Funk Programming Language Final Report

Naser AlDuaij, Senyao Du, Noura Farra, Yuan Kang, Andrea Lottarini
{nya2102, sd2693, naf2116, yjk2106, al3125} @columbia.edu

19 December, 2012

1 Introduction

The Funk language is a multi-paradigm, general purpose programming language. It is designed to let programmers who are more familiar with imperative programming languages to reap the benefits of functional programming and parallel programming. The input language of Funk syntactically resembles the Go programming language. The output of the translator is C code, which, coupled with a library, compiles to a native binary.

Funk is like a functional programming language. It treats functions as values. In other words, a Funk program can assign functions to variables, pass them as parameters, and return them from other functions. More significantly, Funk uses functional programming concepts to make parallel programming easy to express, and safe to execute. Aside from these features, Funk's syntax, and most of its semantics resemble those of common imperative languages.

1.1 Background

Parallel programming, although not novel, is still cumbersome on legacy programming languages. Consider the C language as an example. It has really great support from compilers. Comparing with other languages, the generated binary executes very fast on every platform (including embedded devices). The primary method to achieve parallelism in the C language is through using the pthread library. However, unbounded concurrency that offered by the pthread library almost inevitably leads to errors. On top of that, using the pthread library to setup and execute a simple thread requires long and mechanical steps.

On the other hand, Go is a more novel programming language, and like recent languages, such as Python, Ruby, Go, Scala and C# 4.0 it mixes several programming language paradigms, including imperative, object-oriented and functional programming. It has construct to make parallel programming easier for the programmer. Go's goroutines [1] execute functions of arbitrary types with a syntax that resembles a sequential function call in parallel to the main thread.

We want to leverage the Go language in order to enable safe parallel execution by using some concepts of functional programming. Functions are especially promising for safe, parallel execution, since, in the strictest, mathematical sense, they have no side effects. Unlike the pthread library which does not impose any constrain on the side effects of a thread.

[1] http://golang.org/doc/effective_go.html#goroutines

1.2 Related Work

As previously mentioned, several modern programming languages support parallel programming, and mix at least the three main programming language paradigms. Funk, derived from Go, is no exception. Of course, these languages differ by how much of the language uses functional concepts, and how much of it uses imperative concepts.

Although C is an imperative programming language, and languages such as Python, Go and C# follow multiple paradigms, their syntax draws heavily from that of C. Most notably, these languages use blocked variable scoping, and follow similar operator precedence rules [2].

Programs written in the Go language's web-based interpreter [3] reveal that, like C and Java, the scope is static, and the function parameters are calculated in applicative order. Moreover, global, named functions that change global variables indeed cause side effects, but when function calls are used as parameters, their execution is not left-to-right.

On the function-oriented side, the existing programs show that its goroutines rely on existing functions to execute code in parallel. Besides the use of functions for parallel programming, the greatest similarity to functional programming is its ability to assign and return functions as if they were data.

Using the Go language as a starting reference, we wish to reap more of the benefits of functional programming languages, without violating more of the intuitions of imperative programs.

1.3 Goals

1.3.1 Familiarity

We chose the Go syntax as a model for the input syntax to challenge ourselves with a language that differs from the output language, C. Therefore, a number of features in the Go language is unfamiliar with programmers better versed in Java, or more imperative languages such as C. While we retained novel aspects that would reduce code complexity, and added restrictions to avoid unwanted surprises, our choice in what to retain, and what features of the Go language to add to Funk, and what features to discard is based around our desire to make the style of Funk familiar to more traditional programmers.

1.3.2 Flexibility

Treating functions as data, which a programmer can define inside a function, reflects our goal to allow the programmer to easily apply different code to similar frameworks, without repetition required to create multiple, similar functions, and passing in all the local information required for a program to execute.

1.3.3 Safety

On the other hand, we wanted to restrict the side effects of functions and asynchronous code to avoid possible errors due to the side effects of functions on outside variables, and bugs due to asynchronous manipulation of data.

[2] http://golang.org/ref/spec#Operator_precedence
[3] http://golang.org/#

2 Tutorial

2.1 Using the Compiler

Inside the *src* folder, type **make**. This creates the Funk to C compiler, **frontend/fk2c**, which, given a file in the Funk language, outputs C code to standard output. For full compilation, from Funk to C to binary, use **frontend/fk2c**, which takes the Funk input file, the C output file and the binary output file.

The following sample Funk code demonstrates the following features:

- The mandatory main function, with no parameters and no return
- Calling the built-in **print** function, which takes an arbitrary list of non-function values
- A character array, or string, with escape characters

```
func main () {
    print("Hello World!\a\n")
}
```

Listing 1: Compilable hello world

To compile the sample code above, type:

```
> frontend/funkc hello_world.fk c_code.c binary
```

The output will be:

```
> ./binary
Hello World!
```

2.2 Data manipulation

2.2.1 Primitives

Funk's declaration and assignments work as follows:

- A newly-declared variable is preceded by **var**, and is succeeded by its type, such as **int**, **double**, **bool** or **char**
- A single declaration or assignment can assign multiple values to multiple statements. Moreover, the assignments do not affect the calculation of the values on the right side of the equal sign.

```
func main () {
    /* this is a comment */
    // So is this
    var a, b int = 0,1
    print("Before: a is ", a, "\nb is ", b, "\n")
    a, b = b, a
    print("After: a is ", a, "\nb is ", b, "\n")
    a, b = b + 1, a
    print("Last: a is ", a, "\nb is ", b, "\n")
}
```

Listing 2: Assignments and arithmetic expressions

Note the swap in the output:

```
Before: a is 0
b is 1
After: a is 1
b is 0
Last: a is 1
b is 1
```

2.2.2 Arrays

Arrays are indexed collections of values, including arrays.

- Elements, including subarrays, can be read or overwritten.
- Copies of arrays are deep, so that a change to one copy does not affect another.
- Note that the global variable outside the function, **b**, is never on the left hand side of any assignment, ever since it was declared. In fact, the program cannot modify any global variable.
- The number of elements in an array is denoted by the expression in the right-most bracket pair. The remaining size expressions for the subarrays are only for comments. If the size is not given, then the size is the number of arrays in the literal.

```
var b [3]int = [3]int{1, 2, 3}
func main() {
    /* access single element */
    print("b[1] = ", b[1], "\n")
    /* multi-dimensional matrix */
    var matrix [][]int = [][]int{[]int{1, 2, 3}, []int{4, 5, 6}, []int{7, 8, 9}}
    print("matrix = ", matrix, "\n")
    /* access subarray */
    print("matrix[2] = ", matrix[2], "\n")
    /* deep copy of matrix */
    var cm [][]int = matrix
    /* changes do not affect each other */
    matrix[0][0] = -1
    print("matrix = ", matrix, "\n")
    print("cm = ", cm, "\n")
}
```

Listing 3: Array operations

print iteratively outputs the elements of an array, side-by-side, to be consistent with printing character arrays like strings.

```
b[1] = 2
matrix = 123456789
matrix[2] = 789
matrix = -123456789
cm = 123456789
```

2.3 Control Flow

Control flow statements resemble their counterparts in C and Java, with a few exceptions

- The condition must be a binary
- There are no parentheses around the header, but there must be curly braces around the body
- A **for** statement with a single, binary, expression, denotes a **while** statement

```
func main() {

    var a int = 0
    /* If-else */
    if a<10 {
        print(a," is less than 10\n")
    } else {
        print(a," is not less than 10\n")
    }

    /* for-as-while */
    for a < 10 {
        print("while: a is ", a, "\n")
        a = a + 1
    }
    /* only if */
    if a == 10 {
        print (a, " is equal to 10\n")
    }

    /* for-as-for */
    for a = 0; a < 5; a = a + 1 {
        print("for: a is ", a, "\n")
    }
}
```

Listing 4: Control Flow

Following the control flow, the program outputs:

```
0 is less than 10
while: a is 0
while: a is 1
while: a is 2
while: a is 3
while: a is 4
while: a is 5
while: a is 6
while: a is 7
while: a is 8
while: a is 9
10 is equal to 10
for: a is 0
for: a is 1
for: a is 2
```

```
for: a is 3
for: a is 4
```

2.4 Global Functions

While the "Hello World" example demonstrated the use of one of the built-in functions, the user can also define custom functions. The following function calculates and returns a Fibonacci number. Note that the return type is the last part of the function header, before the left brace:

```
func fib_array(n int) int {
    var fibs []int = [n + 1]int{}
    var i int

    fibs[0] = 0
    fibs[1] = 1
    for i = 2 ; i < n + 1 ; i = i + 1 {
        fibs[i] = fibs[i - 1] + fibs[i - 2]
    }

    /* Return value */
    return fibs[n]
}

func main () {
    /* pass in parameter */
    print(fib_array(10), '\n')
}
```

Listing 5: Function call

This function returns the 10th Fibonacci number:

```
55
```

Global variables can also call themselves, as does the following factorial function:

```
/* fact is declared in global scope */
func fact (a int) int {
    if a == 1 {
            return a
    } else {
        /* fact can be used here */
        return a * fact(a - 1)
    }
}

func main(){
    /* outputs 120 */
    print(fact(5))
}
```

Listing 6: Recursive function

$5! = 120$:

```
120
```

2.5 Anonymous Functions and Closure

Functions can also be declared anonymously, to be passed into variables or arrays, or as return values. Since these functions can exist outside the scope in which they were declared, Funk supports function closure. This means that each function keeps an independent copy of any variables it accesses, and calling an anonymous function could affect its subsequent execution, but not those of other functions.

```
/* adapted from Go's website: http://golang.org/ */
func fib() func () int {
    /* variable declared outside anonymous function */
    var a, b int = 0, 1
    return func () int {
        /* function changes own state only */
        a, b = b, a + b
        return a
    }
}

func main() {
    /* assign function like variables */
    var f func () int = fib()

    /* left-to-right printing and execution */
    print(f(), f(), f(), f(), f(), f(), f())
}
```

Listing 7: Function closure

The sequence of Fibonacci numbers that the program outputs are are (note the 13 squashed at the end, due to the printing format):

```
11235813
```

Like arrays, functions are copied deeply, by value.

```
func main () {
    var i int
    i = 0
    print("i ", i, "\n")
    /* this will not affect i */
    var inc func () int = func () int {
        i = i + 1
        return i
    }

    /* inc2 will be independent of inc */
    var inc2 func () int = inc
    print("inc: ", inc(), "\n")
    print("inc2: ", inc2(), "\n")
    print("inc2: ", inc2(), "\n")
    print("inc2: ", inc2(), "\n")
    /* inc prints from where it left off */
    print("inc: ", inc(), "\n")
    /* i is unchanged */
    print("i ", i, "\n")
```

```
}
```

Listing 8: Function closure

The output of **inc** and **inc2** are independent:

```
i 0
inc: 1
inc2: 1
inc2: 2
inc2: 3
inc: 2
i 0
```

2.6 Asynchronous blocks

Funk allows the program to execute parallel code in designated asynchronous blocks. As with function closure, the asynchronous block copies in all the values it needs, and the only external effect is its return value.

```
func fib (n int) int {
    if n == 0 {
        return 1
    }
    if n == 1 {
        return 1
    }
    /* recurse asynchronously */
    var a int = async {
        return fib(n - 1)
    }
    var b int = async {
        return fib(n - 2)
    }
    return a + b
}

func main () {
    print(fib(5), "\n")
}
```

Listing 9: Asynchronous code

The program outputs a single Fibonacci number:

```
8
```

3 Language Reference Manual

3.1 Introduction

This manual describes Funk, a functional programming language with an imperative structure. Funk is designed to allow simpler parallel programming and higher order functions. Features defining the language include dynamic function declaration, asynchronous block declarations and strong typing. This manual describes in detail the lexical conventions, types, scoping rules, built-in functions, and grammar of the Funk language.

3.2 Lexical conventions

3.2.1 Identifiers

An identifier in Funk represents a programmer-defined object. The identifier starts with a letter or an underscore, and is optionally followed by letters, numbers or underscores. An identifier name is thus defined by the following regular expression:

```
['a' - 'z' 'A' - 'Z' '_']['a' - 'z' 'A' - 'Z' '0' - '9' '_']*
```

3.2.2 Keywords

Funk has a set of reserved keywords that can not be used as identifiers.

3.2.2.1 Statements and Blocks The following keywords indicate types of statements or blocks:

```
var func async
```

3.2.2.2 Control Flow The following keywords are used for control flow

```
if else for break return
```

3.2.2.3 Types Each primitive type, integer, double, boolean and character, has a name that the program uses for declarations.

```
int double bool char
```

A function type, and a function declaration begins with the **func** keyword.

3.2.2.4 Built-in Functions The following keywords are reserved for built in functions.

```
print double2int int2double bool2int int2bool
```

3.2.3 Constants

Funk supports integer, double, character, and boolean constants, otherwise known as literals, inside expressions. Any array with any level of nesting can be also expressed as a literal. Arrays of characters have a more concise representation as string literals.

3.2.3.1 Integer Define a decimal digit with the following regular expression:

```
digit = ['0' - '9']
```

An int is a 32-bit signed integer, and consists of at least one digit.

```
digit+
```

3.2.3.2 Double A double is a 64-bit ("double precision") floating point number. Like the floating-point constant in the C language, according to Kernighan and Ritchie, the constant consists of an integer part of digits, as described in 3.2.3.1, a decimal point, a fractional part –ie. the part of the floating point whose absolute value is less than 1, and an exponential part, starting with ‘e’, followed by an optional sign, and then at least 1 digit, which indicates the power of 10 in the scientific notation. Define the regular expression of the exponential part as follows:

```
exp = 'e' ['+' '-']? ['0'-'9']+
```

If the decimal point is present, at least one of the the integer and fractional parts must also be present –the compiler interprets an absent part as 0. If there is no decimal point, the integer part and the exponent must be present:

```
((digit+ '.' digit* | '.' digit+) exp?) | (digit+ exp)
```

Therefore, the following literals are valid doubles:

```
1.2 .2 1. 1.2e3 .2e3 1.e3 1e+3 1e-3
```

However, strings lacking digits either before or after the exponent marker or decimal point, if each exists, and integer constants are not valid doubles:

```
12 e3 . .e 1.2e 1.2e+
```

3.2.3.3 Boolean The boolean type has two predefined constants for each truth value, and no other values:

```
true false
```

3.2.3.4 Characters and Character Arrays Characters are single, 8-bit, ASCII characters. Generally, the character values are representable in their printable form. However, not all characters are printable, and some printable characters conflict with the lexical conventions of characters and character arrays. They are therefore specially marked with a backslash, and considered single characters. Funk supports the following escape sequences:

- New line:

  ```
  \n
  ```

- Carriage return:

  ```
  \r
  ```

- Tab:

  ```
  \t
  ```

- Alarm:

  ```
  \a
  ```

- Double quotation marks, to fit inside character array literals:

```
\"
```

- The backslash itself:

```
\\
```

A literal of character type consists of a single character or escape sequence inside two single quotes:

```
'c'
```

Note that the printable characters with escape sequences, ie. the double quotation marks and backslash, do not have to be escaped, because the compiler processes a single character between two single quotes without any translations (which is why there is no escape sequence for single quotes). But the compiler also accepts their escaped form. For example, the following two literals are equivalent:

```
'\', '\\'
```

A string is a character array. Since strings are widely used we considered a special representation for string constants which begins and ends with unescaped double quotes.:

```
"foo"
```

This is equivalent to:

```
[3]char {'f','o','o'}
```

Each single character or backslash-character pair is a single entry of the character array.

3.2.4 Punctuation

Funk supports primary expressions using the previously-mentioned identifiers and constants. Primary expressions also include expressions inside parentheses. In addition, parentheses can indicate a list of arguments in a function declaration, a function variable declaration, or a function call:

```
f() //function call
```

Braces indicate a statement in blocks, including blocks that make up function bodies. They are also used for array literals:

```
{
//this is a block
}

var a [2]int = [2]int{1,2} // more about arrays in later sections
```

Brackets indicate array types and access to array elements:

```
a[0] //access to element 0 in array a
[3]int //this is a data type
```

Semicolon is used to separate statement and expression in a for loop:

```
for i=0;i<10;i=i+1
```

Comma is used to separate elements of a list:

```
int a,b = 0,1
```

3.2.5 Comments

Multiline comments in Funk start with /* and terminate with the next */. Therefore, multiline comments do not nest. Funk also supports single-line comments starting with // and ending at the end of the line.

3.2.6 Operations

Funk supports a number of operations that resemble their arithmetic and boolean counterparts.

3.2.6.1 Value Binding A single equal sign indicates assignment in an assignment or declaration statement:

```
=
```

3.2.6.2 Operators Funk supports the following binary operators:

```
-, *, /, %
&, ^, |, <<, >>
==, !=, <=, <, >=, >
&&, ||
```

Funk also supports the following unary negations:

```
!, ~
```

And there are two operations that can be unary or binary, depending on the expression:

```
+, -
```

3.3 Syntax

3.3.1 Program structure

At the highest level, the program consists of declarations only. While a program can initialize global variables with expressions, all other statements, including assignments to existing variables, must be inside function bodies:

`Global function or variable declarations (interchangeable)`

program:
  declaration
  program declaration

declaration:
  funcdec
  vardec `new-line`
  `new-line`

3.3.1.1 Variable Declarations

Variables can be declared and initialized globally, or locally in a function, with the *vardec* rule

`var` new-id-list var-type `new-line`

var is the keyword. *new-id-list* can be one or more newly declared identifier tokens. *var-type* is the type, which has to be one of the allowed primitive types or function, or a possibly-multidimensional array of the aforementioned types.

var-type:

array-marker$_{\text{opt}}$ single-type

single-type:

`func` (formal-list$_{\text{opt}}$) var-type$_{\text{opt}}$
`int`
`double`
`char`
`bool`

array-marker:

array-level
array-marker array-level

array-level:

[]
[expression]

The declaration ends with a new line.
Variables can also be initialized:

`var` new-id-list var-type = actual-list `new-line`

actual-list is a comma-separated list of expressions and anonymous function declarations (which cannot be inside expressions, since there are no operations on functions). 3.3.2 will describe expressions in detail.

3.3.1.2 Global and Anonymous Functions

Global functions are defined the following way:

`func` ID (formal-list$_{\text{opt}}$) var-type$_{\text{opt}}$ block

func is the keyword. *ID* identifies the instance of the function. If *var-type* is not specified, the function cannot return a value. *block* contains the function body. *formal-list* (optional) in the parentheses would include the formal arguments, which are clusters of variables of the same type, indicated by a *var-type*. If there is only one variable of the type, the *single-type* use of *formal-list* can omit the name, but for the purpose of function declaration headers, all names must be present:

formal-list:

formal-type-cluster
formal-list **,** formal-type-cluster

formal-type-cluster:

> var-type
> new-id-list var-type

The mandatory global function in Funk is main, which marks the entry point of a program. It takes no parameters (so it does not support command line arguments), and returns no values.

```
func main() {
    print("Hello world!\n")
    print("Goodbye world!\n")
}
```

Listing 10: main function in a sample program

Anonymous functions are declared with a similar syntax and are used like expressions (see 3.3.3) but without an identifier, as the program can refer to them as variables. Except for the fact that the parameters must all have names here, the function header, excluding the body, resembles the corresponding *single-type*:

> `func` (formal-list$_{\text{opt}}$) var-type$_{\text{opt}}$ block

3.3.2 Expressions

3.3.2.1 Primary expressions Primary expressions consist of:

- Literals
  - The primitive literals mentioned in 3.2.3, which the grammar calls *INT_LIT*, *DOUBLE_LIT*, *BOOL_LIT* and *CHAR_LIT*.
  - Character arrays of the token *STRING*
  - Identifiers of the token *ID*
  - Arrays: Array literals begin with their type, and the expressions they contain must be of the same type; they are of the same primitive type as the array, but have one less array level

    actual-arr:
    > arr-header `{` actual-list$_{\text{opt}}$ `}`

    arr-header:
    > arrary-marker single-type

    For example, a 3x2, 2-level array looks like this:

    `[3][2]int {[2]int{1,2},[2]int{3,4},[2]int{5,6}}`

- Asynchronous blocks have the value and type of whatever the block returns.

  > `async` block

  However, the program will not wait for the return value until it is required in another expression. Therefore, if the block is inside another expression, it always blocks until completion. And if it is an rvalue, the program does not block until the corresponding lvalue is used inside another expression.

```
{
    var i int = 3 + async { return 0 } /* blocks immediately */

    i = async { return 1 } /* does not block yet */
    var j int = i /* blocks */

    j = async { return 2 } /* does not block yet */
    i = 1 + j /* blocks */
}
```

Listing 11: Blocking conditions for async

- Parenthesized expressions

    (expression)

3.3.2.2 Precedence of Operations

We list the operations in order from highest to lowest precedence. The full grammar enforces precedence using similar rules as the C grammar, according to its LRM.

1. Under certain conditions, the two postfix expressions appear outside of expressions that the program evaluates.

   The program can call functions if the *expression* before the parentheses is of type *func*, and the types of the actual parameters in *actual-list*, if any, match the formal arguments of the function. The type of the expression is the type of the return value, so, as an expression, the function must return a value.

   func-call-expr:

   expression ($\text{actual-list}_{\text{opt}}$)

   An expression can extract its value from an array. The left *expression* must be an array type, so that the value resulting from the array access has a type that is one array level shallower than that of the left expression (eg. if a has type $[][]int$, then $a[0]$ has type $[]int$). The right *expression*, inside the brackets, must be of integer type.

   obj-get-expr:

   primary-expr
   expression [expression]

   We reveal the *primary-expr* rule not only to show the precedence, but also because accessing a pure primary expression is useful when this expression is used in assignments, which 3.3.3.1 will explain.

2. Prefix, unary operations

   un-op expression

   - `+` optional positive sign for *int* and *double*
   - `-` 2-complement negation for *int* and *double*
   - `!` Boolean negation for *bool*
   - `~` 1-complement (bitwise) negation for *int*

3. Binary operations. In addition to specific restrictions, the two operands of a binary operation must have the same type. Unless noted otherwise, they will return the same type. The subscripts are added for clarity.

$expression_a$ bin-op $expression_b$

The items in the following list start from the highest precedence, and all operations are left-associative.

(a) `*, /, %`: Arithmetic multiplication and division are valid for both *int* and *double* types. The modulo operation gives the positive remainder when $expression_b$ divides $expression_a$. Therefore, the modulo operation only applies to integers.

(b) `+, -`: Arithmetic addition and subtraction of *int* and *double* values.

(c) `<<, >>`: Bitwise shift for *int* values. Return $expression_a$ shifted left or right by $expression_b$ bits. $expression_b$ can be negative, in which case the shift direction is reversed, and the expression shifts by the absolute value of $expression_b$.

(d) `<, >=, >, =<`: Real-number comparison for *int*, *double*, and *char* values. Returns *bool* value.

(e) `==, !=`: Equality and inequality for *int*, *double*, *char*, and *bool* values. Returns *bool* value.

(f) `&`: Bitwise AND for *int* values. The program evaluates both $expression_a$ and $expression_b$.

(g) `^`: Bitwise XOR for *int* values. The program evaluates both $expression_a$ and $expression_b$.

(h) `|`: Bitwise OR for *int* values. The program evaluates both $expression_a$ and $expression_b$.

(i) `&&`: Logical AND for *bool* values. The program evaluates $expression_b$ if and only if $expression_a$ is *true*

(j) `||`: Logical OR for *bool* values. The program evaluates $expression_b$ if and only if $expression_a$ is *false*

3.3.3 Statements

3.3.3.1 Assignments

assign-stmt:

obj-get-expr-list = actual-list `new-line`

An assignment statement defines an *obj-get-expr* list as the lvalues. Unlike its usage in *expression*, each lvalue must be either an *ID* token, or any n-level array access of an array object indicated by an *ID* token. This rule assures that there is an identifier that can reach the assigned value.

The rvalues are in actual-list, an expression list followed by a new line to end the statement. The expressions are of various types, and each i^{th} lvalue will store the evaluated value of the i^{th} rvalue. Therefore, each i^{th} lvalue and rvalue must match in type, and the number of lvalues and rvalues must be the same.

3.3.3.2 Blocks and Control Flow

- Block

  A block is defined inside curly braces, which can include a possibly-empty list of statements.

- Selection statement

  A selection statement is an if or if-else statement that takes an expression that evaluates to a *bool* value:

  `if` expression block
  `if` expression block else block

  There exists ambiguity with the selection statement: "if expression if expression else", which is why Funk selects between blocks rather than statements.

- Iteration statement

  An iteration statement begins with the *for* keyword. We support three types of for loops. The first is a regular for loop with a starting assignment, a boolean loop condition expression, and an assignment for advancing to the next iteration. The three parts are separated by semicolons. The other loops are a for loop with one expression (similar to while loop), and a for loop with no expression. The expressions must evaluate to a *bool* value. A missing expression implies *true* –ie. an infinite loop:

  `for` assign-stmt$_{\text{opt}}$; expression$_{\text{opt}}$; assign-stmt$_{\text{opt}}$ block
  `for` expression$_{\text{opt}}$ block

- Jump statements

  The return keyword accepts an optional anonymous function or expression and ends with a newline. It exits out of the smallest containing function or async body. *async* is an expression described in 3.3.2.1

  return expression$_{\text{opt}}$ `new-line`

  The *break* keyword breaks iteration of the smallest containing for loop. In other words, it jumps to the code immediately following the loop.

- A statement can call functions using the *func-call-expr* syntax. Changes to the state of the function due to the call persist. Therefore, the function *expression* to the left of the parentheses must be an *ID*, or an n-level array access of an *ID*, ie. an lvalue that can store the changed function instance. The function does not need to return a value, and the program discards any value that it does return.

- *vardec*

3.4 Scoping Rules

Funk uses lexical scoping: the scope of an object is limited to the block in which it is declared, and overrides, or suspends, the scope of an object with the same identifier declared in a surrounding block.

3.4.1 Lexical Scoping with Blocks

When the program declares object o with identifier *id* in declaration D_o, D_o can assign a value to o using *id*. Moreover, in B_o, the block that directly contains D_o, any statement after D_o that assigns to or reads from *id* in fact does so from o, with two important exceptions.

The first is function closure, which subsection 3.4.2 will cover in more detail. The second exception is the declaration of the same variable inside the first approximation of the scope of o. When *id* is on the left side of another declaration, D'_o, inside a block B'_o, contained in B_o, then *id* is not bound to o starting from the *left* side of D'_o until the end of B'_o. Instead, D'_o will create a new object, o', and *id* will refer to it until the end B'_o, aside from the previously-mentioned exceptions.

```
1  {
2      var i int = 0
3      print(i) /* 0 */
4      if (i == 0) {
5          i = i + 1 /* this will refer to the i from line 2 */
6          print(i) /* 1 */
7          var i char = 'c' /* suspend scope of i from line 2 */
8          print(i) /* c */
9          if (i == 'c') {
10             var i double = 1.1 /* suspend scope of i from line 7 */
```

```
            print(i) /* 1.1 */
        }
        print(i) /* c */
    }
    print(i) /* 1 */
}
```

Listing 12: Scope suspended in contained block

Note that for globally-declared variables, B_o includes the entire code. However, in this case, id can only be used inside a function body.

In certain contexts, id cannot refer to multiple objects. While D'_o can redeclare id in a block that B_o contains, B_o cannot directly contain D'_o; the programmer cannot redeclare id in the same block.

```
{
    var i int = 0 /* first declaration */
    print(i) /* 0 */
    var i int = 10 /* illegal redeclaration */
}
```

Listing 13: Illegal variable redeclaration

Likewise, the program cannot use the old o when redefining it in block B'_o

```
{
    var i int = 0
    if (i == 0) {
        /* i is 0 */
        var i int = i + 1 /* i is 0 on the right, and 1 on the left */
        /* i is 1 */
    }
    /* i is 0 */
}
```

Listing 14: Cannot use old object for defining new object with the same Id in a nested block.

3.4.2 Function Closure

Each function instance has an environment associated with it. Let C be the scope of object o, with identifier id. For a function, F, inside C, let $S = \{s_0, \ldots s_n\}$ be all the statements in F that use id that would refer to o according to the rules detailed in the previous subsection. If s_i is the first statement in which id appears on the left hand side, or is called as a function instance, all $s_j : j < i$, as well as id on the right side of s_i, use the value of o as it appeared before the declaration of F, or since the last execution of the same instance of F. On the other hand, all statements $s_j : j \geq i$ use a new o' that was a distinct, deep copy from o. Likewise, any changes to o after F does not change the values used in F, even when the program executes the instance after the change. As a consequence, the only effect of a function on the outside scope is through its return values, and not through side effects or parameters, which are passed as values or deep copies.

```
{
    var i int = 0
    print(i) /* 0 */

    var inc func() int = func() int {
        print(i)
```

```

        i = i + 1

        print(i)

        return i
    }

    print(i) /* 0 */

    var j int = inc() /* "01" */
    print(i) /* 0 */
    print(j) /* 1, because it contains the return value */

    i = 100
    print(i) /* 100 */

    j = inc() /* "12", which the last assignment did not change */

    /* deep copy of function */
    var inc_inc func() int = func() int {
        return inc()
    }

    j = inc_inc() /* "23" */
    print(j) /* 3 */

    j = inc_inc() /* "34" */
    print(j) /* 4 */

    j = inc() /* "23", which inc_inc did not change */
    print(j) /* 3 */
}
```

Listing 15: Examples of closures; block comments indicate the value printed to standard output

Closure is necessary primarily in order to support higher order functions. Functions in funk can be passed around and subsequently executed outside of their original scope. Consider the following example as an explanation why closures are necessary:

```
// fib returns a function that returns
// successive Fibonacci numbers.
func fib() func() int {
    var a, b int := 0, 1
    return func() int {
        a, b := b, a+b
        return a
    }
}

func main() {
    f := fib()
    print(f()) //1
    print(f()) //1
    print(f()) //2
```

```
    print(f()) //3
    print(f()) //5
}
```

Listing 16: Function invoked outside its original scope

The fib function returns an anonymous function to the caller. This anonymous function has variables *a* and *b* as free variables and gets executed outside its original scope, specifically in the scope of the main function. What happen is that at function declaration time the anonymous function creates a copy of a and b (the function environment) from the surrounding scope which will be used on subsequent invocations.

Closures are also used to avoid race conditions in async blocks.

```
{
    var i int = 0
    var a, b int

    print(i) /* 0 */

    a = async {
        i = i + 1
        return i
    }
    print(i) /* 0 */

    i = 10
    b = async {
        i = i + 2
        return i
    }

    print(a) /* 1 */
    print(b) /* 12 */
    print(i) /* 10 */
}
```

Listing 17: Closure for async Block

The two async blocks seem to compete for variable i defined in the outer scope. Both blocks will instead create a closure of all their free variables effectively eliminating race conditions.

3.4.3 Assignment and Parameter Passing

In Funk, we try to minimize side effects, including those that may arise in assignments. Therefore, copy-assignment statements, or assignments that use a variable directly on the right without performing any operations, copy the value, rather than reference, of the variable. In addition, the assignment reads the values of the right hand side before it could change any of them.

Let S be the scope of o as defined in the previous subsections. If o is on the right side of an assignment, a, the target object of the assignment, o_a, identified by id_a, stores a deep copy of the value of o.

```
{
    /* functions */
    var i int = 0
    print(i) /* 0 */

```

```
    var inc func() int = func() int {
        i = i + 1

        print(i)

        return i
    }

    inc() /* 1 */

    var inc2 func() int = inc /* copying function */

    inc() /* 2 */
    inc() /* 3 */

    inc2() /* 2 */
    inc() /* 4 */

    /* arrays */
    var arr [2]int = [2]int{0, 1}
    print(arr[0]) /* 0 */

    var arr2 [2]int = arr /* copying array */
    print(arr2[0]) /* 0 */
    arr2[0] = 2
    print(arr2[0]) /* 2 */
    print(arr[0]) /* 0 */
}
```

Listing 18: Deep copies of function instances and arrays

Our language supports assignment of multiple objects. The right hand side of the assignment gets evaluated first **from left to right**, then the results are copied to the objects. Therefore, changes to objects on the left side do not affect their values on the right side. This enables swapping of values of objects.

```
{
    var a, b int = 0, 1
    print(a, b) /* 0 1 */

    /*
     * the assignment changes a and b, but not before it reads the
     * original values
     */
    a, b = b, a
    print(a, b) /* 1 0 --note the swap */
}
```

Listing 19: Assignment to the left side does not affect the right side

However, being free from side effects does not imply that functions are referentially transparent in our language (due to closures). Evaluations of the same function at different times can held different results if the function environment is modified between different function calls. Consider the following example using the Fibonacci closure and multiple assignments.

```
// fib returns a function that returns
```

```
// successive Fibonacci numbers.
func fib() func() int {
    var a, b int := 0, 1
    return func() int {
        a, b := b, a+b
        return a
    }
}

func main() {
     var a,b,c = fib(),fib(),fib() // a=1 b=1 c=2 ..
}
```

Listing 20: Multiple assignment and closures

For consistency with copying-by-value, as well as function closure, function calls also copy parameters by value, evaluating the right side expressions from left to right.

```
{
    var i int = 0
    print(i) /* 0 */

    var inc func(int) int = func(i int) int {
        i = i + 1

        return i
    }

    print(inc(i), i) /* 1 0 */
}
```

Listing 21: No side effects on parameters

3.5 Type Conversions

Funk is a strongly typed language; therefore it performs no implicit type conversion. It is the responsibility of programmers to convert operands to the correct type. For example, consider arithmetic operations between an integer and a floating point operand:

```
var a int = 1
var b double = 2.0
var c double = a + b //the compiler rejects the expression, as the types are not the same
```

The programmer has to explicitly convert operands:

```
var a int = 1
var b double = 2.0
var c double = int2double(a) + b
```

We believe that this approach is less error prone than implicit conversion. For a complete list of conversion functions see Section 3.6.

3.6 Built-in Functions

3.6.1 Conversion Functions

Funk has four conversion functions to and from the int type:

- double2int(x double) int: The function discards the fractional part of x and returns the integer part of x as an int.
- int2double(x int) double: The function returns a double with the fractional part equal to 0 and the integer part equal to x.
- boolean2int(x bool) int: If x is **true**, the function returns 1. Otherwise it returns 0.
- int2boolean(x int) bool: If x is equal to 0, the function returns **false**. Otherwise it returns **true**.

3.6.2 The print function

A Funk program performs printing using the **print** function. The syntax and semantics of the **print** function are inspired by the Python 3 function with the same name. Like the **async** keyword, the **print** function is not present in the Go language that we are using as the baseline. Go uses the Printf function included in the fmt package as the standard function for formatted I/O. Even though the authors of Go claim to have better mechanisms than the C language printf [4], the semantics and syntax of fmt.Printf are almost indistinguishable from its C counterpart. Therefore we decided to implement a function similar to the Python 3 print function for the following reasons:

- The print function does not have a format string, making formatted I/O simpler and less error prone.
- The print function is polymorphic, which is also helpful for the programmer.

For example consider the following snippet:

```
var a int = 1
var b int = 2
print (a,"+",b," is ", (a+b)) // prints 1+2 is 3
```

The syntax of our print command is the following:

```
print([expression, ...])
```

The function prints the concatenation of the string representation of each expression to standard output. It does not automatically end the output with a newline, but the user can include expressions whose string representations include a newline.

In contrast, print function in Python 3 has these features that allow programmers to specify 3 optional parameters:

1. `sep` is a separator that print outputs to I/O between every expression in the list.
2. `end` is a string that the function prints after it has output all the expressions.
3. `file` specifies the destination for the print statement.

We are not implementing the first two because the simplified **print** function can still output any format by manually adding in the separators and the end of line string. And since we are not considering file I/O for our language, we are not going to implement the last as well.

[4] http://golang.org/pkg/fmt/

3.7 Grammar

In the grammar listed below, we have some of undefined terminals like *ID*, *INT_LIT*, *DOUBLE_LIT*, *CHAR_LIT*, *BOOT_LIT* and *STRING*. They are tokens passed in from the lexer. The words in `textwriter` style are terminal symbols given literally, with the exception of `new-line` indicating the line break. And for symbols with the subscript $_{opt}$, they will be expanded in the actual grammar with a non-terminal that consists of an empty part and the actual symbol. "one of" indicates separate tokens, listed in a single line, to which the non-terminal could expand. And with those indicated above, the grammar represented here will be accepted by the ocamlyacc parser-generator.

program:

> declaration
> program declaration

declaration:

> funcdec
> vardec `new-line`
> `new-line`

block:

> `{` stmt-list$_{opt}$ `}`

stmt-list:

> `new-line`
> stmt-list stmt `new-line`
> stmt-list `new-line`

funcdec:

> `func` `ID` `(` formal-list$_{opt}$ `)` var-type$_{opt}$ block

anon:

> `func` `(` formal-list$_{opt}$ `)` var-type$_{opt}$ block

vardec:

> `var` new-id-list var-type
> `var` new-id-list var-type `=` actual-list

formal-list:

> formal-type-cluster
> formal-list `,` formal-type-cluster

formal-type-cluster:

> var-type
> new-id-list var-type

new-id-list

`ID`
new-id-list `, ID`

single-type:

`func (` formal-list$_{\text{opt}}$ `)` var-type$_{\text{opt}}$
`int`
`double`
`char`
`bool`

var-type:

array-marker$_{\text{opt}}$ single-type

arr-header:

array-marker single-type

array-marker:

array-level
array-marker array-level

array-level:

`[]`
`[` expression `]`

actual-arr:

arr-header `{` actual-list$_{\text{opt}}$ `}`

actual-list:

expression
anon
actual-list `,` expression
actual-list `,` anon

obj-get-expr-list:

obj-get-expr
obj-get-expr-list `,` obj-get-expr

assign-stmt:

obj-get-expr-list `=` actual-list `new-line` stmt:

block
func-call-expr
`return` expression
`return` anon
`break`
`if` expression block
`if` expression block `else` block
`for` assign-stmt$_{\text{opt}}$ `;` expression$_{\text{opt}}$ `;` assign-stmt$_{\text{opt}}$ block
`for` expression$_{\text{opt}}$ block
assign-stmt
vardec

expression:

or-expr

or-expr:

and-expr
or-expr `||` and-expr

and-expr:

bor-expr
and-expr `&&` bor-expr

bor-expr:

bxor-expr
bor-expr `|` bxor-expr

bxor-expr:

band-expr
bxor-expr `^` band-expr

band-expr:

eq-expr
band-expr `&` eq-expr

eq-expr:

comp-expr
eq-expr eq-op comp-expr

eq-op: one of

`==` `!=`

comp-expr:

shift-expr
comp-expr comp-op shift-expr

comp-op: one of

`<` `<=` `>` `>=`

shift-expr:

add-expr
shift-expr shift-op add-expr

shift-op: one of

`<<` `>>`

add-expr:

mult-expr
add-expr add-op mult-expr

add-op: one of

`+` `-`

mult-expr:

un-expr
mult-expr mult-op un-expr

mult-op: one of

`*` `/` `%`

un-expr:

post-expr
un-op un-expr

un-op: one of

`-` `+` `!` `~`

post-expr:

obj-get-expr
func-call-expr

obj-get-expr:

primary-expr
post-expr `[` expression `]`

func-call-expr:

post-expr `(` actual-list$_{\text{opt}}$ `)`

primary-expr:

```
INT_LIT
DOUBLE_LIT
CHAR_LIT
BOOL_LIT
STRING
ID
```
actual-arr
`async` block
(expression)

4 Project Plan

4.1 Planning Process

Throughout the project we used an iterative planning process, where we initially set the main goals and milestone deadlines for building the Funk compiler and then iteratively created short-term goals as we worked deeper through each milestone. The main milestones we set and our actual full project log are outlined in the following sections. The choice of milestones was based on the stages we believed existed in the compilation process, as well as suggestions from our meetings with Prof. Edwards.

4.2 Specification Process

We had an initial specification of the set of features we wanted our language to achieve, and the main building blocks that would be required to build the end-to-end compiler. From the beginning, we planned for Funk to be a derivative of Go, thus supporting function closure, and simplified parallel execution semantics. Our first concrete specification were the lexical and syntax specifications, which we implemented in the lexer and parser, at the same time as when we wrote our language reference manual. The reference manual also specified other features of Funk, and in both cases, although each member was responsible for a particular section, other team members could suggest revisions, thus also revising our specifications. Once we turned in the LRM, we still changed specifications as the situation called for it, most notably in our reversal of the decision to allow global functions with closure, which made recursion difficult.

4.3 Development Process

Development followed the stages of the compiler architecture. In other words, we started with the lexer, then continued to the parser, then the semantics checker, then the code generator. Although we already had common interfaces for communication between each stage, we found it tedious, for example, to generate an SAST for the code generator by hand. We therefore decided to give higher priority and more time to finishing earlier stages first.

4.4 Testing Process

Before completion, we unit tested each stage, and tested the compilation process up to that stage. When we had a compilable end-to-end program, we designed tests for various features of the language, to check how they pass through the entire compiler into possibly-broken or buggy C code. If we were unsure about which stage caused the problem, we would consult the visual representation of the various intermediate data structures, to look for inconsistencies. Since these representations tend to grow, we were often forced to write smaller test programs to pinpoint the cause for the bug. More details on our testing plan is included in the test plan section.

4.5 Team Responsibilities

The team responsibilities were divided roughly across the five members as in the table below; however, there was no strict division of responsibilities as multiple members contributed to multiple parts, depending on the stage of the project.

Team Member	Responsibility
Naser AlDuaij	Compiler Front end, Test case creation
Senyao Du	Code generation, Compiler Front end
Noura Farra	Semantics, Documentation
Yuan Kang	Code generation, Semantics, C libraries
Andrea Lottarini	SAST visualization, Compiler Front end, Testing automation and vizualization

4.6 Project Timeline

The project timeline we aimed for is shown in the below table.

Date	Milestone
September 26	Language proposal and whitepaper complete
October 31	Language Reference Manual Complete
October 31	Compiler front end (lexer and parser) complete
December 1	Semantics / typechecking complete
December 10	Code generation complete
December 11	Hello World runs
December 15	Regression Testing , Debugging Complete
December 18	Final report complete

4.7 Project Log

Our project log was as follows:

Date	Milestone
September 20	Language defined
September 26	Language proposal and whitepaper complete
October 15	Agreement on main language features
October 20	Grammar defined
October 31	Language Reference Manual Complete
October 31	Compiler front end (lexer and parser) complete
November 3	Semantics and type checking initiated
November 10	End-end Hello World initiated
November 15	AST Printing (Dot) Complete
November 24	Testable components for code generation
December 12	'Noisy' Hello World
December 13	Semantics / typechecking complete
December 14	Code generation debuggable version
December 14	Hello World runs
December 15	Regression Testing , Debugging Complete
December 17	All modules complete
December 18	Final report complete

4.8 Software Development Environment

We had the following programming and devlopment environment:

- Programming language for building compiler : Ocaml version 4.00.1 . Ocamlyacc and Ocamllex extensions were used for compiling the scanner and parser front end.
- Development environments: Different members preferred to code in different environments including: Eclipse, emacs, vim, Komodo. Eclipse, and its plugins, turned out to be especially useful for debugging.

4.9 Programming Style Guide

We generally followed the following guidelines while programming our compiler:

- Formatting and indents: We generally followed the Ocaml editing and formatting style. We used one space for indentation in all files except Semantics.ml where one tab for indentation was used. Note that in Semantics.ml, the 'in' keyword was placed at the end of the line while in all other files it was atthe beginning of the line.
- Comments and documentation: We preceded each block of code by multiline Ocaml comments describing the code below.

5 Architectural Design

5.1 The Compiler

The architecture of the Funk translator consists of the following major components: Lexer, Parser, Semantic (Type) Checker, and Code Generator, shown in Figure 5.1 below. The lexer and parser constitute the front

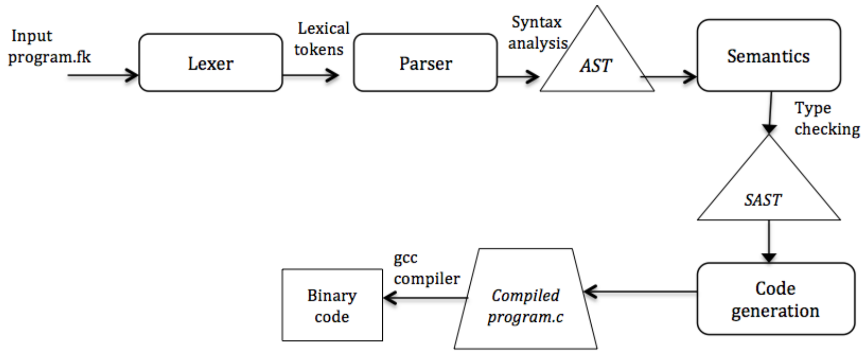

Figure 1: Architecture of Funk compiler

end of the compiler, and the semantic checker and code generator constitute the back end. We implemented all of these components in Ocaml.

The entry point of the compiler is fk2c.ml, which calls each of the above described components sequentially. First, the input Funk code is passed to scanner.ml and parser.ml, generating an AST structure which is passed to semantics.ml. The AST components are defined in commponents.mli and the extended AST components are defined in xcommponents.mli. In semantics.ml, variable declarations and function declarations, anonymous and global, are verified, matched against their declared type, execution scope and return type, and added to the SAST. After type checking, the semantically-checked SAST is passed to codegen.ml. The output of codegen.ml is the compiled C code which is further compiled to binary code using a C compiler, such as gcc or clang.

5.1.1 The Lexer (Naser, Peter, Andrea, Yuan)

The lexer takes as input a Funk source program and generates tokens for identifiers, keywords, operators, and values, as specified by the lexical conventions. Aside from that, its most complex task is processing escape sequences into single characters.

5.1.2 The Parser (Naser, Peter, Andrea, Yuan)

The parser takes in the generated tokens and generates an abstract syntax tree (AST) based on the Funk grammar. For most of the time, single tokens refer to single AST nodes, or none at all (in the case of punctuation). The exception is strings, which needs to be broken into the separate components of an array literal.

5.1.3 The Semantics Checker (Peter, Noura, Andrea, Yuan)

The semantics block recursively traverses the AST and converts it into an extended abstract syntax tree (SAST). The primary value of the SAST is that it keeps track of objects rather than string identifiers for variables, and free and declared variables. It does so using the running_env record, an aggregator argument that acts as a symbol table. This table maps a string identifier to an object stack, with the innermost declaration on top of the stack. In addition, we found it convenient to prune the SAST by folding constants, before passing it to the more mechanical code generator. More importantly, since most of the transformations were type dependent, it was easy to add type checking into this stage.

5.1.4 The Code Generator (Yuan, Naser)

The code generator generally traverses the tree to generate code in post-order fashion, with each node returning at least a variable for its handle, ie. a C primitive or var struct, and the code necessary to evaluate the value for the handle. In addition, depending on the free variables of an anonymous function, this stage also generates the code necessary to initialize a function instance before the program can run it.

5.2 The C Libraries (Yuan, Peter, Andrea)

It is infeasible to generate pure C code that could implement all the functional programming concepts. Instead, we added two additional C header files, with corresponding C code files, to support general features of the program. The two header files are *env.h* and *func.h*.

5.2.1 env.h

The environment header contains general structures for storing and accessing all types of Funk variables, including a stubs for function instances.

The main object is **struct var**, or the variable structure, which contains a union of the various types. The primitive values are directly stored, while function instances and arrays are referred to by pointers. The functions in the header help simply and safely read, write or copy primitive variables, and abstracts away the locking mechanism needed for reading the results of asynchronous blocks. Most of these functions are simple enough to implement inline. However, since C's pthread library requires a function pointer to execute on a thread, *env.c* contains the asynchronous procedure for copying the result of asynchronous blocks.

5.2.2 func.h

The object for representing functions is **struct function_instance**. Aside from a pointer to the function itself, it contains functions for initializing the instance's free variables, and copying a function's free variables to another instance. The instance also points to custom structures for the free variables and parameters, which are written during function definitions and function calls, respectively. Most functions are simply inline wrappers for calling the functions inside the function_instance struct, which may in turn be inside the var struct. However, asynchronous blocks were implemented as headerless functions, as they have the same closure requirements. To run these functions in C, we again needed a function pointer, this time in *func.c*, to run a function asynchronously.

6 Test Plan

All of our tests involved white, black, and grey box testing.

6.1 Testing phases

6.1.1 Unit testing (Lexer, parser, code generation)

We tested those components individually throughout the semester. For the lexer, we hand tested the different cases of input by injecting those cases in to the scanner and printing them at the parser level. The parser was tested by writing Funk code and then verifying it with our own version of printing the AST (dot.ml). The AST would then be verified to be the correct form of the tree. Code generation was hand tested with a basic hello world in addition to a fibonacci program.

6.1.2 Integration testing

Testing syntax and lexical conventions below individually. The purpose of the code is to sanity check the compiler with our language end-to-end (i.e. from lexer to code generation). The generated code is then compiled with gcc and run to verify the expected output and/or result.

- Identifiers

  Testing variable names or function names that start with a letter or underscore followed by alphanumeric or underscores. e.g. var i int, var _i9 int

- Keywords

  Keywords are a subset of identifiers which are reserved for our language. To verify, sufficient testing was conducted on cases such as var i int for a positive case or var if int for a negative case.

- Statements and Blocks

  These were tested standalone and nested in a function and then referenced in later code: var func async

- Control flow

  Control flow was unit tested in the following way: if, if/else, for (with and without loop condition expressions), break, and return. Negative cases were also tested, such as having an else by itself or an incorrect for loop

- Types

  We support four types int, double, bool, and char. Testing was done on those types including negative test cases such as assigning an int to a bool or a char to a double.

- Arrays

  We test the declaration and usage of arrays of different types to verify the correctness. We use printing to verify the result.

- Built-in functions

  The five built-in functions were tested (print, double2int, int2double, bool2int, int2bool)

- Constants/Literals

  Integer/double/boolean/characters/strings were tested individually and verified by using if statements or printing them to the screen. Escape sequences were tested

  `(\n, \r, \t, \a, \", \\)`

- Comments

  Comments were placed randomly in the code to ensure these were scanned correctly. We tested for both single and multi-line comments.

- Operators

  Binary, unary, and value binding (=) were tested by using them with a declare variable or two and printing the result to the screen to ensure their correctness

- Variable and function declarations

  This is similar to the Statement and Blocks section above. Here we implement global and nested variables and functions. We try different parameters for functions and different return types. In addition, we test multiple variable declarations on one line e.g. var a, b int = 0, 1

6.1.3 System testing

Hello world created and tested.

More meaningful tests were run here. A few selected algorithms were fed to our compiler (such as Fibonacci) and the output verified. Scoping rules and function closures were also tested. The test files are located in directories listed in the Test suite subsection below. An example of hello world and a closure "double" is also listed below in the Funk to C subsection.

6.2 Automation

Our compile script in the funklang directory takes in a directory name and compiles all the files (must have extension "fk") in that directory to C code that can be compiled with gcc and run.

6.3 Test suites

There are two folders with our tests. should_pass with test cases that should pass and should_fail with test cases that should fail. Testing specific for components can be found in /src/backend/compile_test/ for backend testing and /src/front_end/regression_test/ for frontend. These are manually run using the "main" binary produced by make.

We tested the following features of our language:

- Primitive data assignment and operations (should_pass/data.fk): We tested basic declaration, assignment and an arithmetic, binary operation. The test stressed the feature of Funk that allowed swapping of variables by assignment.
- Array assignment (should_pass/arr_copy.fk): The test program tried to read and assign elements of the array, including members that are arrays themselves, and checked for the deep copy feature.
- Control flow (should_pass/ctrlflow.fk): We checked if-else blocks whose bodies should or should not execute, and verified the number iterations of loops. We also tried stressing the language's ability to handle nested blocks.
- Global recursion (should_pass/recursion.fk): Global recursion revealed the difficulty of a function keeping a copy of itself. This led us to opt for stateless global functions, and test recursions of purely mathematical functions, such as factorials.
- Closure (should_pass/closure_single.fk): We checked that the anonymous function could be returned like any other value, and that executing it can change its state.
- Async (should_pass/async.fk): At first, when the code was less complete regarding recursion, we tried a simple test for an asynchronous execution. Once we had recursion, we tried implementing the Fibonacci recursion asynchronously, to see how the two features combined.

6.4 Funk to C

Note: complete.h is a file located in src/backend/c/include/ with built-in C functions

```
func main () {
	print("hello world!\n")
}
```

Listing 22: hello world in funk language

```c
#include <complete.h>
static struct function_instance main_global_instance;
struct var gv_0;
static void *main_global_function(void *data)
{
struct function_instance *self = (struct function_instance *)data;
struct var t_0;
init_var((&t_0));
init_array((&t_0), (13));
struct var t_1;
init_var((&t_1));
set_char((&t_1), 'h');
set_element((&t_0), 0, (&t_1));
struct var t_2;
init_var((&t_2));
set_char((&t_2), 'e');
set_element((&t_0), 1, (&t_2));
struct var t_3;
init_var((&t_3));
set_char((&t_3), 'l');
set_element((&t_0), 2, (&t_3));
struct var t_4;
init_var((&t_4));
set_char((&t_4), 'l');
set_element((&t_0), 3, (&t_4));
struct var t_5;
init_var((&t_5));
set_char((&t_5), 'o');
set_element((&t_0), 4, (&t_5));
struct var t_6;
init_var((&t_6));
set_char((&t_6), ' ');
set_element((&t_0), 5, (&t_6));
struct var t_7;
init_var((&t_7));
set_char((&t_7), 'w');
set_element((&t_0), 6, (&t_7));
struct var t_8;
init_var((&t_8));
set_char((&t_8), 'o');
set_element((&t_0), 7, (&t_8));
struct var t_9;
init_var((&t_9));
set_char((&t_9), 'r');
set_element((&t_0), 8, (&t_9));
struct var t_10;
init_var((&t_10));
set_char((&t_10), 'l');
set_element((&t_0), 9, (&t_10));
struct var t_11;
init_var((&t_11));
set_char((&t_11), 'd');
set_element((&t_0), 10, (&t_11));
```

```
struct var t_12;
init_var((&t_12));
set_char((&t_12), '!');
set_element((&t_0), 11, (&t_12));
struct var t_13;
init_var((&t_13));
init_var((&t_13));
set_char((&t_13), '\n');
set_element((&t_0), 12, (&t_13));
;
{
int arr_i;
for (arr_i = 0; arr_i < (&t_0)->arr_size; arr_i++) {
struct var *member_1 = (&(&t_0)->val.array[arr_i]);
PRINT_CHAR(member_1);

}

}

RETURN: return NULL;

}
static void main_global_copy(struct function_instance *dst, struct function_instance
                                   *src)
{

}
static void main_global_init(struct function_instance *new_inst)
{
new_inst->function = main_global_function;
new_inst->init = main_global_init;
new_inst->copy = main_global_copy;
init_var((&new_inst->ret_val));

}
int main(void)
{
init_var((&gv_0));
main_global_init((&main_global_instance));
gv_0.val.ptr = (&main_global_instance);
main_global_instance.function(&main_global_instance);
return 0;
}
```

Listing 23: funk hello world compiled to C

```
/* adapted from Go's website: http://golang.org/ */
func test() func() func() int {
    var parent int = 1
    return func() func() int {
        return func() int {
            var grandson int = parent + 1
            return grandson
        }
```

```
    }
}

func main() {
    print("About to print nested call\n")
    print(test()()(),"\n")
}
```

Listing 24: funk closure example

```
#include <complete.h>
struct test_globalanon1anon0_env{
    struct var v_0;
};
struct test_globalanon1_env{
    struct var v_0;
};
static struct function_instance test_global_instance;
static struct function_instance main_global_instance;
struct var gv_0;
struct var gv_1;
static void *test_globalanon1anon0_function(void *data)
{
struct function_instance *self = (struct function_instance *)data;
struct var bv_0;
init_var((&bv_0));
struct var t_0;
init_var((&t_0));
struct var t_2;
init_var((&t_2));
shallow_copy((&t_2), (&((struct test_globalanon1anon0_env *) self->scope)->v_0));
set_int((&t_0), (get_int((&t_2)))+(1));
shallow_copy((&bv_0), (&t_0));
struct var t_4;
init_var((&t_4));
shallow_copy((&t_4), (&bv_0));
shallow_copy((&self->ret_val), (&t_4));
;
goto RETURN;

RETURN: return NULL;

}
static void test_globalanon1anon0_copy(struct function_instance *dst, struct
                                    function_instance *src)
{
struct test_globalanon1anon0_env *src_env = (struct test_globalanon1anon0_env *)
                                    src->scope;
struct test_globalanon1anon0_env *dst_env = (struct test_globalanon1anon0_env *)
                                    dst->scope;
shallow_copy((&((struct test_globalanon1anon0_env *) dst_env)->v_0), (&((struct
                                    test_globalanon1anon0_env *) src_env)->v_0));

}
static void test_globalanon1anon0_init(struct function_instance *new_inst)
```

```
{
new_inst->function = test_globalanon1anon0_function;
new_inst->init = test_globalanon1anon0_init;
new_inst->copy = test_globalanon1anon0_copy;
struct test_globalanon1anon0_env *dst_env = malloc(sizeof(struct
                                    test_globalanon1anon0_env));
new_inst->scope = dst_env;
init_var((&((struct test_globalanon1anon0_env *) new_inst->scope)->v_0));
init_var((&new_inst->ret_val));
```

Listing 25: funk closure example compiled to C

```
/* fact is declared in global scope */
func fact (a int) int {
    if a == 1 {
            return a
    }else{
        /* fact can be used here */
        return a * fact(a - 1)
    }
}

func main(){
    /* outputs 120 */
    print(fact(5))
}
```

Listing 26: funk recursion example

```
#include <complete.h>
struct param_1{
    struct var p_0;
};
static struct function_instance fact_global_instance;
static struct function_instance main_global_instance;
struct var gv_0;
struct var gv_1;
static void *fact_global_function(void *data)
{
struct function_instance *self = (struct function_instance *)data;
struct param_1 *my_params = (struct param_1 *) (self->params);
struct var bv_0;
init_var((&bv_0));
shallow_copy((&bv_0), (&my_params->p_0));
struct var t_0;
init_var((&t_0));
struct var t_1;
init_var((&t_1));
struct var t_3;
init_var((&t_3));
shallow_copy((&t_3), (&bv_0));
set_int((&t_1), (get_int((&t_3)))==(1));
copy_primitive((&t_0), (&t_1));
if ((get_bool((&t_0)))){
struct var t_5;
```

```
init_var((&t_5));
shallow_copy((&t_5), (&bv_0));
shallow_copy((&self->ret_val), (&t_5));
;
goto RETURN;

}
else{
struct var t_6;
init_var((&t_6));
struct var t_8;
init_var((&t_8));
shallow_copy((&t_8), (&bv_0));
struct var t_15;
init_var((&t_15));
struct var t_9;
init_var((&t_9));
struct var t_11;
init_var((&t_11));
shallow_copy((&t_11), (&bv_0));
set_int((&t_9), (get_int((&t_11)))-(1));
;
struct function_instance fi_14;
(&gv_0)->val.ptr->init((&fi_14));
```

Listing 27: funk recursion example compiled to C

6.5 Testing Roles

Andrea created the testing infrastructure, including automation of regression tests, and visualization of the AST and SAST. Naser designed test cases, and reported bugs to the member responsible for the code (Peter, Yuan or Noura), who would in turn find and solve the reported error.

7 Lessons Learned

- Naser AlDuaij

  Communication is key to being organized for a term project such as funk. Utilizing available tools instead of creating our own or duplicating code is also very important. For example, using github for version control or graphiz for visualizing the AST/SAST was extremely essential. Creating test cases along the way instead of at the end of each phase was also an important lesson learned to save precious time. Meeting regularly to discuss issues or plans was paramount to the success of this team.

  Advice: Start early and work together, dividing up the work is not that easy and understanding someone else's (OCaml) code is even tougher. Keep testing along the way and try to get basic things working before complicating things.

- Senyao Du

  I feel OCaml is easy to use at first as a functional programming language, as it is very close the way we think about the problems. However, as the complexity of the project goes, it is increasingly harder for me to grasp the recent changes towards the way we generate SAST and subsequent code generation. It requires a consistent effort to parse and absorb both the code generation process as well the SAST, so that I could make some meaningful changes. One key aspect I have gained from this project is that the complexity of the project is tantamount to the effort of maintainability.

Advice: If you stick with OCaml, you might want to use better IDE for coding and debugging. Eclipse with OcaIDE[5] plugin is really useful for visualizing your code structure. It plays an important role for debugging and stepping through your code if necessary. And on top of that, learning the "evil" side of ocaml that includes reference and mutable field in records will save your precious time. And I would suggest you might want to take a look at various packages ocaml offers online as well.

- Noura Farra

As my first encounter with functional programming languages, OCaml was quite a challenge: requiring serious consideration of every line of code I write, very different than the imperative programming style I am used to. I saw the task of actually writing a compiler in Ocaml, and for a functional language nontheless , as an intimidating task. But I have seen how elegant the language can be, with easy type declarations , hash maps that are ideal for defining structures like symbol tables and variable maps, and match statements ideal for type checking.

As a relatively new student to CS coming from CE, this project was my first encounter not only with Ocaml and functional programming but also with the entire integrated project development experience: such as using a version control system in a unix environment, even latex editing. I also had to work quite a bit on getting up to speed on abstract progrmaming language topics needed to understand our language. The concept of function closures and how we are implementing them was not so easy to grasp.

Looking back to the beginning of the semester, I can safely say I've filled a gap between what I knew then and what I know now about writing functional language compilers using a functional language.

Advice: Communication is definitely essential in a team project such as this one. While I used to hesitate before admitting that I did not understand or agree with something, I learned that to learn and be effective, you have to ask questions and get clear on what is happening, especially when this is something very new to you : there is no shame in asking when you don't know.

- Yuan Kang

I learned several lessons involving both the technical aspects of writing a compiler in Ocaml, and how to effectively work in a team. Since our project required the use of a functional programming language, and our language derived some concepts from functional programming, the project has taught me both about how to use and how to implement a functional programming language. As somebody used to programming in C, I was pleasantly surprised to learn about the conveniences of functional programming, which let me avoid writing too many functions that did about the same thing. Specifically, I found that function nesting and currying made creating functions on the fly very convenient. Of course there were challenges: I learned to program without side effects, and also realized the need to enforce a consistent order of Ocaml lists, which would often be processed into a backward output. At the same time, implementing functional programming concepts forced me to explore different ways to keep track of nested scope, and simulate function instances with an imperative language.

Working in the team also taught me about what kind behavior is helpful or unhelpful to a team. For me, the most important lesson is that explaining to teammates the code you wrote is just as important as writing the code. When teammates understand your code, they can more easily work with it. And if you are stuck with a bug, they will be more able to lend

[5] http://www.algo-prog.info/ocaide/

a fresh pair of eyes. And while it goes without saying that a team can always start early, the team should also make sure to plan ealier, and plan to make changes. Even if you don't follow your plan exactly, the process reveals potential challenges ahead of time, and also builds concensus among the team members.

Advice: Aside from the interpersonal lessons, future teams could also benefit from an early familiary with Ocaml. While it is not necessary –or feasible –for students to study every new programming language ahead of time, I suggest that they treat the Ocaml lecture not just as a lecture, but as a guide for hands-on practice, tweaking the examples not only to gain familiarity with the language that runs the project, but also to gain an instinct for finding strange cases that could trip up their own language.

- Andrea Lottarini

  Working on a software project in a five people group is tough. The fact that the project is a compiler makes the process even more difficult as every design choice has many consequences throughout the whole development. Moreover, splitting the job in multiple task is difficult as every task is necessary to start working on subsequent ones. Therefore, every task inevitably require some synchronization between the team members.
  Everyone carries his set of different skills as well as inexperience, it is fundamental to exploit the former while trying to tackle the latter as soon as possible. Moreover, everyone has is own set of preferred development tools and practices; compromise is necessary here in order to have smooth software development. I have to admit that I kind of refused to learn git commands in the beginning. That lead me to continuously use google to get the equivalent of the svn commands that I knew. Obviously a not optimal approach. Learning new tools is generally good so it is important to give them a fair chance, and yes, this include OCaml.

  Advice: Meet a lot in the beginning of the semester when you have more time and make sure that everyone is on the same page. Try to learn each other skill as well as inexperience from the beginning instead when deadlines are approaching. Aim for a "simple" project that has plenty of opportunities for expansion. You are probably underestimating the difficulties of working in a group using a language presumably unknown by all the team members. Aiming for a big project with plenty of grey areas is a recipe for spending a lot of all nighters coding during finals' weeks.

8 Appendix

```
open Parser
open Printf
open Scanner
open Semantics
open Codegen
open Dot
open Xdot

let _ =
  let cin =
    if Array.length Sys.argv > 1
       then open_in Sys.argv.(1)
       else stdin
    in (* Let's make sure we can parse a file *)
  let lexbuf = Lexing.from_channel cin in
  let ast = Parser.program Scanner.free_form lexbuf in
  (*fprintf out "digraph g {"; *)
```

```
  Dot.print_program ast ;
  (*fprintf out "}" ; *)
  let sast, globals = Semantics.check_ast_type ast in
  (*fprintf out "digraph g {";*)
  Xdot.print_program sast ;
  (*fprintf out "}";*)
  let code = gen_prog sast globals
  in print_endline code
```

Listing 28: Funk to C main file

```
{
  open Parser
  let translate_escape = function
    | 'n' -> '\n'
    | 'r' -> '\r'
    | 't' -> '\t'
    | 'a' -> '\007'
    (*
     * the rest are printable characters that had special meaning, but
     * the backslash removes it
     *)
    | '\"' -> '\"'
    | '\\' -> '\\'
    | x -> raise (Failure("illegal escape sequence: \\" ^ String.make 1 x))
  let rec translate_escapes s =
    if (String.contains s '\\') then
      let esc_i = String.index s '\\' in
      let before = String.sub s 0 esc_i in
      let after_start = esc_i + 2 in
      let after = String.sub s after_start ((String.length s) - after_start) in
      let replaced = translate_escape(s.[esc_i + 1]) in
      String.concat "" [before ; String.make 1 replaced ;
            translate_escapes(after)]
    else
      s
}

let fp1 = ['0'-'9']* '.' ['0'-'9']+
let fp2 = ['0'-'9']+ '.' ['0'-'9']*
let exp = 'e' ['+' '-']? ['0'-'9']+

(* code reference from ocaml lex *)
(* http://caml.inria.fr/svn/ocaml/trunk/lex/lexer.mll *)

let back_slash = '\\'
let back_slash_escapes = back_slash ['n' 'r' 't' 'a' '"' '\\']
let string_char = ([^ '"'] | back_slash_escapes)*

rule free_form = parse
| [' ' '\t'] { free_form lexbuf }
| ['\n' '\r']+ { NEWLINE }
| "//" { linecomment lexbuf }
| "/*" { blockcomment lexbuf }
| "||" { OR }
```

```
| "&&" { AND }
| '!' { NOT }
| '+' { PLUS }
| '*' { MULT }
| '-' { MINUS }
| '/' { DIV }
| '=' { ASSIGN }
| '<' { LT }
| '>' { GT }
| "<=" { LTE }
| ">=" { GTE }
| "==" { EQ }
| "!=" { NE }
| ';' { SEMI }
| ',' { COMMA }
| '(' { LPAREN }
| ')' { RPAREN }
| '{' { LBRACE }
| '}' { RBRACE }
| '[' { LBRACKET }
| ']' { RBRACKET }
| '~' { BNOT }
| '&' { BAND }
| '|' { BOR }
| '%' { MOD }
| '^' { BXOR }
| "<<" { LSHIFT }
| ">>" { RSHIFT }
| "func" { FUNC }
| "var" { VAR }
| "async" { ASYNC }
| "bool" { BOOL }
| "double" { DOUBLE }
| "char" { CHAR }
| "int" { INT }
| "if" { IF }
| "else" { ELSE }
| "for" { FOR }
| "break" { BREAK }
| "return" { RETURN }
| "true" | "false" as tf { BOOL_LIT (bool_of_string tf) }
| fp1 as fp { DOUBLE_LIT (float_of_string fp) }
| fp2 as fp { DOUBLE_LIT (float_of_string fp) }
| fp1 exp as fp { DOUBLE_LIT (float_of_string fp) }
| fp2 exp as fp { DOUBLE_LIT (float_of_string fp) }
| ['0'-'9']+ exp as fp { DOUBLE_LIT (float_of_string fp) }
| ['0'-'9']+ as lxm { INT_LIT (int_of_string lxm) }
| ['a'-'z' 'A'-'Z' '_'] ['0'-'9' 'a'-'z' 'A'-'Z' '_']* as id { ID (id) }
| eof { EOF }
| '"' (string_char as content) '"' { STRING (translate_escapes content) }
(* escaped character *)
| '\'' back_slash (_ as esc_char) '\''
    { CHAR_LIT (translate_escape(esc_char)) }
(* unescaped single character *)
```

```
| '\'' (_ as char_match) '\'' { CHAR_LIT (char_match) }
| _ as char { raise (Failure("illegal character:[" ^ Char.escaped char ^ "]")) }

and linecomment = parse
| ['\r' '\n'] { NEWLINE }
| _ {linecomment lexbuf}

and blockcomment = parse
| "*/" { free_form lexbuf }
| _ {blockcomment lexbuf}
```

Listing 29: Lexer/Scanner

```
%{
    open Commponents
    let parse_error s = print_endline s

    type formal_cluster =
        | ImplicitParam of vartype
        | ExplicitParams of vartype * (string list)
    let formalize_cluster =
        let formalize_list_rev (running, vt) next =
            ({id = next; bare_type = Some(vt)}::running, vt)
        in let formalize_list vt id_list =
            List.rev
                (fst
                    (List.fold_left
                        formalize_list_rev
                        ([], vt)
                        id_list
                    )
                )
        in function
            | ImplicitParam(vt) ->
                [{id = ""; bare_type = Some(vt)}]
            | ExplicitParams(vt, id_list) ->
                formalize_list vt id_list
%}

%token OR AND NOT PLUS MULT MINUS DIV LT LTE GT GTE EQ NE ASSIGN
%token BNOT BAND BOR MOD BXOR LSHIFT RSHIFT
%token SEMI COMMA LPAREN RPAREN LBRACE RBRACE LBRACKET RBRACKET NEWLINE DOUBLE_QUOTE EOF
%token FUNC VAR ASYNC ARR BOOL DOUBLE INT CHAR IF ELSE NOELSE FOR BREAK RETURN
%token PRINTLN
%token <string> ESCAPESEQ
%token <string> ID
%token <string> STRING
%token <int> INT_LIT
%token <float> DOUBLE_LIT
%token <bool> BOOL_LIT
%token <char> CHAR_LIT

%start program
%type <Commponents.program> program

```

```
%%

program:
    | program_rev { List.rev $1 }

program_rev:
    | declaration { [$1] }
    | program_rev declaration { $2::$1 }

declaration:
    | funcdec { $1 }
    | vardec NEWLINE { Vardec($1) }
    | NEWLINE { Newline() }

block: LBRACE stmt_list_opt RBRACE { $2 }

stmt_list_opt:
    | /* empty */ { [] }
    | stmt_list { List.rev $1 }

stmt_list:
    | NEWLINE { [] }
    | stmt_list stmt NEWLINE { $2::$1 }
    | stmt_list NEWLINE { $1 }

funcdec: FUNC ID LPAREN formal_list_opt RPAREN opt_vartype block
    {  Funcdec{fid=$2; func_header = {ret_type=$6; params= $4} ; body = $7 } }

anon: FUNC LPAREN formal_list_opt RPAREN opt_vartype block {
    ({ret_type = $5; params = $3}, $6)
}

vardec:
    | VAR new_id_list var_type
    { {id_list=List.rev $2; var_type=$3; actual_list = [] } }

    | VAR new_id_list var_type ASSIGN actual_list
    { {id_list=List.rev $2; var_type=$3; actual_list = List.rev $5 } }

formal_list_opt:
    /* empty */ { [] }
    | formal_list { List.rev $1 }

formal_list:
    formal_type_cluster { formalize_cluster $1 }
    | formal_list COMMA formal_type_cluster { (formalize_cluster $3)@$1 }

formal_type_cluster:
    | var_type { ImplicitParam($1) }
    | new_id_list var_type { ExplicitParams($2, List.rev $1) }

new_id_list:
    | ID { [ $1 ] }
    | new_id_list COMMA ID { $3::$1 }
```

```

opt_vartype:
    /* empty */ { None }
    | var_type { Some($1) }

single_type:
    | FUNC LPAREN formal_list_opt RPAREN opt_vartype {
        FunkFunc({ret_type = $5; params = $3})
    }
    | INT {FunkInt}
    | DOUBLE {FunkDouble}
    | CHAR {FunkChar}
    | BOOL {FunkBool}

var_type:
    | opt_array_marker single_type { ($2, $1) }
arr_header:
    | array_marker single_type { ($2, $1) }

opt_array_marker:
    | /* empty */ { [] }
    | array_marker { List.rev $1 }

array_marker:
    | array_level { [$1] }
    | array_marker array_level { $2::$1 }

array_level:
    | LBRACKET RBRACKET { SingleConst(IntVal(-1)) }
    | LBRACKET expr RBRACKET { $2 }

/* Array value */
actual_arr:
    | arr_header LBRACE actual_list_opt RBRACE { ($1, $3) }
actual_list_opt:
    /* empty */ { [] }
    | actual_list { List.rev $1 }

/*
 * Any list of expressions. Could be paramater values (possibly of different
 * types) or array elements (must be of same type)
 */
actual_list:
    | expr { [ExprRVal($1)] }
    | anon { [FuncRVal($1)] }
    | actual_list COMMA expr { ExprRVal($3)::$1 }
    | actual_list COMMA anon { FuncRVal($3)::$1 }

obj_get_expr_list:
    obj_get_expr { [$1] }
    | obj_get_expr_list COMMA obj_get_expr { $3::$1 }

assign_stmt:
    | obj_get_expr_list ASSIGN actual_list { Assignment(List.rev $1, List.rev $3) }
```

```
assign_stmt_opt:
    /* empty */ { None }
    | assign_stmt { Some($1) } /* Will need to run regression on this */
stmt:
    | block { Block($1) }
    | func_call_expr { FunctionCall($1) }
    | RETURN expr { Return(Some(ExprRVal($2))) }
    | RETURN anon { Return(Some(FuncRVal($2))) }
    | BREAK { Break }
    | IF expr block { IfBlock(($2, $3)) }
    | IF expr block ELSE block { IfElseBlock(($2, $3, $5))}
    | FOR assign_stmt_opt SEMI expr_opt SEMI
      assign_stmt_opt block { ForBlock(($2, $4, $6, $7)) } //for loop
    | FOR expr block { WhileBlock(($2, $3)) } // while loop
    | FOR block { WhileBlock((SingleConst(BoolVal(true)), $2)) } //forever loop
    | assign_stmt { $1 }
    | vardec { Declaration($1) }

expr_opt:
    /* empty */ { None }
    | expr { Some($1) }
expr:
    | or_expr { $1 }
or_expr:
    | and_expr { $1 }
    | or_expr OR and_expr { FunkBinExpr($1, Or, $3) }
and_expr:
    | bor_expr { $1 }
    | and_expr AND bor_expr { FunkBinExpr($1, And, $3) }
bor_expr:
    | bxor_expr { $1 }
    | bor_expr BOR bxor_expr { FunkBinExpr($1, BOr, $3) }
bxor_expr:
    | band_expr { $1 }
    | bxor_expr BXOR band_expr { FunkBinExpr($1, BXor, $3) }
band_expr:
    | eq_expr { $1 }
    | band_expr BAND eq_expr { FunkBinExpr($1, BAnd, $3) }
eq_expr:
    | comp_expr { $1 }
    | eq_expr eq_op comp_expr { FunkBinExpr($1, $2, $3) }
eq_op:
    | EQ { Eq }
    | NE { NEq }
comp_expr:
    | shift_expr { $1 }
    | comp_expr comp_op shift_expr { FunkBinExpr($1, $2, $3) }
comp_op:
    | LT { LeT }
    | LTE { LE }
    | GT { GrT }
    | GTE { GE }
shift_expr:
```

```
    | add_expr { $1 }
    | shift_expr shift_op add_expr { FunkBinExpr($1, $2, $3) }
shift_op:
    | LSHIFT { LSh }
    | RSHIFT { RSh }
add_expr:
    | mult_expr { $1 }
    | add_expr add_op mult_expr { FunkBinExpr($1, $2, $3) }
add_op:
    | PLUS { Add }
    | MINUS { Sub }
mult_expr:
    | un_expr { $1 }
    | mult_expr mult_op un_expr { FunkBinExpr($1, $2, $3) }
mult_op:
    | MULT { Mult }
    | DIV { Div }
    | MOD { Mod }
un_expr:
    | post_expr { $1 }
    | un_op un_expr { FunkUnExpr($1, $2) }
un_op:
    | MINUS { IntNeg }
    | BNOT { BitNot }
    | NOT { Not }
    | PLUS { Positive }
post_expr:
    | obj_get_expr { $1 }
    | func_call_expr { FunkCallExpr($1) }
obj_get_expr:
    | primary_expr { $1 }
    | post_expr LBRACKET expr RBRACKET { FunkArrExpr($1, $3) }
func_call_expr:
    | post_expr LPAREN actual_list_opt RPAREN { ($1, $3) }
primary_expr:
    | INT_LIT { SingleConst(IntVal($1)) }
    | DOUBLE_LIT { SingleConst(DoubleVal($1)) }
    | CHAR_LIT { SingleConst(CharVal($1)) }
    | BOOL_LIT { SingleConst(BoolVal($1)) }
    | STRING {
       let rec listify i s =
           if i < String.length s then
              ExprRVal(SingleConst(CharVal(s.[i])))
                  ::(listify (i + 1) s)
           else [] in
              let listed = listify 0 $1 in
       ArrayLit((FunkChar,[SingleConst(IntVal(List.length
                                                            listed)
                                                    )
                                         ]
                                ), listed
                      )
    }
    | ID { Variable({id = $1; bare_type = None}) }
```

```
    | actual_arr { ArrayLit(fst $1, snd $1) }
    | ASYNC block { FunkAsyncExpr($2) }
    | LPAREN expr RPAREN { $2 }
```

Listing 30: Parser

```
(*
 * Common components between the front end and the back end.
 * These include the parts of the AST.
 *)

type unop = IntNeg | BitNot | Not | Positive

type binop = Mult | Div | Mod | Add | Sub | LSh | RSh | LeT | GrT | LE |
        GE | Eq | NEq | BAnd | BXor | BOr | And | Or

(* nameless header *)
type func_dec_header = { ret_type : vartype option ; params : var list }
(* variable types *)
and single_vartype =
  | FunkInt
  | FunkDouble
  | FunkChar
  | FunkBool
  (* function instance's value not calculated statically *)
  | FunkFunc of func_dec_header
and vartype =
  (*
   * type, size of each level. If integer list is empty, the type is just
   * single
   *)
  single_vartype * (expr list)
(* var is used only for the formal parameters of a function declaration
   notice how a var_dec has a list of id opposed to a single id
*)
and var = { id : string; bare_type : vartype option }
and single_funk_value =
  | IntVal of int
  (* this is what OCaml calls double, according to
   *http://blog.frama-c.com/index.php?post/2010/11/20/IEEE-754-single-precision-
   *numbers-in-Frama-C
   *)
  | DoubleVal of float
  | CharVal of char
  | BoolVal of bool
  (* function instance's value not calculated statically *)
  | FuncVal of func_dec_header * (statement list)
and func_call = expr * (rvalue list)

and expr =
  | SingleConst of single_funk_value
  | ArrayLit of vartype * rvalue list
  | Variable of var
  | FunkUnExpr of unop * expr
  | FunkBinExpr of expr * binop * expr
```

```
  | FunkCallExpr of func_call
  | FunkArrExpr of expr * expr
  | FunkAsyncExpr of (statement list) (* async block *)

and var_dec = { id_list: string list ; var_type: vartype ; actual_list: rvalue list}

(* notice this is the same as funcdec without a fid *)
and anon = func_dec_header * (statement list)

and rvalue =
  | ExprRVal of expr (* simple expression *)
  | FuncRVal of anon (* anonymous function declaration *)

and statement =
  | Assignment of (expr list) * (rvalue list)
  | Declaration of var_dec
  (* the state of the function may change *)
  | FunctionCall of func_call
  | Block of statement list
  | ForBlock of (statement option) * (expr option) * (statement option) *
    (statement list)
  | IfBlock of expr * (statement list)
  | IfElseBlock of expr * (statement list) * (statement list)
  | WhileBlock of expr * (statement list)
  | Break
  | Return of rvalue option



type funcdec = { fid: string; func_header: func_dec_header ; body: statement list}

type declaration =
  | Vardec of var_dec
  | Funcdec of funcdec
  | Newline of unit

type program = declaration list
```

Listing 31: commponents (Parser's frontend to backend)

```
open Commponents

(*
 * where variable is declared
 *)
type setting = {
  own_func : string ; (* containing function or asynchronous block *)
  (*
   * ID of immediately containing block. Unique in this function, and generated
   * like in dot.ml, ie. pre-order.
   *)
  own_block : int ;
}
(*
 * object pointed to by identifier
```

```
 *)
type id_object = {
   name : string;
  obj_type : xvartype ; (* type of value the variable points to *)
  scope_setting : setting ; (* where was the function declared? *)
  (*
   * if this variable needs to be copied from a higher scope for closure,
   * then this is the free variable id that is unique in this function or
   * async block. Otherwise, it is the function-unique bound variable id.
   * the free id and bound variable id follow the same rules, but are in
   * distinct namespaces, eg. we can only have one free variable of id
   * 100 in a function, but we can also have a bound variable of id 100 in
   * that function, as long as there are no other bound variables of id 100
   * in that function.
   *)
  var_id : int ;
  is_free : bool
}
and xfunc_dec_header = { xret_type : xvartype option ; xparams : xvar list }
and xsingle_vartype =
  | XFunkInt
  | XFunkDouble
  | XFunkChar
  | XFunkBool
  (* function instance's value not calculated statically *)
  | XFunkFunc of xfunc_dec_header
and xvartype =
  (*
   * type, size of each level. If integer list is empty, the type is just
   * single
   *)
  xsingle_vartype * (xexpr list)
and xvar = { xbare_type : xvartype }
(* generalize value, which could be known in an expression *)
and funk_value =
  | SingleValue of single_funk_value
  | ArrayValue of xvartype * xrvalue list
(* declared variables already known, so only handle assignments *)
and xassignment = { lvals: xexpr list ; rvals: xrvalue list}
and called_func =
  | PrintCall
  | Double2IntCall
  | Int2DoubleCall
  | Boolean2IntCall
  | Int2BooleanCall
  | GenericCall of xexpr

(*
 * extended version of func_call just uses extended versions of
 * func_call's members
 *)
and xfunc_call = xfunc_dec_header * called_func * (xrvalue list)
(*
 * with some exceptions, the only change is the addition of the xvartype
```

```
 * argument, and the use of the extended types of the original arguments,
 * eg. rvalue -> xrvalue
 *)
and xexpr =
  | XSingleConst of xsingle_vartype * single_funk_value
  | XArrayLit of xvartype * xrvalue list
  | XVariable of id_object (* point to object, rather than ID *)
  | XFunkUnExpr of xvartype * unop * xexpr
  | XFunkBinExpr of xvartype * xexpr * binop * xexpr
  | XFunkCallExpr of xvartype * xfunc_call
  | XFunkArrExpr of xvartype * xexpr * xexpr (* array access: array and index *)
  | XFunkAsyncExpr of xvartype * xblock (* return type, async block *)
and xanon = xfunc_dec_header * xblock
and xrvalue =
  | XExprRVal of xvartype * xexpr (* simple expression *)
  | XFuncRVal of xanon (* anonymous function declaration *)
and xstatement =
  | XAssignment of xassignment
  | XFunctionCall of xfunc_call
  | XBlock of xblock
  | XForBlock of (xstatement option) * (xexpr option) * (xstatement option) *
    xblock
  | XIfBlock of xexpr * xblock
  | XIfElseBlock of xexpr * xblock * xblock
  | XWhileBlock of xexpr * xblock
  | XBreak
  | XReturn of xrvalue option
(*
 * unlike the AST block, the SAST block needs to know about the variables
 * that need to be declared in it, and those it has declared
 *)
and xblock = { xstmts : xstatement list ; (* statements in code *)
              (*
               * free variable that needs a copy, paired with the higher-scope
               * variable that needs to be copied
               *)
              need_copy : (id_object * id_object) list ;
              (* variables it has declared, including copies *)
              declared : id_object list
            }

(* same as funcdec, except xbody contains xstatements *)
type xfuncdec = { global_id : id_object ; xfid : string ;
         xfunc_header: xfunc_dec_header ;
                xbody : xblock }

type xdeclaration =
  (*
   * we already know which variables are being declared, so code generation
   * only needs to know about assignment declarations
   *)
  | XVardec of xassignment
  | XFuncdec of xfuncdec

```

```
(* exactly analogous to xprogram *)
type xprogram = xdeclaration list

(*
 * result of eval on expression, including a possibly pruned expression AST
 * node, a value, if it can be statically determined, and the type
 *)
type evalue = { e_node : xexpr ; result : funk_value option ;
                e_type : xvartype }

val check_types : Commponents.program -> bool
```

Listing 32: xcommponents (Code generation's frontend to backend)

```
open Commponents
open Xcommponents
open Printf

(* maps variable name to stack of objects *)
module VarMap = Map.Make(struct
       type t = string
       let compare x y = Pervasives.compare x y
   end)

(* set of variables declared in scope *)
module VarSet = Set.Make(struct
       type t = string
       let compare x y = Pervasives.compare x y
   end)

(*
* Running scope environment, including variable-object stack map,
* variables declared in the block, and current block
*)
type running_env = {
    scope : id_object list VarMap.t ; (* maps name to id_object stack *)
    (*
    * any newly-declared variables in this block that will need to be popped
    * from scope after this block is completed
    *)
    new_objects : VarSet.t ;
    (* function-wide list version of new_objects *)
    new_objects_ordered : id_object list ;
    mutable free_objects_ordered: (id_object* id_object) list ;
    current_block : setting ; (* new variables will use this setting *)
    (*
    * next free id_object will have this value as var_id.
    * unique inside the function or async block, so this value is not passed
    * outside of a function, after it is checked.
    *)
    next_free : int ;
    (*
    * next bound id_object will have this value as var_id.
    * unique inside the function or async block, so this value is not passed
    * outside of a function, after it is checked.
```

```
    *)
    next_bound : int ;
    (* function-wide fields for keeping track of return type *)
    (* has a return type been found? If not, then any new return value is OK *)
    has_ret : bool ;
    (* Return type. If it's None, either that means that all return statements don't
                                      return anything, or has_ret = false *)
    r_ret_type : xvartype option
}

let filter_out_block new_env old_env =
    { scope = old_env.scope ; new_objects = old_env.new_objects ;
      new_objects_ordered = new_env.new_objects_ordered ;
      free_objects_ordered = new_env.free_objects_ordered ;
      current_block = old_env.current_block;
      next_free = new_env.next_free ; next_bound = new_env.next_bound ;
      has_ret = new_env.has_ret ; r_ret_type = new_env.r_ret_type }
let filter_in_function old_env name_opt =
    let name = match name_opt
        with Some(name) -> name
        | None -> old_env.current_block.own_func ^ "_"
    in { scope = old_env.scope; new_objects = VarSet.empty;
    current_block = { own_func = name ;
              own_block = 0 };
    next_free = 0; next_bound = 0 ;
    new_objects_ordered = [] ; has_ret = false ; free_objects_ordered = [];
    r_ret_type = None }

(* debugging support *)
let debug_setting setting =
        eprintf "setting %s,%d\n" setting.own_func setting.own_block

let debug_single_vartype typ =
  eprintf "#SINGLE_VARTYPE\n";
  match typ with
  | XFunkInt -> eprintf "int"

  | XFunkChar -> eprintf "char"

  | XFunkDouble -> eprintf "double"

  | XFunkBool -> eprintf "bool"

  | XFunkFunc x -> eprintf "function header"

let debug_vartype var=
  eprintf "#VARTYPE: ";
    debug_single_vartype (fst(var))
  (*print_expr_list next (next+1) (snd(var))
  List.iter eprint "expr" (snd(var))
*)

let debug_idobject id =
        eprintf "id_object: %s\n" id.name;
```

```
        debug_setting id.scope_setting

let debug_idobject_free id =
  eprintf("\027[36;40m[\027[0m");
  debug_idobject(fst(id));
  debug_idobject(snd(id));
  eprintf("\027[36;40m]\027[0m\n")

let debug_env env =
    eprintf "DEBUGGING ENV\n";
    eprintf "VarMap\n";
    VarMap.iter (fun k v -> eprintf "\tvariable_name: %s\n" k) env.scope;
    eprintf "VarSet\n";
    VarSet.iter (fun k -> eprintf "\tvariable_name: %s\n" k) env.new_objects;
    eprintf "Free Vars\n";
    List.iter debug_idobject_free env.free_objects_ordered;
    eprintf "Binded Vars\n";
    List.iter debug_idobject env.new_objects_ordered

let debug_xblock xblock =
    eprintf("Declared variables:\n");
    List.iter debug_idobject xblock.declared;
    eprintf("Need to copy variables:\n");
    List.iter (fun (x,y) -> debug_idobject x; debug_idobject y; eprintf("\n"))
                                         xblock.need_copy

(* end of debugging support *)

(*
 * polymorphic list checker, iterate through a list and returns a list
 * of corresponding xtypes and the new_env
 *)
let list_checker checker env values =
    let env, checked_values = List.fold_left
        (fun (old_env, old_list) value ->
         let new_env, new_checked = checker old_env value
         in new_env, new_checked::old_list) (env, [])
        values
    in env, (List.rev checked_values)

let global_scope = "global"

let new_r_env = { scope = VarMap.empty ; new_objects = VarSet.empty ;
        new_objects_ordered = [] ;
        current_block = {own_func = global_scope ; own_block = 0} ;
        next_free = 0 ; next_bound = 0 ; has_ret = false ; r_ret_type = None;
        free_objects_ordered = []}


(* helper functions for extracting single_vartype values *)
let get_single = function
    | SingleValue(sv) -> sv
    | ArrayValue(_) -> raise (Failure "Expected single value")
let get_int = function
```

```
    | IntVal(i_v) -> i_v
    | _ -> raise (Failure "Expected int for get_int")
let get_double = function
    | DoubleVal(d_v) -> d_v
    | _ -> raise (Failure "Expected double for get_double")
let get_bool = function
    | BoolVal(b_v) -> b_v
    | _ -> raise (Failure "Expected bool for get_bool")
let get_char = function
    | CharVal(c_v) -> c_v
    | _ -> raise (Failure "Expected char for get_char")

(* turns single_vartype into vartype *)
let generalize_single st = (st, [])

(* check for duplicate variable in the current scope-block *)

let is_not_duplicate env a =
    if VarMap.mem a env.scope
    then let obj = List.hd (VarMap.find a env.scope) in
    (*need to check if it's in a block at the same depth*)
    if obj.scope_setting = env.current_block
    then false
    else true
    else
        true


let check_duplicate env a =
    if is_not_duplicate env a
    then
        true
    else
        raise (Failure ("Variable already declared:" ^ a))

(* must update environment and return y *)
let mapAdd key xtype is_free env =
    (* get list of variables with name key *)
    let old_list = if VarMap.mem key env.scope
        then VarMap.find key env.scope
        else []
        in
    (*check that is not a redeclaration*)
    let _ = check_duplicate env key
    (* assign a unique id which depends on whether this variable is free or not *)
    in let var_id, next_bound, next_free = if is_free
            then env.next_free, env.next_bound, env.next_free + 1
            else env.next_bound, env.next_bound + 1, env.next_free
    (* create a new id_object using the newly created uid *)
    in let new_obj = { name = key; obj_type = xtype ; scope_setting = env.current_block ;
        var_id = var_id ; is_free = is_free }
    (* create new binded variables list *)
    in let new_obj_list = if is_free then env.new_objects_ordered
        else new_obj::env.new_objects_ordered
```

```
    (* create new free variables list *)
    in let new_free_list = if is_free then
       if old_list = []
       then raise(Failure ("Variable " ^ key ^ " has never been declared before"))
       else let old_var = List.hd old_list in
       (new_obj,old_var)::env.free_objects_ordered
       else env.free_objects_ordered
    (* modify the scope *)
    in let new_scope = VarMap.add key (new_obj::old_list) env.scope
    (* return the moloch *)
    in { current_block = env.current_block ; scope = new_scope ;
       next_free = next_free ; next_bound = next_bound ; new_objects =
          VarSet.add key env.new_objects ;
       new_objects_ordered = new_obj_list ; has_ret = env.has_ret ;
       r_ret_type = env.r_ret_type; free_objects_ordered = new_free_list
    }, new_obj


(*
* finds member type of an array
* --simply deletes first level
* (vartype) arr_type: type with at least one entry in sizes list
* returns type that the array stores, which is one level lower
*)
let find_lower arr_type =
    let size_list = snd arr_type
    in if (List.length size_list > 0)
       then fst arr_type, List.tl (snd arr_type)
       else raise (Failure "Array has 0 levels")
(*
* finds member xcommponents type of an array
* --simply deletes first level
* (xvartype) arr_type: type with at least one entry in sizes list
* returns type that the array stores, which is one level lower
*)
let find_lowerx arr_type =
    let size_list = snd arr_type
    in if (List.length size_list > 0)
       then fst arr_type, List.tl (snd arr_type)
       else raise (Failure "Array has 0 levels")
(*
* matches any two vartype objects. Note that in this implementation,
* general and special are effectively interchangeable
* (xvartype) general: the required type, as is specified in the formal
* parameters list, lvalue, or array type
* (xvartype) special: the type of the given value, as specified in the
* rvalue expression
* returns true if special matches general
*)
let rec match_type general special =
    if List.length (snd general) != List.length (snd special) then false
    else match fst general, fst special
    with XFunkFunc(g_f), XFunkFunc(s_f) -> match_func_type g_f s_f
    | g_t, s_t -> g_t == s_t
```

```
(*
* matches function types according to their headers.
* Note, again, that switching general and special is alright for now
* (xfunc_dec_header) general: the required function header
* (xfunc_dec_header) special: given function header
* returns true if both return type and formal parameters match
*)
and match_func_type general special =
    (* iteratively matches parameter types *)
    let rec match_param_types = function
        | [], [] -> true
        | g_v::g_tl, s_v::s_tl ->
            let g_vt, s_vt = g_v.xbare_type, s_v.xbare_type
            in match_type g_vt s_vt &&
                (match_param_types (g_tl, s_tl))
        | _, _ -> false
    in
    (* matches optional return type *)
    let match_ret_type = function
        | None, None -> true
        | Some(g_rv), Some(s_rv) -> match_type g_rv s_rv
        | _, _ -> false
    in (match_ret_type (general.xret_type, special.xret_type)) &&
        (match_param_types (general.xparams, special.xparams))
(*
* Converts vartype into xvartype, calling eval on any size expressions
* (vartype) vt: the value to convert
* return xvartype form of vt
*)
let rec extend_vartype vt r_env =
    let extend_sizes sizes r_env =
        let xsizes, r_env = List.fold_left
                (fun (old_xs, old_r_env) size ->
                    let r_env, ev = eval old_r_env size
                    in (ev.e_node::old_xs, r_env)
                )
                ([], r_env) sizes
        in (List.rev xsizes), r_env
    in let extend_single r_env = function
        | FunkInt -> XFunkInt, r_env
        | FunkDouble -> XFunkDouble, r_env
        | FunkChar -> XFunkChar, r_env
        | FunkBool -> XFunkBool, r_env
        | FunkFunc(hdr) -> let xhdr, r_env = extend_func_header hdr
                            r_env
        in XFunkFunc(xhdr), r_env
    in let xsingle, r_env = extend_single r_env (fst vt)
    in let xsizes, r_env = extend_sizes (snd vt) r_env
    in (xsingle, xsizes), r_env


(*
* Checks expression types by traversing its tree
* (running_env) r_env: running environment of variables, as passed from
```

```
*                      a higher level
* (expr) expr_nd: AST-node representation of expression
* [? pending decision on whether eval should also generate code] (boolean) wait:
                                      Determines if evaluation should wait on an async value.
* returns the updated r_env and the evalue that stores the result of
*         the check
*)
and eval r_env expr_nd =
    (*
    * helper function for checking that unary and binary operations are
    * applied to single_vartype
    *)
    let narrow_single et = match snd et
    with [] -> fst et
        | levels -> raise (Failure
                        "Math and logic operations cannot be applied to arrays")
    in match expr_nd
    (* single literal is constant, and always known *)
    with SingleConst(c) ->
        (* extract type from single_vartype *)
        let find_single_type = function
            | IntVal(_) -> FunkInt
            | DoubleVal(_) -> FunkDouble
            | CharVal(_) -> FunkChar
            | BoolVal(_) -> FunkBool
            (*
            * sanity check on compiler: should only have above tokens for constant
            *)
            | _ -> raise (Failure "Improper constant")
        in let st = generalize_single (find_single_type c)
        in let xct, r_env = extend_vartype st r_env
        in let const_xval = XSingleConst(fst xct, c)
        in r_env, { e_node = const_xval; result = Some(SingleValue(c)) ;
            e_type = xct }
    (* Variable value unknown, but can extract type from last declaration *)
    | Variable(v) ->
            let find_var_obj fv r_env=
                let name = fv.id in
                if VarMap.mem name r_env.scope
                then
                    let obj = List.hd (VarMap.find name r_env.scope) in
                    if obj.scope_setting = r_env.current_block then
                        obj, r_env
                    else
                        if obj.scope_setting.own_func = r_env.current_block.own_func ||
                            obj.scope_setting.own_func = global_scope then
                            obj, r_env
                        else
                            let r_env, free_copy = mapAdd name obj.obj_type true r_env
                            in
                            free_copy, r_env
                else raise (Failure ("Variable " ^ name ^ " has not been declared"))
            in let v_o, r_env = find_var_obj v r_env in
            r_env, { e_node = XVariable(v_o); result = None ; e_type = v_o.obj_type }
```

```
(*
* we know array value if all members are constant.
* but first, we need to eval all the members
*)
| ArrayLit(arr_type, elements) ->
    let member_type = find_lower (arr_type) in
    (*
  * evaluates each member, returning the new env_r, filtered rvalue, and
  * if it's constant. The expression or function type is also checked
  * against member_type, the second part of the second argument tuple,
  * but is not returned
  *)
    let eval_member r_env = function
      | ExprRVal(e), mt -> let r_env, expr_member = eval r_env e
      in let xmt, r_env = extend_vartype mt r_env
      in let _ = if not (match_type xmt expr_member.e_type)
        then raise (Failure "expression in array gives wrong type")
      in
      (r_env, XExprRVal(expr_member.e_type, expr_member.e_node),
      expr_member.result != None)
      | FuncRVal(af), (FunkFunc(mtf), []) -> let r_env, fheader, fbody, _ =
          (check_func r_env (fst(af)) (snd(af)) None)
      in let xmtf, r_env = extend_func_header mtf r_env
      in let matched = match_func_type xmtf fheader
      in let _ = if not matched
        then raise (Failure "Function in array has wrong type")
      in (r_env, XFuncRVal(fheader, fbody), false)
      | _, _ -> raise (Failure "Wrong array member")
    (* cumulatively applies eval_member to all members *)
    in let eval_members es = List.fold_left
        (fun (c_env, known, rlist) next ->
          let (n_env, rval, n_known) =
            eval_member c_env
              (next, member_type)
          in
          (n_env, known & n_known, rval::rlist)
        )
        (r_env, true, []) es
    in let r_env, known, rlist = eval_members elements
    in let evaled_list = List.rev rlist
    in let xarr_type, r_env = extend_vartype arr_type r_env
    in let xarr_type = if (List.length evaled_list > 0)
      then let highest = List.hd (snd xarr_type)
      in let have_size = match highest
      with XSingleConst(_, sv) -> (match sv
      with IntVal(si) -> (si > 0)
      | _ -> true
      )
        | _ -> true
      in if have_size then xarr_type
      else let base = fst xarr_type
      in let remainder = List.tl (snd xarr_type)
      in let replacement = (XSingleConst(XFunkInt,
          IntVal(List.length evaled_list)
```

```
                    )
                )
                in (base, replacement::remainder)
                else xarr_type
            in let tentative = ArrayValue(xarr_type, evaled_list)
            (*
        * "value" is necessary for ArrayLit expression, and if it's fully
        * known, it is put into result
        *)
            in let known_value = if known then Some(tentative) else None
            in r_env, { e_node = XArrayLit(xarr_type, evaled_list) ;
                result = known_value ; e_type = xarr_type }
    (*
    * Prefix unary operator can only work on correct type. If the value of
    * the expression to operate on is known, it is straightforward to calculate
    * the result of the unary expression
    *)
    | FunkUnExpr(op, e) ->
            let r_env, evaled = eval r_env e
            in let unop_func = match narrow_single evaled.e_type
            with XFunkInt ->
                let do_unop_int op_f = fun oprnd -> IntVal(op_f (get_int oprnd))
                in (match op
                with IntNeg -> do_unop_int (fun x -> -x)
                | Positive -> do_unop_int (fun x -> x)
                | BitNot -> do_unop_int (fun x -> lnot x)
                | _ -> raise (Failure ("Integers can only have unary operators \'-\',
                                    \'+\' and \'~\'"))
                )
                | XFunkDouble ->
                        let do_unop_double op_f = fun oprnd -> DoubleVal(op_f
                                    (get_double oprnd))
                        in (match op
                        with IntNeg -> do_unop_double (fun x -> -. x)
                        | Positive -> do_unop_double (fun x -> x)
                        | _ -> raise (Failure ("Doubles can only have unary
                                            operators\'-\', and \'+\'"))
                        )
                | XFunkBool ->
                        let do_unop_bool op_f = fun oprnd -> BoolVal(op_f (get_bool oprnd))
                        in (match op
                        with Not -> do_unop_bool (fun x -> not x)
                        | _ -> raise (Failure "Booleans can only have unary operators
                                            \'!\'")
                        )
                | _ -> raise (Failure "Invalid type for unary operator")
            in let conv_eval = match evaled.result
            (* LOOOK HERE THIS IS AN EXAMPLE OF HOW YOU INIT EVALUE *)
            with None -> { e_node = XFunkUnExpr(evaled.e_type, op, evaled.e_node);
                result = None ; e_type = evaled.e_type }
                | Some(rv) -> match rv
                with ArrayValue(_) -> raise (Failure
                            "Cannot evaluate array with unary operator")
                | SingleValue(sv) -> let const_result = unop_func(sv)
```

```
            in { e_node = XSingleConst(fst evaled.e_type, const_result) ;
                 result = Some(SingleValue(const_result)) ;
                 e_type = evaled.e_type }
        in r_env, conv_eval
    (*
    * Binary operators can only work on certain pairs of operands.
    * If both operands have known values, calculating the result is possible
    * here. And certain operations can take advantage of the value of only 1
    * operand
    *)
    | FunkBinExpr(e1, op, e2) ->
        let r_env, evaled1 = eval r_env e1
        in let r_env, evaled2 = eval r_env e2
        in let type1 = narrow_single evaled1.e_type
        in let type2 = narrow_single evaled2.e_type
        in let type12 = if type1 == type2 then type1
            else raise (Failure "Binary operands don't match.")
        in let type_out = if (op = LeT) || (op = GrT) || (op = LE) ||
                                            (op = GE) || (op = Eq) || (op = NEq)
            then XFunkBool else type12
        in let st = generalize_single(type12)
        in let st_out = generalize_single(type_out)
        in let binop_func = match type12
        with XFunkInt ->
            let do_binop_int op_f = fun oprnd1 oprnd2 ->
                IntVal(op_f (get_int oprnd1) (get_int oprnd2))
            in (match op
            with Mult -> do_binop_int (fun a b -> a * b)
            | Div -> do_binop_int (fun a b -> a / b)
            | Mod -> do_binop_int (fun a b -> a mod b)
            | Add -> do_binop_int (fun a b -> a + b)
            | Sub -> do_binop_int (fun a b -> a - b)
            | LSh -> do_binop_int (fun a b -> if b >= 0 then a lsl b
                        else a lsr -b)
            | RSh -> do_binop_int (fun a b -> if b >= 0 then a lsr b
                        else a lsl -b)
            | LeT -> (fun a b -> BoolVal((get_int a) < (get_int b)))
            | GrT -> (fun a b -> BoolVal((get_int a) > (get_int b)))
            | LE -> (fun a b -> BoolVal((get_int a) <= (get_int b)))
            | GE -> (fun a b -> BoolVal((get_int a) >= (get_int b)))
            | Eq -> (fun a b -> BoolVal((get_int a) = (get_int b)))
            | NEq -> (fun a b -> BoolVal((get_int a) != (get_int b)))
            | BAnd -> do_binop_int (fun a b -> a land b)
            | BXor -> do_binop_int (fun a b -> a lxor b)
            | BOr -> do_binop_int (fun a b -> a lor b)
            | _ -> raise (Failure ("Invalid binary operator for integers"))
            )
            | XFunkDouble ->
                    let do_binop_double op_f = fun oprnd1 oprnd2 ->
                        DoubleVal(op_f (get_double oprnd1)
                                (get_double oprnd2))
                    in (match op
                    with Mult -> do_binop_double (fun a b -> a *. b)
                    | Div -> do_binop_double (fun a b -> a /. b)
```

```
                        | Add -> do_binop_double (fun a b -> a +. b)
                        | Sub -> do_binop_double (fun a b -> a -. b)
                        | LeT -> (fun a b -> BoolVal((get_double a) < (get_double b)))
                        | GrT -> (fun a b -> BoolVal((get_double a) > (get_double b)))
                        | LE -> (fun a b -> BoolVal((get_double a) <= (get_double b)))
                        | GE -> (fun a b -> BoolVal((get_double a) >= (get_double b)))
                        | Eq -> (fun a b -> BoolVal((get_double a) = (get_double b)))
                        | NEq -> (fun a b -> BoolVal((get_double a) != (get_double b)))
                        | _ -> raise (Failure ("Invalid binary operator for doubles"))
                        )
                | XFunkBool ->
                        (match op
                        with Eq -> (fun a b -> BoolVal((get_bool a) = (get_bool b)))
                        | NEq -> (fun a b -> BoolVal((get_bool a) != (get_bool b)))
                        | And -> (fun a b -> BoolVal((get_bool a) && (get_bool b)))
                        | Or -> (fun a b -> BoolVal((get_bool a) || (get_bool b)))
                        | _ -> raise (Failure ("Invalid binary operator for booleans"))
                        )
                | XFunkChar ->
                        (match op
                        with Eq -> (fun a b -> BoolVal((get_char a) = (get_char b)))
                        | NEq -> (fun a b -> BoolVal((get_char a) != (get_char b)))
                        | _ -> raise (Failure ("Invalid binary operator for chars"))
                        )
                | _ -> raise (Failure "Invalid type for unary operator")
            in let conv_eval = match evaled1.result, evaled2.result
            with Some(r1), Some(r2) ->
                let const_result =
                    let v1 = get_single(r1)
                    in let v2 = get_single(r2)
                    in (binop_func v1 v2)
                in { e_node = XSingleConst(type_out, const_result) ;
                    result = Some(SingleValue(const_result)) ;
                    e_type = st_out }
                (*
            * some boolean operations can be calculated with only one operand,
            * others, we can find out the result, but we should evaluate them
            * anyways for potential side effects
            *)
                | Some(r1), None ->
                        let v1 = get_single r1
                        in let first_result = match op
                        with And -> (match v1
                        with BoolVal(false) -> Some(SingleValue(BoolVal(false)))
                        | _ -> None
                        )
                            | Or -> (match v1
                            with BoolVal(true) -> Some(SingleValue(BoolVal(true)))
                            | _ -> None
                            )
                            | _ -> None
                        in { e_node = XFunkBinExpr(st, evaled1.e_node, op, evaled2.e_node);
                            result = first_result ; e_type = st_out }
                | _, _ ->
```

```
                        { e_node =
                                XFunkBinExpr(st, evaled1.e_node, op, evaled2.e_node);
                            result = None ; e_type = st_out }
            in r_env, conv_eval
    (*
     * Check that function executes to return correct type, and was given
     * correct parameters
     *)
    | FunkCallExpr(fc) ->
        let xfc, r_env = check_func_call fc r_env
        in let ret_type =
            let header, _, _ = xfc
            in match header.xret_type
            with None -> raise (Failure "Function call expression has to return a value")
            | Some(rtype) -> rtype
        in r_env, { e_node = XFunkCallExpr(ret_type , xfc) ; result = None ; e_type =
                                        ret_type}
    (*
     * Check that an integer index is given. Have constant result if both
     * tuple members are constant
     *)
    | FunkArrExpr(array_expr, index_expr) ->
        (* evaluate both the array and the indexes *)
        let r_env,array_evalue = eval r_env array_expr in
        let r_env,index_evalue = eval r_env index_expr in
        let _ =
            let index_type = index_evalue.e_type in
            match index_type
            with XFunkInt, [] -> true
            | _ -> raise (Failure "Array index must be an integer.")
        in let xvartype = find_lowerx array_evalue.e_type in
        let xarrexpr = XFunkArrExpr(xvartype, array_evalue.e_node , index_evalue.e_node)
                                        in
        (* running_env evalue *)
        r_env, { e_node = xarrexpr ; result = None ; e_type = xvartype }
    (*
     * Similar to FunkCallExpr, but parameters do not need to be checked, but
     * entire body needs to be checked here
     *)
    | FunkAsyncExpr(block) ->
        let async_env = filter_in_function r_env None in
        let xblocco, async_env = check_block block async_env in
        let xvartype = match async_env.r_ret_type
            with None -> raise (Failure "async block must return a value")
            | Some(xrt) -> xrt
        in let asyncExpr = XFunkAsyncExpr(xvartype,xblocco)
        in let evalue = { e_node = asyncExpr ; result = None ;
                e_type = xvartype }
        in r_env ,evalue

(* Evaluate the lvalues of assignment *)
and eval_lvalues id_list var_type env =
    let xvt, env = extend_vartype var_type env in
    let new_env, variables = List.fold_left (fun (original_env, original_list) next_id ->
```

```
        let new_env, new_obj = mapAdd next_id xvt false original_env in
        new_env, new_obj::original_list
    ) (env, []) id_list in
    let variables = List.rev variables in
    let var_exprs = List.map (fun v -> XVariable(v)) variables in
    var_exprs, new_env

(* Evaluate the rvalues of assignment *)
and eval_rvalue env rvalue =
    match rvalue with
    | ExprRVal x ->
        let new_env,evalue = eval env x
        in new_env, XExprRVal(evalue.e_type, evalue.e_node)
    | FuncRVal x ->
        let func_header, func_body = x in
        let env, xheader, xbody, _ = check_func env func_header func_body None in
        env, XFuncRVal(xheader, xbody)

(* evalute a variable declaration against the environment *)
and eval_vardec env dec =
    let env, rvalues = list_checker eval_rvalue env dec.actual_list in
    let lvalues, env = eval_lvalues dec.id_list dec.var_type env in
    let xa =
        if (List.length rvalues) = 0
        then
            {lvals=[];rvals=[]}
        else
            {lvals=lvalues;rvals=rvalues}
    in
    let _ = check_xassignment xa in
    env, xa

and check_func_call fc env =
    let (f_expr, f_rvalues) = fc in
    (* evaluate rvalues *)
    let env, evaluated_rvalues = list_checker eval_rvalue env f_rvalues in
    (* helper function: takes an X function call*)
    let check_param_types xfc =
        (*made of x function header, called func and rvals *)
        let xhdr, _, rvals = xfc in
        let param_types =
            List.map (fun p_v -> p_v.xbare_type) xhdr.xparams
        in let n_formal = List.length xhdr.xparams
        in let n_actual = List.length rvals
        (* check if the number of parameters is the same *)
        in let _ = if (n_formal = n_actual) then ()
            else raise (Failure ("Expected " ^ string_of_int(n_formal) ^
                    " parameters, but " ^ string_of_int(n_actual) ^ " passed in\n"))
        in
        if (check_rvalues_types param_types rvals) then xfc
        else raise (Failure "Function parameters are of wrong type.")
    in
    let handle_generic env generic_func =
        let env, fevalue = eval env generic_func
```

```
        in let function_vartype = fevalue.e_type
        in let function_header = (match (function_vartype) with
            | XFunkFunc(header), [] -> header
            | _ -> raise (Failure "This is not a function")
        )
        in check_param_types (function_header, GenericCall(fevalue.e_node),
                                        evaluated_rvalues), env
    in
    (match f_expr with
    | Variable v ->
        (match v.id with
        | "print" -> ({xret_type=None; xparams = []}, PrintCall, evaluated_rvalues), env
        | "double2int" -> check_param_types ({xret_type=Some(generalize_single XFunkInt)
                                        ; xparams = [{ xbare_type = (XFunkDouble,[])
                                        }]},
            Double2IntCall, evaluated_rvalues), env
        | "int2double" -> check_param_types ({xret_type=Some(generalize_single
                                        XFunkDouble); xparams = [{ xbare_type =
                                        (XFunkInt,[]) }]},
            Int2DoubleCall, evaluated_rvalues), env
        | "bool2int"-> check_param_types ({xret_type=Some(generalize_single XFunkInt);
                                        xparams = [{ xbare_type = (XFunkBool,[]) }]},
            Boolean2IntCall, evaluated_rvalues), env
        | "int2bool"-> check_param_types ({xret_type=Some(generalize_single XFunkBool);
                                        xparams = [{ xbare_type = (XFunkInt,[]) }]},
            Int2BooleanCall, evaluated_rvalues), env
        | _ -> handle_generic env f_expr
        )
    | generic_func -> handle_generic env generic_func
    )

and check_rvalues_types types rvalues =
    let get_rtype = function
        | XExprRVal(rt, _) -> rt
        | XFuncRVal(hdr, _) -> generalize_single (XFunkFunc hdr)
    in
    (* Match lvalues with rvalues in semantics *)
    let ret_value =
        if (List.length types) = (List.length rvalues)
        then
        List.fold_left2
            (fun _ t rv ->
                if (match_type t (get_rtype rv))
                then
                    true
                else
                    raise (Failure ("Left" ^ "right value types do not match."))
            ) true types rvalues
        else
            false
    in
        ret_value

(** check the assignment statment types*)
```

```
and check_xassignment xa =
    let get_ltype = function
        | XVariable(v) -> v.obj_type
        | XFunkArrExpr(xvt, _, _) -> xvt
        | _ -> raise (Failure "Left value must be variable or array.")
    in
    let ltypes = List.map get_ltype xa.lvals
    in
    check_rvalues_types ltypes xa.rvals

and check_statement stmt env =
    match (stmt) with
    | Break -> XBreak, env
    | Assignment (expressions, rvalues) ->
        let right_checker = list_checker eval_rvalue
        in let env, xrvalues = right_checker env rvalues
        in let left_checker = list_checker eval
        in let env, lvalues = left_checker env expressions
        in let xexprs = List.map (fun x -> x.e_node) lvalues
        in let xa = { lvals = xexprs ; rvals = xrvalues }
        in let _ = check_xassignment xa
        in XAssignment(xa), env
    | Declaration x ->
        let env, xvardec = eval_vardec env x in
        XAssignment(xvardec), env
    | FunctionCall x ->
        let call, env = check_func_call x env
        in XFunctionCall(call), env
    | Block x ->
            let block, env = check_block x env
                in XBlock(block), env
    | ForBlock (a,b,c,d) ->
            let am, env = (match a with
                | None -> None, env
                | Some(stmt) -> let as1, as2 = check_statement stmt env in Some(as1), as2) in
            let bm, env = (match b with
            | None -> None, env
            | Some(expr) -> let bs1, bs2 = eval env expr in Some(bs2.e_node), bs1) in
            let cm, env = (match c with
                | None -> None, env
                | Some(stmt) -> let cs1, cs2 = check_statement stmt env in Some(cs1), cs2) in
            let dm, env = check_block d env in
            XForBlock(am, bm, cm, dm), env
    | IfBlock (x,y) ->
            let env, xexpr = eval env x in
            let xblock, env = check_block y env in
            XIfBlock(xexpr.e_node, xblock), env
    | IfElseBlock (x,y,z) ->
            let env,xm = eval env x in
            let ym, env = check_block y env in
            let zm, env = check_block z env in
            XIfElseBlock(xm.e_node, ym,zm) , env
```

```
    | WhileBlock(x,y) ->
          let env, xexpr = eval env x in
          let xblock, env = check_block y env in
          XWhileBlock(xexpr.e_node, xblock), env
    | Return x ->
          let replace_ret_type old_env new_has_ret new_type = { scope = old_env.scope ;
                                  new_objects = old_env.new_objects ;
                                  new_objects_ordered = old_env.new_objects_ordered ;
                                  current_block = old_env.current_block ;
                                  next_free = old_env.next_free ;
                                  next_bound = old_env.next_bound ;
                                  has_ret = new_has_ret ; r_ret_type = new_type;
                                  free_objects_ordered =
                                                                 old_env.free_objects_ordered
                                                                 }
          in match (x) with
          (* Check return values for consistency *)
          (* Does return a value. Must be consistent with any old type *)
          | Some(rval) ->
                let new_env, new_xvalue = eval_rvalue env rval
                (* figure out returned type *)
                in let ret_type = (match new_xvalue
                   with XExprRVal(xvt, rexpr) -> xvt
                   | XFuncRVal(anon) -> generalize_single (XFunkFunc(fst anon)))
                in let has_ret, r_ret_type = if (env.has_ret) then
                      (* match running return type to new return type *)
                      let type_matched = match env.r_ret_type
                         with None -> raise (Failure ("Function does not need to
                                                           return a value," ^
                                  " according to previous return statements."))
                         | Some(r_ret) -> match_type r_ret ret_type
                      in (if (type_matched) then true, env.r_ret_type
                         else raise (Failure "Return statements do not match in
                                                           type."))
                      (* always alright if no previous return type *)
                   else (true, Some(ret_type))
                in XReturn(Some(new_xvalue)), replace_ret_type new_env has_ret
                                                   r_ret_type
          | None ->
                (* if there is a previous return type, it must be None *)
                let has_ret, r_ret_type = if (env.has_ret) then
                   (if env.r_ret_type = None then env.has_ret, env.r_ret_type
                    else raise (Failure "Function needs to return a value, according
                                                     to previous return
                                                     statements."))
                (* always alright if no previous return type *)
                else (true, None)
                in XReturn(None), replace_ret_type env has_ret r_ret_type

and check_block stmts env =
    let deeper_env = { scope = env.scope; new_objects = env.new_objects;
        current_block = { own_func =
                env.current_block.own_func;
            own_block = env.current_block.own_block + 1 };
```

```
      next_free = env.next_free;
      next_bound = env.next_bound ;
      new_objects_ordered = env.new_objects_ordered ;
      has_ret = env.has_ret ; r_ret_type = env.r_ret_type;
      free_objects_ordered = env.free_objects_ordered}
   in let old_env = env
   in let xstmts, new_env = List.fold_left
         (fun (old_xstmts, old_env) stmt ->
            let xstmt, env = check_statement stmt old_env
            in (xstmt::old_xstmts, env)) ([], deeper_env) stmts

   in let env = filter_out_block new_env old_env
   in
   { xstmts = List.rev xstmts ; need_copy = List.rev env.free_objects_ordered ;
      declared = List.rev env.new_objects_ordered }, env

and extend_func_header fh r_env =
   (*TO_XRET: helper function to extend function header to XFH *)
   let to_xret r_env = function
      | None -> None, r_env
      | Some(rt) -> let xrt, r_env = extend_vartype rt r_env
            in Some(xrt), r_env
   (* TO_XPARAM: helper function that should check parameters of a function and put
                                  them in the scope *)
   in let to_xparam prms r_env =
      let xprms, r_env = List.fold_left
            (* ok so this is the function that does the job *)
            (fun (old_prms, r_env) prm ->
               (* first extend the vartype of the parameter *)
               let xbt, r_env = match prm.bare_type
               with None -> raise (Failure
                           "Missing paramter type in function header")
               | Some(bt) -> extend_vartype bt r_env
               (* then create this object which is just a wrapper of the type ?!*)
               in let xprm = { xbare_type = xbt }
               in (xprm::old_prms, r_env)
            )
            ([], r_env) prms
      in (List.rev xprms), r_env
   in let xparams, r_env = to_xparam fh.params r_env in
   let xrt, r_env = to_xret r_env fh.ret_type
   in { xret_type = xrt ; xparams = xparams }, r_env



(** Check function body and find the return type
      (running_env) r_env: the most recent environment, as passed by caller
      (statement list) body: list of statements in the body of the function level
      returns updated r_env, updated body, which will be futher used in
      check_func, return type, and id_objects that need to be copied from a
      higher scope *)
and check_func_body r_env body =
   let func_env, xbody = List.fold_left
            (fun (old_env, old_xbody) stmt ->
```

```
                    let xstmt, r_env = check_statement stmt
                                old_env
                    in (r_env, xstmt::old_xbody)
                ) (r_env, []) body
    in

    r_env, { xstmts = List.rev xbody ;
    need_copy = List.rev func_env.free_objects_ordered;
    declared = List.rev func_env.new_objects_ordered }, func_env.r_ret_type

(**
* Check entire function, including header and body USED by Rvalue and FuncDec
* (running_env) r_env: the most recent environment, as passed by caller
* (func_dec_header) header: header of function, which gives the type that
* return statements should have, and types of the formal parameters, which
* will be used as variables in the body
* (statement list) body: list of statements in the body of the function
* returns updated r_env, tuple containing header and updated body, which
* will be futher used as an anon value, and id_objects that need to be
* copied from higher scope
*)
and check_func r_env header body name_opt =
    let xheader, r_env = extend_func_header header r_env in
    let r_env, global_obj, name_opt = match name_opt
        with None -> r_env, None, None
        | Some(name) ->
            let r_env, global_obj = mapAdd name (XFunkFunc(xheader),[]) false
                        r_env in
            r_env, Some(global_obj), Some(name)
    in
    let func_env = filter_in_function r_env name_opt in
    let func_env = List.fold_left2 (fun old_env prm xprm ->
        (* add name to the env *)
        fst (mapAdd prm.id xprm.xbare_type false old_env))
        func_env header.params xheader.xparams
    in
    let func_env, body, rtype = check_func_body func_env body in
    let hdr_rtype = xheader.xret_type in
    let _ = match rtype
        with None -> (match hdr_rtype
                with None -> ()
                | _ -> raise (Failure
                    "Function returns no value, but the header says it does")
                 )
        | Some(rt) -> (match hdr_rtype
            with Some(hrt) -> if (match_type hrt rt) then ()
                else raise (Failure "Return type of function does not match header")
            | None -> raise (Failure
                "Function returns value, but the header says it does not")
                 )
    in
    let check_if_copy_required_variables_in_scope r_env xblock =
        let r_body_need_copy = ref [] in
        let r_r_env = ref r_env in
```

```
        List.iter (fun (dest,org) ->
            if (List.mem org r_env.new_objects_ordered) || not (is_not_duplicate r_env
                                                  org.name)
            then
                r_body_need_copy := (dest,org)::!r_body_need_copy
            else
                let n_env, n_variable = mapAdd org.name org.obj_type true !r_r_env in
                r_r_env := n_env;
                r_body_need_copy:= (dest,n_variable)::!r_body_need_copy;
            ) xblock.need_copy;
        !r_r_env, {xstmts = xblock.xstmts; need_copy = (!r_body_need_copy); declared =
                                              xblock.declared}
    in
    let new_r_env, new_r_body = check_if_copy_required_variables_in_scope r_env body in
    new_r_env, xheader, new_r_body, global_obj



let check_declaration declaration env =
    match declaration with
    | Newline x -> None, env
    | Funcdec f ->
        (* invoke the check_func keeping the resulting env for the variables,
         * but not propagating it directly *)
        let env, xheader, xbody, obj_opt = (check_func env f.func_header f.body (Some
                                              f.fid)) in
        let global_obj = match obj_opt
            with Some(g_o) -> g_o
            | None -> raise (Failure "check_func should have returned global_object")
        in
        Some(XFuncdec({ global_id = global_obj ; xfid = f.fid ; xfunc_header = xheader ;
                                              xbody = xbody })), env

    | Vardec x ->
        let env, xvardec = eval_vardec env x in
        Some(XVardec(xvardec)), env

let check_ast_type prog =
    let check_dec_foldable (old_list, old_env) line =
        let xdecl, n_env = check_declaration line old_env in
            match xdecl with
            | Some(xdecl) -> (xdecl::old_list), n_env
            | None -> old_list, n_env
    in let prog_list, env = List.fold_left (check_dec_foldable) ([], new_r_env) prog
    in (List.rev prog_list), List.rev env.new_objects_ordered
```

Listing 33: Semantic and type checking

```
val check_ast_type : Commponents.program -> Xcommponents.xprogram *
    Xcommponents.id_object list
```

Listing 34: Semantic and type checking

```
open Commponents
open Xcommponents
```

```
(* maps parameter types to the the C struct *)
module ParamTable = Map.Make(String)

(*
 * aggregator for global program code
 *)
type struct_table = {
  struct_decs : string ; (* struct declarations *)
  prog_decs : string ; (* C function declarations *)
  (* paramater structs that have already been declared *)
  params : string ParamTable.t
}

(*
 * helper function for writing C-style function calls
 * (string) func_name: name or C-style expression for C function or macro
 * (string list) params: handlers for expressions
 * returns C function call
 *)
let c_func_call func_name params =
  let params_code = String.concat ", " params
  in func_name ^ "(" ^ params_code ^ ")"

let default_env = "self->scope"

let global_scope = "global"

let is_global iobj = iobj.scope_setting.own_func = global_scope

(*
 * get C expression to acces free variable
 * (string) struct_id: the function instance struct
 * (int) free_id: var_id of a free id_object
 * (string) estruct: type of the environment structure for the function
 * returns C expression that accesses free variable in environment of
 *   function
 *)
let get_free_name struct_id free_id estruct =
  "((" ^ estruct ^ " *) " ^ struct_id ^ ")->v_" ^ string_of_int(free_id)

let get_bound_name id = "bv_" ^ string_of_int(id)

let get_global_name id = "gv_" ^ string_of_int(id)

let get_non_free_name iobj = if iobj.scope_setting.own_func = global_scope
  then get_global_name iobj.var_id
  else get_bound_name iobj.var_id

(*
 * get C expression for free or bound variable inside a function
 * (id_object) iobj: ocaml representation of the variable to access
 * (string) estruct: type of the environment structure for the function, and
 *                  contains iobj, if it is a free variable
```

```
 *)
let get_name iobj estruct estruct_var = if iobj.is_free
  then get_free_name estruct_var iobj.var_id estruct
  else get_non_free_name iobj

(*
 * extract single_funk_value from from funk_value
 * (funk_value): assumed to be SingleValue
 * returns the single_funk_value inside the single_value
 *)
let get_single = function
  | SingleValue(sv) -> sv
  | ArrayValue(_) -> raise (Failure "Expected single value")

(*
 * Helper functions to extract primitive values from single_funk_value
 * Each function, "get_[type]" takes the following form:
 * (single_funk_value): assumed to contain a value of type [type]
 * returns value of type [type]
 *)
let get_int = function
  | IntVal(i_v) -> i_v
  | _ -> raise (Failure "Expected int for get_int")
let get_double = function
  | DoubleVal(d_v) -> d_v
  | _ -> raise (Failure "Expected double for get_double")
let get_bool = function
  | BoolVal(b_v) -> b_v
  | _ -> raise (Failure "Expected bool for get_bool")
let get_char = function
  | CharVal(c_v) -> c_v
  | _ -> raise (Failure "Expected char for get_char")

(*
 * Returns string of how type would appear in one of the C-library functions
 * (xsingle_vartype): type to convert to string
 * returns lower-case string corresponding to type for primitive functions only
 *)
let get_type_name = function
  | XFunkInt -> "int"
  | XFunkDouble -> "double"
  | XFunkChar -> "char"
  | XFunkBool -> "bool"
  (* there are no families of C functions that can directly access a function *)
  | _ -> raise (Failure "Cannot access functions like this")

(*
 * generate C function name for setting a primitive value
 * (xsingle_vartype) st: type of value to set
 * returns name of type-appropriate function to write to var struct
 *)
let set_func_type st = "set_" ^ (get_type_name st)

(*
```

```
 * generate C function name for reading a primitive value
 * (xsingle_vartype) st: type of value to read
 * returns name of type-appropriate function to read from var struct
 *)
let get_func_type st = "get_" ^ (get_type_name st)

(*
 * Given non-pointer variable, returns C expression for its reference
 * (string) var: C variable of a non-pointer type
 * returns reference to var
 *)
let point var = "(&" ^ var ^ ")"

(*
 * returns C expression that returns raw, primitive value of a constant or
 * a non-pointer variable
 * (string) sv: value could be constant or non-pointer C variable
 * (xsingle_vartype) st: type of primitive value
 * (bool) is_var: is sv a variable? If false, it's a constant
 * return C expression that returns value contained in sv
 *)
let get_single_raw sv st is_var =
  let bare = if is_var then
      let getter_func = get_func_type st
      in c_func_call getter_func [point sv]
    else sv
  in "(" ^ bare ^ ")"

(*
 * helper function for creating variable name
 * (string) prefix: alphabetic prefix, eg. "t"
 * (int) id: index of variable in function
 * returns C-style variable
 *)
let get_field_name prefix id = prefix ^ "_" ^ (string_of_int id)

(*
 * generate structure
 * (string) name: struct's name, including "struct"
 * (string) prefix: alphabetic prefix, eg. "t"
 * (int) count: number. indexes are expected to include 0 to count - 1
 * returns C-style struct declaration
 *)
let gen_struct name prefix count =
  let rec get_member_codes member_codes current =
    if current < count then
      (get_member_codes
        (("\tstruct var " ^
          (get_field_name prefix current) ^ ";\n")::member_codes)
         (current + 1))
    else member_codes
  in name ^ "{ \n" ^
    (String.concat "\n" (List.rev (get_member_codes [] 0))) ^
    "};\n"
```

```

(*
 * get the C-style struct name
 * (string) sname: bare name of struct
 * returns
 *)
let get_struct_type sname = "struct " ^ sname

(*
 * get string encoding for list of types for parameters. The string is
 * used to look up the param structure in a ParamTable
 * params
 * (xvar list): the list of parameters from xfunc_dec_header
 * returns string representation of parameters
 *)
let param_encode params = string_of_int (List.length params)

(*
 * get C struct type for param struct
 * (xvar list): the list of parameters from xfunc_dec_header
 * returns C code for the param struct type
 *)
let get_pstruct_type params = get_struct_type ("param_" ^ (param_encode params))

(*
 * helper function for getting param struct name
 * (var list) params: list of parameter variables
 * (struct_table) struct_tbl: current struct declarations and map of structures
 *                          returns
 *   struct name (including "struct ")
 *   the updated struct_tbl, which will have the new name added if it was not
 *     there before
 *   the code for declaring the struct, if the type is new, and contains
 *     at least 1 parameter. Otherwise, return empty string. Also return
 *     empty string if there are no parameters.
 *)
let get_param_struct params struct_tbl =
  if ((List.length params) = 0) then "", struct_tbl else
    let encoding = param_encode params in
    if ParamTable.mem encoding struct_tbl.params
      then (ParamTable.find encoding struct_tbl.params, struct_tbl)
      else
        let param_struct = get_pstruct_type params in
        let param_dec = if (List.length params >= 1)
          then gen_struct param_struct "p" (List.length params)
          else ""
        in (param_struct,
          { params = ParamTable.add encoding param_struct struct_tbl.params ;
            struct_decs = struct_tbl.struct_decs ^ param_dec ;
            prog_decs = struct_tbl.prog_decs })

(*
 * get name for per-function free-variable struct
 * (string) fname: function name
```

```
 * returns "struct [fname]_env"
 *)
let get_estruct_type fname = get_struct_type fname ^ "_env"

(*
 * Generate struct for free variables
 * (string) fname: function name
 * (id_object list) needed: needed sources for free variables, ie. the need_copy
 *                          field of the function body's xblock
 * return C-style declaration of free-variable struct
 *)
let get_env_struct fname needed =
  if (List.length needed = 0) then ("", "")
    else let env_struct = get_estruct_type fname in
      (env_struct, gen_struct env_struct "v" (List.length needed))

(*
 * get pointer to array member of var struct
 * (string) name: name of the struct var value that is an array
 * (string) index: array index. Could be a constant or variable
 * returns C-style expression for getting the pointer to the arrat member
 *)
let get_arr_memb name index = name ^ "->val.array[" ^ index ^ "]"

(*
 * generic helper for traversing a var that could be an array
 * (string) name: name of the struct var value that is an array or single value
 * (int) depth: number of levels in array. a single value has depth 0
 * (fun string -> string) pre_iter: if the object is an array, the action to
 *                                  take before iterating through the array
 * (fun string -> string) single_handler : if the object is a single value,
 *                                         the action to take
 * returns C-style nested for loops for handling all the single values the
 * variable contains
 *)
let rec trav_arrs name depth pre_iter single_handler = match depth
  with 0 -> single_handler name
  | d -> let memb_access = point (get_arr_memb name "arr_i")
        in let memb_name = "member_" ^ string_of_int(depth)
        in "{\n" ^
        "int arr_i;\n" ^
        (pre_iter name) ^
        "for (arr_i = 0; arr_i < " ^ name ^ "->arr_size; arr_i++) {\n" ^
        "struct var *" ^ memb_name ^ " = " ^ memb_access ^ ";\n" ^
        (trav_arrs memb_name (depth - 1) pre_iter single_handler) ^
        "\n}\n" ^
        "\n}\n"


(* "struct function_instance" *)
let func_inst_type = get_struct_type "function_instance"

(*
 * string to cast a variable to a C pointer type
```

```
 * (string) tn: C type to cast an expression to
 * (string) castee: the expression to cast as [tn] *
 *)
let cast_ptr tn castee = "(" ^ tn ^ " *) (" ^ castee ^ ")"

(*
 * output temporary function and its initialization code
 * (int) id: the temporary id of variable, derived from temp_cnt
 * returns temporary variable, and its initialization code
 *)
let gen_temp id =
  let t_name = "t_" ^ string_of_int(id)
  in let t_dec = "struct var " ^ t_name ^ ";\n" ^
    (c_func_call "init_var" [point t_name]) ^ ";\n"
  in t_name, t_dec

(*
 * allocates function instance
 * (string) temp: base name for instance to which "_inst" is appended.
 *                Could be variable for var struct.
 * returns
 *   pointer variable to function instance
 *   code to malloc function instance
 *)
let alloc_inst temp = let inst_var = temp ^ "_inst"
  in let init_inst = func_inst_type ^ " *" ^ inst_var ^
    " = malloc(sizeof(" ^ func_inst_type ^ "));\n"
  in inst_var, init_inst

(*
 * wrapper for alloc_inst that also has code for setting the instance of temp
 * (string) temp: base name and variable of type struct var
 * returns
 *   pointer variable to function instance
 *   code to malloc function instance, and set it to ptr of temp
 *)
let set_alloc_inst temp =
  let inst_var, init_inst = alloc_inst temp
  in let set_alloced = temp ^ ".val.ptr = " ^ inst_var ^ ";\n"
  in inst_var, init_inst ^ set_alloced

(*
 * like set_alloc_inst, but given a pointer, and bare variable is not available
 * (string) dst_ptr: C expression for pointer to struct var
 * returns
 *   pointer variable to function instance
 *   code to malloc function instance, and set it to ptr of temp
 *   updated temp_cnt
 *)
let set_alloc_inst_ptr dst_ptr =
  let temp_var, _ = gen_temp 0
  in let inst_var, init_inst = alloc_inst temp_var
  in let set_alloced = dst_ptr ^ "->val.ptr = " ^ inst_var ^ ";\n"
  in inst_var, init_inst ^ set_alloced
```

```
(*
 * Helper function for wrapping C statements inside block braces
 * (string) bare_blocks: lines of C code
 * returns bare_blocks inside braces
 *)
let enclose bare_block = "{\n" ^ bare_block ^ "\n}\n"

(*
 * generic helper function for creating a function definition out of a function
 * body
 * (string) fname: function name
 * (string) fbody_code: C-style function body code
 * (string) ret_type: C return type, which could be "void"
 * (string list) params: parameter types and names
 * returns C-style function definition
 *)
let wrap_c_func fname fbody_code ret_type params =
  let param_strs = String.concat ", " params
  in "static " ^ ret_type ^ fname ^ "(" ^ param_strs ^ ")\n" ^
  (enclose fbody_code)

(*
 * helper functions for generating the names of the C functions for each
 * funk function that will be the members of struct function_instance
 * (string) fname: internal name of function
 *)
(* returns name of .function member *)
let get_inst_function fname = fname ^ "_function"
(* returns name of .init member *)
let get_inst_init fname = fname ^ "_init"
(* returns name of .copy member *)
let get_inst_copy fname = fname ^ "_copy"

(*
 * Creates C expression that returns cast pointer to function instance
 *)
let get_func_inst var = "(" ^ var ^ "->val.ptr" ^ ")"

(*
 * calls initialization function from a function var struct on an instance
 * (string) dst_inst_ptr: C pointer to function instance to initialize
 * (string) initer_ptr: C pointer to var struct to initialize as function
 * returns call of init function in variables function instance on dst_ptr
 *)
let gen_init_func dst_inst_ptr initer_ptr =
  c_func_call ((get_func_inst initer_ptr) ^ "->init") [dst_inst_ptr]

(*
 * copy function instance from var struct pointer to var struct
 * (string) dst_ptr: C expression that is of type "struct var *"
 * (string) src_ptr: C expression that is of type "struct var *"
 * returns C code to allocate instance at destination, code to initialize it
 *   copy source's env to it
```

```
 *)
let copy_func dst_ptr src_ptr =
  let copy_line = (c_func_call "copy_funcvar"
    [dst_ptr ; src_ptr]) ^ ";\n"
  in copy_line

(*
 * helper function that puts function code into a C function
 * (string) fname: function name
 * (string) fbody_code: function's C code
 *)
let wrap_func_body fname fbody_code =
  let fbody_code =
    func_inst_type ^ " *self = (" ^ func_inst_type ^ " *)data;\n" ^
    fbody_code
  in wrap_c_func fname fbody_code "void *" ["void *data"]

(*
 * generates default copy function output by gen_expr
 * (string) src: variable of type struct var
 * returns function for generating 1-level copy variable function
 *)
let default_copy src =
  fun dst -> (c_func_call "copy_primitive" [point dst ; point src]) ^ ";\n"

(*
 * generates copy function output by gen_expr for non-primitive types
 * (string) src: variable of type struct var
 * returns function for generating 1-level copy variable function
 *)
let shallow_copy src =
  fun dst -> (c_func_call "shallow_copy" [point dst ; point src]) ^ ";\n"

(*
 * generates internal name for anonymous function. Can also be used for async
 * blocks
 * (string) fname: function that contains this function. Could be anonymous,
 *                 in which case its name was also created using this function
 * (int) temp_cnt: id used to name this function
 * returns name of anon and possibly-updated temp_cnt
 *)
let get_anon_name fname temp_cnt =
  fname ^ "anon" ^ string_of_int(temp_cnt), temp_cnt

(*
 * Generates C code for a function body, which could be an anonymous block
 * (string) fname: internal version of name
 * (xbody) fbody: function body or anonymous block
 * (xvar list) params: function's paramaters. always empty for async blocks
 * (struct_table) struct_tbl: current struct declarations and map of structures
 *                          returns
 * returns
 *   the function to instantiate the function
 *   the updated struct_tbl
```

```
 *)
let rec gen_func_body fname fbody params struct_tbl =
  let estruct = get_estruct_type fname
  in let c_body, _, struct_tbl = gen_block fbody estruct fname 0
    struct_tbl
  in let n_params = List.length params
  in let pstruct_var = "my_params"
  in let pstruct, dec_pstruct, struct_tbl = if n_params > 0 then
    let pstruct, struct_tbl = get_param_struct params struct_tbl
    in let dec_pstruct = pstruct ^ " *" ^ pstruct_var ^ " = " ^
      (cast_ptr pstruct "self->params") ^
      ";\n"
    in pstruct, dec_pstruct, struct_tbl
    else "", "", struct_tbl
  in let get_param = fun i -> point (pstruct_var ^ "->p_" ^ string_of_int(i))
  in let bound_decs, _ = List.fold_left
    (fun (old_code, id) dec ->
      let bv_name = "bv_" ^ string_of_int(id)
      in let param_copy = if id < n_params
        then (c_func_call "shallow_copy" [point bv_name ; get_param id]) ^ ";\n"
        else ""
      in let bv_dec = "struct var " ^ bv_name ^ ";\n" ^
        (c_func_call "init_var" [point bv_name]) ^ ";\n" ^ param_copy
      in (old_code ^ bv_dec, id + 1)
    ) ("", 0) fbody.declared
  in let initer = (fun inst_var env_var temp_cnt struct_tbl ->
    gen_new_func inst_var fbody fname estruct env_var temp_cnt struct_tbl)
  in let estruct_name, estruct_dec =
    get_env_struct fname fbody.need_copy
  in let func_def = wrap_func_body (get_inst_function fname)
                  (dec_pstruct ^ bound_decs ^ c_body ^
           "\nRETURN: return NULL;\n")
  in let func_init = gen_init fname fbody estruct_name pstruct
  in let func_copy = gen_copy fname fbody estruct_name
  in let struct_tbl = { struct_decs = struct_tbl.struct_decs ^ estruct_dec ;
                        prog_decs = struct_tbl.prog_decs ^ func_def ^
                        func_copy ^ (fst func_init) ;
                        params = struct_tbl.params }
  in initer, struct_tbl
(*
 * generates code to evaluate expression
 * (xexpr) expr_nd: expression node of SAST
 * (string) estruct: environment structure
 * (string) fname: function name
 * (int) temp_cnt:-k 0123456789abcdef cd running count of temporary variables
 * (struct_table) struct_tbl: current struct declarations and map of
 *                          structures returns
 * returns
 *   C representation of value
 *   is value a temporary variable, rather than raw constant?
 *   function to generate code to copy value into a var struct
 *   C code to generate value
 *   the value's type
 *   new temp_cnt
```

```
 *   new struct_tbl
 *)
and gen_expr expr_nd estruct fname temp_cnt struct_tbl =
  match expr_nd
  (* different code and copier for different types of expressions *)
  (* a simple constant; no C variable needed *)
  with XSingleConst(vt, sv) ->
    let const_copy type_pre = fun src -> fun dst ->
      (c_func_call ("set_" ^ type_pre)
       [point dst ; src]) ^ ";\n"
    in let val_code, val_copy = (match sv
    with IntVal(iv) -> string_of_int(iv), const_copy "int"
    | DoubleVal(dv) -> string_of_float(dv), const_copy "double"
    | BoolVal(dv) -> (if dv then "1" else "0"), const_copy "bool"
    | CharVal(cv) -> "\'" ^ (Char.escaped cv) ^ "\'", const_copy "char"
    | _ -> raise (Failure "Only support primitives as XSingleConst")
    )
    in val_code, false, val_copy val_code, "", (vt, []), temp_cnt, struct_tbl
  (* array has its own temp variable, and needs to evaluate each member *)
  | XArrayLit(xvt, xrvs) ->
      let arr_temp, arr_gen = gen_temp temp_cnt
      in let arr_ptr = point arr_temp
      in let temp_cnt = temp_cnt + 1
      in let size_expr = List.hd (snd xvt)
      in let size_temp, size_is_temp, _, size_gen, size_xvt, temp_cnt,
        struct_tbl = gen_expr size_expr estruct fname temp_cnt struct_tbl
      in let arr_gen = arr_gen ^ size_gen ^
        (c_func_call "init_array"
         [arr_ptr ; get_single_raw size_temp XFunkInt size_is_temp]) ^ ";\n"
      in let _, arr_fill, temp_cnt, struct_tbl = List.fold_left
        (fun (i, old_code, old_cnt, old_tbl) xrv ->
          let memb_code, handle, _, temp_cnt, struct_tbl =
            gen_rvalue xrv estruct fname old_cnt old_tbl
          in let fill_memb = (c_func_call "set_element" [arr_ptr;
                                                   (string_of_int i) ;
                                                   (point handle)]) ^ ";\n"
          in i + 1, old_code ^ memb_code ^ fill_memb, temp_cnt, struct_tbl
        )
        (0, "", temp_cnt, struct_tbl) xrvs
      in let arr_gen = arr_gen ^ arr_fill
      in arr_temp, true, shallow_copy arr_temp, arr_gen, xvt, temp_cnt,
        struct_tbl
  (* variable object. Access free or bound variable *)
  | XVariable(iobj) -> let name = get_name iobj estruct default_env
    in let temp_cnt = temp_cnt + 1
    (* do allow inside effects due to anonymous functions *)
    in let is_function = match iobj.obj_type
      with XFunkFunc(_), [] -> true
      | _ -> false
    in let fetch_var, var_temp, temp_cnt =
      if (is_global iobj || is_function) then "", name, temp_cnt
      else
        let var_temp, var_gen = gen_temp temp_cnt
        in let temp_cnt = temp_cnt + 1
```

```
      in let copy_var = gen_copy_var_wrap (point var_temp) (point name)
        iobj.obj_type estruct
      in let fetch_var = var_gen ^ copy_var
      in fetch_var, var_temp, temp_cnt
  in var_temp, true, default_copy var_temp, fetch_var, iobj.obj_type,
    temp_cnt, struct_tbl
  (*in name, true, default_copy name, "", iobj.obj_type, temp_cnt,
    struct_tbl*)
(*
 * Unary expression. Evaluate expression first, then apply a primitive C
 * operation on it
 *)
| XFunkUnExpr(xvt, op, xpr) ->
  let un_temp, un_gen = gen_temp temp_cnt
  in let temp_cnt = temp_cnt + 1
  in let opstring = match op
    with IntNeg -> "-"
    | BitNot -> "~"
    | Not -> "!"
    | Positive -> ""
  in let st = fst xvt
  in let xpr_temp, xpr_is_temp, _, xpr_gen, xpr_xvt, temp_cnt,
    struct_tbl = gen_expr xpr estruct fname temp_cnt struct_tbl
  in let set_result = c_func_call (set_func_type st) [point un_temp ;
                      opstring ^
                      (get_single_raw xpr_temp st xpr_is_temp)] ^ ";\n"
  in let un_gen = xpr_gen ^ un_gen ^ set_result
  in un_temp, true, default_copy un_temp, un_gen, xvt, temp_cnt,
    struct_tbl
(*
 * binary function. Generate code for evaluating both expressions,
 * but depending on the operation and the result of one of the expressions,
 * the other expression may not be executed during runtime, or a different
 * operator is used
 *)
| XFunkBinExpr(xvt, xpr0, op, xpr1) ->
  let bin_temp, bin_gen = gen_temp temp_cnt
  in let bin_ptr = point bin_temp
  in let temp_cnt = temp_cnt + 1
  in let st = fst xvt
  in let t0, is_temp0, _, gen0, xvt0, temp_cnt,
    struct_tbl = gen_expr xpr0 estruct fname temp_cnt struct_tbl
  in let t1, is_temp1, _, gen1, xvt1, temp_cnt,
    struct_tbl = gen_expr xpr1 estruct fname temp_cnt struct_tbl
  (* most operations always need to evaluate both expressions *)
  in let eval_both gen0 gen1 = gen0 ^ gen1
  (* for most operations, always use the same C operator *)
  in let single_op_raw op = fun val0 val1 ->
      c_func_call (set_func_type st) [bin_ptr ; val0 ^ op ^ val1 ] ^ ";\n"
  (* wrapper for single_op that figures out how to get the input values *)
  in let single_op op xvt0 xvt1 = fun t0 t1 is_temp0 is_temp1 ->
    single_op_raw op (get_single_raw t0 (fst xvt0) is_temp0)
      (get_single_raw t1 (fst xvt1) is_temp1)
  (* condition on the expressions' results determines what operator to use *)
```

```
  in let cond_op op0 op1 xvt0 xvt1 cond = fun t0 t1 is_temp0 is_temp1
   -> let val0 = get_single_raw t0 (fst xvt0) is_temp0
   in let val1 = get_single_raw t1 (fst xvt1) is_temp0
   in "if (" ^ (cond val0 val1) ^ ") {\n" ^
    single_op_raw op0 val0 val1 ^
    "} else {\n" ^
    single_op_raw op1 val0 val1 ^
    "}\n"
  (* most operations always evaluate both expressions and use one operand *)
  in let default_code op xvt0 xvt1 = fun gen0 gen1 t0 t1 is_temp0
   is_temp1 ->
   (eval_both gen0 gen1) ^
   (single_op op xvt0 xvt1 t0 t1 is_temp0 is_temp1)
  (* evaluate second expression if first condition is met *)
  in let cond_exec1 op xvt0 xvt1 cond only0 = fun gen0 gen1 t0 t1 is_temp0
   is_temp1 ->
   let val0 = (get_single_raw t0 (fst xvt0) is_temp0)
   in let val1 = (get_single_raw t1 (fst xvt1) is_temp1)
   in gen0 ^
    "if (" ^ (cond val0) ^ ") {\n" ^
    gen1 ^
    single_op_raw op val0 val1 ^
    "} else {\n" ^
    only0 val0 ^
    "}\n"
  (*
   * for And and Or, if condition is not met, we just use the first
   * expression
   *)
  in let get0 val0 = c_func_call (set_func_type st) [bin_ptr ; val0] ^
   ";\n"
  (*
   * evaluate both expressions, but condition determines operation
   *)
  in let both_cond op0 op1 xvt0 xvt1 cond = fun gen0 gen1 t0 t1 is_temp0
   is_temp1 ->
   (eval_both gen0 gen1) ^ (cond_op op0 op1 xvt0 xvt1 cond t0 t1 is_temp0
               is_temp1)
  (*
   * condition function that compares expression output to some value
   *)
  in let comp_cond eq target =
   fun cmpval -> cmpval ^ " " ^ eq ^ " " ^ target
  (*
   * condition for non-negative shift
   *)
  in let nonneg val0 val1 = comp_cond ">=" "0" val1
  (*
   * wrapper for both_cond with shift operations, which reverses shift for
   * negative shifts
   *)
  in let shift_cond op0 op1 xvt0 xvt1 = both_cond op0 op1 xvt0 xvt1
   nonneg
  (*
```

```
   * get code generator depending on the operation
   *)
  in let gen_bin_code = match op
    with Mult -> default_code "*" xvt0 xvt1
    | Div -> default_code "/" xvt0 xvt1
    | Mod -> default_code "%" xvt0 xvt1
    | Add -> default_code "+" xvt0 xvt1
    | Sub -> default_code "-" xvt0 xvt1
    | LSh -> shift_cond "<<" ">>" xvt0 xvt1
    | RSh -> shift_cond ">>" "<<" xvt0 xvt1
    | LeT -> default_code "<" xvt0 xvt1
    | GrT -> default_code ">" xvt0 xvt1
    | LE -> default_code "<=" xvt0 xvt1
    | GE -> default_code ">=" xvt0 xvt1
    | Eq -> default_code "==" xvt0 xvt1
    | NEq -> default_code "!=" xvt0 xvt1
    | BAnd -> default_code "&" xvt0 xvt1
    | BXor -> default_code "^" xvt0 xvt1
    | BOr -> default_code "|" xvt0 xvt1
    | And -> cond_exec1 "&&" xvt0 xvt1 (comp_cond "!=" "0")
            get0
    | Or -> cond_exec1 "||" xvt0 xvt1 (comp_cond "==" "0")
            get0
  in let bin_code = bin_gen ^ (gen_bin_code gen0 gen1 t0 t1 is_temp0
                           is_temp1)
  in bin_temp, true, default_copy bin_temp, bin_code, xvt, temp_cnt,
    struct_tbl
(* call function, using the generic gen_func_call *)
| XFunkCallExpr(xvt, xfc) ->
  let ret_var, call_code, temp_cnt, struct_tbl = gen_func_call xfc estruct
                                                  fname temp_cnt struct_tbl
  in let ret_temp, ret_gen = gen_temp temp_cnt
  in let temp_cnt = temp_cnt + 1
  in let store_ret = gen_copy_var_wrap (point ret_temp) (point ret_var)
    xvt estruct
  in let get_call_code = ret_gen ^ call_code ^ store_ret
  in ret_temp, true, default_copy ret_var, get_call_code, xvt, temp_cnt,
    struct_tbl
(*
 * evaluating an async expressions is like declaring a function, and then
 * running it as a thread
 *)
| XFunkAsyncExpr(xvt, ab) ->
  let out_temp, out_gen = gen_temp temp_cnt
  in let temp_cnt = temp_cnt + 1
  in let async_fname, temp_cnt = get_anon_name fname temp_cnt
  in let async_var, init_async_inst = alloc_inst async_fname
  in let temp_cnt = temp_cnt + 1
  in let st = fst xvt
  in let initer, struct_tbl = gen_func_body async_fname ab [] struct_tbl
  in let init_func, temp_cnt, struct_tbl = initer async_var default_env
                                            temp_cnt struct_tbl
  in let call_func = c_func_call "run_async" [(point out_temp) ; async_var] ^
    ";\n"
```

```
    in let async_code = out_gen ^ init_async_inst ^ init_func ^ call_func
    in out_temp, true, default_copy out_temp, async_code, xvt, temp_cnt,
      struct_tbl
  | XFunkArrExpr(xvt, arr_expr, i_expr) ->
    let arr_temp, arr_gen = gen_temp temp_cnt
    in let temp_cnt = temp_cnt + 1
    in let arr_temp, arr_is_temp, _, gen_arr, arr_xvt, temp_cnt,
      struct_tbl = gen_expr arr_expr estruct fname temp_cnt struct_tbl
    in let i_temp, i_is_temp, _, gen_i, i_xvt, temp_cnt,
      struct_tbl = gen_expr i_expr estruct fname temp_cnt struct_tbl
    in let i_val = get_single_raw i_temp XFunkInt i_is_temp
    in let arr_gen = gen_arr ^ gen_i
    in let el_temp, el_gen = gen_temp temp_cnt
    in let temp_cnt = temp_cnt + 1
    in let arr_get = (c_func_call "get_element" [(point arr_temp) ;
                                                  i_val])
    in let el_copy = gen_copy_var_wrap (point el_temp) arr_get xvt estruct
    in let el_copy = el_gen ^ el_copy
    in el_temp, true, default_copy arr_temp, arr_gen ^ el_copy, xvt, temp_cnt,
      struct_tbl
(*
 * Generates code to copy between variables
 * (string) dst_ptr: C pointer variable of output
 * (string) src_ptr: C pointer variable of input
 * (int) depth: depth of array. 0 for single type
 * (xsingle_vartype) base_type: the first member of the variable's type tuple
 * (string) estruct: env struct
 * (int) temp_cnt: running count of temporary variables
 * (struct_table) struct_tbl: current struct declarations and map of
 *                          structures returns
 * returns
 *   code to copy variable
 *   updated temp_cnt
 *   updated struct_tbl
 *)
and gen_copy_var dst_ptr src_ptr depth base_type estruct_var =
  if depth = 0 then
    let try_copy = match base_type
      with XFunkFunc(_) -> copy_func dst_ptr src_ptr
      | _ -> (c_func_call "shallow_copy" [dst_ptr ; src_ptr]) ^ ";\n"
    in try_copy
  else
    let src_memb_access = point (get_arr_memb src_ptr "arr_i")
    in let dst_memb_access = point (get_arr_memb dst_ptr "arr_i")
    in let src_memb = "member_src_" ^ string_of_int(depth)
    in let dst_memb = "member_dst_" ^ string_of_int(depth)
    in let alloc_code = (c_func_call "init_array" [dst_ptr ; src_ptr ^
                                                    "->arr_size"]) ^ ";\n"
    in let gen_members =
      let gen_member = gen_copy_var dst_memb
        src_memb (depth - 1) base_type estruct_var
      in "{\n" ^
      "int arr_i;\n" ^
      "for (arr_i = 0; arr_i < " ^ dst_ptr ^ "->arr_size; arr_i++) {\n" ^
```

```
      "struct var *" ^ dst_memb ^ " = " ^ dst_memb_access ^ ";\n" ^
      "struct var *" ^ src_memb ^ " = " ^ src_memb_access ^ ";\n" ^
      gen_member ^
      "\n}\n" ^
      "\n}\n"
    in alloc_code ^ gen_members
(*
 * wrapper for gen_copy_var, that takes a full type, from which it finds the
 * array depth and the single base type
 * (string) dst_ptr: C variable of output
 * (string) src_ptr: C variable of input
 * (xvartype) full_type: contains base type, and depth is length of its second
 *                        part, the list of sizes
 * (string) estruct: env struct
 * (int) temp_cnt: running count of temporary variables
 * (struct_table) struct_tbl: current struct declarations and map of
 *                            structures returns
 * returns
 *   code to copy variable
 *   updated temp_cnt
 *   updated struct_tbl
 *)
and gen_copy_var_wrap dst_ptr src_ptr full_type estruct_name =
  gen_copy_var dst_ptr src_ptr (List.length (snd full_type))
    (fst full_type) estruct_name
(*
 * generates C code that initializes a new function instance
 * (string) inst_var: pointer to instance
 * (xblock) fbody: contains free variables that need copies
 * (string) dst_estruct: type of instance's env
 * (int) temp_cnt: running count of temporary variables
 * (struct_table) struct_tbl: current struct declarations and map of
 *                            structures returns
 * returns
 *   code to initialize function
 *   updated temp_cnt
 *   updated struct_tbl
 *)
and gen_new_func inst_var fbody fname dst_estruct dst_env temp_cnt struct_tbl =
  let setup_env, temp_cnt, struct_tbl = if ((List.length fbody.need_copy) > 0)
    then
      let alloc_env = c_func_call "SETUP_FUNC" [inst_var ; fname] ^ ";\n"
      in let env_var = "t_" ^ string_of_int(temp_cnt) ^ "_env"
      in let temp_cnt = temp_cnt + 1
      in let src_estruct = get_estruct_type fname
      in let create_env_var = dst_estruct ^ " *" ^ env_var ^ " = (" ^
        dst_estruct ^ "*) " ^ inst_var ^ "->scope;\n"
      in let copy_code, temp_cnt, struct_tbl =
        List.fold_left (fun (old_code, old_cnt, old_tbl)
        (dst, src) ->
        let dst_var = get_name dst dst_estruct env_var
        in let src_var = get_name src src_estruct dst_env
        in let sdt = dst.obj_type
        in let new_code =
```

```
        gen_copy_var (point dst_var) (point src_var) (List.length (snd sdt))
        (fst sdt) dst_estruct
      in old_code ^ new_code, temp_cnt, struct_tbl)
      ("", temp_cnt, struct_tbl) fbody.need_copy
    in alloc_env ^ create_env_var ^ copy_code, temp_cnt, struct_tbl
   else "", temp_cnt, struct_tbl
  in let func_init_name = get_inst_init fname
  in (c_func_call func_init_name [inst_var]) ^ ";\n" ^ setup_env,
   temp_cnt, struct_tbl

(*
 * generate code for xrvalue
 * (xrvalue) rv: the xrvalue for which to generate code
 * (string) estruct: environment structure
 * (int) temp_cnt: running count of temporary variables
 * (struct_table) struct_tbl: current struct declarations and map of
 * returns:
 *   C code to calculate xrvalue
 *   the C-code representation of the variable handler of the xrvalue
 *   the type of the rvalue
 *   the updated temp_cnt
 *   the updated struct_tbl
 *)
and gen_rvalue rv estruct fname temp_cnt struct_tbl =
  let rcode, temp, rtype, temp_cnt, struct_tbl = match rv
   with XExprRVal(vt, ex) ->
     let handle, have_temp, copier, excode, _, temp_cnt, struct_tbl =
       gen_expr ex estruct fname temp_cnt struct_tbl
     in if have_temp then excode, handle, vt, temp_cnt, struct_tbl
     else
       let temp, temp_code = gen_temp temp_cnt
       in let temp_cnt = temp_cnt + 1
       in let cpcode = match vt
         with (st, []) -> copier temp
         (* sanity check *)
         | (_, _) -> raise (Failure "Arrays must have a temporary variable")
       in excode ^ temp_code ^ cpcode, temp, vt, temp_cnt, struct_tbl
   | XFuncRVal(anf) ->
     let header, body = anf
     in let anon_name, temp_cnt = get_anon_name fname temp_cnt
     in let temp, temp_code = gen_temp temp_cnt
     in let temp_cnt = temp_cnt + 1
     in let inst_var, init_inst = set_alloc_inst temp
     in let gen_vars = temp_code ^ init_inst
     in let initer, struct_tbl =
       gen_func anon_name header body struct_tbl
     in let init_code, temp_cnt, struct_tbl = initer inst_var default_env
       temp_cnt struct_tbl
     in gen_vars ^ init_code, temp, (XFunkFunc(header), []), temp_cnt,
       struct_tbl
  in rcode, temp, rtype, temp_cnt, struct_tbl
(*
 * generate code for a list of xrvalues
 * (xrvalue list) rvalues: the xrvalues for which to generate code
```

```
 * (string) estruct: environment structure
 * (string) fname: function name
 * (int) temp_cnt: running count of temporary variables
 * (struct_table) struct_tbl: current struct declarations and map of
 * returns:
 *   C code to calculate xrvalue
 *   list of C-code representations of the variable handlers of the xrvalue
 *   the type of the rvalue
 *   the updated temp_cnt
 *   the updated struct_tbl
 *)
and gen_rvalues rvalues estruct fname temp_cnt struct_tbl =
  let rvalue_code, handles, types, temp_cnt, struct_tbl = List.fold_left
    (fun (old_code, old_handles, old_types, old_cnt, old_tbl) rv ->
      let next_piece, handle, rtype, temp_cnt, struct_tbl =
        gen_rvalue rv estruct fname old_cnt old_tbl
      in old_code ^ next_piece, handle::old_handles, rtype::old_types,
        temp_cnt, struct_tbl
    )
    ("", [], [], temp_cnt, struct_tbl) rvalues
  in rvalue_code ^ "", List.rev handles, List.rev types,
    temp_cnt, struct_tbl
(*
 * generate code for function calls
 * (xfunc_call) fc: the function call SAST node
 * (string) estruct: environment structure
 * (string) fname: function name
 * (int) temp_cnt: running count of temporary variables
 * (struct_table) struct_tbl: current struct declarations and map of
 * returns:
 *   a variable handle for the return value
 *   code to call or execute the function
 *   the updated temp_cnt
 *   the updated struct_tbl
 *)
and gen_func_call fc estruct fname temp_cnt struct_tbl =
  let hdr, callee, params = fc
  in let param_code, handles, types, temp_cnt, struct_tbl =
    gen_rvalues params estruct fname temp_cnt struct_tbl
  in let conv_func dst_st src_st conv = fun src temp_cnt ->
    let dst_temp, dst_gen = gen_temp temp_cnt
    in let temp_cnt = temp_cnt + 1
    in let get_src_func = get_func_type src_st
    in let set_dst_func = set_func_type dst_st
    in let get_src = c_func_call get_src_func [point src]
    in let set_dst = c_func_call set_dst_func [point dst_temp ; conv get_src] ^
      ";\n"
    in let gen_conv = dst_gen ^ set_dst
    in dst_temp, gen_conv, temp_cnt
  in let ret_var, exec_code, temp_cnt, struct_tbl = match callee
    (* built-in functions, and generic function *)
    with PrintCall ->
      let print_single stype = fun sname ->
        let print_type = match stype
```

```
          with XFunkInt -> "INT"
          | XFunkDouble -> "DOUBLE"
          | XFunkChar -> "CHAR"
          | XFunkBool -> "BOOL"
          | _ -> raise (Failure "Only primitives can be printed directly")
        in (c_func_call ("PRINT_" ^ print_type) [sname]) ^ ";\n"
      in let print_code = List.fold_left2
        (fun old_code rhand rtype ->
          let new_code = trav_arrs (point rhand) (List.length (snd rtype))
            (fun _ -> "") (print_single (fst rtype))
          in old_code ^ new_code
        )
        "" handles types
      in "", print_code, temp_cnt, struct_tbl
    | Double2IntCall ->
      let conv_ret, conv_code, temp_cnt =
        conv_func XFunkInt XFunkDouble (fun x -> x) (List.hd handles) temp_cnt
      in conv_ret, conv_code, temp_cnt, struct_tbl
    | Int2DoubleCall ->
      let conv_ret, conv_code, temp_cnt =
        conv_func XFunkDouble XFunkInt (fun x -> x) (List.hd handles) temp_cnt
      in conv_ret, conv_code, temp_cnt, struct_tbl
    | Boolean2IntCall ->
      let conv_ret, conv_code, temp_cnt =
        conv_func XFunkInt XFunkBool (fun x -> x) (List.hd handles) temp_cnt
      in conv_ret, conv_code, temp_cnt, struct_tbl
    | Int2BooleanCall ->
      let conv_ret, conv_code, temp_cnt =
        conv_func XFunkInt XFunkBool (fun x -> x ^ " ? 1 : 0")
          (List.hd handles) temp_cnt
      in conv_ret, conv_code, temp_cnt, struct_tbl
    (* generic case, where we use the struct var containing ptr *)
    | GenericCall(xpr) ->
      let func_var, _, _, func_gen, func_type, temp_cnt, struct_tbl =
        gen_expr xpr estruct fname temp_cnt struct_tbl
      (* passing in parameters *)
      in let pstruct, struct_tbl = get_param_struct hdr.xparams struct_tbl
      in let pstruct_name = "params_" ^ string_of_int(temp_cnt)
      in let temp_cnt = temp_cnt + 1
      in let func_var_ptr = point func_var
      in let func_ptr = func_var_ptr ^ "->val.ptr"
      in let rec check_global = function
        | XVariable(iobj) -> is_global iobj
        | XFunkArrExpr(_, arr_xpr, _) -> check_global arr_xpr
        | _ -> false
      (* have to declare own function instance if global *)
      in let func_ptr, dec_structs, read_pstruct, run_func, temp_cnt =
        if (check_global xpr)
        then let fsname = "fi_" ^ string_of_int(temp_cnt)
          in let set_pstructs = if (String.length pstruct = 0) then "" else
            (c_func_call (func_ptr ^ "->init") [(point fsname)]) ^ ";\n"
          in let dec_structs = "struct function_instance " ^ fsname ^ ";\n" ^
            set_pstructs
          in let temp_cnt = temp_cnt + 1
```

```
          in let read_pstruct = (cast_ptr pstruct (fsname ^ ".params"))
          in let run_func = (c_func_call (func_ptr ^ "->function")
            [(point fsname)]) ^ ";\n"
          in (point fsname), dec_structs, read_pstruct, run_func, temp_cnt
        else let run_func = (c_func_call "run_funcvar" [func_var_ptr]) ^ ";\n"
         in func_ptr, "",
           (cast_ptr pstruct (cast_ptr func_inst_type (func_ptr)) ^
           "->params"), run_func, temp_cnt
      in let get_pstruct =
        if ((String.length pstruct) = 0) then "" else
        pstruct ^ " *" ^ pstruct_name ^ " = " ^
        read_pstruct ^ ";\n"
      (* set parameters *)
      in let _, set_params =
        let set_param (count, old_code) src src_type =
          let new_code = gen_copy_var
            (point (pstruct_name ^ "->p_" ^ string_of_int(count))) (point src)
            (List.length (snd src_type)) (fst src_type) default_env
          in count + 1, old_code ^ new_code
        in List.fold_left2 set_param (0, "") handles
          types
      (* value is in ret_field *)
      in let ret_field = func_ptr ^ "->ret_val"
      in ret_field, func_gen ^ dec_structs ^ get_pstruct ^ set_params ^
        run_func, temp_cnt, struct_tbl
  in ret_var, param_code ^ ";\n" ^ exec_code, temp_cnt, struct_tbl
(*
 * generate code for statement
 * (string) estruct: environment structure
 * (string) fname: function name
 * (int) temp_cnt: running count of temporary variables
 * (struct_table) struct_tbl: current struct declarations and map of
 * (xstatement): the statement from which to generate code
 * returns:
 *   code to execute the statement
 *   the updated temp_cnt
 *   the updated struct_tbl
 *)
and gen_stmt estruct fname temp_cnt struct_tbl =
  (*
   * generic loop code generation --implemented as C while loop
   * (xexpr option) cond_expr: expression for calculation condition. default to
   *                          1
   * (xstmt option) start: statement before the first iteration
   * (xstmt option) progress: statement at end of iteration
   * (xbody) body: loop body
   *)
  let gen_loop cond_expr start progress body =
    (*
     * declaration of condition variable
     * calculation of condition value
     * C expression that returns condition value
     * updated temp_cnt and struct_tbl
     *)
```

```
    let gen_check, calc_cond, get_cond, temp_cnt, struct_tbl = match cond_expr
      (* None expression defaults to 1, and no calculation needed *)
      with None -> "", "", "1", temp_cnt, struct_tbl
      (*
       * Some expression requires calculation before the while check, which
       * is done before entering the while loop, and at the end of each
       * iteration
       *)
      | Some(cond_expr) ->
        (*
         * declare temporary variable that is used for "while" check. We need
         * a single variable that can be accessed before and inside the loop
         * body, and in the while check
         *)
        let check_temp, gen_check = gen_temp temp_cnt
        in let temp_cnt = temp_cnt + 1
        (*
         * generate code for calculating condition result
         * this code is used before and at the end of the loop body, so
         * redeclaration is alright
         *)
        in let cond_temp, cond_is_temp, copy_cond, cond_gen, _, temp_cnt,
          struct_tbl = gen_expr cond_expr estruct fname temp_cnt struct_tbl
        (*
         * full condition value calculation also includes moving the value
         * to the common condition check variable
         *)
        in let calc_cond = cond_gen ^ (copy_cond check_temp)
        (*
         * the condition check variable is a variable of type bool, so we need
         * to use a C function to access it
         *)
        in let get_cond = (get_single_raw check_temp XFunkBool true)
        in gen_check, calc_cond, get_cond, temp_cnt, struct_tbl
    (* evaluate start and progression statements if they exist *)
    in let run_start, temp_cnt, struct_tbl = match start
      with None -> "", temp_cnt, struct_tbl
      | Some(xstmt) -> gen_stmt estruct fname temp_cnt struct_tbl xstmt
    in let run_progress, temp_cnt, struct_tbl = match progress
      with None -> "", temp_cnt, struct_tbl
      | Some(xstmt) -> gen_stmt estruct fname temp_cnt struct_tbl xstmt
    (* the loop body block code *)
    in let body_code, temp_cnt, struct_tbl = gen_block body estruct fname
      temp_cnt struct_tbl
    (* code before "while", outside the loop body *)
    in let pre_code = gen_check ^ run_start ^ calc_cond
    (* the C loop header *)
    in let while_header = "while (" ^ get_cond ^ ")"
    (*
     * each body includes loop body, as well as progress and
     * calculation of next condition
     *)
    in let full_body_code = body_code ^ run_progress ^ calc_cond
    in pre_code ^ while_header ^ (enclose full_body_code),
```

```
      temp_cnt, struct_tbl
  and gen_if_block cond_expr body =
    let check_temp, gc = gen_temp temp_cnt
    in let temp_cnt = temp_cnt + 1
    in let cond_temp, cond_is_temp, copy_cond, cond_gen, _, temp_cnt,
         struct_tbl = gen_expr cond_expr estruct fname temp_cnt struct_tbl
    in let calc_cond = cond_gen ^ (copy_cond check_temp)
    in let get_cond = get_single_raw check_temp XFunkBool true
    in let pre_code = gc ^ calc_cond
    in let if_header = "if (" ^ get_cond ^ ")"
    in let new_code, temp_cnt, struct_tbl = gen_block body estruct fname temp_cnt
                        struct_tbl
    in pre_code ^ if_header ^ (enclose new_code), temp_cnt, struct_tbl
  in function
  | XAssignment(xa) ->
    let assign_code, temp_cnt, struct_tbl = gen_assign xa estruct fname temp_cnt
                                           struct_tbl
    in assign_code, temp_cnt, struct_tbl
  | XFunctionCall(fc) ->
    let _, call_code, temp_cnt, struct_tbl = gen_func_call fc estruct fname
                                            temp_cnt struct_tbl
    in call_code, temp_cnt, struct_tbl
  | XBlock(blk) ->
    let new_code, temp_cnt, struct_tbl = gen_block blk estruct fname temp_cnt
                                        struct_tbl
    in enclose new_code, temp_cnt, struct_tbl
  | XForBlock(start, cond_expr, progress, body) ->
    gen_loop cond_expr start progress body
  | XIfBlock(cond_expr, body) -> gen_if_block cond_expr body
  | XIfElseBlock(cond_expr, ifbody, elsebody) ->
    let codei, temp_cnt, struct_tbl = gen_if_block cond_expr ifbody
    in let codee, temp_cnt, struct_tbl = gen_block elsebody estruct fname
                      temp_cnt struct_tbl
    in codei ^ "else" ^ (enclose codee), temp_cnt, struct_tbl
  | XWhileBlock(cond_expr, body) -> gen_loop (Some cond_expr) None None body
  | XBreak -> "break;", temp_cnt, struct_tbl
  | XReturn(rvo) ->
    (* if there is return value, copy it to ret_val of the function instance *)
    let put_out, temp_cnt, struct_tbl = match rvo
      with Some(rv) ->
        let out_var = "(&self->ret_val)"
        in let rcode, temp, rtype, temp_cnt, struct_tbl =
          gen_rvalue rv estruct fname temp_cnt struct_tbl
        in let copy_code = (gen_copy_var_wrap
                        out_var (point temp) rtype estruct)^ ";\n"
        in rcode ^ copy_code, temp_cnt, struct_tbl
      | None -> "", temp_cnt, struct_tbl
    in put_out ^ "goto RETURN;\n", temp_cnt, struct_tbl
(*
 * generate code for block
 * (xblock): a block that contains statements to turn into code, and variables
 *          to declare
 * (string) estruct: environment structure
 * (string) fname: function name
```

```
 * (int) temp_cnt: running count of temporary variables
 * (struct_table) struct_tbl: current struct declarations and map of param
 *                          structs
 * returns:
 *   code to execute the block
 *   the updated temp_cnt
 *   the updated struct_tbl
 *)
and gen_block xb estruct fname temp_cnt struct_tbl = List.fold_left
  (fun (old_code, old_cnt, old_tbl) stmt ->
    let new_code, new_cnt, new_tbl = gen_stmt estruct fname old_cnt old_tbl stmt
    in (old_code ^ new_code, new_cnt, new_tbl)
  )
  ("", temp_cnt, struct_tbl) xb.xstmts
(*
 * generates access to generates C expression to access array
 * (string) arr_ptr: variable of type struct var * that is an array
 * (xexpr) i_expr: integer expression
 * (string) estruct: environment structure
 * (string) fname: function name
 * (int) temp_cnt: running count of temporary variables
 * (struct_table) struct_tbl: current struct declarations and map of param
 *                          structs
 * returns
 *   array element, of type struct var
 *   updated temp_cnt
 *   updated struct_tbl
 *)
and gen_arr_access arr_ptr i_expr estruct fname temp_cnt struct_tbl =
  let i_val, i_is_temp, _, i_code, i_xvt, temp_cnt, struct_tbl =
    gen_expr i_expr estruct fname temp_cnt struct_tbl
  in let e_var = c_func_call "get_element" [arr_ptr ;
    (get_single_raw i_val (fst i_xvt) i_is_temp) ]
  in e_var, i_code, temp_cnt, struct_tbl
(*
 * generate code for assignment or assigning declaration
 * (xassignment): an assignment, that contains rvalues to evaluate, and
 *               lvalues to fill
 * (string) estruct: environment structure
 * (string) fname: function name
 * (int) temp_cnt: running count of temporary variables
 * (struct_table) struct_tbl: current struct declarations and map of
 * returns:
 *   code to execute the assignment
 *   the updated temp_cnt
 *   the updated struct_tbl
 *)
and gen_assign assign estruct fname temp_cnt struct_tbl =
  let rvalue_code, handles, types, temp_cnt, struct_tbl =
    gen_rvalues assign.rvals estruct fname temp_cnt struct_tbl
  in let rec get_lvalue (temp_cnt, struct_tbl) = function
    | XVariable(iobj) ->
      if ((is_global iobj) && fname != global_scope)
      then raise (Failure
```

```
                  ("You are attempting to change a global constant:\n" ^
                   iobj.name))
      else (point (get_name iobj estruct default_env)), "", temp_cnt, struct_tbl
    | XFunkArrExpr(xvt, a_expr, i_expr) ->
      let arr_var, arr_code, temp_cnt, struct_tbl = get_lvalue
                                                    (temp_cnt, struct_tbl)
                                                    a_expr
      in let e_var, arr_access, temp_cnt, struct_tbl = gen_arr_access arr_var
                                                       i_expr estruct fname
                                                       temp_cnt struct_tbl
      in e_var, arr_code ^ arr_access, temp_cnt, struct_tbl
    | _ -> raise (Failure "Invalid lvalue")
  in let lvars, lcode, temp_cnt, struct_tbl = List.fold_left
    (fun (old_vars, old_code, old_cnt, old_tbl) lexpr ->
     let new_var, new_code, temp_cnt, param_tbl = get_lvalue
                                                  (old_cnt, old_tbl) lexpr
     in new_var::old_vars, old_code ^ new_code, temp_cnt, struct_tbl)
     ([], "", temp_cnt, struct_tbl) assign.lvals
  in let lvars = List.rev lvars
  in let gen_pair_code lvar (rval, rvar) temp_cnt struct_tbl =
    match rval
      with XExprRVal(xvt, xpr) -> (match xpr
        with XFunkAsyncExpr(_) -> ((c_func_call "run_async_assign"
                                 [lvar ; rvar])) ^ ";\n",
                                 temp_cnt, struct_tbl
        | _ -> let direct_copy =
            gen_copy_var lvar rvar (List.length (snd xvt)) (fst xvt) default_env
          in direct_copy, temp_cnt, struct_tbl
      )
      | XFuncRVal(hdr, _) ->
        let ft = XFunkFunc(hdr)
        in let direct_copy =
          gen_copy_var lvar rvar 0 ft default_env
        in direct_copy, temp_cnt, struct_tbl
  in let rvals_vars =
    let rev = List.fold_left2 (fun old_list rval rvar ->
      (rval, point rvar)::old_list
    ) [] assign.rvals handles
    in List.rev rev
  in let pair_copies, temp_cnt, struct_tbl = List.fold_left2
    (fun (old_copies, old_cnt, old_tbl) lvar rva ->
     let new_copy, temp_cnt, struct_tbl = gen_pair_code lvar rva old_cnt
       struct_tbl
     in old_copies ^ new_copy, temp_cnt, struct_tbl)
    ("", temp_cnt, struct_tbl) lvars rvals_vars
  in let assign_code = rvalue_code ^ lcode ^ pair_copies
  in assign_code, temp_cnt, struct_tbl
(*
 * Generates C code for a function
 * (string) fname: the name for the function. If the function is anonymous,
 *       the name was declared automatically
 * (xfunc_header) fheader: the parameter and return header of the function
 * (xbody) fbody: the body of the function
 * (struct_table) struct_tbl: current struct declarations and map of
```

```
 *                             structures returns
 * returns:
 *   code that declares the env struct, if needed
 *   code that declares param struct, if needed
 *   C version of function
 *   (code for init function, its name)
 *   code for copy function
 *   updated struct_tbl
 *)
and gen_func fname fheader fbody struct_tbl =
  let func_initer, struct_tbl = gen_func_body fname fbody
    fheader.xparams struct_tbl
  in let pstruct_name, struct_tbl =
    get_param_struct fheader.xparams struct_tbl
  in func_initer, struct_tbl
(*
 * Wrapper function for generating code for global function
 * (xfuncdec) g_func: global function declaration, which contains all
 *                    the information that is needed for the function itself.
 *                    the main difference is that the C code function will
 *                    append "_global" to the name
 * (struct_table) struct_tbl: current struct declarations and map of
 *                             structures returns
 * returns:
 *   the static variable for the global function's instance
 *   the declaration of that static variable
 *   the function for initializing the function instance
 *   updated struct_tbl
 *)
and gen_global_func g_func struct_tbl =
  let global_name = g_func.xfid ^ "_global"
  in let global_inst = global_name ^ "_instance"
  in let global_inst_dec = "static " ^ func_inst_type ^ " " ^ global_inst ^
    ";\n"
  in let initer, struct_tbl =
    gen_func global_name g_func.xfunc_header g_func.xbody struct_tbl
  in global_inst, global_inst_dec, initer, struct_tbl
(*
 * defines .init member of function
 * (string) fname: internal name of function
 * (xblock) fbody: function body
 * (string) estruct_name: name of free-variable struct type
 * returns C function for initializing a function instance
 *)
and gen_init fname fbody estruct_name pstruct_name =
  let ni = "new_inst"
  in let ne = ni ^ "->scope"
  in let ni_memb = ni ^ "->"
  in let f_func, f_init, f_copy = get_inst_function fname, get_inst_init fname,
                                  get_inst_copy fname;
  in let inst_funcs = ni_memb ^ "function = " ^ f_func ^ ";\n" ^
                      ni_memb ^ "init = " ^ f_init ^ ";\n" ^
                      ni_memb ^ "copy = " ^ f_copy ^ ";\n"
  in let dec_dst = if ((String.length estruct_name) > 0)
```

```
    then estruct_name ^ " *dst_env = malloc(sizeof(" ^
      estruct_name ^ "));\n" ^
      ne ^ " = dst_env;\n"
    else ""
  in let dec_param = if ((String.length pstruct_name) > 0)
    then pstruct_name ^ " *params = malloc(sizeof(" ^
      pstruct_name ^ "));\n" ^
      ni ^ "->params = params;\n"
    else ""
  in let rec init_vars count old_code =
    let init_this = c_func_call "init_var"
      [(point (get_free_name ne count estruct_name))] ^ ";\n"
    in if count < (List.length fbody.need_copy)
      then init_vars (count + 1) (old_code ^ init_this)
      else old_code
  in let init_vars_code = init_vars 0 ""
  in let init_ret = (c_func_call "init_var" [point (ni ^ "->ret_val")]) ^ ";\n"
  in let fbody_code = inst_funcs ^ dec_dst ^ dec_param ^ init_vars_code ^
    init_ret
  in (wrap_c_func f_init fbody_code "void " [func_inst_type ^ " *" ^ ni],
    f_init)
(*
 * defines .copy member of function
 * (string) fname: internal name of function
 * (xblock) fbody: function body
 * (string) estruct_name: name of free-variable struct type
 * returns C function for copying a function instance
 *)
and gen_copy fname fbody estruct_name =
  let fbody_code = if (List.length fbody.need_copy > 0) then
    let dec_src = estruct_name ^ " *src_env = (" ^ estruct_name ^
      " *) src->scope;\n"
    in let dec_dst = estruct_name ^ " *dst_env = (" ^ estruct_name ^
      " *) dst->scope;\n"
    in let copy_vars, _ = List.fold_left
      (fun (old_code, i) (free_src, _) ->
        let copy_code = (gen_copy_var_wrap (point (get_free_name "dst_env" i
                                           estruct_name))
                         (point (get_free_name "src_env" i estruct_name))
                         free_src.obj_type estruct_name)
        in old_code ^ copy_code, i + 1
      ) ("", 0) fbody.need_copy
    in dec_src ^ dec_dst ^ copy_vars
    else ""
  in wrap_c_func (get_inst_copy fname) fbody_code "void "
    [func_inst_type ^ " *" ^ "dst" ; func_inst_type ^ " *" ^ "src"]


(*
 * main function for generating C code
 * (xprogram) prog: root of the SAST
 * returns the complete C code
 *)
let gen_prog prog globals =
```

```
let build_dec temp_cnt struct_tbl = function
  | XVardec(xa) ->
    let new_code, temp_cnt, struct_tbl = gen_assign xa "" global_scope
      temp_cnt struct_tbl
    in new_code, "", temp_cnt, { struct_decs = struct_tbl.struct_decs ;
                        prog_decs = struct_tbl.prog_decs ;
                        params = struct_tbl.params }
  | XFuncdec(xf) ->
    let global_inst, global_inst_dec, initer, struct_tbl =
      gen_global_func xf struct_tbl
    in let try_start = if xf.xfid = "main" then
      c_func_call (global_inst ^ ".function") ["&" ^ global_inst] ^ ";\n"
      else ""
    in let init_code, temp_cnt, struct_tbl = initer (point global_inst)
      (global_inst ^ "->env") temp_cnt struct_tbl
    in let copy_code = (get_global_name xf.global_id.var_id) ^
      ".val.ptr = " ^ (point global_inst) ^ ";\n"
    in init_code ^ copy_code, try_start, temp_cnt,
      { struct_decs = struct_tbl.struct_decs ^ global_inst_dec ;
        prog_decs = struct_tbl.prog_decs ; params = struct_tbl.params }
in let main_init, main_start, _, struct_tbl =
  List.fold_left
  (fun (old_main, old_start, old_cnt, old_tbl) dec ->
    let new_main, new_start, temp_cnt, struct_tbl =
      build_dec old_cnt old_tbl dec
    in old_main ^ new_main,
    old_start ^ new_start, temp_cnt, struct_tbl)
  ("", "", 0, { struct_decs = "" ; prog_decs = "" ;
                         params = ParamTable.empty}) prog
in let global_decs, global_inits, _ = List.fold_left
  (fun (old_dec, old_code, id) dec ->
    let bv_name = get_global_name id
    in let bv_dec = "struct var " ^ bv_name ^ ";\n"
    in let bv_init = (c_func_call "init_var" [point bv_name]) ^ ";\n"
    in (old_dec ^ bv_dec, old_code ^ bv_init, id + 1)
  ) ("", "", 0) globals
in "#include <complete.h>\n" ^
  struct_tbl.struct_decs ^
  global_decs ^
  struct_tbl.prog_decs ^
  "int main(void)\n" ^
  "{\n" ^
  global_inits ^ main_init ^ main_start ^
  "return 0;\n" ^
  "}"
```

Listing 35: Code generation

```
open Commponents
open Xcommponents

val gen_prog : xprogram -> id_object list -> string
```

Listing 36: Code generation

```c
/* The C output includes this file to use all types, macros and inlines */
#include <env.h>
#include <func.h>
#include <built_in.h>
```

Listing 37: C include files: complete.h

```c
/*
 * Runtime handling of variables in and between scopes
 */
#ifndef ENV_H
#define ENV_H

#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <pthread.h>

#include <funk_error.h>

struct function_instance;

union value {
    long int int_v;
    int bool_v;
    double double_v;
    char char_v;
    struct var *array; /* member objects */
    struct function_instance *ptr; /* function pointer */
};

struct var {
    size_t arr_size; /* number of elements */
    /* primitive value, or pointer to array or function */
    union value val;
    pthread_mutex_t await_lock;
    pthread_cond_t await_cond;
};

/*
 * Holds assignment pair for pthread function to handle
 */
struct async_assignment {
    struct var *lval;
    struct var *rval;
};

void *assign_async(void *data);

#define LOCK_VAR(var) pthread_mutex_lock(&(var)->await_lock)
#define UNLOCK_VAR(var) pthread_mutex_unlock(&(var)->await_lock)

static inline void run_async_assign(struct var *lval, struct var *rval)
{
```

```
    pthread_t t;
    struct async_assignment *aa = malloc(sizeof(struct async_assignment));

    aa->lval = lval;
    aa->rval = rval;

    //Lock a global lock
    LOCK_VAR(lval);
    if (pthread_create(&t, NULL, assign_async, aa)) {
        perror(ASYNC_ERROR);
        exit(-1);
    }
    pthread_join(t, NULL);
    UNLOCK_VAR(lval);
}

static inline void copy_primitive(struct var *dst, struct var *src)
{
    union value temp;

    LOCK_VAR(src);
    memcpy(&temp, &src->val, sizeof(dst->val));
    UNLOCK_VAR(src);

    LOCK_VAR(dst);
    memcpy(&dst->val, &temp, sizeof(dst->val));
    UNLOCK_VAR(dst);
}

static inline void init_var(struct var *new_var)
{
    memset(&new_var->val, 0, sizeof(union value));
    pthread_mutex_init(&new_var->await_lock, NULL);
    pthread_cond_init(&new_var->await_cond, NULL);
}

static inline void init_array(struct var *new_array, int size)
{
    int i;

    if (size <= 0) {
        fprintf(stderr, ERROR "Array size must be positive");
        exit(-1);
    }
    new_array->arr_size = (size_t) size;
    new_array->val.array = malloc(sizeof(struct var) * size);
    for (i = 0; i < size; i++) {
        init_var(&(new_array->val.array[i]));
    }
}

static inline void check_bounds(struct var *array, int index)
{
    if (array->val.array == NULL) {
```

```
        fprintf(stderr, ERROR
            "Array has not been initialized\n");
        exit(-1);
    }
    if (index >= array->arr_size || index < 0) {
        fprintf(stderr, ERROR
            "Array out of bounds: Size is %d, but %d was requested",
            (int) array->arr_size, index);
        exit(-1);
    }
}

static inline struct var *get_element(struct var *array, int index)
{
    check_bounds(array, index);
    return &array->val.array[index];
}

static inline void shallow_copy(struct var *dst, struct var *src)
{
    union value temp;
    size_t arr_size;

    LOCK_VAR(src);
    memcpy(&temp, &src->val, sizeof(src->val));
    arr_size = src->arr_size;
    UNLOCK_VAR(src);

    LOCK_VAR(dst);
    memcpy(&dst->val, &temp, sizeof(dst->val));
    dst->arr_size = arr_size;
    UNLOCK_VAR(dst);
}

static inline void _shallow_copy(struct var *dst, struct var *src)
{
    union value temp;
    size_t arr_size;

    memcpy(&temp, &src->val, sizeof(src->val));
    arr_size = src->arr_size;

    memcpy(&dst->val, &temp, sizeof(dst->val));
    dst->arr_size = arr_size;
}

static inline void set_element(struct var *out, int index,
                    struct var *new_member)
{
    struct var *out_slot = get_element(out, index);

    shallow_copy(out_slot, new_member);
}

```

```c
/* getting primitives */
static inline int get_int(struct var *int_var)
{
    int int_val;
    LOCK_VAR(int_var);
    int_val = (int_var)->val.int_v;
    UNLOCK_VAR(int_var);
    return int_val;
}

static inline int get_bool(struct var *bool_var)
{
    int bool_val;
    LOCK_VAR(bool_var);
    bool_val = (bool_var)->val.bool_v;
    UNLOCK_VAR(bool_var);
    return bool_val;
}

static inline double get_double(struct var *double_var)
{
    double double_val;
    LOCK_VAR(double_var);
    double_val = (double_var)->val.double_v;
    UNLOCK_VAR(double_var);
    return double_val;
}

static inline char get_char(struct var *char_var)
{
    char char_val;
    LOCK_VAR(char_var);
    char_val = (char_var)->val.char_v;
    UNLOCK_VAR(char_var);
    return char_val;
}

/* setting primitives */
static inline void set_int(struct var *int_var, int int_val)
{
    LOCK_VAR(int_var);
    (int_var)->val.int_v = int_val;
    UNLOCK_VAR(int_var);
}

static inline void set_bool(struct var *bool_var, int bool_val)
{
    LOCK_VAR(bool_var);
    (bool_var)->val.bool_v = bool_val;
    UNLOCK_VAR(bool_var);
}

static inline void set_double(struct var *double_var, double double_val)
{
```

```c
    LOCK_VAR(double_var);
    (double_var)->val.double_v = double_val;
    UNLOCK_VAR(double_var);
}

static inline void set_char(struct var *char_var, char char_val)
{
    LOCK_VAR(char_var);
    (char_var)->val.char_v = char_val;
    UNLOCK_VAR(char_var);
}

#endif /* ENV_H */
```

Listing 38: C include files: env.h

```c
/*
 * Runtime handling of function instances
 */
#ifndef FUNC_H
#define FUNC_H

#include <funk_error.h>
#include <env.h>

/* function, with pointers to parameters, in-scope functions, return values */
struct function_instance {
    void *params; /* explicit parameters given in Funk language */
    void *scope; /* variables manipulated inside scope of function */
    struct var ret_val; /* placeholder for return values */
    /*
     * backend implementation of function in Funk. Note that it takes in
     * all the previous members of this struct
     * self: void form of this function_instance
     */
    void *(*function)(void *self);
    /* automatically generated function for initializing environments */
    void (*init)(struct function_instance *new_inst);
    /* automatically generated function for copying function instances */
    void (*copy)(struct function_instance *dst,
            struct function_instance *src);
};

struct async_data {
    struct var *out;
    struct function_instance *async_block;
};

void *exec_async_block(void *data);

static inline void run_async(struct var *out,
                struct function_instance *async_block)
{
    pthread_t t;
    struct async_data *ad = malloc(sizeof(struct async_data));
```

```
    ad->out = out;
    ad->async_block = async_block;

    LOCK_VAR(out);
    if (pthread_create(&t, NULL, exec_async_block, ad)) {
        perror(ASYNC_ERROR);
        exit(-1);
    }
}

static inline void check_funcvar(struct var *func_var)
{
    if (func_var->val.ptr == NULL) {
        fprintf(stderr, ERROR "Using uninitialized function value\n");
        exit(-1);
    }
}

static inline void run_funcvar(struct var *func_var)
{
    check_funcvar(func_var);

    LOCK_VAR(func_var);
    func_var->val.ptr->function(func_var->val.ptr);
    UNLOCK_VAR(func_var);
}

static inline void copy_funcvar(struct var *dst, struct var *src)
{
    struct function_instance *inst;

    check_funcvar(src);
    inst = malloc(sizeof(struct function_instance));

    LOCK_VAR(src);
    src->val.ptr->init(inst);
    src->val.ptr->copy(inst, src->val.ptr);
    UNLOCK_VAR(src);

    LOCK_VAR(dst);
    dst->val.ptr = inst;
    UNLOCK_VAR(dst);

}

#define SETUP_FUNC_0(inst_ptr, base_name) \
    base_name##_init(inst_ptr);
#define SETUP_FUNC(inst_ptr, base_name) \
    (inst_ptr)->scope = malloc(sizeof(struct base_name##_env)); \
    SETUP_FUNC_0(inst_ptr, base_name)
#define CREATE_FUNC(inst_ptr, base_name) \
    inst_ptr = malloc(sizeof(struct function_instance)); \
    SETUP_FUNC(inst_ptr, base_name)
```

```c
#define CREATE_FUNC_0(inst_ptr, base_name) \
    inst_ptr = malloc(sizeof(struct function_instance)); \
    SETUP_FUNC_0(inst_ptr, base_name)

#endif /* FUNC_H */
```

Listing 39: C include files: func.h

```c
/* C-support for built-in functions */
#ifndef BUILT_IN_H
#define BUILT_IN_H

/* printing primitives */
#define PRINT_INT(int_var) printf("%ld", (int_var)->val.int_v)
#define PRINT_BOOL(bool_var) printf((bool_var)->val.bool_v ? "true" : "false")
#define PRINT_DOUBLE(double_var) printf("%f", (double_var)->val.double_v)
#define PRINT_CHAR(char_var) printf("%c", (char_var)->val.char_v)

#endif /* BUILT_IN_H */
```

Listing 40: C include files: built_in.h

```c
/*
 * Error handling macros
 */
#ifndef FUNK_ERROR_H
#define FUNK_ERROR_H

#define ERROR "Error: "
#define ASYNC_ERROR ERROR "Could not execute async block"

#endif /* FUNK_ERROR_H */
```

Listing 41: C include files: funk_error.h

```c
#include <env.h>

void *assign_async(void *data)
{
    struct async_assignment *assignment = (struct async_assignment *) data;
    struct var *lval = assignment->lval, *rval = assignment->rval;

    LOCK_VAR(rval);
    _shallow_copy(lval, rval);
    UNLOCK_VAR(rval);

    free(data);

    return NULL;
}
```

Listing 42: C libraries: env.c

```c
#include <func.h>

```

```c
void *exec_async_block(void *data)
{
    struct async_data *ad = (struct async_data *) data;
    struct var *out = ad->out;
    struct function_instance *async_block = ad->async_block;

    async_block->function((void *) async_block);
    _shallow_copy(out, &async_block->ret_val);
    UNLOCK_VAR(out);
    free(data);

    return NULL;
}
```

Listing 43: C libraries: func.c