=true
        fi

        if [ "$keepc" = true ]
        then
            echo "Keeping intermediate compiler files..."
        else
            echo "Removing all .s, .llvm, and .exe files..."
            make clean_intermediates
            make clean_exes
        fi

        if [ "$keept" = true ]
        then
            echo "Keeping intermediate testing files..."
        else
            echo "Removing all .out and .diff files created..."
            make clean_tests
        fi
        echo "Done. Bye!"
    fi

    exit
fi

# -s runs one single test
if [ $1 = "-s" ]
    then
    # $2 is the file name
    run_single_test $2
    cd ../
```

```
    clean_up "dummy" $2 $3 $4
    exit
fi

Usage


# keep intermediatery tests
```

make-gsts.sh

```
#!/bin/sh


# Creates new gold standards for tests. Type of test (e2e,
    sast, scanner-parser)
# is specified through flags


# info for e2e tests
e2e_tests=$(make print_succtests)
e2e_fail=$(make print_failtests)
test_dir="tests/"

sast_tests=$(make print_succsast)
sast_fail=$(make print_failsast)
sast_dir="sast-tests"

sp_tests=$(make print_succsp)
sp_fail=$(make print_failsp)
sp_dir="sp-tests"

## sp test info


Usage() {
    echo "./make-gsts.sh [-sp | -sast | -e2e] [test-name]
        -sp :create gsts for scanner-parser tests
        -sast: create gsts for SAST tests
        -e2e: create gsts for e2e tests
        test-name: optional string that makes gold standard
    for test test-name"
}


check_success() {
    if [ $1 -ne 0 ];
        # last command failed
        then
            echo "Previous command failed with exit code
    {$1}\n"
            echo  "Exiting script now...."
            exit
    fi
}

# compile one .bs test into an executable
compile_one_test() {
    ./toplevel.native < "tests/$1.bs" > $1.llvm
    check_success $?
    llc "-relocation-model=pic" $1.llvm
```

```
    check_success $?
    cc -c exec.c # links with our c file
    cc $1.llvm.s exec.o -o $1.exe
    check_success $?
}

# create one GST
run_one_gst() {
    test=$1
    file_name="${test_dir}${test}.bs"
    echo "making gst file $file_name...\n"
    if [ ! -f $file_name ]; then
        echo "****ALERT***** Test ${test} doesn't exist. we're not gonna try :/ \n\n\n\n"
        continue;
    fi
    gold_standard="${gsts_dir}${test}.gst"
    touch $gold_standard
    # echo $gold_standard

    type=${test::4}
    if [ $type == "fail" ]; then
        # get from stderr
        ./toplevel.native -s < $file_name 2> $gold_standard
    else
        if [ $type == "test" ]; then
            # actually compile and run
            compile_one_test $test
            output="$test.exe"
            ./$output > $gold_standard
        else
            echo "*****Alert**** test type is not of fail or tests :( - test name should start with fail- or test-\n"
        fi
    fi
}

# create one sast gst
run_sast_gst()
{
    test=$1
    file_name="sast-tests/${test}.bs"
    echo "making gst file $file_name...\n"
    if [ ! -f $file_name ]; then
        echo "****ALERT***** Test ${test} doesn't exist. we're not gonna try :/ \n\n\n\n"
        continue;
    fi
    gold_standard="sast-tests/${test}.gst"
    touch $gold_standard
    type=${test::4}
    if [ $type == "fail" ]; then
```

```
        # get from stderr
        ./toplevel.native -s < $file_name 2> $gold_standard
    else
        ./toplevel.native -s < $file_name > $gold_standard
    fi

}

#create a single gst of a scanner-parser test
run_sp_gst()
{
    test=$1
    file_name="sp-tests/${test}.bs"
    echo "making gst file $file_name...\n"
    if [ ! -f $file_name ]; then
        echo "****ALERT***** Test ${test} doesn't exist. we're not gonna try :/ \n\n\n\n"
        continue;
    fi
    gold_standard="sp-tests/${test}.gst"
    touch $gold_standard
    type=${test::4}
    if [ $type == "fail" ]; then
        # get from stderr
        ./toplevel.native -a < $file_name 2> $gold_standard
    else
        # get standard output
        ./toplevel.native -a < $file_name > $gold_standard
    fi

}


if [ $# -lt 1 ];
    then
    Usage
fi

make

# scanner parser gsts
if [ $1 = "-sp" ]; then
    if [ "$#" -eq 1 ]; then
    echo "Making gold standard for all scanner-parser tests:"
        for test in $sp_tests; do
            run_sp_gst $test
        done

        for test in $sp_fail; do
            run_sp_gst $test
        done
    fi
```

```
148     exit
149 else
150     if [ "$#" -eq 2 ]; then
151         echo "Making gold standard for one scanner-parser test
        $2:"
152         run_sp_gst $2
153     fi
154     exit
155 fi
156
157 # sast gsts
158 if [ $1 = "-sast"]; then
159     if [ "$#" -eq 1 ]; then
160     echo "Making gold standard for all sast tests:"
161         for test in $sast_tests; do
162             run_sast_gst $test
163         done
164         for test in $sast_fail; do
165             run_sast_gst $test
166         done
167     fi
168     exit
169 else
170     if [ "$#" -eq 2 ]; then
171         echo "Making gold standard for one sast test $2:"
172         run_sast_gst $2
173     fi
174     exit
175 fi
176
177
178 # create e2e gsts
179 if [ $1 = "-e2e"]; then
180     if [ "$#" -eq 1 ]; then
181     echo "Making gold standard for all e2e tests:"
182         for test in $tests; do
183             run_one_gst $test
184         done
185         for test in $fail_tests; do
186             run_one_gst $test
187         done
188     fi
189     exit
190 else
191     if [ "$#" -eq 2 ]; then
192         echo "Making gold standard for one e2e test $2:"
193         run_one_gst $2
194     fi
195     exit
196 fi
197
198
```

```
199 make clean
200 echo "bye"
```

compile.sh

```
#!/bin/bash
# compiles the BlueShell compiler and compiles a BlueShell
    file with the
# BlueShell compiler

Usage() {
    echo "Usage: ./compile.sh [file].bs \n"
    exit
}

if [ $# -lt 1 ]
    then
    Usage
fi


full_filename=$1
# strips extension from full filename
extension="${full_filename##*.}"
# enforces the .bs extension
if [ "$extension" != "bs" ];
    then
    Usage
fi

# strips filename from path and extension
filename=$(basename -- "$full_filename")
filename=${filename%.*}

make # compiles compiler
./toplevel.native < $full_filename > "$filename.llvm"
llc "-relocation-model=pic" $filename.llvm
cc -c exec.c # links with our c file
cc $filename.llvm.s exec.o -o $filename.exe
```

8.3 Demonstration Programs

hello-world.bs

```
/* Create an exeuctable to run echo with argument "Hello
   world", then run it */

./<"echo" withargs ["Hello World"]>;
```

run-programs.bs

```
/* run-programs.bs
   compiles and runs some BlueShell test programs */

// Executes an executable type
string execute_exec(exec e) {
    return ./e;
}

// Executes an executable type
exec create_execs(string s) {
    return <s>;
}

//compiles a bs executable
exec create_compile_execs(string s) {
    return <"./compile.sh" withargs [s]>;
}

//maps a list of executables over a function and outputs a
   list of strings
list of exec map_string_to_exec(function (string -> exec)
   func, list of string strings) {
    int l = len strings;

    list of exec new_execs = [];
    for (int i = l - 1; i >= 0; i = i - 1) {
        new_execs = func(strings[i]) :: new_execs;
    }
    return new_execs;
}

// maps a list of strings over a function and outputs a list
   of executables
list of string map_exec_to_string(function (exec -> string)
   func, list of exec execs) {
    int l = len execs;
    list of string new_strings = [];
    for (int i = l - 1; i >= 0; i = i - 1) {
        new_strings = func(execs[i]) :: new_strings;
    }
    return new_strings;
}

//creates exeutables that compiles the following tests
list of string execs_to_compile =
   ["tests/test-echo1.bs","tests/test-pipe1.bs",
   "tests/test-concatseq1.bs"];
list of exec compile_execs =
   map_string_to_exec(create_compile_execs, execs_to_compile);

```

```
44 // compiles the scripts
45 map_exec_to_string(execute_exec, compile_execs);
46
47 // executes the compiled bs programs
48 list of string run_execs = ["./test-echo1.exe",
       "./test-pipe1.exe", "./test-concatseq1.exe"];
49 list of exec final_execs = map_string_to_exec(create_execs,
       run_execs);
50 map_exec_to_string(execute_exec, final_execs);
```

counts-bs.bs

```
/* creates some executables and combines them with concat and
   pipe operators */

exec wc = <"wc" withargs ["-l"]>;
exec cat_README = <"cat" withargs ["README"]>;
exec cat_Makefile = <"cat" withargs ["compile.sh"]>;
exec grep_BS = <"grep" withargs ["-a","BlueShell"]>;

exec final_exec = (cat_README + cat_Makefile) | grep_BS| wc;
./final_exec;
```

misc-ops.bs

```
// test combinations of executable operators
// expected results are printed when this program is run

exec e1 = <"cat" withargs ["sample-files/test_file.txt"]>;
exec e2 = <"cat" withargs ["sample-files/test_file2.txt"]>;

exec e3 = <"grep" withargs ["-a", "zip"]>;
exec e4 = <"grep" withargs ["-a", "BlueShell"]>;

exec e5 = (e1 | e3) + (e2 | e4);
./<"echo" withargs ["TEST 1 OUTPUT: ZIPS FROM FIRST FILE AND
    BLUESHELLS FROM SECOND"]>;
string s = ./e5;



./<"echo" withargs ["_______________"]>;
exec e5 = (e1 | e3) * (e2 | e4);
./<"echo" withargs ["\n\nTEST 2 OUTPUT: BLUESHELLS FROM
    SECOND"]>;
string s = ./e5;




./<"echo" withargs ["_______________"]>;
exec e5 = (e1 + e2) | e3;
./<"echo" withargs ["\n\nTEST 3 OUTPUT: ZIPS FROM BOTH"]>;
string s = ./e5;




./<"echo" withargs ["_______________"]>;
exec e5 = (e1 * e2) | e4;
./<"echo" withargs ["\n\nTEST 4 OUTPUT: BLUESHELLS FROM
    SECOND"]>;
string s = ./e5;




./<"echo" withargs ["_______________"]>;
exec e6 = <"echo" withargs ["hello world"]>;
exec e5 = (e2 * e1) | (e3 + (e6 + e6)) | <"grep" withargs
    ["-a", "hello"]> ;
./<"echo" withargs ["\n\nTEST 5 OUTPUT: ZIPS FROM FIRST PLUS
    HELLO WORLDS"]>;
string s = ./e5;

```

```
46
47
48
49 exec e5 = (e2 * e1) | (e3 + (e6 + e6) | <"grep" withargs
       ["-a", "hello"]>);
50 ./<"echo" withargs ["\n\nTEST 6 OUTPUT: GETS OUTPUT OF FIRST
       FILE, GREPS FOR ZIP WHICH GETS CONCATENATED WITH HELLO
       WORLD AND WE GREP FOR HELLO WORLD"]>;
51 string s = ./e5;
```

8.4 Git Log

```
commit c12f834f82db1797c62a3943ca12b59ca82fbc44
Author: mjsidhom <maryjoyis9@gmail.com>
Date:   Fri May 5 19:15:28 2023 -0400

    exec args cannot be lists, execs, or funtions

commit c5d8abf878c8aea301dc8ff634046a7b22a6fc1c
Author: mjsidhom <maryjoyis9@gmail.com>
Date:   Fri May 5 18:26:11 2023 -0400

    added precedence and deleted bool equality

commit c01883527245e728ebe463f2126856a743e0be0c
Author: Alan Luc <alanluc2001@gmail.com>
Date:   Fri May 5 16:12:32 2023 -0400

    final deliverable

commit 62a7a8396fac2329299fe2edbbb3cfb4cdfa4487
Merge: eb1d650 1242834
Author: klin303 <kenny@Kennys-MacBook-Pro.local>
Date:   Fri May 5 15:57:24 2023 -0400

    Merge branch 'main' of https://github.com/klin303/BlueShell
    into main

commit eb1d650a908393c40dc2a9d3675f11eda28a1d31
Author: klin303 <kenny@Kennys-MacBook-Pro.local>
Date:   Fri May 5 15:55:10 2023 -0400

    final fixes, removed exitcode from all instances, commented
    semant

commit 12428343f74a2abe9c6a13c65d38290625c15f9e
Author: Tina <tma03@cs.tufts.edu>
Date:   Fri May 5 15:32:50 2023 -0400

    make gsts script can now make sp, sast, or e2e gsts

commit 9ce9f889da5930535c790b5a74d18a284eb452ac
Author: Tina <tma03@cs.tufts.edu>
Date:   Fri May 5 13:47:17 2023 -0400

    updated test files with language changes; updated test all to
    also run sast and ast tests as well as e2e

commit d679249d6db2e9c1dfef8544117e553f5166d266
Author: klin303 <kenny@Kennys-MacBook-Pro.local>
Date:   Fri May 5 11:33:11 2023 -0400
```

```
    edited some tests and scripts - gsts missing?

commit 82b9b683dff98f18a4571cfd43bfdbd54288e546
Author: klin303 <kenny@Kennys-MacBook-Pro.local>
Date:   Fri May 5 10:56:48 2023 -0400

    added .bs enforcement to compile.sh

commit 5aa04b414113e7816e596e493ec9b027fad15814
Author: klin303 <kenny@Kennys-MacBook-Pro.local>
Date:   Thu May 4 13:39:59 2023 -0400

    I made hello world hehe

commit 7aff5a22cdd1ac3ba7701d9d11c5e42d19d537a6
Author: mjsidhom <maryjoyis9@gmail.com>
Date:   Wed May 3 21:43:30 2023 -0400

    added demos

commit c43389bc58fd3444a632d1832f6bf8d0ae3643f4
Author: Alan Luc <alanluc2001@gmail.com>
Date:   Wed May 3 17:07:15 2023 -0400

    working on demo

commit c7c79b190c8c7266b9fcccfa0010202cc515a788
Author: Alan Luc <alanluc2001@gmail.com>
Date:   Wed May 3 16:07:05 2023 -0400

    compiler itself should be done

commit bf7ac1c1703f7cfc45bdc5178686074869818622
Author: Alan Luc <alanluc2001@gmail.com>
Date:   Tue May 2 18:53:19 2023 -0400

    pipe works for one thing for sure

commit 71b070b0198efdb2bcdb00eedbb7863d06b3f9d6
Author: Alan Luc <alanluc2001@gmail.com>
Date:   Tue May 2 16:30:21 2023 -0400

    concat and seq work

commit d963ff4888c9025f86efd4c87b0763e829f36e2a
Author: Alan Luc <alanluc2001@gmail.com>
Date:   Tue May 2 15:30:29 2023 -0400

    exec ops are now handled by a C function

commit 4d036a7af7be5d788b81ce6016870148fa1b41c9
Author: Alan Luc <alanluc2001@gmail.com>
```

```
Date:   Mon May 1 12:26:04 2023 -0400

    comments for codegen

commit 4fb6ded4f9bab7b1ef9f1ddef66c2e6de4263a0d
Author: Alan Luc <alanluc2001@gmail.com>
Date:   Mon May 1 10:44:27 2023 -0400

    path works

commit 44551bdfae6d0de6cb3a24593958ccf6172affd7
Author: Alan Luc <alanluc2001@gmail.com>
Date:   Sun Apr 30 20:01:56 2023 -0400

    index fully working

commit 536b3bb69ece6c51f7e1f49587499b417f038a9d
Author: Alan Luc <alanluc2001@gmail.com>
Date:   Sun Apr 30 19:51:00 2023 -0400

    index works in both directions for ints, but need to create a
    switch to cast to diff types

commit e9dde0145ddb9edaed8138d4c183b30e28541ca6
Author: Alan Luc <alanluc2001@gmail.com>
Date:   Fri Apr 28 16:58:30 2023 -0400

    exec return string doesn't quite work yet but it doesn't
    break anything, i have an idea for index but need mj to do it
    with me

commit 4482c71df51416b037296c7515cd666c49306a9c
Author: Alan Luc <alanluc2001@gmail.com>
Date:   Fri Apr 28 13:04:56 2023 -0400

    fixed the bug in exec, you can run it in vscode again

commit b55333aab7f311a489e9e2f7aa2560fa8df281dc
Author: Alan Luc <alanluc2001@gmail.com>
Date:   Fri Apr 28 00:46:04 2023 -0400

    got list indexing working

commit 7bbb2b0eaf6f5909474379d5ac5bcea0a2856331
Author: mjsidhom <maryjoyis9@gmail.com>
Date:   Thu Apr 27 17:10:29 2023 -0400

    Changes semant to allow functions to be called in other
    functions. Codegen doesnt work

commit f51f062e1e3d387f3877bcb57f48f97e5eb0bd60
Author: mjsidhom <maryjoyis9@gmail.com>
```

```
Date:   Thu Apr 27 16:16:09 2023 -0400

    Got higher order functions to work

commit 632284fd2433bd60e091d1ee7cdf57ca221b8271
Author: Alan Luc <alanluc2001@gmail.com>
Date:   Wed Apr 26 13:45:20 2023 -0400

    finished statements in codegen

commit 6d047e922bdb8f45d22b0dcd86a54c51f663eaa4
Author: klin303 <kenny@Kennys-MacBook-Pro.local>
Date:   Wed Apr 26 12:09:14 2023 -0400

    documented tests, gave up on hofs, got blocks to work and
    added fdecl to statement call... need to finish statements next

commit f4a1e6a1e66cd2725038555ad05153bb830fdd05
Author: Alan Luc <alanluc2001@gmail.com>
Date:   Mon Apr 24 17:25:55 2023 -0400

    function pointers work within statements, but not in functions

commit 80c2c3185c4a7637b6638ec90de0bfbe9c17124b
Author: mjsidhom <maryjoyis9@gmail.com>
Date:   Thu Apr 20 14:15:47 2023 -0400

    Changed bools back to i1_t but added bitwise and to exec.c.
    Got functions with returns and params to work, finished all
    bool, int, float operators with tests

commit a555111e88f82e993770cc17eb62874ff325a346
Author: mjsidhom <maryjoyis9@gmail.com>
Date:   Thu Apr 20 12:58:23 2023 -0400

    Void functions with parameters work. Arith Binops work

commit 3092b1b91b47cc33c3d8e7dd3004c0e3844c79ac
Author: mjsidhom <maryjoyis9@gmail.com>
Date:   Thu Apr 20 12:02:32 2023 -0400

    Got floats to print

commit 16603608785e8343835d1efdc19d07c15c8b7c0b
Author: mjsidhom <maryjoyis9@gmail.com>
Date:   Wed Apr 19 19:01:35 2023 -0400

    Made bools i32_t so that way things don't get messed up when
    casting

commit 3a1d9135c8756bd981b2701be06af461caded95c
Author: mjsidhom <maryjoyis9@gmail.com>
```

```
Date:   Wed Apr 19 18:56:43 2023 -0400

    Worked on casting primitive types to strings for execs, added
    a new struct member to list struct that holds the type, add
    derefencing binop values and storing the new num in a variable
    (this needs to be done for all types and all binops).
    Currently, floats and bools don't work when printing out.

commit 7b3572320cbcab3f77e40434eea5798996615239
Author: Alan Luc <alanluc2001@gmail.com>
Date:   Wed Apr 19 17:40:41 2023 -0400

    kinda stuck trying to get autocasting to work (this doesnt
    compile)

commit c763f5f19f5079adf3a7dca8259240512dc59dcb
Author: mjsidhom <maryjoyis9@gmail.com>
Date:   Wed Apr 19 16:39:15 2023 -0400

    updated makefile zip and created gsts

commit a87054edbc847fad6d9929be561a4336fefaee44
Author: Alan Luc <alanluc2001@gmail.com>
Date:   Wed Apr 19 16:17:00 2023 -0400

    fixed the oopsie

commit 6796a63a939a8104db9b70cf9e311dde5c6d1b4b
Author: mjsidhom <maryjoyis9@gmail.com>
Date:   Wed Apr 19 16:13:52 2023 -0400

    Created gsts for tests, allowed calling of functions

commit 83516f0df86ca1270e884734cac750bcdfc33a72
Author: Alan Luc <alanluc2001@gmail.com>
Date:   Tue Apr 18 17:00:53 2023 -0400

    funcs with no params + no return work (i think), implement
    call next to test

commit 089cdd5bd2bfeb9cee9883e718df37f4ff72f854
Author: klin303 <kenny@Kennys-MacBook-Pro.local>
Date:   Tue Apr 18 16:44:12 2023 -0400

    organized tests for submission and added new ones, edited
    README with descriptions of end-to=end tests, edited Makefile
    to unzip properly.

commit 1112dcdc9ba77d09a719571027574399b499fff6
Author: Alan Luc <alanluc2001@gmail.com>
Date:   Tue Apr 18 12:18:22 2023 -0400
```

```
    assign now works thoroughly with all types, work on casting
    next

commit e217b603291d13fd29f363f739001b98d3d5da97
Author: Alan Luc <alanluc2001@gmail.com>
Date:   Mon Apr 17 15:54:20 2023 -0400

    fixed assign

commit 4e1b5c1a0b52c932d98e0ae08cbdd485673a9f40
Author: Alan Luc <alanluc2001@gmail.com>
Date:   Mon Apr 17 15:30:25 2023 -0400

    got execs with args workinggit add . and also some function
    semant stuff TODO: assign

commit 2e04aed0ec2929fa76d4aa8580ab1321d8f1acac
Author: Alan Luc <alanluc2001@gmail.com>
Date:   Sun Apr 16 22:28:18 2023 -0400

    finished all cases of stmt in semant besides return, next
    step is to add stuff to check_func for body

commit 321720ad064289d50fa9a3742a61ca34cd3b3d79
Author: Alan Luc <alanluc2001@gmail.com>
Date:   Fri Apr 14 11:16:49 2023 -0400

    added indexing to semant, added arithmetic ops to codegen

commit aa07b37ec47dd2542d0f8c6e42bd914f44c74ecb
Author: Alan Luc <alanluc2001@gmail.com>
Date:   Thu Apr 13 15:48:38 2023 -0400

    added one line to codegen

commit 7bc80bf002da1c1ec50dcb41b10a933cf8582dbd
Author: Alan Luc <alanluc2001@gmail.com>
Date:   Thu Apr 13 13:58:32 2023 -0400

    lists compile! basically did a bunch of pointer casting

commit ecc7057b0e937283f7d60f8b2bb1541a9410c454
Author: Alan Luc <alanluc2001@gmail.com>
Date:   Wed Apr 12 16:04:06 2023 -0400

    semant is more thorough now, execassign cases are more fine
    grained, codegen is stuck on lists

commit f8bda1c18fbfcd122b4a4b1e09865797279bde65
Author: mjsidhom <maryjoyis9@gmail.com>
Date:   Wed Apr 12 10:25:20 2023 -0400
```

```
    Created semantically checked functions, added functions to
    symbol table, implemented part of check_function, changed
    call, created tests

commit 60ff1306d00636fb6787a218edaaae3537ebaf54
Author: Tina <tma03@cs.tufts.edu>
Date:   Tue Apr 11 17:49:12 2023 -0400

    we can now run sast tests i'm pretty sure but we can't make
    the gold standards wah

commit b6d5eaf5a4fc76cf7f7919f99134e12db1000f34
Author: Alan Luc <alanluc2001@gmail.com>
Date:   Tue Apr 11 17:22:18 2023 -0400

    more hof stuff, this doesn't really work

commit ae286bb0849f9074e248701aee0dd3e813f94c68
Author: mjsidhom <maryjoyis9@gmail.com>
Date:   Mon Apr 10 14:36:07 2023 -0400

    Added emptylist case to semant, added first class function
    syntax to scanner and parser, added first class function
    sematic checks in semant. It compiles and scanner/parser have
    been tested. However, semant changes have not been tested.

commit 2b3ba787521e6d50332924d8ca94f6e803ae7128
Author: mjsidhom <maryjoyis9@gmail.com>
Date:   Mon Apr 10 12:57:20 2023 -0400

    Added more tests. EmptyList case in progress.

commit 62231bb438cc80a222e0318b158ce6d9a6f9e776
Author: mjsidhom <maryjoyis9@gmail.com>
Date:   Sun Apr 9 15:59:06 2023 -0400

    Added semantic checks for indexing and function calls. When
    adding semantic checking for function calls I copied and
    pasted microC code along with the following functions:
    find_func that finds the function in a mapping of function
    declarations, function_decls that  creates the function
    declarations mapping, and add_func that adds a function
    declaration to the function declarations mapping. Currently
    this compiles but is not tested.

commit aa7ad064b20a23abd01d12bf34a1835218e09c3a
Author: Alan Luc <alanluc2001@gmail.com>
Date:   Sun Apr 9 15:35:24 2023 -0400

    all operators in semant

commit a40a05197ecead81e3f4d5da1aeebefa6bd46821
```

```
Author: Tina <tma03@cs.tufts.edu>
Date:   Sun Apr 9 15:29:49 2023 -0400

    - testing now happens only within testing directory
    - make_gsts.sh deprecated

commit 34aa1a43026d6ea30760d709c88254363a0b8e52
Author: mjsidhom <maryjoyis9@gmail.com>
Date:   Wed Mar 29 22:01:00 2023 -0400

    updated readme

commit 750d6d4fa7044231ca8f6a807ffcc475481217dd
Author: Tina <tma03@cs.tufts.edu>
Date:   Wed Mar 29 20:20:01 2023 -0400

    uhhh string parsing messed up in makefile we got it now

commit 9dd1daa35c733314f3fbcd49a83457d51437bc75
Author: klin303 <kenny@Kennys-MacBook-Pro.local>
Date:   Wed Mar 29 17:54:47 2023 -0400

    working on submission scripts

commit b0f9d0827e089bdd925efb85d4d0d54ab4e4e045
Author: klin303 <kenny@Kennys-MacBook-Pro.local>
Date:   Wed Mar 29 13:43:46 2023 -0400

    updated make/compile scripts, need to finish testall

commit 2de19018004d27a897013f8d5761945e069906c6
Author: Alan Luc <alanluc2001@gmail.com>
Date:   Wed Mar 29 12:30:42 2023 -0400

    closer to submitting

commit ade4f02e97ca4189f45bc79ace4179c9fb3f7619
Author: mjsidhom <maryjoyis9@gmail.com>
Date:   Tue Mar 28 21:38:51 2023 -0400

    execvp works ... maybe

commit 90b637f3514a382d632bc50d992ff2b7d0b873e2
Author: mjsidhom <maryjoyis9@gmail.com>
Date:   Tue Mar 28 21:21:46 2023 -0400

    exec almost woks

commit f63d22f6465cf278659bd79940ad530f470995b3
Author: Alan Luc <alanluc2001@gmail.com>
Date:   Tue Mar 28 13:51:32 2023 -0400
```

```
    made codegen work for strings

commit e67b674de54a6e23e70da6807d8871b8324feaf2
Author: mjsidhom <maryjoyis9@gmail.com>
Date:   Tue Mar 28 00:01:07 2023 -0400

    We tried.

commit fa74306ed51485482ba4e05aef67a69b95544850
Author: Alan Luc <alanluc2001@gmail.com>
Date:   Mon Mar 27 16:38:52 2023 -0400

    simplified run test, figured out strings, lists, and execs
    for codegen

commit 173ee9e1d63eafaa11fc85db7f44458ade7639f3
Author: Alan Luc <alanluc2001@gmail.com>
Date:   Tue Mar 21 21:26:01 2023 -0400

    codegen compiles

commit 34a15f9cfba10da0bfd6794c7caeb7dcd897a950
Author: Alan Luc <alanluc2001@gmail.com>
Date:   Thu Mar 16 16:17:24 2023 -0400

    started codegen, fixed semant run

commit 2d873c10c55a4de7553efbeb66f28a786e118a88
Author: Alan Luc <alanluc2001@gmail.com>
Date:   Wed Mar 15 21:13:58 2023 -0400

    semant done for hello world

commit f9355a5b89e127680d1c90366b4249bf3540e3b1
Author: Alan Luc <alanluc2001@gmail.com>
Date:   Wed Mar 15 15:24:12 2023 -0400

    assignment and bind type checking

commit 81d152d917afd3f0e3e44d5e554705575b8cc517
Author: Alan Luc <alanluc2001@gmail.com>
Date:   Tue Mar 14 17:59:18 2023 -0400

    added scope stuff to semant

commit 031c20def1f84af98c040556d3924dc10a21998a
Author: Alan Luc <alanluc2001@gmail.com>
Date:   Mon Mar 13 19:36:34 2023 -0400

    basically starter code for semant

commit 5810dd2bb48e075ed8d76293003b7987767cb2a2
```

```
Author: Alan Luc <alanluc2001@gmail.com>
Date:   Mon Mar 13 11:07:52 2023 -0400

    fixed sast

commit 17d25a55e801acdcbd2322b353f5d62c392a9363
Author: Alan Luc <alanluc2001@gmail.com>
Date:   Mon Mar 13 11:02:45 2023 -0400

    sast done, someone check

commit f68da9254f652f7a4750e3ac2e6158b69667464f
Author: Tina <tma03@cs.tufts.edu>
Date:   Mon Mar 13 10:49:14 2023 -0400

    updated testing script to provide more information on why a
    test failed + if a test passed

commit 5a2e640ee8f35daaef9dae35737ebb3b5bb8dbd2
Author: Alan Luc <alanluc2001@gmail.com>
Date:   Wed Mar 8 13:48:34 2023 -0500

    update executable syntax, lists are monomorphic, added list
    shrink, updated tests accordingly

commit ea6924dce03d34d185552ff9d1978bc61f628143
Author: Alan Luc <alanluc2001@gmail.com>
Date:   Wed Mar 8 13:31:52 2023 -0500

    update executable syntax, lists are monomorphic, added list
    shrink, updated tests accordingly

commit 40a50f729c7f912207d5627b9da7275993023dfd
Author: klin303 <kenny@Kennys-MacBook-Pro.local>
Date:   Tue Feb 28 20:58:01 2023 -0500

    todo after LRM

commit d096aa6fe5eaa2c229924bc10baf794c0037bd6b
Author: klin303 <kenny@Kennys-MacBook-Pro.local>
Date:   Thu Feb 23 18:17:19 2023 -0500

    fixed makefile rule for zip

commit 296c86cd196e9b72dfcbe6555dec81b5eb9bd0f5
Author: klin303 <kenny@Kennys-MacBook-Pro.local>
Date:   Thu Feb 23 18:10:05 2023 -0500

    fixed the tests to updated parser and ast print, added new
    rules to the makefile for cleaning tests, added info to readme

commit 8339f5711b793ac966a449a50e49ff1adcdb2cb0
```

```
Author: Tina <tma03@cs.tufts.edu>
Date:   Thu Feb 23 17:30:29 2023 -0500

    fixed two lines on the script

commit 6f2a6430eb4d7fe251cd2637589b3a6819f30b93
Author: Tina <tma03@cs.tufts.edu>
Date:   Thu Feb 23 17:28:10 2023 -0500

    modified readme

commit 5bee8b548c68716bdce0854a41dfa2286573d7e4
Author: Tina <tma03@cs.tufts.edu>
Date:   Thu Feb 23 17:23:56 2023 -0500

    added some tests and now gst can make one specific test

commit 16164baa9511db087bd4c40dcaa9ad48792ab177
Author: Tina <tma03@cs.tufts.edu>
Date:   Wed Feb 22 23:01:35 2023 -0500

    Revert "Revert "moar tests""

    This reverts commit 500d350643009fd78096195c0056593021f036d8.

commit 233576ef7cf885f120f493cac7a0549d6a524f97
Author: Tina <tma03@cs.tufts.edu>
Date:   Wed Feb 22 23:01:08 2023 -0500

    what

commit 3038eed87332add62100096aee6251b369264e91
Author: Tina <tma03@cs.tufts.edu>
Date:   Wed Feb 22 22:56:38 2023 -0500

    moar tests

commit 51b2f57546a16911d014e051dbdaf9e1d2901657
Author: Alan Luc <alanluc2001@gmail.com>
Date:   Thu Feb 23 13:50:31 2023 -0500

    FINISH README NEXT, style done, testing in progress

commit 257202c7fab09e0655eb2ed65a873c4355fe4860
Author: mjsidhom <maryjoyis9@gmail.com>
Date:   Thu Feb 23 12:37:57 2023 -0500

    variable decl/assignment test

commit 1fd2475007db5695f621247ea4e17d3771544b49
Author: mjsidhom <maryjoyis9@gmail.com>
Date:   Thu Feb 23 12:28:38 2023 -0500
```

```
    Allowed variable declaration and assignment to happen on the
    same line

commit 5d8ac31bb761946330fb90d65c4f09e84e068951
Author: Alan Luc <alanluc2001@gmail.com>
Date:   Thu Feb 23 11:48:13 2023 -0500

    Progress

commit 4d8e281a728e715a7b6b5d94573269edc03c1589
Author: Alan Luc <alanluc2001@gmail.com>
Date:   Thu Feb 23 10:16:28 2023 -0500

    Delete toplevel.native

commit ab3a9cfab9099687359afe9acc3bc81f6814de4f
Author: Tina <61843781+t-a-ma@users.noreply.github.com>
Date:   Wed Feb 22 23:04:10 2023 -0500

    we have somethign that tests things (#2)

    * -wrote the testing script
    -wrote a script that makes gsts
    -added fail and pass tests

    * moar tests

    * Revert "moar tests"

    This reverts commit c87bcc859752c310f4d3b0790848092171f0f5f8.

    * what

    * Revert "Revert "moar tests""

    This reverts commit 500d350643009fd78096195c0056593021f036d8.

    ---------

    Co-authored-by: Tina <tma03@cs.tufts.edu>

commit 594243a9164609ccc8124eed39855d11e7643582
Author: klin303 <kenny@Kennys-MacBook-Pro.local>
Date:   Wed Feb 22 23:01:08 2023 -0500

    added expr = expr and some better printing

commit a250133aa7af0a85f12e2e41d9ef8620aa7bd53e
Author: klin303 <kenny@Kennys-MacBook-Pro.local>
Date:   Wed Feb 22 21:42:20 2023 -0500
```

```
    working version for now

commit c04847db9ef18aa94cc26ac00b958f677960c0f3
Author: klin303 <kenny@Kennys-MacBook-Pro.local>
Date:   Wed Feb 22 21:09:05 2023 -0500

    added emptylist printing and reversed printing of program so
   its in order of declaration

commit 1c939713a9a1f96eb682eb704691a13296a39e44
Author: Alan Luc <alanluc2001@gmail.com>
Date:   Wed Feb 22 19:32:57 2023 -0500

    statements are legal now

commit e30c5a8c096da6541d49410bd50bd8ff902947b6
Merge: 1613a92 9b31e90
Author: mjsidhom <maryjoyis9@gmail.com>
Date:   Wed Feb 22 19:11:19 2023 -0500

    commit merge

commit 1613a92f67f8274eaaa7a8d6e710ef47317a9b30
Author: mjsidhom <maryjoyis9@gmail.com>
Date:   Wed Feb 22 19:10:31 2023 -0500

    redid tests to be right

commit 9b31e904d91b31637d2cc22191d331d079b6c1d1
Author: Tina <61843781+t-a-ma@users.noreply.github.com>
Date:   Wed Feb 22 18:18:29 2023 -0500

    Update Makefile

    sorry i forgot the slahes

commit f0ab6e091b38b379ab4434c9dcf3334733d5aa95
Merge: 69e409d 07e98b9
Author: mjsidhom <maryjoyis9@gmail.com>
Date:   Wed Feb 22 18:09:27 2023 -0500

    Merge branch 'main' of https://github.com/klin303/BlueShell

commit 69e409d4d711f95707a484348bee99731164d9ac
Author: mjsidhom <maryjoyis9@gmail.com>
Date:   Wed Feb 22 18:05:25 2023 -0500

    pretty print execs and function args

commit c514d71801ec1a187de43717ebf11179d2e87277
Author: mjsidhom <maryjoyis9@gmail.com>
Date:   Wed Feb 22 17:55:26 2023 -0500
```

```
    Revert "Merge branch 'test-cases' of
    https://github.com/klin303/BlueShell"

    This reverts commit f2fe4c07412275063bcc3c110d364ff93ac5e4d3,
    reversing
    changes made to 236b8e8207600289a5308424cd898f74dcb9980a.

commit da412881875d754951372e5560ce448de5c818ca
Author: mjsidhom <maryjoyis9@gmail.com>
Date:   Wed Feb 22 17:52:36 2023 -0500

    Revert "more changes"

    This reverts commit 8a1910ec091290461e94fa839c2911cd70da757e.

commit 8a1910ec091290461e94fa839c2911cd70da757e
Author: mjsidhom <maryjoyis9@gmail.com>
Date:   Wed Feb 22 17:47:36 2023 -0500

    more changes

commit f2fe4c07412275063bcc3c110d364ff93ac5e4d3
Merge: 236b8e8 bb311e9
Author: mjsidhom <maryjoyis9@gmail.com>
Date:   Wed Feb 22 17:38:45 2023 -0500

    Merge branch 'test-cases' of
    https://github.com/klin303/BlueShell

commit 07e98b98b8d3bd27b194eb289c5e9ba545ab14fe
Author: Tina <61843781+t-a-ma@users.noreply.github.com>
Date:   Wed Feb 22 17:20:40 2023 -0500

    Test cases + testing script (#1)

    * added succ and fail test cases; added tests in the Makefile
    and code to get tests in testing scripting

    * updated to tests to actually test scanner and parser

    ---------

    Co-authored-by: Tina <tma03@cs.tufts.edu>

commit 236b8e8207600289a5308424cd898f74dcb9980a
Author: Alan Luc <alanluc2001@gmail.com>
Date:   Wed Feb 22 16:08:20 2023 -0500

    TAKE A LOOK AT EMPTYSTMTLIST IN PARSER IDK WHY IT WORKED

commit f61bbf6132cc4b3a3bcadc2af082477bb43793fb
```

```
Author: Alan Luc <alanluc2001@gmail.com>
Date:   Wed Feb 22 15:47:51 2023 -0500

    eargs now wrapped in braces

commit 1aec94b4aecc5e2fb58e103f5f8d8325bedeb9f5
Author: Alan Luc <alanluc2001@gmail.com>
Date:   Wed Feb 22 13:55:43 2023 -0500

    fixed a bunch of parser errors, made path ID | STRING, still
    a bit confused about expr vs exec

commit bb311e9c9060f44af3963a0c0eefe83f38f906ce
Author: Tina <tma03@cs.tufts.edu>
Date:   Wed Feb 22 10:31:47 2023 -0500

    change tests

commit 858f740b1e3b8de10ed060d0c1b1344e643ac907
Author: Tina <tma03@cs.tufts.edu>
Date:   Wed Feb 22 10:11:52 2023 -0500

    added succ and fail test cases; added tests in the Makefile
    and code to get tests in testing scripting

commit b7e054e94116d1fee34719579d524e80470f4923
Author: Alan Luc <alanluc2001@gmail.com>
Date:   Tue Feb 21 22:33:47 2023 -0500

    man idk what i did

commit 9967e32879481a00c92fcab05efee2e624d5eebd
Author: mjsidhom <maryjoyis9@gmail.com>
Date:   Tue Feb 21 19:59:35 2023 -0500

    Fixed compiler bugs, changed empty list to a NoExpr -- not
    an empty tuple, completed pattern matched in ast, changed CHAR
    to CHR and STRING to STR

commit e75f9992509926b9655ec22ccebf417131d5aaa9
Author: Alan Luc <alanluc2001@gmail.com>
Date:   Tue Feb 21 16:40:08 2023 -0500

    added strings and chars to scanner, made it closer to
    compiling

commit c683c6158d2c99fdb48f391fea76324fb55c0888
Author: Alan Luc <alanluc2001@gmail.com>
Date:   Tue Feb 21 11:46:56 2023 -0500

    fixed gitignore lol
```

```
commit e8fbe9731249ed5978395cb56d3365f417503742
Author: Alan Luc <alanluc2001@gmail.com>
Date:   Tue Feb 21 11:45:54 2023 -0500

    made makefile, fixed a lot of bugs

commit c1b792845e9c58e69c199f5a161b34a8c001eb08
Author: Alan Luc <alanluc2001@gmail.com>
Date:   Tue Feb 21 10:36:52 2023 -0500

    allowed binds within function body, fixed operators in ast,
    allowed top to bottom execution

commit ebd15424a8d0d4d6a9d3417f02849121751ac964
Author: klin303 <kenny@Kennys-MacBook-Pro.local>
Date:   Mon Feb 20 17:35:58 2023 -0500

    added exec args and changed lists a bit

commit 919288395cd1e6a8a62c526293732c0c08f8cc43
Author: klin303 <kenny@Kennys-MacBook-Pro.local>
Date:   Mon Feb 20 17:11:28 2023 -0500

    added path edit to exec and top level

commit 601e33656f9b6311d0386bf2f51102c421330c61
Author: klin303 <kenny@Kennys-MacBook-Pro.local>
Date:   Mon Feb 20 16:13:59 2023 -0500

    additions to printing, removed else if, and added preUnop

commit d268922f67c3db36d4ff43813a29e1730ec73bbd
Author: Alan Luc <alanluc2001@gmail.com>
Date:   Mon Feb 20 14:33:16 2023 -0500

    Added more precedence stuff

commit bdda242015065b5d57f89eb5617be80e66cc8471
Author: Alan Luc <alanluc2001@gmail.com>
Date:   Mon Feb 20 11:59:09 2023 -0500

    Added list operators precedencd to parser, executable
    operators to ast

commit cf968d68261a94a5e4de1beda3810c1d5ad94f5b
Author: Alan Luc <alanluc2001@gmail.com>
Date:   Sun Feb 19 16:09:13 2023 -0500

    Completed scanner, completed list operations in parser, added
    list operations to ast, added starter code to toplevel (not
    sure if it's right)
```

```
commit 8741d7932cac1956921847ad5110a711f035d161
Author: Alan Luc <alanluc2001@gmail.com>
Date:   Fri Feb 17 16:43:17 2023 -0500

    Added functions to parser, added new types and new uops to ast

commit 52c0e7ba2d2971e538f63d58fa781bc2c86b48c2
Author: Tina <tma03@cs.tufts.edu>
Date:   Fri Feb 17 16:12:32 2023 -0500

    added reference directory for any other reference text. Added
    mert's monad example. Thanks mert

commit 8c606103b041062ba72487d6bf4eba2eb8ffef85
Author: Alan Luc <alanluc2001@gmail.com>
Date:   Thu Feb 16 16:58:14 2023 -0500

    added list parsing

commit e5f313bf471c1b9db4968a1d743b2150dc2f3c0b
Author: mjsidhom <maryjoyis9@gmail.com>
Date:   Thu Feb 16 14:48:42 2023 -0500

    Added code to parse executable types, added basic parsing
    functionality, and created ast.ml and toplevel.ml files

commit 28feec307a7b957455918e9fb05d41dca359f044
Author: Alan Luc <alanluc2001@gmail.com>
Date:   Mon Feb 13 22:41:34 2023 -0500

    added list to scanner

commit 1be0fd8fdbe9ada779fbf2baa78650e98cce31d2
Author: mjsidhom <maryjoyis9@gmail.com>
Date:   Sun Feb 12 16:12:06 2023 -0500

    added if

commit 2d9a6441bf39f4545c48e4c471939d44ae255312
Author: klin303 <kenny@Kennys-MacBook-Pro.local>
Date:   Sun Feb 12 15:59:06 2023 -0500

    s&p progress

commit be870c17b570015c4e6cbf18b6d9f7ae0a61f21b
Author: klin303 <kenny@Kennys-MacBook-Pro.local>
Date:   Sun Feb 12 15:12:18 2023 -0500

    created scanner and parser
```