Dice Project Report 

"Java, but worse"

Project Manager:	**David Watkins**	**djw2146**
Language Guru:	**Emily Chen**	**ec2805**
System Architect:	**Phillip Schiffrin**	**pjs2186**
Tester & Verifier:	**Khaled Atef**	**kaa2168**

Contents

1. Introduction

The Dice programming language is an object-oriented, general purpose programming language. It is designed to let programmers who are more familiar with object oriented programming languages to feel comfortable with common design patterns to build useful applications. The syntax of Dice resembles the Java programming language. Dice compiles down to LLVM IR which is a cross-platform runtime environment. This allows Dice code to work on any system as long as there is an LLVM port for it, which includes Windows, Mac OS X, and Linux [1].

Dice lays programs out the same way a Java program would. Variables and methods of a class can be declared with private scope. There is a simple to use inheritance that allows for multiple children inheriting the fields and methods of its parent. Dice also allows for convenient use of functions that exist in C, such as malloc, open, and write. This allows the user to construct objects and call c functions using those objects.

Background

Object-oriented programming (OOP) is a programming paradigm based on the concept of "objects". These objects are data structures that contain data, in the form of fields, often known as attributes. The code itself are contained within methods in the code which are compiled to varying subroutines. The most useful aspect of OOP is that these methods and fields can modify one another allowing for a rich and varied use case.

Class based OOP specifically creates instances of classes, referred to as objects, which have their values modified at runtime. There are many languages that implement their language this way including Java and C#.

Inheritance is when an object or class is based on another class using the same implementation. This allows for a class to serve as a blueprint for subclasses. Polymorphism allows an object to take on many forms. This may include an object being assigned to a type that is a class it inherits from, or being used in place of a class it inherits from.

We want to leverage these capabilities using LLVM code to produce a syntactically Java-like language but offer a cross platform solution that is simple and easy to use. Implementing inheritance and objects in a c-like context like LLVM allows for fine control over the code.

Related Work

Object-oriented programming languages have existed since the late 20th century. Java, C#, C++, Objective-C, Python, and many more languages have facilities for defining custom user classes and manipulating them at runtime.

[1] http://llvm.org/

Implementing an object-oriented paradigm using C is a well-known solution, but compiling object-oriented code down to LLVM is not publicly available. We want to contribute to the LLVM community by adding additional information regarding the creation of a compiler using OCaml that compiles to LLVM code.

Goals

Cross-Platform

Utilizing the LLVM IR we are able to compile the source once and have it work on multiple architectures without fail.

Flexibility

Allowing the user to define their own classes and offering them the ability to inherit functionality from other user defined types offer a wide range of possibilities for their programs and also saves the user time when implementing large programs.

Transparency

Using the LLVM IR allows the user to see exactly what the program is doing after the compiler is done. For a more optimal result it can then be compiled to bitcode representation using the LLVM compiler.

2. Language Tutorial

Using the Compiler

Defining methods

Control Flow

Defining custom classes

Using Inheritance

3. Language Reference Manual

Introduction

Dice is a general purpose, object-oriented programming language. The principal is simplicity, pulling many themes of the language from Java. Dice is a high level language that utilizes LLVM IR to abstract away hardware implementation of code. Utilizing the LLVM as a backend allows for automatic garbage collection of variables as well.
Dice is a strongly typed programming language, meaning that at compile time the language will be type-checked, thus preventing runtime errors of type.
This language reference manual is organized as follows:

- Chapter 2 Describes types, values, and variables, subdivided into primitive types and reference types
- Chapter 3 Describes the lexical structure of Dice, based on Java. The language is written in the ASCII character set
- Chapter 4 Describes the expressions and operators that are available to be used in the language
- Chapter 5 Describes different statements and how to invoke them
- Chapter 6 Describes the structure of a program and how to determine scope
- Chapter 7 Describes classes, how they are defined, fields of classes or their variables, and their methods
- Chapter 8 Discusses the different library classes provided with the compiler and their definitions

The syntax of the language is meant to be reminescent of Java, thereby allowing ease of use for the programmer.

Types

There are two kinds of types in the Dice programming language: primitive types and non-primitive types. There are, correspondingly, two kinds of data values that can be stored in variables, passed as arguments, returned by methods, and operated on: primitive values and non-primitive values.

```
Type:
    PrimitiveType
    NonprimitiveType
```

There is also a special null type, the type of the expression *null*, which has no name. Because the null type has no name, it is impossible to declare a variable of the null type. The null reference is the only possible value of an expression of null type. The null reference can always undergo a widening reference conversion to any reference type. In practice, the programmer can ignore the null type and just pretend that *null* is merely a special literal that can be of any reference type.

Primitive Types and Values

A primitive type is predefined by the Dice programming language and named by its reserved keyword.

```
PrimitiveType:
    NumericType
    bool
NumericType:
    IntegralType
    float
IntegralType: one of
    int char
```

int

A value of type *int* is stored as a 32-bit signed two's-complement integer. The *int* type can hold values ranging from -2,147,483,648 to 2,147,483,647, inclusive.

float

The float type stores the given value in 64 bits. The *float* type can hold values ranging from 1e-37 to 1e37. Since all values are represented in binary, certain floating point values must be approximated.

char

The *char* data type is a 8-bit ASCII character. A *char* value maps to an integral ASCII code. The decimal values 0 through 31, and 127, represent non-printable control characters. All other characters can be printed by the computer, i.e. displayed on the screen or printed on printers, and are called printable characters. The character 'A' has the code value of 65, 'B' has the value 66, and so on. The ASCII values of letters 'A' through 'Z' are in a contiguous increasing numeric sequence. The values of the lower case letters 'a' through 'z' are also in a contiguous increasing sequence starting at the code value 97. Similarly, the digit symbol characters '0' through '9' are also in an increasing contiguous sequence starting at the code value 48.

bool

A variable of type *bool* can take one of two values, *true* or *false*. A bool could also be *null*.

Non-Primitive Types

Non-primitive types include arrays and classes.

Arrays

An array stores one or more values of the same type contiguously in memory. The type of an array can be any primitive or an array type. This allows the creation of an n-dimensional array, the members of which can be accessed by first indexing to the desired element of the outermost array, which is of type *array*, and then accessing into the desired element of the immediately nested array, and continuing n-1 times.

Classes

Classes are user-defined types. See chapter 7 to learn about the usage of objects.

Casting

Casting is not supported in this language. There are interesting behaviors between ints and float defined in the section on operators that imitate casting, but there is no syntax to support casting between types directly.

Lexical Conventions

This chapter describes the lexical elements that make up Dice source code. These elements are called tokens. There are six types of tokens: identifiers, keywords, literals, separators, and operators. White space, sometimes required to separate tokens, is also described in this chapter.

Identifiers

Identifiers are sequences of characters used for naming variables, functions and new data types. Valid identifier characters include ASCII letters, decimal digits, and the underscore character '_'. The first character must be alphabetic.
An identifier cannot have the same spelling (character sequence) as a keyword, boolean or null literal, a compile-time error occurs. Lowercase letters and uppercase letters are distinct, such that foo and Foo are two different identifiers.

```
ID = "['a'-'z' 'A'-'Z'](['a'-'z' 'A'-'Z']|['0'-'9']|'\textunderscore')*"
```

Keywords

Keywords are special identifiers reserved for use as part of the programming language itself. You cannot use them for any other purpose. Dice recognizes the following keywords:

if	else	for	while	
break	continue	return		
int	float	bool	char	void
null	true	false	class	constructor
public	private	extends	include	this

Literals

A literal is the source code representation of a value of a primitive type or the null type.

Integer Literals

An integer literal is expressed in decimal (base 10). It is represented with either the single ASCII digit 0, representing the integer zero, or an ASCII digit from 1 to 9 optionally followed by one or more ASCII digits from 0 to 9.

```
INT = "['0'-'9']+"
```

Float Literals

A float literal has the following parts: an integer part, a decimal point (represented by an ASCII period character), and a fraction part. The integer and fraction parts are defined by a single digit 0 or one digit from 1-9 followed by more ASCII digits from 0 to 9.

```
FLOAT = "['0'-'9']+ ['.'] ['0'-'9']+"
```

Boolean Literals

The boolean type has two values, represented by the boolean literals true and false, formed from ASCII letters.

```
BOOL = "true|false"
```

Character Literals

A character literal is always of type *char*, and is formed by an ascii character appearing between two single quotes. The following characters are represented with an escape sequence, which consists of a backslash and another character:

- '\\' - backslash
- '\"' - double-quote
- '\'' - single-quote
- '\n' - newline
- '\r' - carriage return
- '\t' - tab character

It is a compile-time error for the character following the character literal to be other than a single-quote character '.

```
CHAR = "\' ( ([' '-'!' '#'-'[' ']'-'~'] | '\\' [ '\\' '\"' 'n' 'r' 't' ]) )\' "
```

String Literals

A string literal is always of type char[] and is initialized with zero or more characters or escape sequences enclosed in double quotes.

```
char[] x = "abcdef\n";
```

```
STRING = "\"( ([' '-'!' '#'-'[' ']'-'~'] | '\\' [ '\\' '\"' 'n' 'r' 't' ]) )*\""
```

Separators

A separator separates tokens. White space is a separator but it is not a token. The other separators are all single-character tokens themselves: ( ) [ ] ; , .

```
'('     { LPAREN }
')'     { RPAREN }
'{'     { LBRACE }
'}'     { RBRACE }
';'     { SEMI }
','     { COMMA }
'['     { LBRACKET }
']'     { RBRACKET }
'.'     { DOT }
```

Operators

The following operators are reserved lexical elements in the language. See the expression and operators section for more detail on their defined behavior.

```
+    -    *    /    =
==   !=   <    <=   >
>=
```

White Space

White space refers to one or more of the following characters:

- the ASCII SP character, also known as "space"
- the ASCII HT character, also known as "horizontal tab"
- the ASCII FF character, also known as "form feed"
- LineTerminator

White space is ignored, except when it is used to separate tokens. Aside from its use in separating tokens, it is optional. Hence, the following two snippets of source code are equivalent.

```
public int foo()
{
        print( "hello, world\n" );
        return 0;
}

public int foo(){print("hello, world\n"); return 0;}
```

```
WHITESPACE = "[' ' '\t' '\r' '\n']"
```

Comments

The characters (∗ introduce a comment, which terminates with the characters ∗). Multiline comments can be distinguished from code by preceding each line of the comment with a * similar to the following:

```
(* This is a long comment
* that spans multiple lines because
* there is a lot to say. *)
```

```
COMMENT = "(\* [^ \*)]* \*)"
```

Expressions and Operators

The precedence of expression operators is the same as the order of the major subsections of this section (highest precedence first). Within each subsection, the operators have the same precedence. Left- or right-associativity is specified in each subsection for the operators discussed therein.

Primary Expressions

Primary expressions involving . , subscripting, and function calls group left to right.

Identifier

An identifier is a primary expression, provided it has been suitably declared as discussed below. Its type is specified by its declaration.

Literal

Any of the literal types discussed in Chapter 3 is a primary expression, which evaluates to the type of the literal.

(expression)

A parenthesized expression is a primary expression whose type and value are identical to those of the unadorned expression. The presence of parentheses does not affect whether the expression is an lvalue.

primary-expression [expression]

A primary expression followed by an expression in square brackets is a primary expression. The intuitive meaning is that of a subscript. The primary expression has type *array* of . . . and the type of the result is . . . . The type of the subscript expression must be a type that is convertible to an integral type, or a compile-time error occurs.

primary-expression (expression-list-opt)

A function call is a primary expression followed by parentheses containing a possibly empty, comma-separated list of expressions which constitute the actual arguments to the function. The result of the function call is the function's return type. Recursive calls to any function are permissible.

primary-lvalue . member-of-structure

An lvalue expression followed by a dot followed by the name of a class member is a primary expression. The object referred to by the lvalue is assumed to be an instance of the class defining the class member. The given lvalue can be an instance of any user-defined class.

Unary operators

Expressions with unary operators group right-to-left.

expression

The result is the negative of the expression, and has the same type. The type of the expression must be *char*, *int*, or $float$.

not expression

The result of the logical negation operator *not* is *true* if the value of the expression is $false$, $false$ if the value of the expression is *true*. The type of the result is *bool*. This operator is applicable only to operands that evaluate to *bool*.

Multiplicative operators

The multiplicative operators * and / group left-to-right.

expression * expression

The binary * operator indicates multiplication. Operands of *int*, $float$, and *char* types are allowed. If both operands are of type ..., the result is type .... If the operands are of two different types of the ones listed above, the result is the type of the left-most operand.

expression / expression

The binary / operator indicates division. The same type considerations as for multiplication apply.

Additive operators

The additive operators + and group left-to-right.

expression + expression

The value of the result is the sum of the expressions. The same type considerations as for multiplication apply. Overflow of a *char* type during an addition operation results in wraparound.

expression - expression

The value of the result is the difference of the expressions. The same type considerations as for multiplication apply.

Relational operators

The relational operators group left-to-right.

expression < expression

expression > expression

expression <= expression

expression >= expression

The operators < (less than), > (greater than), <= (less than or equal to) and >= (greater than or equal to) all yield *true* is the specified relation is true and $false$ otherwise. The same type considerations as for multiplication apply.

Equality operators

expression == expression

expression != expression

The == (equal to) and the != (not equal to) operators are exactly analogous to the relational operators except for their lower precedence.

Logical operators

expression and expression

Both operands must evaluate to a value of type *bool*. The *and* operator returns *true* if both its operands evaluate to *true*, $false$ otherwise. The second expression is not evaluated if the first evaluates to $false$.

expression or expression

Both operands must evaluate to a value of type *bool*. The *or* operator returns *true* if either of its operands evaluate to *true*, and *false* otherwise. The second operand is not evaluated if the value of the first operand evaluates to *true*.

Assignment operators

lvalue = expression

The value of the expression replaces that of the object referred to by the lvalue. Both operands must have the same type.

Statements

A statement forms a complete unit of execution.

Include Statement

If a .dice file contains a statement of the following form:

```
include(mylib)
```

then all classes defined in *mylib* are available to be used in definitions of classes in the .dice file in which the include statement appears.

Expression Statements

An expression statement consists of an expression followed by a semicolon. The execution of such a statement causes the associated expression to be evaluated. The following types of expressions can be made into a statement by terminating the expression with a semicolon (;):

```
(* Assignment expressions *)
aValue = 8933.234;
(* Method invocations *)
game.updateScore(Player1, 5);
(* Object creation expressions *)
Bicycle myBike = Bicycle();
```

Declaration Statements

A declaration statement declares a variable by specifying its data type and name.

```
float aValue;
```

Control Flow Statements

The statements inside source files are generally executed from top to bottom, in the order that they appear. Control flow statements, however, break up the flow of execution by employing decision making, looping, and branching, enabling your program to conditionally execute particular blocks of code. This section describes the decision-making statements (if-then, if-then-else), the looping statements (for, while), and the branching statements (break, continue, return) supported by the Dice programming language.

if-then, if-then-else

The 'if-then' statement tells the program to execute a certain section of code only if a particular test evaluates to true. The conditional expression that is evaluated is enclosed in balanced parentheses. The section of code that is conditionally executed is specified as a sequence of statements enclosed in balanced braces. If the conditional expression evaluates to false, control jumps to the end of the if-then statement.

```
if (condition) {
        <stmt>
}

if (not condition) {
        <stmt>
} (* if statement is skipped *)
```

The 'if-then-else' statement provides an alternate path of execution when "if" clause evaluates to false. This alternate path of execution is denoted by a sequence of statements enclosed in balanced braces, in the same format as the path of execution to take if the conditional evaluates to true, prefixed by the keyword "else".

```
if (condition) {
        <stmt>
} else {
        <stmt2>
} (* <stmt2> executed when not condition *)
```

Looping: for, while

The 'for' statement allows the programmer the iterate over a range of values. The 'for' statement has the following format:

```
for (initialization; termination; update) { <stmt> }
```

- The 'initialization' expression initializes the loop counter. It is executed once at the beginning of the 'for' statement
- When the 'termination' expression evaluates to false, the loop terminates.
- The 'update' expression is invoked after each iteration and can either increment or decrement the value of the loop counter.

The following example uses a 'for' statement to print the numbers from 1 to 10:

```
int loopCounter;
for (loopCounter=1; loopCounter<11; loopCounter++) {
        print(loopCounter);
}
```

The 'while' statement executes a user-defined block of statements as long as a particular conditional expression evaluates to true. The syntax of a 'while' statement is:

```
while (expression) {
        <stmt>
}
```

The following example uses a 'while' statement to print the numbers from 1 to 10:

```
int loopCounter;
loopCounter = 1;
while (loopCounter < 11) {
        print(loopCounter);
        loopCounter = loopCounter + 1;
}
```

Branching: break, continue, return

If a 'break' statement is included within either a 'for' or 'while' statement, then it terminates execution of the innermost looping statement it is nested within. All break statements have the same syntax:

```
break;
```

In the following example, the 'break' statement terminates execution of the inner 'while' statement and does not prevent the 'for' statement from executing its block of statements for all iterations of i from 1 to 10. This results in the the values of j from 100 to 110 being printed, in each of the 10 iterations of the 'for' loop.

```
int i;
int j;
for (i=1; i<11; i++) {
        j = 100;
        while (j<120) {
                if (j>110) {
                        break;
                }
                print(j);
                j = j + 1;
        }
}
```

In the following example, the 'break' statement terminates execution of the inner 'for' statement and does not prevent the 'while' statement from executing its block of statements for all iterations of i from 1 to 1000. This results in the the values of j from 100 to 110 being printed, in each of the 1000 iterations of the 'while' loop.

```
int i;
int j;

i = 1;
while (i<1001) {
        for (j=100; j<120; j++) {
                if (j>110) {
                        break;
                }
        }
        i = i + 1;
}
```

The continue statement skips the current iteration of a 'for' or 'while' statement, causing the flow of execution to skip to the end of the innermost loop's body and evaluate the conditional expression that controls the loop. The following example uses a 'continue' statement within a 'for' loop to print only the odd integers

between 1 and 10. The code prints "hello" 1000 times and on each of the 1000 'while' loop iterations, prints the odd integers.

```
int i;
int counter;
counter = 1;
while (counter < 1001) {
        print("hello");
        for (i=1; i<11; i++) {
                if (i - 2*(i/2) == 0) {
                        continue;
                } else {
                        print(i);
                }
        }
        counter = counter + 1;
}
```

The 'return' statement exits from the current method, and control flow returns to where the method was invoked. To return a value, simply put the value (or an expression that calculates the value) after the return keyword:

```
return count + 4;
```

The data type of the returned value must match the type of the method's declared return value. When a method is declared void, either no return statement is needed or the following 'return' statement is used:

```
return;
```

Blocks

A block is a group of zero or more statements between balanced braces and can be used anywhere a single statement is allowed. The following example, BlockDemo, illustrates the use of blocks:

```
class BlockDemo {
        public void main(char[,] args) {
                bool condition;
                condition = true;
                if (condition) { (* begin block 1 *)
                        print("Condition is true.");
                } (* end block one *)
                else { (* begin block 2 *)
                        print("Condition is false.");
                } (* end block 2 *)
        }
}
```

Dice Functions

There are several reserved functions in Dice that cannot be overridden and follow a particular syntax and return type.

File I/O

Manipulating files is an important aspect of any programming languages. Open files are denoted by a particular $intfd$; that can be used to read or write from a file. A file must be closed by the end of a program or else undefined behavior may occur.

int fopen(char[] filename, bool isWriteEnabled)

Accepts a filename and a flag to determine whether the file will be written to. If the file exists, it will be opened in append mode, otherwise a new file will be created. If it is in read mode, it will return a file descriptor as normal, or if the file doesn't exist will return '-1'. Likewise for write enabled, if there is an error it will return -1.

```
int fd;
fd = fopen("hello.txt", false);
```

bool fwrite(int fd, char[] values, int num, int offset)

Accepts an array of values to be written to a file, the number of characters it should write, and the offset into the value array it should write from. If there is an error, returns false, otherwise returns true.

```
bool success;
success = fwrite(fd, "This should work", 4, 1); (* Writes "his " to a file *)
```

bool fread(int fd, char[] storage, int num)

Accepts an array to store values from the file that are to be read, and will read in num bytes. Returns true on success and false on error.

```
char[] a;
bool success;

a = char[100];
success = fread(fd, a, 20);
```

bool fclose(int fd)

Closes a file. Returns true on success, false on error.

```
bool success;
success = fclose(fd);
```

Reading and Writing from Console

Reading and writing to the console is defined by two simple to use functions that cannot be overriden.

void print(char[] string)

Accepts a char array and prints the string to the console.

```
print("hello world");
```

void print(int num)

Accepts an int and prints the int to the console.

```
print(1);
```

void input(char[] buf)

Accepts a buf that will hold read bytes from the console. Then it will write those bytes to the array passed. Terminates when a user enters a newline or an EOF.

```
char[] a;
a = char[100];
input(a);
```

Program Structure and Scope

Program structure and scope define what variables are accessible and where. When inside a class, there are many different cases of scope, however those are better defined in chapter 7.

Program Structure

A Dice program may exist either within one source file or spread among multiple files which can be linked at compile-time. An example of such a linked file is the standard library, or *stdlib.dice*. When an include statement is executed at compile time, it will load in the files mentioned at the includes and insert the code at that location as if it were part of the head source file. Therefore at compilation, one only needs to compile with *dicecmaster.dice*. If an included module defines a class that has the same name as one of the classes defined in the including module, then the compiler throws an error. The compiler does not resolve recursive includes; if *foo.dice* includes *bar.dice* and *bar.dice* includes *foo.dice*, the compiler throws an error.
A program consists of zero or more include statements, followed by one or more class definitions. Each class defined in a module must have a distinct name. Only one class out of all classes may have a main method, defined with *public void main(char[][] args)* which designates the entry point for a program to begin executing code. All Dice files are expected to end with the file extension *.dice* and follow the following syntactic layout.

```
include(stdlib)
include(mylib)

class FOO {

        (* my code *)

}

class BAR {

        (* my code *)

        public void main(char[][] args)

}
```

Scope

Scope refers to which variables, methods, and classes are available at any given time in the program. All classes are available to all other classes regardless of their relative position in a program or library. Variable

scope falls into two categories: fields (instance variables) which are defined at the top of a class, and local variables, which are defined within a method. Fields can be public or private. If a field is public then it is accessible whenever an instance of that class is instantiated. For instance, if I have a class X, then class Y can be defined as follows:

```
class Y {

        public int num;

        constructor() {

                X myObj;
                myObj = X();
                this.num = myObj.number;
        }
}

class X {

        public int number;

}
```

In this example, class Y has one field which is an int. In its constructor, an instance of class X is declared, and a public field within that object is used to set the value for the given int. If a field is declared private, however, it can only be accessed by the methods in the same class. For example, if there is a class Y with a private field, the following is valid:

```
class Y {

        private int num;

        constructor() {

                this.num = 5;
        }

        private int getNum() {

                return this.num;
        }
}
```

However, if I have a class X, that class cannot access the private field within Y. The following is invalid:

```
class X {

        public int number;

        constructor() {
```

```
                Y myObj;
                myObj = Y();
                (* This code is invalid since num is a private field within Y *)
                this.number = myObj.num;
        }
}
```

Methods are also declared as public or private, and their accessibility is the same as fields. They must have a scope defined on them.

Local variables are variables that are declared inside of a method. Local variables are only accessible within the same method in which they are declared, and they may have the same name as fields within the same class since fields in a class are only accessible by calling the *this* keyword.

Classes

Classes are the constructs whereby a programmer defines their own types. All state changes in a Dice program must happen in the context of changes in state maintained by an object that is an instance of a user-defined class.

Class definition

A class definition starts with the keyword 'class' followed by the class name (see identifiers in chapter 2) and the class body. The class body, enclosed by a pair of curly braces, declares one or more of each of the following: fields, methods, and constructors.
The members of a class type are all of the following:

- Members inherited from its ancestors (its direct superclass and its ancestors)
- Members declared in the body of the class, with the exception of constructors

Access modifiers

Class member declarations must include access modifiers but the class declaration itself does not; there is no notion of a private class in Dice. Field and method declarations must include one of the access modifiers: *public* or *private*. Fields and methods with the access modifier *public* can be accessed by methods defined in any class. Fields and methods with the access modifier *private* can be accessed by methods defined either in the same class or in successor classes (classes derived directly from that class and their successors).

Fields

The only fields that can be declared are instance variables, which are freshly incarnated for each instance of the class. Field declarations have the following format:

```
<access modifier> <type> <VariableDeclaratorId>;
(* Example *) private int myInstanceVariable;
```

All instance variables must be declared before methods and constructors.

Methods

A method declares executable code that can be invoked, passing a fixed number of values as arguments. The only methods that can be declared are the 'main' method and instance methods. Instance methods are invoked with respect to some particular object that is an instance of a class type.

Method declarations constitute a method header followed by a method body. The method header has the following format:

```
<access modifier> <return type> <method name> <comma-separated list of parameters>
(* Example *) public double amountPaid(double wage, int duration)
```

The method body contains, enclosed between the ASCII characters '{' and '}', zero or more variable declarations followed by zero or more statements. If the type of the return value is not void, then the method body must include a return statement.

One and only one of the classes to be compiled must contain a definition for a method named "main" that executes when the program runs. The *main* method is not callable as an instance method. The *main* method must have a void return type and accept a single parameter of type char[][]. Hence, its signature must be:

```
public void main (char[][] args)
```

If either zero or more than one class contains a definition for a method with the signature above, this results in a compile-time error.

Methods can be overloaded: If two methods of a class (whether both declared in the same class, or both inherited by a class, or one declared and one inherited) have the same name but signatures that are not equivalent, then the method name is said to be overloaded. There can be multiple methods with the same name defined for a class, as long as each has a different number and/or type of parameters. The *main* method can never be overloaded because it has one and only one accepted signature. If two methods in the same class have the same signature, the compiler throws an error.

Constructors

Constructors are similar to methods but cannot be invoked as an instance method; they are used to initialize new class instances. A constructor has no return type and its formal parameters are identical in syntax and semantics to those of a method. A constructor definition has the following format:

```
constructor (<comma-separated formal arguments>) {
        <list of variable declarations>
        <list of statements>
}
(* Example *) constructor (int a, char[] b) {...}
```

Unlike fields and methods, access to constructors is not governed by access modifiers. Constructors are accessible from any class.

Constructor declarations are never inherited and therefore are not subject to overriding.

If no constructors are defined, the compiler defines a default constructor. Like methods, they may be overloaded. It is a compile-time error to declare two constructors with equivalent signatures in a class.

When the programmer declares an instance of the class, either a user-defined constructor or the default constructor is automatically called.

```
class Foo {
        constructor (int x) {...}
        ...
}
class Bar {
        public void main (char[][] args) {
                int x;
                Foo myFooObj;
                x = 5;
                myFooObj = Foo(x);
        }
}
```

Referencing instances

The keyword 'this' is used in the body of method and constructor declarations to reference the instance of the object that the method or constructor will bind to at runtime.

Inheritance

The members of a class include both declared and inherited members. A class inherits all members of its direct superclass and superclasses of that class. To define a class Y that inherits members of an existing class named "X" and all superclasses of X, use the keyword *extends* when defining Y.

```
class Y extends X {...}
```

Overriding

Newly declared methods can override methods declared in any ancestor class. An instance method m1, declared in class C, overrides another instance method m2, declared in class A iff both of the following are true:

- C is a subclass of A
- The signature of m1 is identical to the signature of m2

Built in Functions

Standard Library Classes

Accessing the Standard Library

To access the standard library, enter 'include(stdlib);' at the top of the source code. As noted earlier, including a file can only occur once, so do not include a class a second time.

String

Dice provides certain standard library classes to assist the user with string manipulation and file I/O.

Fields

String has no public fields

Constructors

String(char[] a) Accepts a char array, such as a string literal or a char array, and creates a String object

Methods

public bool contains(char[] chrs) Returns true if and only if this string contains the specified sequence of char values.

public int indexOf(int ch) Returns the index within this string of the first occurrence of the specified character.

public bool isEmpty() Returns true if and only if length() is 0.

public int length() Returns the length of the string.

public char[] toCharArray() Returns the char array of this string.

File

The File class constructor takes one argument which is a char[] that points to a file on which the user wishes to operate. The constructor stores the given path in a field and then calls open() on the given path and, if successful, sets the objects file descriptor field to the return of open(). If open() fails, the program exits with error.

Fields

File has no public fields

Constructors

File(char[] path, bool isWriteEnabled) Accepts a char array to open a file on, then creates a file object with the file descriptor. isWriteEnabled is a parameter that is used to determine whether the file can be written to or just read from.

Methods

public char[] read(int num) Reads num bytes from the open file and returns the bytes in a char array.

public void close() Closes the open file. On error, the program exits with error.

public void write(char[] arr) Writes the contents of the char[] array to the file

Grammar

4. Project Plan

Planning Process

Specification Process

Development Process

Testing Process

Team Responsibilities

Project Timeline

Project Log

Software Development Environment

Programming Style Guide

5. Architecture

The Compiler

The Lexer

The Parser

The Semantic Analyzer

The Code Generator

The Utilities

Pretty printing, token printing, JSON printing

Supplementary Code

The Standard Library

Built-in Functions

Functions Implemented in C

6. Test Plan

Testing Phases

Unit Testing

We hand tested the scanner and parser to make sure it worked. We also printed out the AST and tokens using custom written code.

Integration Testing

Discuss how we ensured the outputted code was correct and some of the thought that went behind different testers

Automation

Testing was very simple using ./tester.sh. We can verify that a test works individually by running lli on the outputted ll file

Test Suites

EDIT MEEEEEE 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. For example: Arrays, control flow, classes, inheritance, etc. Please list as many as possible.

Dice to LL IR

Please provide an example Dice program and the corresponding LLVM IR (DemoAnimals.dice?)

Testing Roles

Khal was the brains behind testing. Everyone contributed by making github issues

7. Lessons Learned

David

Most critically I learned that if you want to make something good, put as much effort as physically possible into it. I was told frequently "get started early" with respect to this project. After starting early I also learned that working often and with purpose helped not only myself get through the project but also the rest of my team. WIP

Emily

Khaled

Philip

8. Code Listing

_tags

```
1  <filepath.*> or <**/*.native> or <**/*.byte>: package(unix)
```

analyzer.ml

```
open Sast
open Ast
open Processor
open Utils
open Filepath
open Conf

module StringMap = Map.Make (String)

module StringSet = Set.Make (String)

let struct_indexes:(string, int) Hashtbl.t = Hashtbl.create 10
let predecessors:(string, string list) Hashtbl.t = Hashtbl.create 10

module SS = Set.Make(
struct
let compare = Pervasives.compare
type t = datatype
end )

type class_map = {
        field_map       : Ast.field StringMap.t;
        func_map        : Ast.func_decl StringMap.t;
        constructor_map : Ast.func_decl StringMap.t;
        reserved_map         : sfunc_decl StringMap.t;
        cdecl                        : Ast.class_decl;
}

type env = {
        env_class_maps: class_map StringMap.t;
        env_name      : string;
        env_cmap           : class_map;
        env_locals    : datatype StringMap.t;
        env_parameters: Ast.formal StringMap.t;
        env_returnType: datatype;
        env_in_for    : bool;
        env_in_while  : bool;
        env_reserved  : sfunc_decl list;
}

let update_env_name env env_name =
{
        env_class_maps = env.env_class_maps;
        env_name       = env_name;
        env_cmap            = env.env_cmap;
        env_locals     = env.env_locals;
        env_parameters = env.env_parameters;
```

```
48          env_returnType = env.env_returnType;
49          env_in_for     = env.env_in_for;
50          env_in_while   = env.env_in_while;
51          env_reserved   = env.env_reserved;
52  }
53
54  let update_call_stack env in_for in_while =
55  {
56          env_class_maps = env.env_class_maps;
57          env_name       = env.env_name;
58          env_cmap            = env.env_cmap;
59          env_locals     = env.env_locals;
60          env_parameters = env.env_parameters;
61          env_returnType = env.env_returnType;
62          env_in_for     = in_for;
63          env_in_while   = in_while;
64          env_reserved   = env.env_reserved;
65  }
66
67  let append_code_to_constructor fbody cname ret_type =
68  let key = Hashtbl.find struct_indexes cname in
69  let init_this = [SLocal(
70  ret_type,
71  "this",
72  SCall(        "cast",
73  [SCall("malloc",
74  [
75  SCall("sizeof", [SId("ignore", ret_type)], Datatype(Int_t), 0)
76  ],
77  Arraytype(Char_t, 1), 0)
78  ],
79  ret_type,
80  0
81  )
82  );
83  SExpr(
84  SAssign(
85  SObjAccess(
86  SId("this", ret_type),
87  SId(".key", Datatype(Int_t)),
88  Datatype(Int_t)
89  ),
90  SInt_Lit(key),
91  Datatype(Int_t)
92  ),
93  Datatype(Int_t)
94  )
95  ]
96  in
```

```
97  let ret_this =
98  [
99  SReturn(
100 SId("this", ret_type),
101 ret_type
102 )
103 ]
104 in
105 (* Need to check for duplicate default constructs *)
106 (* Also need to add malloc around other constructors *)
107 init_this @ fbody @ ret_this
108
109 let default_constructor_body cname =
110 let ret_type = Datatype(Objecttype(cname)) in
111 let fbody = [] in
112 append_code_to_constructor fbody cname ret_type
113
114 let default_sc cname =
115 {
116         sfname                          = Ast.FName (cname ^ "." ^ "constructor");
117         sreturnType         = Datatype(Objecttype(cname));
118         sformals                = [];
119         sbody                           = default_constructor_body cname;
120         func_type               = Sast.User;
121         overrides       = false;
122         source                          = "NA";
123 }
124
125 let default_c cname =
126 {
127         scope                           = Ast.Public;
128         fname                           = Ast.Constructor;
129         returnType                = Datatype(ConstructorType);
130         formals                 = [];
131         body                            = [];
132         overrides                 = false;
133         root_cname                = None;
134 }
135
136 let process_includes filename includes classes =
137 (* Bring in each include  *)
138 let processInclude include_statement =
139 let file_in = open_in include_statement in
140 let lexbuf = Lexing.from_channel file_in in
141 let token_list = Processor.build_token_list lexbuf in
142 let program = Processor.parser include_statement token_list in
143 ignore(close_in file_in);
144 program
145 in
```

```
146 let rec iterate_includes classes m = function
147 [] -> classes
148 | (Include h) :: t ->
149 let h = if h = "stdlib" then Conf.stdlib_path else h in
150 (* Check each include against the map *)
151 let realpath = Filepath.realpath h in
152 if StringMap.mem realpath m then
153 iterate_includes (classes) (m) (t)
154 else
155 let result = processInclude realpath in
156 match result with Program(i,c) ->
157 iterate_includes (classes @ c) (StringMap.add realpath 1 m) (i @ t)
158 in
159 iterate_includes classes (StringMap.add (Filepath.realpath filename) 1 StringMap.empty)
    ↪ includes
160
161 let get_name cname fdecl =
162 (* We use '.' to separate types so llvm will recognize the function name and it won't
    ↪ conflict *)
163 (* let params = List.fold_left (fun s -> (function Formal(t, _) -> s ^ "." ^
    ↪ Utils.string_of_datatype t | _ -> "" )) "" fdecl.formals in *)
164 let name = Utils.string_of_fname fdecl.fname in
165 if name = "main"
166 then "main"
167 else cname ^ "." ^ name(*  ^ params *)
168
169 let get_constructor_name cname fdecl =
170 let params = List.fold_left (fun s -> (function Formal(t, _) -> s ^ "." ^
    ↪ Utils.string_of_datatype t | _ -> "" )) "" fdecl.formals in
171 let name = Utils.string_of_fname fdecl.fname in
172 cname ^ "." ^ name ^ params
173
174 let get_name_without_class fdecl =
175 (* We use '.' to separate types so llvm will recognize the function name and it won't
    ↪ conflict *)
176 let params = List.fold_left (fun s -> (function Formal(t, _) -> s ^ "." ^
    ↪ Utils.string_of_datatype t | _ -> "" )) "" fdecl.formals in
177 let name = Utils.string_of_fname fdecl.fname in
178 let ret_type = Utils.string_of_datatype fdecl.returnType in
179 ret_type ^ "." ^ name ^ "." ^ params
180
181 (* Generate list of all classes to be used for semantic checking *)
182 let build_class_maps reserved cdecls =
183 let reserved_map = List.fold_left (fun m f -> StringMap.add (Utils.string_of_fname
    ↪ f.sfname) f m) StringMap.empty reserved in
184 let helper m (cdecl:Ast.class_decl) =
185 let fieldfun = (fun m -> (function Field(s, d, n) -> if (StringMap.mem (n) m) then
    ↪ raise(Exceptions.DuplicateField) else (StringMap.add n (Field(s, d, n)) m))) in
186 let funcname = get_name cdecl.cname in
```

```
187 let funcfun m fdecl =
188 if (StringMap.mem (funcname fdecl) m)
189 then raise(Exceptions.DuplicateFunction(funcname fdecl))
190 else if (StringMap.mem (Utils.string_of_fname fdecl.fname) reserved_map)
191 then raise(Exceptions.CannotUseReservedFuncName(Utils.string_of_fname fdecl.fname))
192 else (StringMap.add (funcname fdecl) fdecl m)
193 in
194 let constructor_name = get_constructor_name cdecl.cname in
195 let constructorfun m fdecl =
196 if fdecl.formals = [] then m
197 else if StringMap.mem (constructor_name fdecl) m
198 then raise(Exceptions.DuplicateConstructor)
199 else (StringMap.add (constructor_name fdecl) fdecl m)
200 in
201 let default_c = default_c cdecl.cname in
202 let constructor_map = StringMap.add (get_constructor_name cdecl.cname default_c)
    ↪  default_c StringMap.empty in
203 (if (StringMap.mem cdecl.cname m) then raise (Exceptions.DuplicateClassName(cdecl.cname))
    ↪  else
204 StringMap.add cdecl.cname
205 {         field_map = List.fold_left fieldfun StringMap.empty cdecl.cbody.fields;
206         func_map = List.fold_left funcfun StringMap.empty cdecl.cbody.methods;
207         constructor_map = List.fold_left constructorfun constructor_map
          ↪  cdecl.cbody.constructors;
208         reserved_map = reserved_map;
209         cdecl = cdecl }
210 m) in
211 List.fold_left helper StringMap.empty cdecls
212
213 let rec get_all_descendants cname accum =
214 if Hashtbl.mem predecessors cname then
215 let direct_descendants = Hashtbl.find predecessors cname in
216 let add_childs_descendants desc_set direct_descendant =
217 get_all_descendants direct_descendant (StringSet.add direct_descendant desc_set)
218 in
219 List.fold_left add_childs_descendants accum direct_descendants
220 else accum
221
222 let inherited potential_predec potential_child =
223 match potential_predec, potential_child with
224 Datatype(Objecttype(predec_cname)), Datatype(Objecttype(child_cname)) ->
225 let descendants = get_all_descendants predec_cname StringSet.empty in
226 if (predec_cname = child_cname) || (StringSet.mem child_cname descendants) then true
227 else raise (Exceptions.LocalAssignTypeMismatch(predec_cname, child_cname))
228 | _ , _ -> false
229
230 let get_equality_binop_type type1 type2 se1 se2 op =
231 (* Equality op not supported for float operands. The correct way to test floats
232 for equality is to check the difference between the operands in question *)
```

```
233 if (type1 = Datatype(Float_t) || type2 = Datatype(Float_t)) then raise
    ↪ (Exceptions.InvalidBinopExpression "Equality operation is not supported for Float
    ↪ types")
234 else
235 match type1, type2 with
236 Datatype(Char_t), Datatype(Int_t)
237 |         Datatype(Int_t), Datatype(Char_t)
238 |         Datatype(Objecttype(_)), Datatype(Null_t)
239 |         Datatype(Null_t), Datatype(Objecttype(_))
240 |         Datatype(Null_t), Arraytype(_, _)
241 |         Arraytype(_, _), Datatype(Null_t) -> SBinop(se1, op, se2, Datatype(Bool_t))
242 | _ ->
243 if type1 = type2 then SBinop(se1, op, se2, Datatype(Bool_t))
244 else raise (Exceptions.InvalidBinopExpression "Equality operator can't operate on
    ↪ different types, with the exception of Int_t and Char_t")
245
246 let get_logical_binop_type se1 se2 op = function
247 (Datatype(Bool_t), Datatype(Bool_t)) -> SBinop(se1, op, se2, Datatype(Bool_t))
248 | _ -> raise (Exceptions.InvalidBinopExpression "Logical operators only operate on Bool_t
    ↪ types")
249
250 let get_comparison_binop_type type1 type2 se1 se2 op =
251 let numerics = SS.of_list [Datatype(Int_t); Datatype(Char_t); Datatype(Float_t)]
252 in
253 if SS.mem type1 numerics && SS.mem type2 numerics
254 then SBinop(se1, op, se2, Datatype(Bool_t))
255 else raise (Exceptions.InvalidBinopExpression "Comparison operators operate on numeric
    ↪ types only")
256
257
258 let get_arithmetic_binop_type se1 se2 op = function
259 (Datatype(Int_t), Datatype(Float_t))
260 |         (Datatype(Float_t), Datatype(Int_t))
261 |         (Datatype(Float_t), Datatype(Float_t))         -> SBinop(se1, op, se2,
    ↪ Datatype(Float_t))
262
263 |         (Datatype(Int_t), Datatype(Char_t))
264 |         (Datatype(Char_t), Datatype(Int_t))
265 |         (Datatype(Char_t), Datatype(Char_t))         -> SBinop(se1, op, se2,
    ↪ Datatype(Char_t))
266
267 |         (Datatype(Int_t), Datatype(Int_t))                 -> SBinop(se1, op, se2,
    ↪ Datatype(Int_t))
268
269 | _ -> raise (Exceptions.InvalidBinopExpression "Arithmetic operators don't support these
    ↪ types")
270
271 let rec get_ID_type env s =
272 try StringMap.find s env.env_locals
```

```
273 with | Not_found ->
274 try let formal = StringMap.find s env.env_parameters in
275 (function Formal(t, _) -> t | Many t -> t ) formal
276 with | Not_found -> raise (Exceptions.UndefinedID s)
277
278 and check_array_primitive env el =
279 let rec iter t sel = function
280 [] -> sel, t
281 |         e :: el ->
282 let se, _ = expr_to_sexpr env e in
283 let se_t = get_type_from_sexpr se in
284 if t = se_t
285 then iter t (se :: sel) el
286 else
287 let t1 = Utils.string_of_datatype t in
288 let t2 = Utils.string_of_datatype se_t in
289 raise(Exceptions.InvalidArrayPrimitiveConsecutiveTypes(t1, t2))
290 in
291 let se, _ = expr_to_sexpr env (List.hd el) in
292 let el = List.tl el in
293 let se_t = get_type_from_sexpr se in
294 let sel, t = iter se_t ([se]) el in
295 let se_t = match t with
296 Datatype(x) -> Arraytype(x, 1)
297 |         Arraytype(x, n) -> Arraytype(x, n+1)
298 |         _ as t -> raise(Exceptions.InvalidArrayPrimitiveType(Utils.string_of_datatype
    ↪  t))
299 in
300 SArrayPrimitive(sel, se_t)
301
302 and check_array_init env d el =
303 (* Get dimension size for the array being created *)
304 let array_complexity = List.length el in
305 let check_elem_type e =
306 let sexpr, _ = expr_to_sexpr env e in
307 let sexpr_type = get_type_from_sexpr sexpr in
308 if sexpr_type = Datatype(Int_t)
309 then sexpr
310 else raise(Exceptions.MustPassIntegerTypeToArrayCreate)
311 in
312 let convert_d_to_arraytype = function
313 Datatype(x) -> Arraytype(x, array_complexity)
314 |         _ as t ->
315 let error_msg = Utils.string_of_datatype t in
316 raise (Exceptions.ArrayInitTypeInvalid(error_msg))
317 in
318 let sexpr_type = convert_d_to_arraytype d in
319 let sel = List.map check_elem_type el in
320 SArrayCreate(d, sel, sexpr_type)
```

```
321
322 and check_array_access env e el =
323 (* Get dimensions of array, ex: foo[10][4][2] is dimen=3 *)
324 let array_dimensions = List.length el in
325 (* Check every e in el is of type Datatype(Int_t). Ensure all indices are ints *)
326 let check_elem_type arg =
327 let sexpr, _ = expr_to_sexpr env arg in
328 let sexpr_type = get_type_from_sexpr sexpr in
329 if sexpr_type = Datatype(Int_t)
330 then sexpr
331 else raise(Exceptions.MustPassIntegerTypeToArrayAccess)
332 in
333 (* converting e to se also checks if the array id has been declared  *)
334 let se, _ = expr_to_sexpr env e in
335 let se_type = get_type_from_sexpr se in
336
337 (* Check that e has enough dimens as e's in el. Return overall datatype of access*)
338 let check_array_dim_vs_params num_params = function
339 Arraytype(t, n) ->
340 if num_params < n then
341 Arraytype(t, (n-num_params))
342 else if num_params = n then
343 Datatype(t)
344 else
345 raise (Exceptions.ArrayAccessInvalidParamLength(string_of_int num_params, string_of_int
    ↪  n))
346 |         _ as t ->
347 let error_msg = Utils.string_of_datatype t in
348 raise (Exceptions.ArrayAccessExpressionNotArray(error_msg))
349 in
350 let sexpr_type = check_array_dim_vs_params array_dimensions se_type in
351 let sel = List.map check_elem_type el in
352
353 SArrayAccess(se, sel, sexpr_type)
354
355 and check_obj_access env lhs rhs =
356 let check_lhs = function
357 This                        -> SId("this", Datatype(Objecttype(env.env_name)))
358 |        Id s                          -> SId(s, get_ID_type env s)
359 |         ArrayAccess(e, el)        -> check_array_access env e el
360 |         _ as e         -> raise (Exceptions.LHSofRootAccessMustBeIDorFunc
    ↪  (Utils.string_of_expr e))
361 in
362 let ptype_name parent_type = match parent_type with
363 Datatype(Objecttype(name))         -> name
364 |         _ as d                                                -> raise
    ↪  (Exceptions.ObjAccessMustHaveObjectType (Utils.string_of_datatype d))
365 in
366 let rec check_rhs (env) parent_type (top_level_env) =
```

```
367 let pt_name = ptype_name parent_type in
368 let get_id_type_from_object env (id) cname tlenv =
369 let cmap = StringMap.find cname env.env_class_maps in
370 let match_field f = match f with
371 Field(scope, d, n) ->
372 (* Have to update this with all parent classes checks *)
373 if scope = Ast.Private && tlenv.env_name <> env.env_name then
374 raise(Exceptions.CannotAccessPrivateFieldInNonProperScope(n, env.env_name,
    ↪ tlenv.env_name))
375 else d
376 in
377 try match_field (StringMap.find id cmap.field_map)
378 with | Not_found -> raise (Exceptions.UnknownIdentifierForClass(id, cname))
379 in
380 function
381 (* Check fields in parent *)
382 Id s                                -> SId(s, (get_id_type_from_object env s pt_name
    ↪ top_level_env)), env
383 (* Check functions in parent *)
384 |         Call(fname, el)         ->
385 let env = update_env_name env pt_name in
386 check_call_type top_level_env true env fname el, env
387 (* Set parent, check if base is field *)
388 |         ObjAccess(e1, e2)         ->
389 let old_env = env in
390 let lhs, env = check_rhs env parent_type top_level_env e1 in
391 let lhs_type = get_type_from_sexpr lhs in
392
393 let pt_name = ptype_name lhs_type in
394 let lhs_env = update_env_name env pt_name in
395
396 let rhs, env = check_rhs lhs_env lhs_type top_level_env e2 in
397 let rhs_type = get_type_from_sexpr rhs in
398 SObjAccess(lhs, rhs, rhs_type), old_env
399 |         _ as e                                -> raise (Exceptions.InvalidAccessLHS
    ↪ (Utils.string_of_expr e))
400 in
401 let arr_lhs, _ = expr_to_sexpr env lhs in
402 let arr_lhs_type = get_type_from_sexpr arr_lhs in
403 match arr_lhs_type with
404 Arraytype(Char_t, 1) -> raise(Exceptions.CannotAccessLengthOfCharArray)
405 |        Arraytype(_, _) ->
406 let rhs = match rhs with
407 Id("length") -> SId("length", Datatype(Int_t))
408 |         _ -> raise(Exceptions.CanOnlyAccessLengthOfArray)
409 in
410 SObjAccess(arr_lhs, rhs, Datatype(Int_t))
411 | _ ->
412 let lhs = check_lhs lhs in
```

```
413 let lhs_type = get_type_from_sexpr lhs in
414
415 let ptype_name = ptype_name lhs_type in
416 let lhs_env = update_env_name env ptype_name in
417
418 let rhs, _ = check_rhs lhs_env lhs_type env rhs in
419 let rhs_type = get_type_from_sexpr rhs in
420 SObjAccess(lhs, rhs, rhs_type)
421
422 and check_call_type top_level_env isObjAccess env fname el =
423 let sel, env = exprl_to_sexprl env el in
424 (* check that 'env.env_name' is in the list of defined classes *)
425 let cmap =
426 try StringMap.find env.env_name env.env_class_maps
427 with | Not_found -> raise (Exceptions.UndefinedClass env.env_name)
428 in
429
430 let handle_param formal param =
431 let fty = match formal with Formal(d, _) -> d | _ -> Datatype(Void_t) in
432 let pty = get_type_from_sexpr param in
433 match fty, pty with
434 Datatype(Objecttype(f)), Datatype(Objecttype(p)) ->
435 if f <> p then
436 try let descendants = Hashtbl.find predecessors f in
437 let _ = try List.find (fun d -> p = d) descendants
438 with | Not_found -> raise(Exceptions.CannotPassNonInheritedClassesInPlaceOfOthers(f, p))
439 in
440 let rt = Datatype(Objecttype(f)) in
441 SCall("cast", [param; SId("ignore", rt)], rt, 0)
442 with | Not_found -> raise(Exceptions.ClassIsNotExtendedBy(f, p))
443 else param
444 |         _ -> if fty = pty then param else
    ↪  raise(Exceptions.IncorrectTypePassedToFunction(fname, Utils.string_of_datatype pty))
445 in
446
447 let index fdecl fname =
448 let cdecl = cmap.cdecl in
449 (* Have to update this with all parent classes checks *)
450 let _ =
451 if fdecl.scope = Ast.Private && top_level_env.env_name <> env.env_name then
452 raise(Exceptions.CannotAccessPrivateFunctionInNonProperScope(get_name env.env_name fdecl,
    ↪  env.env_name, top_level_env.env_name))
453 in
454 (* Not exactly sure why there needs to be a list.rev *)
455 let fns = List.rev cdecl.cbody.methods in
456 let rec find x lst =
457 match lst with
458 | [] -> raise (Failure ("Could not find " ^ fname))
459 | fdecl :: t ->
```

```
460 let search_name = (get_name env.env_name fdecl) in
461 if x = search_name then 0
462 else if search_name = "main" then find x t
463 else 1 + find x t
464 in
465 find fname fns
466 in
467
468 let handle_params (formals) params =
469 match formals, params with
470 [Many(Any)], _ -> params
471 |          [], [] -> []
472 |          [], _
473 |          _, [] -> raise(Exceptions.IncorrectTypePassedToFunction(fname,
    ↪  Utils.string_of_datatype (Datatype(Void_t))))
474 |          _ ->
475 let len1 = List.length formals in
476 let len2 = List.length params in
477 if len1 <> len2 then raise(Exceptions.IncorrectNumberOfArguments(fname, len1, len2))
478 else
479 List.map2 handle_param formals sel
480 in
481
482 let sfname = env.env_name ^ "." ^ fname in
483 try let func = StringMap.find fname cmap.reserved_map in
484 let actuals = handle_params func.sformals sel in
485 SCall(fname, actuals, func.sreturnType, 0)
486 with | Not_found ->
487 try let f = StringMap.find sfname cmap.func_map in
488 let actuals = handle_params f.formals sel in
489 let index = index f sfname in
490 SCall(sfname, actuals, f.returnType, index)
491 with | Not_found -> raise(Exceptions.FunctionNotFound(env.env_name, sfname)) | _ as ex ->
    ↪  raise ex
492
493 and check_object_constructor env s el =
494 let sel, env = exprl_to_sexprl env el in
495 (* check that 'env.env_name' is in the list of defined classes *)
496 let cmap =
497 try StringMap.find s env.env_class_maps
498 with | Not_found -> raise (Exceptions.UndefinedClass s)
499 in
500 (* get a list of the types of the actuals to match against defined function formals *)
501 let params = List.fold_left (fun s e -> s ^ "." ^ (Utils.string_of_datatype
    ↪  (get_type_from_sexpr e))) "" sel in
502 let constructor_name = s ^ "." ^ "constructor" ^ params in
503 let _ =
504 try StringMap.find constructor_name cmap.constructor_map
505 with | Not_found -> raise (Exceptions.ConstructorNotFound constructor_name)
```

```
in
let ftype = Datatype(Objecttype(s)) in
(* Add a reference to the class in front of the function call *)
(* Must properly handle the case where this is a reserved function *)
SObjectCreate(constructor_name, sel, ftype)

and check_assign env e1 e2 =
let se1, env = expr_to_sexpr env e1 in
let se2, env = expr_to_sexpr env e2 in
let type1 = get_type_from_sexpr se1 in
let type2 = get_type_from_sexpr se2 in
match (type1, se2) with
Datatype(Objecttype(_)), SNull
|         Arraytype(_, _), SNull -> SAssign(se1, se2, type1)
|   _ ->
match type1, type2 with
Datatype(Char_t), Datatype(Int_t)
|         Datatype(Int_t), Datatype(Char_t) -> SAssign(se1, se2, type1)
|         Datatype(Objecttype(d)), Datatype(Objecttype(t)) ->
if d = t then SAssign(se1, se2, type1)
else if inherited type1 type2 then
SAssign(se1, SCall("cast", [se2; SId("ignore", type1)], type1, 0), type1)
else raise (Exceptions.AssignmentTypeMismatch(Utils.string_of_datatype type1,
↪  Utils.string_of_datatype type2))
| _ ->
if type1 = type2
then SAssign(se1, se2, type1)
else raise (Exceptions.AssignmentTypeMismatch(Utils.string_of_datatype type1,
↪  Utils.string_of_datatype type2))

and check_unop env op e =
let check_num_unop t = function
Sub         -> t
|         _                 -> raise(Exceptions.InvalidUnaryOperation)
in
let check_bool_unop = function
Not         -> Datatype(Bool_t)
|         _                 -> raise(Exceptions.InvalidUnaryOperation)
in
let se, env = expr_to_sexpr env e in
let t = get_type_from_sexpr se in
match t with
Datatype(Int_t)
|        Datatype(Float_t)         -> SUnop(op, se, check_num_unop t op)
|          Datatype(Bool_t)         -> SUnop(op, se, check_bool_unop op)
|         _ -> raise(Exceptions.InvalidUnaryOperation)

and check_binop env e1 op e2 =
let se1, env = expr_to_sexpr env e1 in
```

```
553 let se2, env = expr_to_sexpr env e2 in
554 let type1 = get_type_from_sexpr se1 in
555 let type2 = get_type_from_sexpr se2 in
556 match op with
557 Equal | Neq -> get_equality_binop_type type1 type2 se1 se2 op
558 | And | Or -> get_logical_binop_type se1 se2 op (type1, type2)
559 | Less | Leq | Greater | Geq -> get_comparison_binop_type type1 type2 se1 se2 op
560 | Add | Mult | Sub | Div | Mod -> get_arithmetic_binop_type se1 se2 op (type1, type2)
561 | _ -> raise (Exceptions.InvalidBinopExpression ((Utils.string_of_op op) ^ " is not a
    ↪  supported binary op"))
562
563 and check_delete env e =
564 let se, _ = expr_to_sexpr env e in
565 let t = get_type_from_sexpr se in
566 match t with
567 Arraytype(_, _) | Datatype(Objecttype(_)) -> SDelete(se)
568 |         _ -> raise(Exceptions.CanOnlyDeleteObjectsOrArrays)
569
570 and expr_to_sexpr env = function
571 Int_Lit i           -> SInt_Lit(i), env
572 |   Boolean_Lit b       -> SBoolean_Lit(b), env
573 |   Float_Lit f         -> SFloat_Lit(f), env
574 |   String_Lit s        -> SString_Lit(s), env
575 |   Char_Lit c          -> SChar_Lit(c), env
576 |   This                -> SId("this", Datatype(Objecttype(env.env_name))), env
577 |   Id s                -> SId(s, get_ID_type env s), env
578 |   Null                -> SNull, env
579 |   Noexpr              -> SNoexpr, env
580
581 |   ObjAccess(e1, e2)   -> check_obj_access env e1 e2, env
582 |   ObjectCreate(s, el) -> check_object_constructor env s el, env
583 |   Call(s, el)         -> check_call_type env false env s el, env
584
585 |   ArrayCreate(d, el)  -> check_array_init env d el, env
586 |   ArrayAccess(e, el)  -> check_array_access env e el, env
587 |   ArrayPrimitive el   -> check_array_primitive env el, env
588
589 |   Assign(e1, e2)      -> check_assign env e1 e2, env
590 |   Unop(op, e)         -> check_unop env op e, env
591 |   Binop(e1, op, e2)   -> check_binop env e1 op e2, env
592 |         Delete(e)                          -> check_delete env e, env
593
594
595 and get_type_from_sexpr = function
596 SInt_Lit(_)                                -> Datatype(Int_t)
597 |         SBoolean_Lit(_)                        -> Datatype(Bool_t)
598 |         SFloat_Lit(_)                       -> Datatype(Float_t)
599 |         SString_Lit(_)                         -> Arraytype(Char_t, 1)
600 |         SChar_Lit(_)                       -> Datatype(Char_t)
```

```
601 |         SId(_, d)                                -> d
602 |         SBinop(_, _, _, d)                   -> d
603 |         SAssign(_, _, d)                   -> d
604 |         SNoexpr                                -> Datatype(Void_t)
605 |         SArrayCreate(_, _, d)         -> d
606 |         SArrayAccess(_, _, d)          -> d
607 |         SObjAccess(_, _, d)                -> d
608 |         SCall(_, _, d, _)                  -> d
609 |   SObjectCreate(_, _, d)          -> d
610 |         SArrayPrimitive(_, d)         -> d
611 |          SUnop(_, _, d)                        -> d
612 |         SNull                                      -> Datatype(Null_t)
613 |         SDelete _                                -> Datatype(Void_t)
614
615 and exprl_to_sexprl env el =
616 let env_ref = ref(env) in
617 let rec helper = function
618 head::tail ->
619 let a_head, env = expr_to_sexpr !env_ref head in
620 env_ref := env;
621 a_head::(helper tail)
622 | [] -> []
623 in (helper el), !env_ref
624
625 let rec local_handler d s e env =
626 if StringMap.mem s env.env_locals
627 then raise (Exceptions.DuplicateLocal s)
628 else
629 let se, env = expr_to_sexpr env e in
630 let t = get_type_from_sexpr se in
631 if t = Datatype(Void_t) || t = Datatype(Null_t) || t = d || (inherited d t)
632 then
633 let new_env = {
634         env_class_maps = env.env_class_maps;
635         env_name = env.env_name;
636         env_cmap = env.env_cmap;
637         env_locals = StringMap.add s d env.env_locals;
638         env_parameters = env.env_parameters;
639         env_returnType = env.env_returnType;
640         env_in_for = env.env_in_for;
641         env_in_while = env.env_in_while;
642         env_reserved = env.env_reserved;
643 } in
644 (* if the user-defined type being declared is not in global classes map, it is an
    ↪  undefined class *)
645 (match d with
646 Datatype(Objecttype(x)) ->
647 (if not (StringMap.mem (Utils.string_of_object d) env.env_class_maps)
648 then raise (Exceptions.UndefinedClass (Utils.string_of_object d))
```

```
649 else
650 let local = if inherited d t then SLocal(t, s, se) else SLocal(d, s, se)
651 in local, new_env)
652 | _ -> SLocal(d, s, se), new_env)
653 else
654 (let type1 = (Utils.string_of_datatype t) in
655 let type2 = (Utils.string_of_datatype d) in
656 let ex = Exceptions.LocalAssignTypeMismatch(type1, type2) in
657 raise ex)
658
659 let rec check_sblock sl env = match sl with
660 [] -> SBlock([SExpr(SNoexpr, Datatype(Void_t))]), env
661 |          _  ->
662 let sl, _ = convert_stmt_list_to_sstmt_list env sl in
663 SBlock(sl), env
664
665 and check_expr_stmt e env =
666 let se, env = expr_to_sexpr env e in
667 let t = get_type_from_sexpr se in
668 SExpr(se, t), env
669
670 and check_return e env =
671 let se, _ = expr_to_sexpr env e in
672 let t = get_type_from_sexpr se in
673 match t, env.env_returnType with
674 Datatype(Null_t), Datatype(Objecttype(_))
675 |          Datatype(Null_t), Arraytype(_, _) -> SReturn(se, t), env
676 |          _ ->
677 if t = env.env_returnType
678 then SReturn(se, t), env
679 else raise (Exceptions.ReturnTypeMismatch(Utils.string_of_datatype t,
    ↪  Utils.string_of_datatype env.env_returnType))
680
681 and check_if e s1 s2 env =
682 let se, _ = expr_to_sexpr env e in
683 let t = get_type_from_sexpr se in
684 let ifbody, _ = parse_stmt env s1 in
685 let elsebody, _ = parse_stmt env s2 in
686 if t = Datatype(Bool_t)
687 then SIf(se, ifbody, elsebody), env
688 else raise Exceptions.InvalidIfStatementType
689
690 and check_for e1 e2 e3 s env =
691 let old_val = env.env_in_for in
692 let env = update_call_stack env true env.env_in_while in
693
694 let se1, _ = expr_to_sexpr env e1 in
695 let se2, _ = expr_to_sexpr env e2 in
696 let se3, _ = expr_to_sexpr env e3 in
```

```
697 let forbody, _ = parse_stmt env s in
698 let conditional = get_type_from_sexpr se2 in
699 let sfor =
700 if (conditional = Datatype(Bool_t) || conditional = Datatype(Void_t))
701 then SFor(se1, se2, se3, forbody)
702 else raise Exceptions.InvalidForStatementType
703 in
704
705 let env = update_call_stack env old_val env.env_in_while in
706 sfor, env
707
708 and check_while e s env =
709 let old_val = env.env_in_while in
710 let env = update_call_stack env env.env_in_for true in
711
712 let se, _ = expr_to_sexpr env e in
713 let t = get_type_from_sexpr se in
714 let sstmt, _ = parse_stmt env s in
715 let swhile =
716 if (t = Datatype(Bool_t) || t = Datatype(Void_t))
717 then SWhile(se, sstmt)
718 else raise Exceptions.InvalidWhileStatementType
719 in
720
721 let env = update_call_stack env env.env_in_for old_val in
722 swhile, env
723
724 and check_break env =
725 if env.env_in_for || env.env_in_while then
726 SBreak, env
727 else
728 raise Exceptions.CannotCallBreakOutsideOfLoop
729
730 and check_continue env =
731 if env.env_in_for || env.env_in_while then
732 SContinue, env
733 else
734 raise Exceptions.CannotCallContinueOutsideOfLoop
735
736 and parse_stmt env = function
737 Block sl                                -> check_sblock sl env
738 |         Expr e                                    -> check_expr_stmt e env
739 |         Return e                                  -> check_return e env
740 |         If(e, s1, s2)                             -> check_if e s1 s2        env
741 |         For(e1, e2, e3, e4)          -> check_for e1 e2 e3 e4 env
742 |         While(e, s)                                -> check_while e s env
743 |          Break                                     -> check_break env (* Need to
    ↪  check if in right context *)
```

```
744 |   Continue                                -> check_continue env (* Need to check if in
    ↪  right context *)
745 |   Local(d, s, e)                         -> local_handler d s e env
746
747 (* Update this function to return an env object *)
748 and convert_stmt_list_to_sstmt_list env stmt_list =
749 let env_ref = ref(env) in
750 let rec iter = function
751 head::tail ->
752 let a_head, env = parse_stmt !env_ref head in
753 env_ref := env;
754 a_head::(iter tail)
755 | [] -> []
756 in
757 let sstmt_list = (iter stmt_list), !env_ref in
758 sstmt_list
759
760 let append_code_to_main fbody cname ret_type =
761 let key = Hashtbl.find struct_indexes cname in
762 let init_this = [SLocal(
763 ret_type,
764 "this",
765 SCall(        "cast",
766 [SCall("malloc",
767 [
768 SCall("sizeof", [SId("ignore", ret_type)], Datatype(Int_t), 0)
769 ],
770 Arraytype(Char_t, 1), 0)
771 ],
772 ret_type, 0
773 )
774 );
775 SExpr(
776 SAssign(
777 SObjAccess(
778 SId("this", ret_type),
779 SId(".key", Datatype(Int_t)),
780 Datatype(Int_t)
781 ),
782 SInt_Lit(key),
783 Datatype(Int_t)
784 ),
785 Datatype(Int_t)
786 )
787 ]
788 in
789 init_this @ fbody
790
791 let convert_constructor_to_sfdecl class_maps reserved class_map cname constructor =
```

```
792 let env = {
793         env_class_maps         = class_maps;
794         env_name              = cname;
795         env_cmap                 = class_map;
796         env_locals             = StringMap.empty;
797         env_parameters         = List.fold_left (fun m f -> match f with Formal(d, s) ->
        ↪  (StringMap.add s f m) | _ -> m) StringMap.empty constructor.formals;
798         env_returnType         = Datatype(Objecttype(cname));
799         env_in_for                 = false;
800         env_in_while          = false;
801         env_reserved          = reserved;
802 } in
803 let fbody = fst (convert_stmt_list_to_sstmt_list env constructor.body) in
804 {
805         sfname                  = Ast.FName (get_constructor_name cname constructor);
806         sreturnType = Datatype(Objecttype(cname));
807         sformals          = constructor.formals;
808         sbody                 = append_code_to_constructor fbody cname
        ↪  (Datatype(Objecttype(cname)));
809         func_type         = Sast.User;
810         overrides          = false;
811         source                  = "NA";
812 }
813
814 let check_fbody fname fbody returnType =
815 let len = List.length fbody in
816 if len = 0 then () else
817 let final_stmt = List.hd (List.rev fbody) in
818 match returnType, final_stmt with
819 Datatype(Void_t), _ -> ()
820 |         _, SReturn(_, _) -> ()
821 |         _ -> raise(Exceptions.AllNonVoidFunctionsMustEndWithReturn(fname))
822
823 let convert_fdecl_to_sfdecl class_maps reserved class_map cname fdecl =
824 let root_cname = match fdecl.root_cname with
825 Some(x) -> x
826 | None -> cname
827 in
828 let class_formal =
829 if fdecl.overrides then
830 Ast.Formal(Datatype(Objecttype(root_cname)), "this")
831 else
832 Ast.Formal(Datatype(Objecttype(cname)), "this")
833 in
834 let env_param_helper m fname = match fname with
835 Formal(d, s) -> (StringMap.add s fname m)
836 |         _ -> m
837 in
```

```
838 let env_params = List.fold_left env_param_helper StringMap.empty (class_formal ::
    ↪  fdecl.formals) in
839 let env = {
840         env_class_maps         = class_maps;
841         env_name             = cname;
842         env_cmap                 = class_map;
843         env_locals            = StringMap.empty;
844         env_parameters        = env_params;
845         env_returnType        = fdecl.returnType;
846         env_in_for                = false;
847         env_in_while         = false;
848         env_reserved         = reserved;
849 }
850 in
851 let fbody = fst (convert_stmt_list_to_sstmt_list env fdecl.body) in
852 let fname = (get_name cname fdecl) in
853 ignore(check_fbody fname fbody fdecl.returnType);
854 let fbody = if fname = "main"
855 then (append_code_to_main fbody cname (Datatype(Objecttype(cname))))
856 else fbody
857 in
858 (* We add the class as the first parameter to the function for codegen *)
859 {
860         sfname                          = Ast.FName (get_name cname fdecl);
861         sreturnType         = fdecl.returnType;
862         sformals                = class_formal :: fdecl.formals;
863         sbody                       = fbody;
864         func_type               = Sast.User;
865         overrides       = fdecl.overrides;
866         source                          = cname;
867 }
868
869 let convert_cdecl_to_sast sfdecls (cdecl:Ast.class_decl) =
870 {
871         scname = cdecl.cname;
872         sfields = cdecl.cbody.fields;
873         sfuncs = sfdecls;
874 }
875
876 (*
877 * Given a list of func_decls for the base class and a single func_decl
878 * for the child class, replaces func_decls for the base class if any of them
879 * have the same method signature
880 *)
881 let replace_fdecl_in_base_methods base_cname base_methods child_fdecl =
882 let replace base_fdecl accum =
883 let get_root_cname = function
884 None -> Some(base_cname)
885 | Some(x) -> Some(x)
```

```
in
let modify_child_fdecl =
{
        scope = child_fdecl.scope;
        fname = child_fdecl.fname;
        returnType = child_fdecl.returnType;
        formals = child_fdecl.formals;
        body = child_fdecl.body;
        overrides = true;
        root_cname = get_root_cname base_fdecl.root_cname;
}
in
if (get_name_without_class base_fdecl) = (get_name_without_class child_fdecl)
then modify_child_fdecl::accum
else base_fdecl::accum
in
List.fold_right replace base_methods []

let merge_methods base_cname base_methods child_methods =
let check_overrides child_fdecl accum =
let base_checked_for_overrides =
replace_fdecl_in_base_methods base_cname (fst accum) child_fdecl
in
if (fst accum) = base_checked_for_overrides
then ((fst accum), child_fdecl::(snd accum))
else (base_checked_for_overrides, (snd accum))
in
let updated_base_and_child_fdecls =
List.fold_right check_overrides child_methods (base_methods, [])
in
(fst updated_base_and_child_fdecls) @ (snd updated_base_and_child_fdecls)

let merge_cdecls base_cdecl child_cdecl =
(* return a cdecl in which cdecl.cbody.fields contains the fields of
the extended class, concatenated by the fields of the child class *)
let child_cbody =
{
        fields = base_cdecl.cbody.fields @ child_cdecl.cbody.fields;
        constructors = child_cdecl.cbody.constructors;
        methods = merge_methods base_cdecl.cname base_cdecl.cbody.methods
        ↪  child_cdecl.cbody.methods
}
in
{
        cname = child_cdecl.cname;
        extends = child_cdecl.extends;
        cbody = child_cbody
}

```

```
(* returns a list of cdecls that contains inherited fields *)
let inherit_fields_cdecls cdecls inheritance_forest =
(* iterate through cdecls to make a map for lookup *)
let cdecl_lookup = List.fold_left (fun a litem -> StringMap.add litem.cname litem a)
    StringMap.empty cdecls in
let add_key key pred maps =
let elem1 = StringSet.add key (fst maps) in
let accum acc child = StringSet.add child acc in
let elem2 = List.fold_left (accum) (snd maps) pred in
(elem1, elem2)
in
let empty_s = StringSet.empty in
let res = StringMap.fold add_key inheritance_forest (empty_s, empty_s) in
let roots = StringSet.diff (fst res) (snd res) in
let rec add_inherited_fields predec desc map_to_update =
let merge_fields accum descendant =
let updated_predec_cdecl = StringMap.find predec accum in
let descendant_cdecl_to_update = StringMap.find descendant cdecl_lookup in
let merged = merge_cdecls updated_predec_cdecl descendant_cdecl_to_update in
let updated = (StringMap.add descendant merged accum) in
if (StringMap.mem descendant inheritance_forest) then
let descendants_of_descendant = StringMap.find descendant inheritance_forest in
add_inherited_fields descendant descendants_of_descendant updated
else updated
in
List.fold_left merge_fields map_to_update desc
in
(* map class name of every class_decl in `cdecls` to its inherited cdecl *)
let inherited_cdecls =
let traverse_tree tree_root accum =
let tree_root_descendant = StringMap.find tree_root inheritance_forest in
let accum_with_tree_root_mapping = StringMap.add tree_root (StringMap.find tree_root
    cdecl_lookup) accum in
add_inherited_fields tree_root tree_root_descendant accum_with_tree_root_mapping
in
StringSet.fold traverse_tree roots StringMap.empty
in
(* build a list of updated cdecls corresponding to the sequence of cdecls in `cdecls` *)
let add_inherited_cdecl cdecl accum =
let inherited_cdecl =
try StringMap.find cdecl.cname inherited_cdecls
with | Not_found -> cdecl
in
inherited_cdecl::accum
in
let result = List.fold_right add_inherited_cdecl cdecls [] in
result

let convert_cdecls_to_sast class_maps reserved (cdecls:Ast.class_decl list) =
```

```
let find_main = (fun f -> match f.sfname with FName n -> n = "main" | _ -> false) in
let get_main func_list =
let mains = (List.find_all find_main func_list) in
if List.length mains < 1 then
raise Exceptions.MainNotDefined
else if List.length mains > 1 then
raise Exceptions.MultipleMainsDefined
else List.hd mains
in
let remove_main func_list =
List.filter (fun f -> not (find_main f)) func_list
in
let find_default_constructor cdecl clist =
let default_cname = cdecl.cname ^ "." ^ "constructor" in
let find_default_c f =
match f.sfname with FName n -> n = default_cname | _ -> false
in
try let _ = List.find find_default_c clist in
clist
with | Not_found ->
let default_c = default_sc cdecl.cname in
default_c :: clist
in
let handle_cdecl cdecl =
let class_map = StringMap.find cdecl.cname class_maps in
let sconstructor_list = List.fold_left (fun l c -> (convert_constructor_to_sfdecl
    ↪ class_maps reserved class_map cdecl.cname c) :: l) [] cdecl.cbody.constructors in
let sconstructor_list = find_default_constructor cdecl sconstructor_list in
let func_list = List.fold_left (fun l f -> (convert_fdecl_to_sfdecl class_maps reserved
    ↪ class_map cdecl.cname f) :: l) [] cdecl.cbody.methods in
let sfunc_list = remove_main func_list in
let scdecl = convert_cdecl_to_sast sfunc_list cdecl in
(scdecl, func_list @ sconstructor_list)
in
let iter_cdecls t c =
let scdecl = handle_cdecl c in
(fst scdecl :: fst t, snd scdecl @ snd t)
in
let scdecl_list, func_list = List.fold_left iter_cdecls ([], []) cdecls in
let main = get_main func_list in
let funcs = remove_main func_list in
(* let funcs = (add_default_constructors cdecls class_maps) @ funcs in *)
{
        classes                 = scdecl_list;
        functions                 = funcs;
        main                        = main;
        reserved                 = reserved;
}

```

```
let add_reserved_functions =
let reserved_stub name return_type formals =
{
        sfname                          = FName(name);
        sreturnType         = return_type;
        sformals                = formals;
        sbody                       = [];
        func_type               = Sast.Reserved;
        overrides                = false;
        source                      = "NA";
}
in
let i32_t = Datatype(Int_t) in
let void_t = Datatype(Void_t) in
let str_t = Arraytype(Char_t, 1) in
let mf t n = Formal(t, n) in (* Make formal *)
let reserved = [
reserved_stub "print"         (void_t)         ([Many(Any)]);
reserved_stub "malloc"         (str_t)         ([mf i32_t "size"]);
reserved_stub "cast"         (Any)                 ([mf Any "in"]);
reserved_stub "sizeof"         (i32_t)         ([mf Any "in"]);
reserved_stub "open"         (i32_t)         ([mf str_t "path"; mf i32_t "flags"]);
reserved_stub "close"         (i32_t)         ([mf i32_t "fd"]);
reserved_stub "read"         (i32_t)         ([mf i32_t "fd"; mf str_t "buf"; mf i32_t
↪  "nbyte"]);
reserved_stub "write"         (i32_t)         ([mf i32_t "fd"; mf str_t "buf"; mf i32_t
↪  "nbyte"]);
reserved_stub "lseek"         (i32_t)         ([mf i32_t "fd"; mf i32_t "offset"; mf
↪  i32_t "whence"]);
reserved_stub "exit"         (void_t)         ([mf i32_t "status"]);
reserved_stub "getchar" (i32_t)            ([]);
reserved_stub "input"         (str_t)         ([]);
] in
reserved

let build_inheritance_forest cdecls cmap =
let handler a cdecl =
match cdecl.extends with
Parent(s)         ->
let new_list = if (StringMap.mem s a) then
cdecl.cname::(StringMap.find s a)
else
[cdecl.cname]
in
Hashtbl.add predecessors s new_list;
(StringMap.add s new_list a)
|         NoParent         -> a
in
let forest = List.fold_left handler StringMap.empty cdecls in
```

```
1074
1075 let handler key value =
1076 if not (StringMap.mem key cmap) then
1077 raise (Exceptions.UndefinedClass key)
1078 in
1079 ignore(StringMap.iter handler forest);
1080 forest
1081
1082 let merge_maps m1 m2 =
1083 StringMap.fold (fun k v a -> StringMap.add k v a) m1 m2
1084
1085 let update_class_maps map_type cmap_val cname cmap_to_update =
1086 let update m map_type =
1087 if map_type = "field_map" then
1088 {
1089         field_map = cmap_val;
1090         func_map = m.func_map;
1091         constructor_map = m.constructor_map;
1092         reserved_map = m.reserved_map;
1093         cdecl = m.cdecl;
1094 }
1095 else m
1096 in
1097 let updated = StringMap.find cname cmap_to_update in
1098 let updated = update updated map_type in
1099 let updated = StringMap.add cname updated cmap_to_update in
1100 updated
1101
1102 let inherit_fields class_maps predecessors =
1103 (* Get basic inheritance map *)
1104 let add_key key pred map = StringMap.add key pred map in
1105 let cmaps_inherit = StringMap.fold add_key class_maps StringMap.empty in
1106 (* Perform accumulation of child classes *)
1107 let add_key key pred maps =
1108 let elem1 = StringSet.add key (fst maps) in
1109 let accum acc child = StringSet.add child acc in
1110 let elem2 = List.fold_left (accum) (snd maps) pred in
1111 (elem1, elem2)
1112 in
1113 let empty_s = StringSet.empty in
1114 let res = StringMap.fold add_key predecessors (empty_s, empty_s) in
1115 let roots = StringSet.diff (fst res) (snd res) in
1116 (*in let _ = print_set_members roots*)
1117 let rec add_inherited_fields predec desc cmap_to_update =
1118 let cmap_inherit accum descendant =
1119 let predec_field_map = (StringMap.find predec accum).field_map in
1120 let desc_field_map = (StringMap.find descendant accum).field_map in
1121 let merged = merge_maps predec_field_map desc_field_map in
1122 let updated = update_class_maps "field_map" merged descendant accum in
```

```
1123 if (StringMap.mem descendant predecessors) then
1124 let descendants_of_descendant = StringMap.find descendant predecessors in
1125 add_inherited_fields descendant descendants_of_descendant updated
1126 else updated
1127 in
1128 List.fold_left cmap_inherit cmap_to_update desc
1129 (* end of add_inherited_fields *)
1130 in
1131 let result = StringSet.fold (fun x a -> add_inherited_fields x (StringMap.find x
     ↪  predecessors) a) roots cmaps_inherit
1132 (*in let _ = print_map result*)
1133 in result
1134
1135 (* TODO Check that this actually works *)
1136 let check_cyclical_inheritance cdecls predecessors =
1137 let handle_predecessor cdecl parent predecessor =
1138 if cdecl.cname = predecessor then
1139 raise(Exceptions.CyclicalDependencyBetween(cdecl.cname, parent))
1140 in
1141 let handle_cdecl cdecl =
1142 if StringMap.mem cdecl.cname predecessors
1143 then
1144 let pred_list = StringMap.find cdecl.cname predecessors in
1145 List.iter (handle_predecessor cdecl (List.hd pred_list)) pred_list
1146 else ()
1147 in
1148 List.iter handle_cdecl cdecls
1149
1150 let build_func_map_inherited_lookup cdecls_inherited =
1151 let build_func_map cdecl =
1152 let add_func m fdecl = StringMap.add (get_name cdecl.cname fdecl) fdecl m in
1153 List.fold_left add_func StringMap.empty cdecl.cbody.methods
1154 in
1155 let add_class_func_map m cdecl = StringMap.add cdecl.cname (build_func_map cdecl) m in
1156 List.fold_left add_class_func_map StringMap.empty cdecls_inherited
1157
1158 let add_inherited_methods cmaps cdecls func_maps_inherited =
1159 let find_cdecl cname =
1160 try List.find (fun cdecl -> cdecl.cname = cname) cdecls
1161 with | Not_found -> raise Not_found
1162 in
1163 let update_with_inherited_methods cname cmap =
1164 let fmap = StringMap.find cname func_maps_inherited in
1165 let cdecl = find_cdecl cname in
1166 {
1167         field_map = cmap.field_map;
1168         func_map = fmap;
1169         constructor_map = cmap.constructor_map;
1170         reserved_map = cmap.reserved_map;
```

```
1171            cdecl = cdecl;
1172    }
1173    in
1174    let add_updated_cmap cname cmap accum = StringMap.add cname
        ↪  (update_with_inherited_methods cname cmap) accum in
1175    StringMap.fold add_updated_cmap cmaps StringMap.empty
1176
1177    let handle_inheritance cdecls class_maps =
1178    let predecessors = build_inheritance_forest cdecls class_maps in
1179    let cdecls_inherited = inherit_fields_cdecls cdecls predecessors in
1180    let func_maps_inherited = build_func_map_inherited_lookup cdecls_inherited in
1181    ignore(check_cyclical_inheritance cdecls predecessors);
1182    let cmaps_with_inherited_fields = inherit_fields class_maps predecessors in
1183    let cmaps_inherited = add_inherited_methods cmaps_with_inherited_fields cdecls_inherited
        ↪  func_maps_inherited in
1184    cmaps_inherited, cdecls_inherited
1185
1186    let generate_struct_indexes cdecls =
1187    let cdecl_handler index cdecl =
1188    Hashtbl.add struct_indexes cdecl.cname index
1189    in
1190    List.iteri cdecl_handler cdecls
1191
1192    (* Main method for analyzer *)
1193    let analyze filename program = match program with
1194    Program(includes, classes) ->
1195    (* Include code from external files *)
1196    let cdecls = process_includes filename includes classes in
1197    ignore(generate_struct_indexes cdecls);
1198
1199    (* Add built-in functions *)
1200    let reserved = add_reserved_functions in
1201    (* Generate the class_maps for look up in checking functions *)
1202    let class_maps = build_class_maps reserved cdecls in
1203    let class_maps, cdecls = handle_inheritance cdecls class_maps in
1204    let sast = convert_cdecls_to_sast class_maps reserved cdecls in
1205    sast
```

ast.ml

```
type op = Add | Sub | Mult | Div | Equal | Neq | Less | Leq | Greater | Geq | And | Not |
↪  Or | Mod
type scope = Private | Public
type primitive = Int_t | Float_t | Void_t | Bool_t | Char_t | Objecttype of string |
↪  ConstructorType | Null_t
type datatype = Arraytype of primitive * int | Datatype of primitive | Any

type extends = NoParent | Parent of string
type fname = Constructor | FName of string
type formal = Formal of datatype * string | Many of datatype

type expr =
Int_Lit of int
|          Boolean_Lit of bool
|          Float_Lit of float
|          String_Lit of string
|          Char_Lit of char
|          This
|          Id of string
|          Binop of expr * op * expr
|          Assign of expr * expr
|          Noexpr
|          ArrayCreate of datatype * expr list
|          ArrayAccess of expr * expr list
|          ObjAccess of expr * expr
|          Call of string * expr list
|    ObjectCreate of string * expr list
|          ArrayPrimitive of expr list
|           Unop of op * expr
|          Null
|          Delete of expr

type stmt =
Block of stmt list
|          Expr of expr
|          Return of expr
|          If of expr * stmt * stmt
|          For of expr * expr * expr * stmt
|          While of expr * stmt
|           Break
|    Continue
|    Local of datatype * string * expr

type field = Field of scope * datatype * string
type include_stmt = Include of string

type func_decl = {
```

```
46          scope : scope;
47          fname : fname;
48          returnType : datatype;
49          formals : formal list;
50          body : stmt list;
51          overrides : bool;
52          root_cname : string option;
53  }
54
55  type cbody = {
56          fields : field list;
57          constructors : func_decl list;
58          methods : func_decl list;
59  }
60
61  type class_decl = {
62          cname : string;
63          extends : extends;
64          cbody: cbody;
65  }
66
67  type program = Program of include_stmt list * class_decl list
```

bindings.c

```c
#include <stdio.h>
#include <stdlib.h>

#define INIT_SIZE 100

struct s {
        int x;
        int y;
};

char* input() {
        int initial_size = INIT_SIZE;
        char* str = malloc(initial_size);
        int index = 0;
        char tmp = '0';
        while((tmp = getchar() )!= '\n') {
                if(index >= initial_size - 1) {
                        str = realloc(str, initial_size *= 2);
                }
                str[index++] = tmp;
        }
        str[index] = '\0';
        return str;
}

void rec_init(long* arr, int curr_offset, int* static_offsets, int* indexes, int* dims,
↪  int dimc, int dim_curr) {

        //Assign length
        arr[curr_offset] = dims[dim_curr];

        if(dim_curr + 1 >= dimc)
        return;

        //Determine the static offset and the dynamic offset
        int static_offset = static_offsets[dim_curr];
        int dynamic_offset = 0;
        for(int i = 0; i < dim_curr; i++) {
                int tmp = indexes[i];
                for(int j = i + 1; j <= dim_curr; j++) {
                        tmp *= dims[j];
                }
                dynamic_offset += tmp;
        }

        //Iterate through position and iniitalize subarrays
        //Set local indexes to pointers to the subarrays
```

```
47          for(int i = 0; i < dims[dim_curr]; i++) {
48                  int offset = (static_offset + (dynamic_offset + i) * (dims[dim_curr + 1]
                    ↪  + 1));
49
50                  long* sub = arr + offset;
51                  arr[curr_offset + 1 + i] = (long) sub;
52
53                  indexes[dim_curr] = i;
54                  rec_init(arr, offset, static_offsets, indexes, dims, dimc, dim_curr + 1);
55          }
56  }
57
58  long* init_arr(int* dims, int dimc) {
59
60          int static_offsets[dimc];
61          int total = 0;
62          for(int i = 0; i < dimc; i++) {
63                  static_offsets[i] = 1;
64                  for(int j = 0; j < i; j++) {
65                          static_offsets[i] *= dims[j];
66                  }
67                  static_offsets[i] *= dims[i] + 1;
68                  static_offsets[i] += total;
69                  total = static_offsets[i];
70          }
71
72          int indexes[dimc];
73          for(int i = 0; i < dimc; i++) {
74                  indexes[i] = 0;
75          }
76
77          //Get total length of array
78          int length = 0;
79          for(int i = 0; i < dimc; i++) {
80                  int tmp = 1;
81                  for(int j = i - 1; j >= 0; j--) {
82                          tmp *= dims[j];
83                  }
84                  tmp *= dims[i] + 1;
85                  length += tmp;
86          }
87
88          //Malloc array
89          long* arr = malloc(length);
90
91          //Set all values to 0 initially
92          for(int i = 0 ; i < length; i++) {
93                  arr[i] = 0;
94          }
```

```
95
96          //Initialize the entire array
97          rec_init(arr, 0, static_offsets, indexes, dims, dimc, 0);
98
99          return arr;
100 }
101
102 // int main() {
103
104         //          //Array creation
105         //          int dims[5] = {2, 3, 4, 5, 6};
106         //          int dimc = 5;
107
108         //          long* arr = init_arr(dims, dimc);
109
110         //          //Get total length of array
111         //          int length = 0;
112         //          for(int i = 0; i < dimc; i++) {
113                 //                  int tmp = 1;
114                 //                  for(int j = i - 1; j >= 0; j--) {
115                         //                          tmp *= dims[j];
116                         //                  }
117                 //                  tmp *= dims[i] + 1;
118                 //                  length += tmp;
119                 //          }
120
121         //          for(int i = 0; i < length; i++) {
122                 //                  printf("val: %ld | addr: %ld\n", arr[i], (long) arr +
                    ↪  i);
123                 //          }
124         //          printf("\n");
125         // }
```

codegen.ml

```
(* ===----------------------------------------------------------------------===
* Code Generation
*===----------------------------------------------------------------------===*)

open Llvm
open Ast
open Sast
open Analyzer
open Exceptions
open Batteries
open Hashtbl
open Conf

open Llvm.MemoryBuffer
open Llvm_bitreader

let context = global_context ()
let the_module = create_module context "Dice Codegen"
let builder = builder context
let named_values:(string, llvalue) Hashtbl.t = Hashtbl.create 50
let named_params:(string, llvalue) Hashtbl.t = Hashtbl.create 50
let struct_types:(string, lltype) Hashtbl.t = Hashtbl.create 10
let struct_field_indexes:(string, int) Hashtbl.t = Hashtbl.create 50

let i32_t = i32_type context;;
let i8_t = i8_type context;;
let f_t = double_type context;;
let i1_t = i1_type context;;
let str_t = pointer_type i8_t;;
let i64_t = i64_type context;;
let void_t = void_type context;;

let str_type = Arraytype(Char_t, 1)

let (br_block) = ref (block_of_value (const_int i32_t 0))
let (cont_block) = ref (block_of_value (const_int i32_t 0))
let is_loop = ref false

let debug = fun s ->
print_endline ("<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<"^s);
dump_module the_module;
print_endline (">>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>"^s);
()

let rec get_ptr_type datatype = match datatype with
Arraytype(t, 0) -> get_type (Datatype(t))
|       Arraytype(t, 1) -> pointer_type (get_type (Datatype(t)))
```

```
48 |        Arraytype(t, i) -> pointer_type (get_ptr_type (Arraytype(t, (i-1))))
49 |         _ -> raise(Exceptions.InvalidStructType "Array Pointer Type")
50
51 and find_struct name =
52 try Hashtbl.find struct_types name
53 with | Not_found -> raise(Exceptions.InvalidStructType name)
54
55 and get_type (datatype:Ast.datatype) = match datatype with
56 Datatype(Int_t) -> i32_t
57 |         Datatype(Float_t) -> f_t
58 |         Datatype(Bool_t) -> i1_t
59 |         Datatype(Char_t) -> i8_t
60 |         Datatype(Void_t) -> void_t
61 |         Datatype(Null_t) -> i32_t
62 |         Datatype(Objecttype(name)) -> pointer_type(find_struct name)
63 |         Arraytype(t, i) -> get_ptr_type (Arraytype(t, (i)))
64 |         d -> raise(Exceptions.InvalidStructType (Utils.string_of_datatype d))
65
66 (* cast will return an llvalue of the desired type *)
67 (* The commented out casts are unsupported actions in Dice *)
68 let cast lhs rhs lhsType rhsType llbuilder =
69 match (lhsType, rhsType) with
70 (* int to,__ ) ( using const_sitofp for signed ints *)
71 (Datatype(Int_t), Datatype(Int_t))                                  -> (lhs, rhs),
   ↪  Datatype(Int_t)
72 |         (Datatype(Int_t), Datatype(Char_t))                                  ->
   ↪  (build_uitofp lhs i8_t "tmp" llbuilder, rhs), Datatype(Char_t)
73 (* |           (Datatype(Int_t), Datatype(Bool_t))                                  ->
   ↪  (lhs, const_zext rhs i32_t) *)
74 |   (Datatype(Int_t), Datatype(Float_t))                           -> (build_sitofp lhs f_t
   ↪  "tmp" llbuilder, rhs), Datatype(Float_t)
75
76 (* char to,__)  ( using uitofp since char isn't signed *)
77 |   (Datatype(Char_t), Datatype(Int_t))                            -> (lhs, build_uitofp rhs
   ↪  i8_t "tmp" llbuilder), Datatype(Char_t)
78 |   (Datatype(Char_t), Datatype(Char_t))                           -> (lhs, rhs),
   ↪  Datatype(Char_t)
79 (* |         (Datatype(Char_t), Datatype(Bool_t))                          -> (lhs,
   ↪  const_zext rhs i8_t) *)
80 (* |         (Datatype(Char_t), Datatype(Float_t))                          ->
   ↪  (const_uitofp lhs f_t, rhs) *)
81
82 (* bool to,__)  ( zext fills the empty bits with zeros, zero extension *)
83 (* |           (Datatype(Bool_t), Datatype(Int_t))                           -> (const_zext
   ↪  lhs i32_t, rhs) *)
84 (* |         (Datatype(Bool_t), Datatype(Char_t))                          -> (const_zext
   ↪  lhs i8_t, rhs) *)
85 |          (Datatype(Bool_t), Datatype(Bool_t))                          -> (lhs, rhs),
   ↪  Datatype(Bool_t)
```

```
86  (* |          (Datatype(Bool_t), Datatype(Float_t))                    ->
    ↪  (const_uitofp lhs f_t, rhs) *)
87
88  (* float to,__) ( using fptosi for signed ints *)
89  |   (Datatype(Float_t), Datatype(Int_t))                        -> (lhs, build_sitofp
    ↪  rhs f_t "tmp" llbuilder), Datatype(Float_t)
90  (* |         (Datatype(Float_t), Datatype(Char_t))                    -> (lhs,
    ↪  const_uitofp rhs f_t) *)
91  (* |          (Datatype(Float_t), Datatype(Bool_t))                     -> (lhs,
    ↪  const_uitofp rhs f_t) *)
92  |   (Datatype(Float_t), Datatype(Float_t))                       -> (lhs, rhs),
    ↪  Datatype(Float_t)
93
94  | Datatype(Objecttype(d)), Datatype(Null_t)                      -> (lhs, rhs), lhsType
95  | Datatype(Null_t), Datatype(Objecttype(d))                  -> (rhs, lhs), rhsType
96  | Datatype(Objecttype(d)), t                                          ->
    ↪  raise(Exceptions.CanOnlyCompareObjectsWithNull(d, (Utils.string_of_datatype t)))
97
98  | Arraytype(d, s), Datatype(Null_t)                                 -> (lhs, rhs),
    ↪  lhsType
99  | Datatype(Null_t), Arraytype(d, s)                               -> (rhs, lhs),
    ↪  rhsType
100 | Arraytype(d, _), t                                                      ->
    ↪  raise(Exceptions.CanOnlyCompareArraysWithNull(Utils.string_of_primitive d,
    ↪  (Utils.string_of_datatype t)))
101
102 |         _
    ↪                                                                       ->
    ↪  raise (Exceptions.CannotCastTypeException(Utils.string_of_datatype lhsType,
    ↪  Utils.string_of_datatype rhsType))
103
104 let rec handle_binop e1 op e2 d llbuilder =
105 (* Get the types of e1 and e2 *)
106 let type1 = Analyzer.get_type_from_sexpr e1 in
107 let type2 = Analyzer.get_type_from_sexpr e2 in
108
109 (* Generate llvalues from e1 and e2 *)
110
111 let e1 = codegen_sexpr llbuilder e1 in
112 let e2 = codegen_sexpr llbuilder e2 in
113
114 let float_ops op e1 e2 =
115 match op with
116 Add                 -> build_fadd e1 e2 "flt_addtmp" llbuilder
117 |         Sub                  -> build_fsub e1 e2 "flt_subtmp" llbuilder
118 |         Mult                  -> build_fmul e1 e2 "flt_multmp" llbuilder
119 |         Div                  -> build_fdiv e1 e2 "flt_divtmp" llbuilder
120 |         Mod                  -> build_frem e1 e2 "flt_sremtmp" llbuilder
121 |         Equal                  -> build_fcmp Fcmp.Oeq e1 e2 "flt_eqtmp" llbuilder
```

```
122 |         Neq                 -> build_fcmp Fcmp.One e1 e2 "flt_neqtmp" llbuilder
123 |         Less                 -> build_fcmp Fcmp.Ult e1 e2 "flt_lesstmp" llbuilder
124 |         Leq                 -> build_fcmp Fcmp.Ole e1 e2 "flt_leqtmp" llbuilder
125 |         Greater                -> build_fcmp Fcmp.Ogt e1 e2 "flt_sgttmp" llbuilder
126 |         Geq                 -> build_fcmp Fcmp.Oge e1 e2 "flt_sgetmp" llbuilder
127 |         _                        -> raise Exceptions.FloatOpNotSupported
128
129 in
130
131 (* chars are considered ints, so they will use int_ops as well*)
132 let int_ops op e1 e2 =
133 match op with
134 Add                 -> build_add e1 e2 "addtmp" llbuilder
135 |         Sub                 -> build_sub e1 e2 "subtmp" llbuilder
136 |         Mult                 -> build_mul e1 e2 "multmp" llbuilder
137 |         Div                 -> build_sdiv e1 e2 "divtmp" llbuilder
138 |         Mod                 -> build_srem e1 e2 "sremtmp" llbuilder
139 |         Equal                  -> build_icmp Icmp.Eq e1 e2 "eqtmp" llbuilder
140 |         Neq                 -> build_icmp Icmp.Ne e1 e2 "neqtmp" llbuilder
141 |         Less                 -> build_icmp Icmp.Slt e1 e2 "lesstmp" llbuilder
142 |         Leq                 -> build_icmp Icmp.Sle e1 e2 "leqtmp" llbuilder
143 |         Greater                -> build_icmp Icmp.Sgt e1 e2 "sgttmp" llbuilder
144 |         Geq                 -> build_icmp Icmp.Sge e1 e2 "sgetmp" llbuilder
145 |         And                 -> build_and e1 e2 "andtmp" llbuilder
146 |         Or                        -> build_or  e1 e2 "ortmp" llbuilder
147 |         _                        -> raise Exceptions.IntOpNotSupported
148 in
149
150 let obj_ops op e1 e2 =
151 match op with
152 Equal -> build_is_null e1 "tmp" llbuilder
153 |         Neq -> build_is_not_null e1 "tmp" llbuilder
154 |         _         -> raise (Exceptions.ObjOpNotSupported(Utils.string_of_op op))
155 in
156
157 let (e1, e2), d = cast e1 e2 type1 type2 llbuilder in
158
159 let type_handler d = match d with
160 Datatype(Float_t)   -> float_ops op e1 e2
161 |        Datatype(Int_t)
162 |   Datatype(Bool_t)
163 |         Datatype(Char_t)         -> int_ops op e1 e2
164 |         Datatype(Objecttype(_))
165 |         Arraytype(_, _) -> obj_ops op e1 e2
166 |   _ -> raise Exceptions.InvalidBinopEvaluationType
167 in
168
169 type_handler d
170
```

```
171 and handle_unop op e d llbuilder =
172 (* Get the type of e *)
173 let eType = Analyzer.get_type_from_sexpr e in
174 (* Get llvalue  *)
175 let e = codegen_sexpr llbuilder e in
176
177 let unops op eType e = match (op, eType) with
178 (Sub, Datatype(Int_t))                ->  build_neg e "int_unoptmp" llbuilder
179 |   (Sub, Datatype(Float_t))          ->          build_fneg e "flt_unoptmp" llbuilder
180 |   (Not, Datatype(Bool_t))          ->  build_not e "bool_unoptmp" llbuilder
181 |    _           -> raise Exceptions.UnopNotSupported        in
182
183 let unop_type_handler d = match d with
184 Datatype(Float_t)
185 |        Datatype(Int_t)
186 |   Datatype(Bool_t)        -> unops op eType e
187 |   _ -> raise Exceptions.InvalidUnopEvaluationType
188 in
189
190 unop_type_handler d
191
192 and func_lookup fname =
193 match (lookup_function fname the_module) with
194 None         -> raise (Exceptions.LLVMFunctionNotFound fname)
195 |          Some f          -> f
196
197 and codegen_print el llbuilder =
198 let printf = func_lookup "printf" in
199 let tmp_count = ref 0 in
200 let incr_tmp = fun x -> incr tmp_count in
201
202 let map_expr_to_printfexpr expr =
203 let exprType = Analyzer.get_type_from_sexpr expr in
204 match exprType with
205 Datatype(Bool_t) ->
206 incr_tmp ();
207 let tmp_var = "tmp" ^ (string_of_int !tmp_count) in
208 let trueStr = SString_Lit("true") in
209 let falseStr = SString_Lit("false") in
210 let id = SId(tmp_var, str_type) in
211 ignore(codegen_stmt llbuilder (SLocal(str_type, tmp_var, SNoexpr)));
212 ignore(codegen_stmt llbuilder (SIf(expr,
213 SExpr(SAssign(id, trueStr, str_type), str_type),
214 SExpr(SAssign(id, falseStr, str_type), str_type)
215 )));
216 codegen_sexpr llbuilder id
217 | _ -> codegen_sexpr llbuilder expr
218 in
219
```

```
220 let params = List.map map_expr_to_printfexpr el in
221 let param_types = List.map (Analyzer.get_type_from_sexpr) el in
222
223 let map_param_to_string = function
224 Arraytype(Char_t, 1)         -> "%s"
225 |         Datatype(Int_t)                -> "%d"
226 |         Datatype(Float_t)                -> "%f"
227 |         Datatype(Bool_t)                -> "%s"
228 |         Datatype(Char_t)                -> "%c"
229 |         _                                                -> raise
    ↪  (Exceptions.InvalidTypePassedToPrintf)
230 in
231 let const_str = List.fold_left (fun s t -> s ^ map_param_to_string t) "" param_types in
232 let s = codegen_sexpr llbuilder (SString_Lit(const_str)) in
233 let zero = const_int i32_t 0 in
234 let s = build_in_bounds_gep s [| zero |] "tmp" llbuilder in
235 build_call printf (Array.of_list (s :: params)) "tmp" llbuilder
236
237 and codegen_func_call fname el d llbuilder =
238 let f = func_lookup fname in
239 let params = List.map (codegen_sexpr llbuilder) el in
240 match d with
241 Datatype(Void_t) -> build_call f (Array.of_list params) "" llbuilder
242 |         _ ->                                 build_call f (Array.of_list params) "tmp"
    ↪  llbuilder
243
244 and codegen_sizeof el llbuilder =
245 let type_of = Analyzer.get_type_from_sexpr (List.hd el) in
246 let type_of = get_type type_of in
247 let size_of = size_of type_of in
248 build_bitcast size_of i32_t "tmp" llbuilder
249
250 and codegen_cast el d llbuilder =
251 let cast_malloc_to_objtype lhs currType newType llbuilder = match newType with
252 Datatype(Objecttype(x)) ->
253 let obj_type = get_type (Datatype(Objecttype(x))) in
254 build_pointercast lhs obj_type "tmp" llbuilder
255 |         _ as t -> raise (Exceptions.CannotCastTypeException(Utils.string_of_datatype
    ↪  currType, Utils.string_of_datatype t))
256 in
257 let expr = List.hd el in
258 let t = Analyzer.get_type_from_sexpr expr in
259 let lhs = match expr with
260 |         Sast.SId(id, d) -> codegen_id false false id d llbuilder
261 |          SObjAccess(e1, e2, d) -> codegen_obj_access false e1 e2 d llbuilder
262 |         SArrayAccess(se, sel, d) -> codegen_array_access true se sel d llbuilder
263 | _ -> codegen_sexpr llbuilder expr
264 in
265 cast_malloc_to_objtype lhs t d llbuilder
```

```
266
267 and codegen_call llbuilder d el = function
268 "print"         -> codegen_print el llbuilder
269 |         "sizeof"        -> codegen_sizeof el llbuilder
270 |         "cast"                -> codegen_cast el d llbuilder
271 |         "malloc"         -> codegen_func_call "malloc" el d llbuilder
272 |         "open"                -> codegen_func_call "open" el d llbuilder
273 |         "write"               -> codegen_func_call "write" el d llbuilder
274 |         "close"               -> codegen_func_call "close" el d llbuilder
275 |         "read"                -> codegen_func_call "read" el d llbuilder
276 |         "lseek"         -> codegen_func_call "lseek" el d llbuilder
277 |         "exit"                -> codegen_func_call "exit" el d llbuilder
278 |         "input"         -> codegen_func_call "input" el d llbuilder
279 |   "getchar"   -> codegen_func_call "getchar" el d llbuilder
280 |         _ as fname         -> raise (Exceptions.UnableToCallFunctionWithoutParent
    ↪  fname)(* codegen_func_call fname el llbuilder *)
281
282 and codegen_id isDeref checkParam id d llbuilder =
283 if isDeref then
284 try Hashtbl.find named_params id
285 with | Not_found ->
286 try let _val = Hashtbl.find named_values id in
287 build_load _val id llbuilder
288 with | Not_found -> raise (Exceptions.UnknownVariable id)
289 else
290 try Hashtbl.find named_values id
291 with | Not_found ->
292 try
293 let _val = Hashtbl.find named_params id in
294 if checkParam then raise (Exceptions.CannotAssignParam id)
295 else _val
296 with | Not_found -> raise (Exceptions.UnknownVariable id)
297
298 and codegen_assign lhs rhs d llbuilder =
299 let rhsType = Analyzer.get_type_from_sexpr rhs in
300 (* Special case '=' because we don't want to emit the LHS as an
301 * expression. *)
302 let lhs, isObjAccess = match lhs with
303 |         Sast.SId(id, d) -> codegen_id false false id d llbuilder, false
304 |          SObjAccess(e1, e2, d) -> codegen_obj_access false e1 e2 d llbuilder, true
305 |         SArrayAccess(se, sel, d) -> codegen_array_access true se sel d llbuilder, true
306 | _ -> raise Exceptions.AssignLHSMustBeAssignable
307 in
308 (* Codegen the rhs. *)
309 let rhs = match rhs with
310 |         Sast.SId(id, d) -> codegen_id false false id d llbuilder
311 |          SObjAccess(e1, e2, d) -> codegen_obj_access true e1 e2 d llbuilder
312 | _ -> codegen_sexpr llbuilder rhs
313 in
```

```
314 let rhs = match d with
315 Datatype(Objecttype(_))        ->
316 if isObjAccess then rhs
317 else build_load rhs "tmp" llbuilder
318 |         Datatype(Null_t) -> const_null (get_type d)
319 | _ -> rhs
320 in
321 let rhs = match d, rhsType with
322 Datatype(Char_t), Datatype(Int_t) -> build_uitofp rhs i8_t "tmp" llbuilder
323 |         Datatype(Int_t), Datatype(Char_t) -> build_uitofp rhs i32_t "tmp" llbuilder
324 |         _ -> rhs
325 in
326 (* Lookup the name. *)
327 ignore(build_store rhs lhs llbuilder);
328 rhs
329
330 and deref ptr t llbuilder =
331 build_gep ptr (Array.of_list [ptr]) "tmp" llbuilder
332
333 and codegen_obj_access isAssign lhs rhs d llbuilder =
334 let codegen_func_call param_ty fptr parent_expr el d llbuilder =
335 let match_sexpr se = match se with
336 SId(id, d) -> let isDeref = match d with
337 Datatype(Objecttype(_)) -> false
338 |         _ -> true
339 in codegen_id isDeref false id d llbuilder
340 |         se -> codegen_sexpr llbuilder se
341 in
342 let parent_expr = build_pointercast parent_expr param_ty "tmp" llbuilder in
343 let params = List.map match_sexpr el in
344 match d with
345 Datatype(Void_t) -> build_call fptr (Array.of_list (parent_expr :: params)) "" llbuilder
346 |         _ -> build_call fptr (Array.of_list (parent_expr :: params)) "tmp" llbuilder
347 in
348 let check_lhs = function
349 SId(s, d)                                -> codegen_id false false s d llbuilder
350 |         SArrayAccess(e, el, d)        -> codegen_array_access false e el d llbuilder
351 |         se         -> raise (Exceptions.LHSofRootAccessMustBeIDorFunc
    ↪  (Utils.string_of_sexpr se))
352 in
353 (* Needs to be changed *)
354 let rec check_rhs isLHS parent_expr parent_type =
355 let parent_str = Utils.string_of_object parent_type in
356 function
357 (* Check fields in parent *)
358 SId(field, d) ->
359 let search_term = (parent_str ^ "." ^ field) in
360 let field_index = Hashtbl.find struct_field_indexes search_term in
361 let _val = build_struct_gep parent_expr field_index field llbuilder in
```

```
362 let _val = match d with
363 Datatype(Objecttype(_)) ->
364 if not isAssign then _val
365 else build_load _val field llbuilder
366 | _ ->
367 if not isAssign then
368 _val
369 else
370 build_load _val field llbuilder
371 in
372 _val
373
374 |         SArrayAccess(e, el, d) ->
375
376 let ce = check_rhs false parent_expr parent_type e in
377 let index = codegen_sexpr llbuilder (List.hd el) in
378 let index = match d with
379 Datatype(Char_t) -> index
380 |         _ -> build_add index (const_int i32_t 1) "tmp" llbuilder
381 in
382 let _val = build_gep ce [| index |] "tmp" llbuilder in
383 if isLHS && isAssign
384 then _val
385 else build_load _val "tmp" llbuilder
386
387 (* Check functions in parent *)
388 |         SCall(fname, el, d, index)         ->
389 let index = const_int i32_t index in
390 let c_index = build_struct_gep parent_expr 0 "cindex" llbuilder in
391 let c_index = build_load c_index "cindex" llbuilder in
392 let lookup = func_lookup "lookup" in
393 let fptr = build_call lookup [| c_index; index |] "fptr" llbuilder in
394 let fptr2 = func_lookup fname in
395 let f_ty = type_of fptr2 in
396 let param1 = param fptr2 0 in
397 let param_ty = type_of param1 in
398 let fptr = build_pointercast fptr f_ty fname llbuilder in
399 let ret = codegen_func_call param_ty fptr parent_expr el d llbuilder in
400 let ret = ret
401 (* if not isLHS && not isAssign then
402 build_load ret "tmp" llbuilder
403 else
404 ret *)
405 in
406 ret
407 (* Set parent, check if base is field *)
408 |         SObjAccess(e1, e2, d)         ->
409 let e1_type = Analyzer.get_type_from_sexpr e1 in
410 let e1 = check_rhs true parent_expr parent_type e1 in
```

```
411 let e2 = check_rhs true e1 e1_type e2 in
412 e2
413 |          _ as e -> raise (Exceptions.InvalidAccessLHS (Utils.string_of_sexpr e))
414 in
415 let lhs_type = Analyzer.get_type_from_sexpr lhs in
416 match lhs_type with
417 Arraytype(_, _) ->
418 let lhs = codegen_sexpr llbuilder lhs in
419 let _ = match rhs with
420 SId("length", _) -> "length"
421 |          _ -> raise(Exceptions.CanOnlyAccessLengthOfArray)
422 in
423 let _val = build_gep lhs [| (const_int i32_t 0) |] "tmp" llbuilder in
424 build_load _val "tmp" llbuilder
425 |          _ ->
426 let lhs = check_lhs lhs in
427 let rhs = check_rhs true lhs lhs_type rhs in
428 rhs
429
430 and codegen_obj_create fname el d llbuilder =
431 let f = func_lookup fname in
432 let params = List.map (codegen_sexpr llbuilder) el in
433 let obj = build_call f (Array.of_list params) "tmp" llbuilder in
434 obj
435
436 and codegen_string_lit s llbuilder =
437 if s = "true" then build_global_stringptr "true" "tmp" llbuilder
438 else if s = "false" then build_global_stringptr "false" "tmp" llbuilder
439 else build_global_stringptr s "tmp" llbuilder
440
441 and codegen_array_access isAssign e el d llbuilder =
442 let index = codegen_sexpr llbuilder (List.hd el) in
443 let index = match d with
444 Datatype(Char_t) -> index
445 |          _ -> build_add index (const_int i32_t 1) "tmp" llbuilder
446 in
447 let arr = codegen_sexpr llbuilder e in
448 let _val = build_gep arr [| index |] "tmp" llbuilder in
449 if isAssign
450 then _val
451 else build_load _val "tmp" llbuilder
452
453 and initialise_array arr arr_len init_val start_pos llbuilder =
454 let new_block label =
455 let f = block_parent (insertion_block llbuilder) in
456 append_block (global_context ()) label f
457 in
458 let bbcurr = insertion_block llbuilder in
459 let bbcond = new_block "array.cond" in
```

```
460 let bbbody = new_block "array.init" in
461 let bbdone = new_block "array.done" in
462 ignore (build_br bbcond llbuilder);
463 position_at_end bbcond llbuilder;
464
465 (* Counter into the length of the array *)
466 let counter = build_phi [const_int i32_t start_pos, bbcurr] "counter" llbuilder in
467 add_incoming ((build_add counter (const_int i32_t 1) "tmp" llbuilder), bbbody) counter;
468 let cmp = build_icmp Icmp.Slt counter arr_len "tmp" llbuilder in
469 ignore (build_cond_br cmp bbbody bbdone llbuilder);
470 position_at_end bbbody llbuilder;
471
472 (* Assign array position to init_val *)
473 let arr_ptr = build_gep arr [| counter |] "tmp" llbuilder in
474 ignore (build_store init_val arr_ptr llbuilder);
475 ignore (build_br bbcond llbuilder);
476 position_at_end bbdone llbuilder
477
478 and codegen_array_create llbuilder t expr_type el =
479 if(List.length el > 1) then raise(Exceptions.ArrayLargerThan1Unsupported)
480 else
481 match expr_type with
482 Arraytype(Char_t, 1) ->
483 let e = List.hd el in
484 let size = (codegen_sexpr llbuilder e) in
485 let t = get_type t in
486 let arr = build_array_malloc t size "tmp" llbuilder in
487 let arr = build_pointercast arr (pointer_type t) "tmp" llbuilder in
488 (* initialise_array arr size (const_int i32_t 0) 0 llbuilder; *)
489 arr
490 |          _ ->
491 let e = List.hd el in
492 let t = get_type t in
493
494 (* This will not work for arrays of objects *)
495 let size = (codegen_sexpr llbuilder e) in
496 let size_t = build_intcast (size_of t) i32_t "tmp" llbuilder in
497 let size = build_mul size_t size "tmp" llbuilder in
498 let size_real = build_add size (const_int i32_t 1) "arr_size" llbuilder in
499
500 let arr = build_array_malloc t size_real "tmp" llbuilder in
501 let arr = build_pointercast arr (pointer_type t) "tmp" llbuilder in
502
503 let arr_len_ptr = build_pointercast arr (pointer_type i32_t) "tmp" llbuilder in
504
505 (* Store length at this position *)
506 ignore(build_store size_real arr_len_ptr llbuilder);
507 initialise_array arr_len_ptr size_real (const_int i32_t 0) 0 llbuilder;
508 arr
```

```
509
510 and codegen_array_prim d el llbuilder =
511 let t = d in
512 let size = (const_int i32_t ((List.length el))) in
513 let size_real = (const_int i32_t ((List.length el) + 1)) in
514 let t = get_type t in
515 let arr = build_array_malloc t size_real "tmp" llbuilder in
516 let arr = build_pointercast arr t "tmp" llbuilder in
517 let size_casted = build_bitcast size t "tmp" llbuilder in
518 ignore(if d = Arraytype(Char_t, 1) then ignore(build_store size_casted arr llbuilder););
    ↪  (* Store length at this position *)
519 (* initialise_array arr size_real (const_int i32_t 0) 1 llbuilder; *)
520
521 let llvalues = List.map (codegen_sexpr llbuilder) el in
522 List.iteri (fun i llval ->
523 let arr_ptr = build_gep arr [| (const_int i32_t (i+1)) |] "tmp" llbuilder in
524 ignore(build_store llval arr_ptr llbuilder);  ) llvalues;
525 arr
526
527 and codegen_delete e llbuilder =
528 let ce = match e with
529 SId(id, d) -> codegen_id false false id d llbuilder
530 |         _ -> codegen_sexpr llbuilder e
531 in
532 build_free ce llbuilder
533
534 and codegen_sexpr llbuilder = function
535 SInt_Lit(i)                          -> const_int i32_t i
536 |   SBoolean_Lit(b)                      -> if b then const_int i1_t 1 else const_int
    ↪  i1_t 0
537 |   SFloat_Lit(f)                        -> const_float f_t f
538 |   SString_Lit(s)                       -> codegen_string_lit s llbuilder
539 |   SChar_Lit(c)                         -> const_int i8_t (Char.code c)
540 |   SId(id, d)                         -> codegen_id true false id d llbuilder
541 |   SBinop(e1, op, e2, d)              -> handle_binop e1 op e2 d llbuilder
542 |   SAssign(e1, e2, d)                 -> codegen_assign e1 e2 d llbuilder
543 |   SNoexpr                          -> build_add (const_int i32_t 0) (const_int i32_t 0)
    ↪  "nop" llbuilder
544 |   SArrayCreate(t, el, d)             -> codegen_array_create llbuilder t d el
545 |   SArrayAccess(e, el, d)             -> codegen_array_access false e el d llbuilder
546 |   SObjAccess(e1, e2, d)              -> codegen_obj_access true e1 e2 d llbuilder
547 |   SCall(fname, el, d, _)               -> codegen_call llbuilder d el fname
548 |   SObjectCreate(id, el, d)           -> codegen_obj_create id el d llbuilder
549 |   SArrayPrimitive(el, d)             -> codegen_array_prim d el llbuilder
550 |   SUnop(op, e, d)                    -> handle_unop op e d llbuilder
551 |   SNull                                  -> const_null i32_t
552 |         SDelete e                                          -> codegen_delete e
    ↪  llbuilder
553
```

```
554 and codegen_if_stmt exp then_ (else_:Sast.sstmt) llbuilder =
555 let cond_val = codegen_sexpr llbuilder exp in
556
557 (* Grab the first block so that we might later add the conditional branch
558 * to it at the end of the function. *)
559 let start_bb = insertion_block llbuilder in
560 let the_function = block_parent start_bb in
561
562 let then_bb = append_block context "then" the_function in
563
564 (* Emit 'then' value. *)
565 position_at_end then_bb llbuilder;
566 let _(* then_val *) = codegen_stmt llbuilder then_ in
567
568 (* Codegen of 'then' can change the current block, update then_bb for the
569 * phi. We create a new name because one is used for the phi node, and the
570 * other is used for the conditional branch. *)
571 let new_then_bb = insertion_block llbuilder in
572
573 (* Emit 'else' value. *)
574 let else_bb = append_block context "else" the_function in
575 position_at_end else_bb llbuilder;
576 let _ (* else_val *) = codegen_stmt llbuilder else_ in
577
578 (* Codegen of 'else' can change the current block, update else_bb for the
579 * phi. *)
580 let new_else_bb = insertion_block llbuilder in
581
582
583 let merge_bb = append_block context "ifcont" the_function in
584 position_at_end merge_bb llbuilder;
585 (* let then_bb_val = value_of_block new_then_bb in *)
586 let else_bb_val = value_of_block new_else_bb in
587 (* let incoming = [(then_bb_val, new_then_bb); (else_bb_val, new_else_bb)] in *)
588 (* let phi = build_phi incoming "iftmp" llbuilder in *)
589
590 (* Return to the start block to add the conditional branch. *)
591 position_at_end start_bb llbuilder;
592 ignore (build_cond_br cond_val then_bb else_bb llbuilder);
593
594 (* Set a unconditional branch at the end of the 'then' block and the
595 * 'else' block to the 'merge' block. *)
596 position_at_end new_then_bb llbuilder; ignore (build_br merge_bb llbuilder);
597 position_at_end new_else_bb llbuilder; ignore (build_br merge_bb llbuilder);
598
599 (* Finally, set the builder to the end of the merge block. *)
600 position_at_end merge_bb llbuilder;
601
602 else_bb_val (* phi *)
```

```
603
604 and codegen_for init_ cond_ inc_ body_ llbuilder =
605 let old_val = !is_loop in
606 is_loop := true;
607
608 let the_function = block_parent (insertion_block llbuilder) in
609
610 (* Emit the start code first, without 'variable' in scope. *)
611 let _ = codegen_sexpr llbuilder init_ in
612
613 (* Make the new basic block for the loop header, inserting after current
614 * block. *)
615 let loop_bb = append_block context "loop" the_function in
616 (* Insert maintenance block *)
617 let inc_bb = append_block context "inc" the_function in
618 (* Insert condition block *)
619 let cond_bb = append_block context "cond" the_function in
620 (* Create the "after loop" block and insert it. *)
621 let after_bb = append_block context "afterloop" the_function in
622
623 let _ = if not old_val then
624 cont_block := inc_bb;
625 br_block := after_bb;
626 in
627
628 (* Insert an explicit fall through from the current block to the
629 * loop_bb. *)
630 ignore (build_br cond_bb llbuilder);
631
632 (* Start insertion in loop_bb. *)
633 position_at_end loop_bb llbuilder;
634
635 (* Emit the body of the loop.  This, like any other expr, can change the
636 * current BB.  Note that we ignore the value computed by the body, but
637 * don't allow an error *)
638 ignore (codegen_stmt llbuilder body_);
639
640 let bb = insertion_block llbuilder in
641 move_block_after bb inc_bb;
642 move_block_after inc_bb cond_bb;
643 move_block_after cond_bb after_bb;
644 ignore(build_br inc_bb llbuilder);
645
646 (* Start insertion in loop_bb. *)
647 position_at_end inc_bb llbuilder;
648 (* Emit the step value. *)
649 let _ = codegen_sexpr llbuilder inc_ in
650 ignore(build_br cond_bb llbuilder);
651
```

```
652 position_at_end cond_bb llbuilder;
653
654 let cond_val = codegen_sexpr llbuilder cond_ in
655 ignore (build_cond_br cond_val loop_bb after_bb llbuilder);
656
657 (* Any new code will be inserted in after_bb. *)
658 position_at_end after_bb llbuilder;
659
660 is_loop := old_val;
661
662 (* for expr always returns 0.0. *)
663 const_null f_t
664
665 and codegen_while cond_ body_ llbuilder =
666 let null_sexpr = SInt_Lit(0) in
667 codegen_for null_sexpr cond_ null_sexpr body_ llbuilder
668
669 and codegen_alloca datatype var_name expr llbuilder =
670 let t = match datatype with
671 Datatype(Objecttype(name)) -> find_struct name
672 |           _ -> get_type datatype
673 in
674 let alloca = build_alloca t var_name llbuilder in
675 Hashtbl.add named_values var_name alloca;
676 let lhs = SId(var_name, datatype) in
677 match expr with
678 SNoexpr -> alloca
679 |           _ -> codegen_assign lhs expr datatype llbuilder
680
681 and codegen_ret d expr llbuilder =
682 match expr with
683 SId(name, d) ->
684 (match d with
685 | Datatype(Objecttype(_)) -> build_ret (codegen_id false false name d llbuilder)
    ↪  llbuilder
686 | _ -> build_ret (codegen_id true true name d llbuilder) llbuilder)
687 | SObjAccess(e1, e2, d) -> build_ret (codegen_obj_access true e1 e2 d llbuilder)
    ↪  llbuilder
688 | SNoexpr -> build_ret_void llbuilder
689 | _ -> build_ret (codegen_sexpr llbuilder expr) llbuilder
690
691 and codegen_break llbuilder =
692 let block = fun () -> !br_block in
693 build_br (block ()) llbuilder
694
695 and codegen_continue llbuilder =
696 let block = fun () -> !cont_block in
697 build_br (block ()) llbuilder
698
```

```
699 and codegen_stmt llbuilder = function
700 SBlock sl                              -> List.hd(List.map (codegen_stmt llbuilder) sl)
701 |   SExpr(e, d)                        -> codegen_sexpr llbuilder e
702 |   SReturn(e, d)                          -> codegen_ret d e llbuilder
703 |   SIf (e, s1, s2)                     -> codegen_if_stmt e s1 s2 llbuilder
704 |   SFor (e1, e2, e3, s)                -> codegen_for e1 e2 e3 s llbuilder
705 |   SWhile (e, s)                          -> codegen_while e s llbuilder
706 |   SBreak                                 -> codegen_break llbuilder
707 |   SContinue                              -> codegen_continue llbuilder
708 |   SLocal(d, s, e)                        -> codegen_alloca d s e llbuilder
709
710 let codegen_funcstub sfdecl =
711 let fname = (Utils.string_of_fname sfdecl.sfname) in
712 let is_var_arg = ref false in
713 let params = List.rev (List.fold_left (fun l -> (function Formal(t, _) -> get_type t :: l
    ↪  | _ -> is_var_arg := true; l )) [] sfdecl.sformals) in
714 let fty = if !is_var_arg
715 then var_arg_function_type (get_type sfdecl.sreturnType) (Array.of_list params)
716 else function_type (get_type sfdecl.sreturnType) (Array.of_list params)
717 in
718 define_function fname fty the_module
719
720 let init_params f formals =
721 let formals = Array.of_list (formals) in
722 Array.iteri (fun i a ->
723 let n = formals.(i) in
724 let n = Utils.string_of_formal_name n in
725 set_value_name n a;
726 Hashtbl.add named_params n a;
727 ) (params f)
728
729 let codegen_func sfdecl =
730 Hashtbl.clear named_values;
731 Hashtbl.clear named_params;
732 let fname = (Utils.string_of_fname sfdecl.sfname) in
733 let f = func_lookup fname in
734 let llbuilder = builder_at_end context (entry_block f) in
735 let _ = init_params f sfdecl.sformals in
736 let _ = if sfdecl.overrides then
737 let this_param = Hashtbl.find named_params "this" in
738 let source = Datatype(Objecttype(sfdecl.source)) in
739 let casted_param = build_pointercast this_param (get_type source) "casted" llbuilder in
740 Hashtbl.replace named_params "this" casted_param;
741 in
742 let _ = codegen_stmt llbuilder (SBlock (sfdecl.sbody)) in
743 if sfdecl.sreturnType = Datatype(Void_t)
744 then ignore(build_ret_void llbuilder);
745 ()
746
```

```
747 let codegen_vtbl scdecls =
748 let rt = pointer_type i64_t in
749 let void_pt = pointer_type i64_t in
750 let void_ppt = pointer_type void_pt in
751
752 let f = func_lookup "lookup" in
753 let llbuilder = builder_at_end context (entry_block f) in
754
755 let len = List.length scdecls in
756 let total_len = ref 0 in
757 let scdecl_llvm_arr = build_array_alloca void_ppt (const_int i32_t len) "tmp" llbuilder
    ↪  in
758
759 let handle_scdecl scdecl =
760 let index = Hashtbl.find Analyzer.struct_indexes scdecl.scname in
761 let len = List.length scdecl.sfuncs in
762 let sfdecl_llvm_arr = build_array_alloca void_pt (const_int i32_t len) "tmp" llbuilder in
763
764 let handle_fdecl i sfdecl =
765 let fptr = func_lookup (Utils.string_of_fname sfdecl.sfname) in
766 let fptr = build_pointercast fptr void_pt "tmp" llbuilder in
767
768 let ep = build_gep sfdecl_llvm_arr [| (const_int i32_t i) |] "tmp" llbuilder in
769 ignore(build_store fptr ep llbuilder);
770 in
771 List.iteri handle_fdecl scdecl.sfuncs;
772 total_len := !total_len + len;
773
774 let ep = build_gep scdecl_llvm_arr [| (const_int i32_t index) |] "tmp" llbuilder in
775 ignore(build_store sfdecl_llvm_arr ep llbuilder);
776 in
777 List.iter handle_scdecl scdecls;
778
779 let c_index = param f 0 in
780 let f_index = param f 1 in
781 set_value_name "c_index" c_index;
782 set_value_name "f_index" f_index;
783
784 if !total_len == 0 then
785 build_ret (const_null rt) llbuilder
786 else
787 let vtbl = build_gep scdecl_llvm_arr [| c_index |] "tmp" llbuilder in
788 let vtbl = build_load vtbl "tmp" llbuilder in
789 let fptr = build_gep vtbl [| f_index |] "tmp" llbuilder in
790 let fptr = build_load fptr "tmp" llbuilder in
791
792 build_ret fptr llbuilder
793
794 let codegen_library_functions () =
```

```
795 (* C Std lib functions *)
796 let printf_ty = var_arg_function_type i32_t [| pointer_type i8_t |] in
797 let _ = declare_function "printf" printf_ty the_module in
798 let malloc_ty = function_type (str_t) [| i32_t |] in
799 let _ = declare_function "malloc" malloc_ty the_module in
800 let open_ty = function_type i32_t [| (pointer_type i8_t); i32_t |] in
801 let _ = declare_function "open" open_ty the_module in
802 let close_ty = function_type i32_t [| i32_t |] in
803 let _ = declare_function "close" close_ty the_module in
804 let read_ty = function_type i32_t [| i32_t; pointer_type i8_t; i32_t |] in
805 let _ = declare_function "read" read_ty the_module in
806 let write_ty = function_type i32_t [| i32_t; pointer_type i8_t; i32_t |] in
807 let _ = declare_function "write" write_ty the_module in
808 let lseek_ty = function_type i32_t [| i32_t; i32_t; i32_t |] in
809 let _ = declare_function "lseek" lseek_ty the_module in
810 let exit_ty = function_type void_t [| i32_t |] in
811 let _ = declare_function "exit" exit_ty the_module in
812 let realloc_ty = function_type str_t [| str_t; i32_t |] in
813 let _ = declare_function "realloc" realloc_ty the_module in
814 let getchar_ty = function_type (i32_t) [| |] in
815 let _ = declare_function "getchar" getchar_ty the_module in
816
817 (* Dice defined functions *)
818 let fty = function_type (pointer_type i64_t) [| i32_t; i32_t |] in
819 let _ = define_function "lookup" fty the_module in
820 let rec_init_ty = function_type void_t [| (pointer_type i64_t); i32_t; (pointer_type
    ↪  i32_t); (pointer_type i32_t); (pointer_type i32_t); i32_t; i32_t |] in
821 let _ = declare_function "rec_init" rec_init_ty the_module in
822 let init_arr_ty = function_type (pointer_type i64_t) [| (pointer_type i32_t); i32_t |] in
823 let _ = declare_function "init_arr" init_arr_ty the_module in
824 let input_ty = function_type str_t [||] in
825 let _ = declare_function "input" input_ty the_module in
826 ()
827
828 let codegen_struct_stub s =
829 let struct_t = named_struct_type context s.scname in
830 Hashtbl.add struct_types s.scname struct_t
831
832 let codegen_struct s =
833 let struct_t = Hashtbl.find struct_types s.scname in
834 let type_list = List.map (function Field(_, d, _) -> get_type d) s.sfields in
835 let name_list = List.map (function Field(_, _, s) -> s) s.sfields in
836
837 (* Add key field to all structs *)
838 let type_list = i32_t :: type_list in
839 let name_list = ".key" :: name_list in
840
841 let type_array = (Array.of_list type_list) in
842 List.iteri (fun i f ->
```

```
843 let n = s.scname ^ "." ^ f in
844 Hashtbl.add struct_field_indexes n i;
845 ) name_list;
846 struct_set_body struct_t type_array true
847
848 let init_args argv args argc llbuilder =
849 let new_block label =
850 let f = block_parent (insertion_block llbuilder) in
851 append_block (global_context ()) label f
852 in
853 let bbcurr = insertion_block llbuilder in
854 let bbcond = new_block "args.cond" in
855 let bbbody = new_block "args.init" in
856 let bbdone = new_block "args.done" in
857 ignore (build_br bbcond llbuilder);
858 position_at_end bbcond llbuilder;
859
860 (* Counter into the length of the array *)
861 let counter = build_phi [const_int i32_t 0, bbcurr] "counter" llbuilder in
862 add_incoming ((build_add counter (const_int i32_t 1) "tmp" llbuilder), bbbody) counter;
863 let cmp = build_icmp Icmp.Slt counter argc "tmp" llbuilder in
864 ignore (build_cond_br cmp bbbody bbdone llbuilder);
865 position_at_end bbbody llbuilder;
866
867 (* Assign array position to init_val *)
868 let arr_ptr = build_gep args [| counter |] "tmp" llbuilder in
869 let argv_val = build_gep argv [| counter |] "tmp" llbuilder in
870 let argv_val = build_load argv_val "tmp" llbuilder in
871 ignore (build_store argv_val arr_ptr llbuilder);
872 ignore (build_br bbcond llbuilder);
873 position_at_end bbdone llbuilder
874
875 let construct_args argc argv llbuilder =
876 let str_pt = pointer_type str_t in
877 let size_real = build_add argc (const_int i32_t 1) "arr_size" llbuilder in
878
879 let arr = build_array_malloc str_pt size_real "args" llbuilder in
880 let arr = build_pointercast arr str_pt "args" llbuilder in
881 let arr_len_ptr = build_pointercast arr (pointer_type i32_t) "argc_len" llbuilder in
882 let arr_1 = build_gep arr [| const_int i32_t 1 |] "arr_1" llbuilder in
883
884 (* Store length at this position *)
885 ignore(build_store argc arr_len_ptr llbuilder);
886 ignore(init_args argv arr_1 argc llbuilder);
887 arr
888
889 let codegen_main main =
890 Hashtbl.clear named_values;
891 Hashtbl.clear named_params;
```

```
892 let fty = function_type i32_t [| i32_t; pointer_type str_t |] in
893 let f = define_function "main" fty the_module in
894 let llbuilder = builder_at_end context (entry_block f) in
895
896 let argc = param f 0 in
897 let argv = param f 1 in
898 set_value_name "argc" argc;
899 set_value_name "argv" argv;
900 let args = construct_args argc argv llbuilder in
901 Hashtbl.add named_params "args" args;
902
903 let _ = codegen_stmt llbuilder (SBlock (main.sbody)) in
904 build_ret (const_int i32_t 0) llbuilder
905
906 let linker filename =
907 let llctx = Llvm.global_context () in
908 let llmem = Llvm.MemoryBuffer.of_file filename in
909 let llm = Llvm_bitreader.parse_bitcode llctx llmem in
910 ignore(Llvm_linker.link_modules the_module llm)
911
912 let codegen_sprogram sprogram =
913 let _ = codegen_library_functions () in
914 let _ = List.map (fun s -> codegen_struct_stub s) sprogram.classes in
915 let _ = List.map (fun s -> codegen_struct s) sprogram.classes in
916 let _ = List.map (fun f -> codegen_funcstub f) sprogram.functions in
917 let _ = List.map (fun f -> codegen_func f) sprogram.functions in
918 let _ = codegen_main sprogram.main in
919 let _ = codegen_vtbl sprogram.classes in
920 let _ = linker Conf.bindings_path in
921 the_module
922
923 (* Need to handle assignment of two different types *)
924 (* Need to handle private/public access *)
```

conf.ml

```
let bindings_path = "_includes/bindings.bc"
let stdlib_path = "_includes/stdlib.dice"
```

dice.ml

```
open Llvm
open Llvm_analysis
open Analyzer
open Utils
open Ast
open Yojson
open Exceptions
open Filepath

type action = Tokens | TokenEndl | PrettyPrint | Ast | Sast | Compile | CompileToFile |
↪  Help

let get_action = function
"-tendl"         -> TokenEndl
|         "-t"                 -> Tokens
|         "-p"                 -> PrettyPrint
|         "-ast"                 -> Ast
|         "-sast"         -> Sast
|         "-h"                 -> Help
|         "-c"                 -> Compile
|         "-f"                 -> CompileToFile
|          _ as s                -> raise (Exceptions.InvalidCompilerArgument s)

let check_single_argument = function
"-h"          -> Help, ""
|         "-tendl"
|         "-t"
|         "-p"
|         "-ast"
|         "-sast"
|         "-c"
|         "-f"         -> raise (Exceptions.NoFileArgument)
|          _ as s         -> CompileToFile, s

let dice_name filename =
let basename = Filename.basename filename in
let filename = Filename.chop_extension basename in
filename ^ ".ll"

let help_string = (
"Usage: dice [optional-option] <source file>\n" ^
"optional-option:\n" ^
"\t-h: Print help text\n" ^
"\t-tendl: Prints tokens with newlines intact\n" ^
"\t-t: Prints token stream\n" ^
"\t-p: Pretty prints Ast as a program\n" ^
"\t-ast: Prints abstract syntax tree as json\n" ^
```

```
47  "\t-sast: Prints semantically checked syntax tree as json\n" ^
48  "\t-c: Compiles source\n" ^
49  "\t-f: Compiles source to file (<filename>.<ext> -> <filename>.ll)\n" ^
50  "Option defaults to \"-f\"\n"
51  )
52
53  let _ =
54  ignore(Printexc.record_backtrace true);
55  try
56  let action, filename =
57  if Array.length Sys.argv = 1 then
58  Help, ""
59  else if Array.length Sys.argv = 2 then
60  check_single_argument (Sys.argv.(1))
61  else if Array.length Sys.argv = 3 then
62  get_action Sys.argv.(1), Sys.argv.(2)
63  else raise (Exceptions.InvalidNumberCompilerArguments (Array.length Sys.argv))
64  in
65  (* Added fun () -> <x> so that each is evaluated only when requested *)
66  let filename         = Filepath.realpath filename in
67  let file_in          = fun () -> open_in filename in
68  let lexbuf                 = fun () ->        Lexing.from_channel (file_in ()) in
69  let token_list         = fun () -> Processor.build_token_list (lexbuf ()) in
70  let program          = fun () -> Processor.parser filename (token_list ()) in
71  let sprogram         = fun () -> Analyzer.analyze filename (program ()) in
72  let llm                = fun () -> Codegen.codegen_sprogram (sprogram ()) in
73  (* let _ = Llvm_analysis.assert_valid_module llm in *)
74  match action with
75  Help                        -> print_string help_string
76  |         Tokens                           -> print_string (Utils.token_list_to_string
    ↪  (token_list ()))
77  |         TokenEndl                -> print_string (Utils.token_list_to_string_endl
    ↪  (token_list ()))
78  |         Ast                         -> print_string (pretty_to_string
    ↪  (Utils.print_tree (program ())))
79  |         Sast                          -> print_string (pretty_to_string
    ↪  (Utils.map_sprogram_to_json (sprogram ())))
80  |         PrettyPrint         -> print_string (Utils.string_of_program (program ()))
81  |         Compile                -> dump_module (llm ())
82  |         CompileToFile         -> print_module (dice_name filename) (llm ())
83  with
84  Exceptions.IllegalCharacter(filename, c, ln) ->
85  print_string
86  (
87  "In \"" ^ filename ^ "\", Illegal Character, '" ^
88  Char.escaped c ^ "', line " ^ string_of_int ln ^ "\n"
89  )
90  |         Exceptions.UnmatchedQuotation(ln)         -> print_endline("Unmatched
    ↪  Quotation, line " ^ string_of_int ln)
```

```
91  |         Exceptions.IllegalToken(tok)                -> print_endline("Illegal token "
    ↪  ^ tok)
92  |         Exceptions.MissingEOF                               -> print_endline("Missing
    ↪  EOF")
93  |         Parsing.Parse_error ->
94  print_string
95  (
96  "File \"" ^ !Processor.filename ^ "\", " ^
97  "line " ^ string_of_int !Processor.line_number ^ ", " ^
98  "character " ^ string_of_int !Processor.char_num ^ ", " ^
99  "Syntax Error, token " ^ Utils.string_of_token !Processor.last_token ^ "\n"
100 )
101
102 |          Exceptions.InvalidNumberCompilerArguments i -> print_endline ("Invalid
    ↪  argument passed " ^ (string_of_int i)); print_string help_string
103 |         Exceptions.InvalidCompilerArgument s                -> print_endline ("Invalid
    ↪  argument passed " ^ s); print_string help_string
104 |         Exceptions.NoFileArgument                                    ->
    ↪  print_string ("Must include file argument\n" ^ help_string)
105
106 |         Exceptions.IncorrectNumberOfArgumentsException                      ->
    ↪  print_endline("Incorrect number of arguments passed to function")
107 |         Exceptions.ConstructorNotFound(cname)
    ↪                                       -> print_endline("Constructor" ^ cname ^ "
    ↪  not found")
108 |         Exceptions.DuplicateClassName(cname)                                  ->
    ↪  print_endline("Class " ^ cname ^ " not found")
109 |         Exceptions.DuplicateField
    ↪                                                            ->
    ↪  print_endline("Duplicate field defined")
110 |         Exceptions.DuplicateFunction(fname)                                   ->
    ↪  print_endline("Duplicate function defined " ^ fname)
111 |         Exceptions.DuplicateConstructor
    ↪                                          -> print_endline("Duplicate
    ↪  constructor found")
112 |         Exceptions.DuplicateLocal(lname)
    ↪                                          -> print_endline("Duplicate local
    ↪  variable defined " ^ lname)
113 |         Exceptions.UndefinedClass(cname)
    ↪                                          -> print_endline("Undefined class " ^
    ↪  cname)
114 |         Exceptions.UnknownIdentifier(id)
    ↪                                          -> print_endline("Unkown identifier "
    ↪  ^ id)
115 |         Exceptions.InvalidBinopExpression(binop)                              ->
    ↪  print_endline("Invalid binary expression " ^ binop)
116 |         Exceptions.InvalidIfStatementType
    ↪                                          -> print_endline("Invalid type passed
    ↪  to if statement, must be bool")
```

```
117 |       Exceptions.InvalidForStatementType
    ↪                                          -> print_endline("Invalid type passed
    ↪ to for loop, must be bool")
118 |       Exceptions.ReturnTypeMismatch(t1, t2)                                    ->
    ↪ print_endline("Incorrect return type " ^ t1 ^ " expected " ^ t2)
119 |       Exceptions.MainNotDefined
    ↪                                                          ->
    ↪ print_endline("Main not found in program")
120 |
    ↪         Exceptions.MultipleMainsDefined                                        ->
    ↪ print_endline("Multiple mains defined, can only define 1")
121 |       Exceptions.InvalidWhileStatementType                                    ->
    ↪ print_endline("Invalid type passed to while loop, must be bool")
122 |       Exceptions.LocalAssignTypeMismatch(t1, t2)                              ->
    ↪ print_endline("Invalid assignment of " ^ t1 ^ " to " ^ t2)
123 |       Exceptions.InvalidUnaryOperation
    ↪                                          -> print_endline("Invalid unary
    ↪ operator")
124 |       Exceptions.AssignmentTypeMismatch(t1, t2)                               ->
    ↪ print_endline("Invalid assignment of " ^ t1 ^ " to " ^ t2)
125 |       Exceptions.FunctionNotFound(fname, scope)                               ->
    ↪ print_endline("function " ^ fname ^ " not found in scope " ^ scope)
126 |       Exceptions.UndefinedID(id)
    ↪                                                          ->
    ↪ print_endline("Undefined id " ^ id)
127 |       Exceptions.InvalidAccessLHS(t)
    ↪                                                  -> print_endline("Invalid LHS
    ↪ expression of dot operator with " ^ t)
128 |       Exceptions.LHSofRootAccessMustBeIDorFunc(lhs)                           ->
    ↪ print_endline("Dot operator expects ID, not " ^ lhs)
129 |       Exceptions.ObjAccessMustHaveObjectType(t)                               ->
    ↪ print_endline("Can only dereference objects, not " ^ t)
130 |       Exceptions.UnknownIdentifierForClass(c, id)                             ->
    ↪ print_endline("Unknown id " ^ id ^ " for class " ^ c)
131 |       Exceptions.CannotUseReservedFuncName(f)                                 ->
    ↪ print_endline("Cannot use name " ^ f ^ " because it is reserved")
132 |       Exceptions.InvalidArrayPrimitiveConsecutiveTypes(t1,t2)        ->
    ↪ print_endline("Array primitive types must be equal, not " ^ t1 ^ " " ^ t2)
133 |       Exceptions.InvalidArrayPrimitiveType(t)                                 ->
    ↪ print_endline("Array primitive type invalid, " ^ t)
134 |       Exceptions.MustPassIntegerTypeToArrayCreate                             ->
    ↪ print_endline("Only integer types can be passed to an array initializer")
135 |       Exceptions.ArrayInitTypeInvalid(t)
    ↪                                          -> print_endline("Only integer types
    ↪ can be passed to an array initializer, not " ^ t)
136 |       Exceptions.MustPassIntegerTypeToArrayAccess                             ->
    ↪ print_endline("Only integer types can be passed to an array access")
137 |       Exceptions.ArrayAccessInvalidParamLength(o,a)                           ->
    ↪ print_endline("Only arrays can have access to length, not " ^ o ^ " " ^ a)
```

```
138 |       Exceptions.ArrayAccessExpressionNotArray(a)                          ->
    ↪ print_endline("This expression is not an array " ^ a)
139 |       Exceptions.CanOnlyAccessLengthOfArray
    ↪                                     -> print_endline("Can only access the length
    ↪ of an array")
140 |       Exceptions.CanOnlyDeleteObjectsOrArrays                              ->
    ↪ print_endline("Can only delete objects or arrays")
141 |       Exceptions.CannotAccessLengthOfCharArray                              ->
    ↪ print_endline("Cannot access the length of a char array")
142 |       Exceptions.AllNonVoidFunctionsMustEndWithReturn(f)                    ->
    ↪ print_endline("Non-void function " ^ f ^ " does not end in return")
143 |       Exceptions.CyclicalDependencyBetween(c1, c2)                          ->
    ↪ print_endline("Class " ^ c1 ^ " and " ^ c2 ^ " have a cylical dependence")
144 |       Exceptions.CannotAccessPrivateFieldInNonProperScope(f, cp, cc) ->
    ↪ print_endline("Cannot access private field " ^ f ^ " in scope " ^ cp ^ " from object
    ↪ " ^ cc)
145 |       Exceptions.CannotCallBreakOutsideOfLoop                              ->
    ↪ print_endline("Cannot call break outside of loop")
146 |       Exceptions.CannotCallContinueOutsideOfLoop                           ->
    ↪ print_endline("Cannot call continue outside of loop")
147 |       Exceptions.CannotAccessPrivateFunctionInNonProperScope(f, cp, cc) ->
    ↪ print_endline("Cannot access private function " ^ f ^ " in scope " ^ cp ^ " from
    ↪ object " ^ cc)
148 |       Exceptions.CannotPassNonInheritedClassesInPlaceOfOthers(c1, c2)      ->
    ↪ print_endline("Cannot pass non-inherited classe" ^ c1 ^ " to parameter " ^ c2)
149 |       Exceptions.IncorrectTypePassedToFunction(id, t)
    ↪                                     -> print_endline("Canot pass type " ^ t ^ "
    ↪ to " ^ id)
150 |       Exceptions.IncorrectNumberOfArguments(f, a1, a2) -> print_endline("Cannot pass
    ↪ " ^ string_of_int a1 ^ " args when expecting " ^ string_of_int a2 ^ " in " ^f)
151 |       Exceptions.ClassIsNotExtendedBy(c1, c2)                              ->
    ↪ print_endline("Class " ^ c1 ^ " not extended by " ^ c2)
152 |
153 |       Exceptions.InvalidTypePassedToPrintf                                  ->
    ↪ print_endline("Invalid type passed to print")
154 |       Exceptions.InvalidBinaryOperator                                      ->
    ↪ print_endline("Invalid binary operator")
155 |       Exceptions.UnknownVariable(id)
    ↪                                         -> print_endline("Unknown variable "
    ↪ ^ id)
156 |       Exceptions.AssignLHSMustBeAssignable                                  ->
    ↪ print_endline("Assignment lhs must be assignable")
157 |       Exceptions.CannotCastTypeException(t1, t2)                            ->
    ↪ print_endline("Cannot cast " ^ t1 ^ " to " ^ t2)
158 |       Exceptions.InvalidBinopEvaluationType                                 ->
    ↪ print_endline("Invalid binary expression evaluation type")
159 |       Exceptions.FloatOpNotSupported
    ↪                                         -> print_endline("Float operation not
    ↪ supported")
```

```
160 |         Exceptions.IntOpNotSupported                                    ->
    ↪ print_endline("Integer operation not supported")
161 |         Exceptions.LLVMFunctionNotFound(f)                          ->
    ↪ print_endline("LLVM function " ^ f ^ " not found")
162 |         Exceptions.InvalidStructType(t)                               ->
    ↪ print_endline("Invalid structure type " ^ t)
163 |         Exceptions.UnableToCallFunctionWithoutParent(f)         ->
    ↪ print_endline("Unable to call function " ^ f ^ " without parent")
164 |         Exceptions.CannotAssignParam(p)                               ->
    ↪ print_endline("Cannot assign to param " ^ p)
165 |         Exceptions.InvalidUnopEvaluationType                        ->
    ↪ print_endline("Invalid unary expression evaluation type")
166 |         Exceptions.UnopNotSupported                                      ->
    ↪ print_endline("Unary operator not supported")
167 |         Exceptions.ArrayLargerThan1Unsupported                       ->
    ↪ print_endline("Array dimensions greater than 1 not supported")
168 |         Exceptions.CanOnlyCompareObjectsWithNull(e1, e2)         -> print_endline("Can
    ↪ only compare objects with null " ^ e1 ^ " " ^ e2)
169 |         Exceptions.ObjOpNotSupported(op)                             ->
    ↪ print_endline("Object operator not supported " ^ op)
170 |         Exceptions.CanOnlyCompareArraysWithNull(e1, e2)         -> print_endline("Can
    ↪ only compare arrays with null " ^ e1 ^ " " ^ e2)
```

exceptions.ml

```
(* Dice Exceptions *)
exception InvalidNumberCompilerArguments of int
exception InvalidCompilerArgument of string
exception NoFileArgument

(* Processor Exceptions *)
exception MissingEOF

(* Scanner Exceptions *)
exception IllegalCharacter of string * char * int
exception UnmatchedQuotation of int
exception IllegalToken of string

(* Analyzer Exceptions *)
exception IncorrectNumberOfArgumentsException
exception ConstructorNotFound of string
exception DuplicateClassName of string
exception DuplicateField
exception DuplicateFunction of string
exception DuplicateConstructor
exception DuplicateLocal of string
exception UndefinedClass of string
exception UnknownIdentifier of string
exception InvalidBinopExpression of string
exception InvalidIfStatementType
exception InvalidForStatementType
exception ReturnTypeMismatch of string * string
exception MainNotDefined
exception MultipleMainsDefined
exception InvalidWhileStatementType
exception LocalAssignTypeMismatch of string * string
exception InvalidUnaryOperation
exception AssignmentTypeMismatch of string * string
exception FunctionNotFound of string * string
exception UndefinedID of string
exception InvalidAccessLHS of string
exception LHSofRootAccessMustBeIDorFunc of string
exception ObjAccessMustHaveObjectType of string
exception UnknownIdentifierForClass of string * string
exception CannotUseReservedFuncName of string
exception InvalidArrayPrimitiveConsecutiveTypes of string * string
exception InvalidArrayPrimitiveType of string
exception MustPassIntegerTypeToArrayCreate
exception ArrayInitTypeInvalid of string
exception MustPassIntegerTypeToArrayAccess
exception ArrayAccessInvalidParamLength of string * string
exception ArrayAccessExpressionNotArray of string
```

```
48 exception CanOnlyAccessLengthOfArray
49 exception CanOnlyDeleteObjectsOrArrays
50 exception CannotAccessLengthOfCharArray
51 exception AllNonVoidFunctionsMustEndWithReturn of string
52 exception CyclicalDependencyBetween of string * string
53 exception CannotAccessPrivateFieldInNonProperScope of string * string * string
54 exception CannotCallBreakOutsideOfLoop
55 exception CannotCallContinueOutsideOfLoop
56 exception CannotAccessPrivateFunctionInNonProperScope of string * string * string
57 exception CannotPassNonInheritedClassesInPlaceOfOthers of string * string
58 exception IncorrectTypePassedToFunction of string * string
59 exception IncorrectNumberOfArguments of string * int * int
60 exception ClassIsNotExtendedBy of string * string
61
62 (* Codegen Exceptions *)
63 exception InvalidTypePassedToPrintf
64 exception InvalidBinaryOperator
65 exception UnknownVariable of string
66 exception AssignLHSMustBeAssignable
67 exception CannotCastTypeException of string * string
68 exception InvalidBinopEvaluationType
69 exception FloatOpNotSupported
70 exception IntOpNotSupported
71 exception LLVMFunctionNotFound of string
72 exception InvalidStructType of string
73 exception UnableToCallFunctionWithoutParent of string
74 exception CannotAssignParam of string
75 exception InvalidUnopEvaluationType
76 exception UnopNotSupported
77 exception ArrayLargerThan1Unsupported
78 exception CanOnlyCompareObjectsWithNull of string * string
79 exception ObjOpNotSupported of string
80 exception CanOnlyCompareArraysWithNull of string * string
```

filepath.ml

```
open Filename
open Unix

exception Safe_exception of (string * string list ref)

let raise_safe fmt =
let do_raise msg = raise @@ Safe_exception (msg, ref []) in
Printf.ksprintf do_raise fmt

let reraise_with_context ex fmt =
let do_raise context =
let () = match ex with
| Safe_exception (_, old_contexts) -> old_contexts := context :: !old_contexts
| _ -> Printf.eprintf "warning: Attempt to add note '%s' to non-Safe_exception!" context
in
raise ex
in Printf.ksprintf do_raise fmt

module StringMap = struct
include Map.Make(String)
let find_nf = find
let find_safe key map = try find key map with Not_found -> raise_safe "BUG: Key '%s' not
  ↪  found in StringMap!" key
let find key map = try Some (find key map) with Not_found -> None
let map_bindings fn map = fold (fun key value acc -> fn key value :: acc) map []
end

type path_component =
| Filename of string  (* foo/ *)
| ParentDir           (* ../ *)
| CurrentDir          (* ./ *)
| EmptyComponent      (* / *)

type filepath = string


let on_windows = Filename.dir_sep <> "/"

let path_is_absolute path = not (Filename.is_relative path)

let string_tail s i =
let len = String.length s in
if i > len then failwith ("String '" ^ s ^ "' too short to split at " ^ (string_of_int
  ↪  i))
else String.sub s i (len - i)

let split_path_str path =
```

```
46  let l = String.length path in
47  let is_sep c = (c = '/' || (on_windows && c = '\\')) in
48
49  (* Skip any leading slashes and return the rest *)
50  let rec find_rest i =
51  if i < l then (
52  if is_sep path.[i] then find_rest (i + 1)
53  else string_tail path i
54  ) else (
55  ""
56  ) in
57
58  let rec find_slash i =
59  if i < l then (
60  if is_sep path.[i] then (String.sub path 0 i, find_rest (i + 1))
61  else find_slash (i + 1)
62  ) else (
63  (path, "")
64  )
65  in
66  find_slash 0
67
68  let split_first path =
69  if path = "" then
70  (CurrentDir, "")
71  else (
72  let (first, rest) = split_path_str path in
73  let parsed =
74  if first = Filename.parent_dir_name then ParentDir
75  else if first = Filename.current_dir_name then CurrentDir
76  else if first = "" then EmptyComponent
77  else Filename first in
78  (parsed, rest)
79  )
80
81  let normpath path : filepath =
82  let rec explode path =
83  match split_first path with
84  | CurrentDir, "" -> []
85  | CurrentDir, rest -> explode rest
86  | first, "" -> [first]
87  | first, rest -> first :: explode rest in
88
89  let rec remove_parents = function
90  | checked, [] -> checked
91  | (Filename _name :: checked), (ParentDir :: rest) -> remove_parents (checked, rest)
92  | checked, (first :: rest) -> remove_parents ((first :: checked), rest) in
93
94  let to_string = function
```

```
95  | Filename name -> name
96  | ParentDir -> Filename.parent_dir_name
97  | EmptyComponent -> ""
98  | CurrentDir -> assert false in
99  String.concat Filename.dir_sep @@ List.rev_map to_string @@ remove_parents ([], explode
    ↪  path)
100
101
102 let abspath path =
103 let (+/) = Filename.concat in
104 normpath (
105 if path_is_absolute path then path
106 else (Sys.getcwd ()) +/ path
107 )
108
109 let realpath path =
110 let (+/) = Filename.concat in   (* Faster version, since we know the path is relative *)
111
112 (* Based on Python's version *)
113 let rec join_realpath path rest seen =
114 (* Printf.printf "join_realpath <%s> + <%s>\n" path rest; *)
115 (* [path] is already a realpath (no symlinks). [rest] is the bit to join to it. *)
116 match split_first rest with
117 | Filename name, rest -> (
118 (* path + name/rest *)
119 let newpath = path +/ name in
120 let link = try Some (Unix.readlink newpath) with Unix.Unix_error _ -> None in
121 match link with
122 | Some target ->
123 (* path + symlink/rest *)
124 begin match StringMap.find newpath seen with
125 | Some (Some cached_path) -> join_realpath cached_path rest seen
126 | Some None -> (normpath (newpath +/ rest), false)    (* Loop; give up *)
127 | None ->
128 (* path + symlink/rest -> realpath(path + target) + rest *)
129 match join_realpath path target (StringMap.add newpath None seen) with
130 | path, false ->
131 (normpath (path +/ rest), false)   (* Loop; give up *)
132 | path, true -> join_realpath path rest (StringMap.add newpath (Some path) seen)
133 end
134 | None ->
135 (* path + name/rest -> path/name + rest (name is not a symlink) *)
136 join_realpath newpath rest seen
137 )
138 | CurrentDir, "" ->
139 (path, true)
140 | CurrentDir, rest ->
141 (* path + ./rest *)
142 join_realpath path rest seen
```

```
143 | ParentDir, rest ->
144 (* path + ../rest *)
145 if String.length path > 0 then (
146 let name = Filename.basename path in
147 let path = Filename.dirname path in
148 if name = Filename.parent_dir_name then
149 join_realpath (path +/ name +/ name) rest seen    (* path/.. +  ../rest -> path/../.. +
    ↪  rest *)
150 else
151 join_realpath path rest seen                      (* path/name + ../rest -> path + rest
    ↪  *)
152 ) else (
153 join_realpath Filename.parent_dir_name rest seen    (* "" + ../rest -> .. + rest *)
154 )
155 | EmptyComponent, rest ->
156 (* [rest] is absolute; discard [path] and start again *)
157 join_realpath Filename.dir_sep rest seen
158 in
159
160 try
161 if on_windows then
162 abspath path
163 else (
164 fst @@ join_realpath (Sys.getcwd ()) path StringMap.empty
165 )
166 with Safe_exception _ as ex -> reraise_with_context ex "... in realpath(%s)" path
```

parser.mly

```
%{  open Ast  %}

%token CLASS EXTENDS CONSTRUCTOR INCLUDE DOT THIS PRIVATE PUBLIC
%token INT FLOAT BOOL CHAR VOID NULL TRUE FALSE
%token SEMI LPAREN RPAREN LBRACE RBRACE LBRACKET RBRACKET COMMA
%token AND NOT OR PLUS MINUS TIMES DIVIDE ASSIGN MODULO
%token EQ NEQ LT LEQ GT GEQ BAR
%token RETURN IF ELSE FOR WHILE BREAK CONTINUE NEW DELETE
%token <int> INT_LITERAL
%token <float> FLOAT_LITERAL
%token <string> STRING_LITERAL
%token <string> ID
%token <char> CHAR_LITERAL
%token EOF

%nonassoc NOELSE
%nonassoc ELSE
%right ASSIGN
%left AND OR
%left EQ NEQ
%left LT GT LEQ GEQ
%left PLUS MINUS
%left TIMES DIVIDE MODULO
%right NOT
%right DELETE
%right RBRACKET
%left LBRACKET
%right DOT

%start program
%type <Ast.program> program

%%

program:
includes cdecls EOF { Program($1, $2) }

/******************
INCLUDE
******************/

includes:
/* nothing */ { [] }
|         include_list  { List.rev $1 }

include_list:
include_decl              { [$1] }
```

```
48  |         include_list include_decl { $2::$1 }
49
50  include_decl:
51  INCLUDE LPAREN STRING_LITERAL RPAREN SEMI { Include($3) }
52
53
54  /******************
55  CLASSES
56  ******************/
57  cdecls:
58  cdecl_list    { List.rev $1 }
59
60  cdecl_list:
61  cdecl             { [$1] }
62  | cdecl_list cdecl  { $2::$1 }
63
64  cdecl:
65  CLASS ID LBRACE cbody RBRACE { {
66                  cname = $2;
67                  extends = NoParent;
68                  cbody = $4
69          } }
70  |         CLASS ID EXTENDS ID LBRACE cbody RBRACE { {
71                  cname = $2;
72                  extends = Parent($4);
73                  cbody = $6
74  } }
75
76  cbody:
77  /* nothing */ { {
78                  fields = [];
79                  constructors = [];
80                  methods = [];
81  } }
82  |         cbody field { {
83                  fields = $2 :: $1.fields;
84                  constructors = $1.constructors;
85                  methods = $1.methods;
86  } }
87  |         cbody constructor { {
88                  fields = $1.fields;
89                  constructors = $2 :: $1.constructors;
90                  methods = $1.methods;
91  } }
92  |         cbody fdecl { {
93          fields = $1.fields;
94          constructors = $1.constructors;
95          methods = $2 :: $1.methods;
96  } }
```

```
97
98
99  /******************
100 CONSTRUCTORS
101 ******************/
102
103 constructor:
104 CONSTRUCTOR LPAREN formals_opt RPAREN LBRACE stmt_list RBRACE {
105         {
106                 scope = Public;
107                 fname = Constructor;
108                 returnType = Datatype(ConstructorType);
109                 formals = $3;
110                 body = List.rev $6;
111                 overrides = false;
112                 root_cname = None;
113         }
114 }
115
116 /******************
117 FIELDS
118 ******************/
119
120 scope:
121 PRIVATE { Private }
122 |         PUBLIC  { Public }
123
124 /* public UserObj name; */
125 field:
126 scope datatype ID SEMI { Field($1, $2, $3) }
127
128 /******************
129 METHODS
130 ******************/
131
132 fname:
133 ID { $1 }
134
135 fdecl:
136 scope datatype fname LPAREN formals_opt RPAREN LBRACE stmt_list RBRACE
137 {
138         {
139                 scope = $1;
140                 fname = FName($3);
141                 returnType = $2;
142                 formals = $5;
143                 body = List.rev $8;
144                 overrides = false;
145                 root_cname = None;
```

```
146         }
147 }
148
149 /******************
150 FORMALS/PARAMETERS & VARIABLES & ACTUALS
151 ******************/
152
153 formals_opt:
154 /* nothing */ { [] }
155 |         formal_list   { List.rev $1 }
156
157 formal_list:
158 formal                   { [$1] }
159 |         formal_list COMMA formal { $3 :: $1 }
160
161 formal:
162 datatype ID { Formal($1, $2) }
163
164 actuals_opt:
165 /* nothing */ { [] }
166 |         actuals_list  { List.rev $1 }
167
168 actuals_list:
169 expr                    { [$1] }
170 |         actuals_list COMMA expr { $3 :: $1 }
171
172
173 /***************
174 DATATYPES
175 ***************/
176 primitive:
177 INT                 { Int_t }
178 |         FLOAT               { Float_t }
179 |         CHAR               { Char_t }
180 |         BOOL                { Bool_t }
181 |         VOID            { Void_t }
182
183 name:
184 CLASS ID { Objecttype($2) }
185
186 type_tag:
187 primitive { $1 }
188 |        name          { $1 }
189
190 array_type:
191 type_tag LBRACKET brackets RBRACKET { Arraytype($1, $3) }
192
193 datatype:
194 type_tag   { Datatype($1) }
```

```
195 |         array_type { $1 }
196
197 brackets:
198 /* nothing */                              { 1 }
199 |         brackets RBRACKET LBRACKET { $1 + 1 }
200
201 /******************
202 EXPRESSIONS
203 ******************/
204
205 stmt_list:
206 /* nothing */  { [] }
207 | stmt_list stmt { $2 :: $1 }
208
209 stmt:
210 expr SEMI { Expr($1) }
211 |         RETURN expr SEMI { Return($2) }
212 |        RETURN SEMI                { Return(Noexpr) }
213 |         LBRACE stmt_list RBRACE { Block(List.rev $2) }
214 |         IF LPAREN expr RPAREN stmt %prec NOELSE { If($3, $5, Block([Expr(Noexpr)])) }
215 |         IF LPAREN expr RPAREN stmt ELSE stmt    { If($3, $5, $7) }
216 |         FOR LPAREN expr_opt SEMI expr_opt SEMI expr_opt RPAREN stmt
217 { For($3, $5, $7, $9) }
218 |         WHILE LPAREN expr RPAREN stmt         { While($3, $5) }
219 |        BREAK SEMI                                              { Break }
220 |        CONTINUE SEMI                                        { Continue }
221 |   datatype ID SEMI                                  { Local($1, $2, Noexpr) }
222 |         datatype ID ASSIGN expr SEMI         { Local($1, $2, $4) }
223
224 expr_opt:
225 /* nothing */ { Noexpr }
226 |         expr          { $1 }
227
228 expr:
229 literals                                                    { $1 }
230 |         expr PLUS   expr                                    { Binop($1, Add,   $3)
    ↪ }
231 |         expr MINUS  expr                                    { Binop($1, Sub,   $3)
    ↪ }
232 |         expr TIMES  expr                                    { Binop($1, Mult,  $3)
    ↪ }
233 |         expr DIVIDE expr                                    { Binop($1, Div,   $3)
    ↪ }
234 |         expr EQ     expr                                    { Binop($1, Equal, $3)
    ↪ }
235 |         expr NEQ    expr                                    { Binop($1, Neq,   $3)
    ↪ }
236 |         expr LT     expr                                    { Binop($1, Less,  $3)
    ↪ }
```

```
237 |         expr LEQ    expr                                    { Binop($1, Leq,   $3)
    ↪ }
238 |         expr GT     expr                                    { Binop($1, Greater,
    ↪ $3) }
239 |         expr GEQ    expr                                    { Binop($1, Geq,   $3)
    ↪ }
240 |         expr AND    expr                                    { Binop($1, And,   $3)
    ↪ }
241 |         expr MODULO expr                                    { Binop($1, Mod,
    ↪ $3)}
242 |         NOT  expr                                                { Unop (Not,
    ↪ $2) }
243 |         expr OR     expr                                    { Binop($1, Or,    $3)
    ↪ }
244 |         expr DOT    expr                                    { ObjAccess($1, $3) }
245 |         expr ASSIGN expr                                    { Assign($1, $3) }
246 |         DELETE expr                                            { Delete($2) }
247 |   MINUS expr                                                    { Unop (Sub, $2) }
248 |         ID LPAREN actuals_opt RPAREN                 { Call($1, $3) }
249 |         NEW ID LPAREN actuals_opt RPAREN         { ObjectCreate($2, $4) }
250 |        NEW type_tag bracket_args RBRACKET         { ArrayCreate(Datatype($2), List.rev
    ↪ $3) }
251 |         expr bracket_args RBRACKET                          { ArrayAccess($1, List.rev
    ↪ $2) }
252 |         LPAREN expr RPAREN                                     { $2 }
253
254 bracket_args:
255 LBRACKET expr                                                 { [$2] }
256 |         bracket_args RBRACKET LBRACKET expr { $4 :: $1 }
257
258 literals:
259 INT_LITERAL                      { Int_Lit($1) }
260 | FLOAT_LITERAL                    { Float_Lit($1) }
261 | TRUE                                        { Boolean_Lit(true) }
262 | FALSE                                        { Boolean_Lit(false) }
263 | STRING_LITERAL                  { String_Lit($1) }
264 | CHAR_LITERAL                       { Char_Lit($1) }
265 | THIS                                         { This }
266 | ID                                        { Id($1) }
267 | NULL                                  { Null }
268 | BAR array_prim BAR         { ArrayPrimitive($2) }
269
270 /* ARRAY LITERALS */
271
272 array_prim:
273 expr                                    { [$1] }
274 |        array_prim COMMA expr         { $3 :: $1 }
```

processor.ml

```
open Parser

type token_attr = {
        lineno: int;
        cnum: int;
}

let line_number = ref 1
let last_token = ref EOF
let char_num = ref 1
let filename = ref ""

let build_token_list lexbuf =
Scanner.filename := !filename;
let rec helper prev_cnum prev_lineno lexbuf token_list =
let token = Scanner.token lexbuf in
let lineno = !Scanner.lineno in
let cnum = (Lexing.lexeme_start_p lexbuf).Lexing.pos_cnum in
let prev_cnum = if lineno > prev_lineno then cnum else prev_cnum in
let cnum = cnum - prev_cnum in
match token with
EOF as eof -> (eof, { lineno = lineno; cnum = cnum } )::token_list
|   t          -> (t, { lineno = lineno; cnum = cnum } )::(helper prev_cnum lineno lexbuf
↪  token_list)
in helper 0 0 lexbuf []

let parser filen token_list =
let token_list = ref(token_list) in
let tokenizer _ =
match !token_list with
| (head, curr) :: tail ->
filename := filen;
line_number := curr.lineno;
char_num    := curr.cnum;
last_token := head;
token_list := tail;
head
| [] -> raise (Exceptions.MissingEOF)
in
let program = Parser.program tokenizer (Lexing.from_string "") in
program
```

sast.ml

```
open Ast

type sexpr =
SInt_Lit of int
|          SBoolean_Lit of bool
|          SFloat_Lit of float
|          SString_Lit of string
|          SChar_Lit of char
|          SId of string * datatype
|          SBinop of sexpr * op * sexpr * datatype
|          SAssign of sexpr * sexpr * datatype
|          SNoexpr
|          SArrayCreate of datatype * sexpr list * datatype
|          SArrayAccess of sexpr * sexpr list * datatype
|          SObjAccess of sexpr * sexpr * datatype
|          SCall of string * sexpr list * datatype * int
|    SObjectCreate of string * sexpr list * datatype
|          SArrayPrimitive of sexpr list * datatype
|           SUnop of op * sexpr * datatype
|          SNull
|          SDelete of sexpr

type sstmt =
SBlock of sstmt list
|          SExpr of sexpr * datatype
|          SReturn of sexpr  * datatype
|          SIf of sexpr * sstmt * sstmt
|          SFor of sexpr * sexpr * sexpr * sstmt
|          SWhile of sexpr * sstmt
|           SBreak
|    SContinue
|    SLocal of datatype * string * sexpr

type func_type = User | Reserved

type sfunc_decl = {
        sfname : fname;
        sreturnType : datatype;
        sformals : formal list;
        sbody : sstmt list;
        func_type : func_type;
        source : string;
        overrides : bool;
}

type sclass_decl = {
        scname : string;
```

```
48          sfields : field list;
49          sfuncs: sfunc_decl list;
50  }
51
52  (* Class Declarations | All method declarations | Main entry method *)
53  type sprogram =  {
54          classes : sclass_decl list;
55          functions : sfunc_decl list;
56          main : sfunc_decl;
57          reserved : sfunc_decl list;
58  }
```

scanner.mll

```
{
        open Parser
        let lineno = ref 1
        let depth = ref 0
        let filename = ref ""

        let unescape s =
        Scanf.sscanf ("\"" ^ s ^ "\"") "%S%!" (fun x -> x)
}

let alpha = ['a'-'z' 'A'-'Z']
let escape = '\\' ['\\' ''' '"' 'n' 'r' 't']
let escape_char = ''' (escape) '''
let ascii = ([' '-'!' '#'-'[' ']'-'~'])
let digit = ['0'-'9']
let id = alpha (alpha | digit | '_')*
let string = '"' ( (ascii | escape)* as s) '"'
let char = ''' ( ascii | digit ) '''
let float = (digit+) ['.'] digit+
let int = digit+
let whitespace = [' ' '\t' '\r']
let return = '\n'

rule token = parse
whitespace { token lexbuf }
| return          { incr lineno; token lexbuf}
| "(*"       { incr depth; comment lexbuf }

| '('       { LPAREN }
| ')'       { RPAREN }
| '{'       { LBRACE }
        | '}'       { RBRACE }
| ';'       { SEMI }
| ','       { COMMA }

(* Operators *)
| '+'       { PLUS }
| '-'       { MINUS }
| '*'       { TIMES }
| '/'       { DIVIDE }
| '%'       { MODULO }
| '='       { ASSIGN }
| "=="      { EQ }
| "!="      { NEQ }
| '<'       { LT }
| "<="      { LEQ }
| ">"       { GT }
```

```
48  | ">="      { GEQ }
49  | "and"     { AND }
50  | "or"      { OR }
51  | "not"     { NOT }
52  | '.'       { DOT }
53  | '['       { LBRACKET }
54  | ']'       { RBRACKET }
55  | '|'           { BAR }
56
57  (* Branch Control *)
58  | "if"      { IF }
59  | "else"    { ELSE }
60  | "for"     { FOR }
61  | "while"   { WHILE }
62  | "return"  { RETURN }
63
64  (* Data Types *)
65  | "int"     { INT }
66  | "float"   { FLOAT }
67  | "bool"    { BOOL }
68  | "char"    { CHAR }
69  | "void"    { VOID }
70  | "null"    { NULL }
71  | "true"    { TRUE }
72  | "false"   { FALSE }
73
74  (* Classes *)
75  | "class"       { CLASS }
76  | "constructor" { CONSTRUCTOR }
77  | "public"      { PUBLIC }
78  | "private"     { PRIVATE }
79  | "extends"     { EXTENDS }
80  | "include"     { INCLUDE }
81  | "this"        { THIS }
82  | "break"                { BREAK }
83  | "continue"        { CONTINUE }
84  | "new"                 { NEW }
85  | "delete"               { DELETE }
86
87  | int as lxm                  { INT_LITERAL(int_of_string lxm) }
88  | float as lxm                { FLOAT_LITERAL(float_of_string lxm) }
89  | char as lxm                 { CHAR_LITERAL( String.get lxm 1 ) }
90  | escape_char as lxm{ CHAR_LITERAL( String.get (unescape lxm) 1) }
91  | string                      { STRING_LITERAL(unescape s) }
92  | id as lxm                   { ID(lxm) }
93  | eof                         { EOF }
94
95  | '"'                        { raise (Exceptions.UnmatchedQuotation(!lineno)) }
96  | _ as illegal  { raise (Exceptions.IllegalCharacter(!filename, illegal, !lineno)) }
```

```
97
98  and comment = parse
99  return          { incr lineno; comment lexbuf }
100 |         "*)"          { decr depth; if !depth > 0 then comment lexbuf else token lexbuf }
101 |    "(*"          { incr depth; comment lexbuf }
102 |         _               { comment lexbuf }
```

stdlibe.dice

```
class Integer {

        private int my_int;

        constructor(int input) {
                this.my_int = input;
        }

        public int num() {
                return this.my_int;
        }


        public char toChar(int digit) {

                if (digit == 0) {
                        return '0';
                } else if (digit == 1) {
                        return '1';
                } else if (digit == 2) {
                        return '2';
                } else if (digit == 3) {
                return '3';
                } else if (digit == 4) {
                return '4';
                } else if (digit == 5) {
                return '5';
                } else if (digit == 6) {
                return '6';
                } else if (digit == 7) {
                return '7';
                } else if (digit == 8) {
                return '8';
                } else if (digit == 9) {
                return '9';
                }

        return 'z';
}




public class String toString() {

        (* integer cannot be greater than 10 digits in 32 bit *)
```

```
48          int temp = this.my_int;
49          int i = 0;
50          char[] str = new char[9];
51
52          int digit = temp % 10;
53          str[i] = this.toChar(digit);
54          i = i + 1;
55          temp = temp / 10;
56          while (temp > 0) {
57
58                  digit = temp % 10;
59                  str[i] = this.toChar(digit);
60                  temp = temp / 10;
61                  i = i + 1;
62          }
63
64          str[i] = 0;
65          class String newString = new String(str);
66          class String a = newString.reverse();
67          return newString.reverse();
68  }
69  }
70
71
72
73  class String {
74
75          private char[] my_string;
76          private int length;
77
78          constructor(char[] input) {
79
80                  this.my_string = this.copy_internal(input);
81
82                  this.length = this.length();
83          }
84
85          (* PRIVATE CLASSES ----------------------------------------   *)
86
87          private int length_internal(char[] input) {
88                  int length = 0;
89
90                  while(input[length] != 0) {
91                          length = length + 1;
92                  }
93
94                  return length;
95          }
96
```

```
97          private char[] copy_internal(char[] input) {
98
99                  char[] newString = new char[this.length_internal(input) + 1];
100
101                 int i = 0;
102                 for (; input[i] != 0; i = i + 1) {
103                         newString[i] = input[i];
104                 }
105
106                 newString[i] = 0;
107                 return newString;
108         }
109
110         (* PUBLIC CLASSES ----------------------------------------   *)
111
112         public char[] string() {
113                 return this.my_string;
114         }
115
116         public char getChar(int index) {
117
118                 return this.my_string[index];
119         }
120
121         public int length() {
122
123                 int length = 0;
124
125                 while(this.my_string[length] != 0){
126                         length = length + 1;
127                 }
128
129                 return length;
130         }
131
132         public int toInteger() {
133
134                 char[] temp = this.string();
135                 int ndigit = 0;
136                 int i;
137                 int j;
138                 for (i = 0; i < this.length; i = i + 1) {
139
140                         int exp = 1;
141                         int xdigit = this.toDigit(temp[i]);
142                         for (j = 0; j < (this.length-i-1); j = j + 1) {
143                                 exp = exp * 10;
144                         }
145                         xdigit = xdigit * exp;
```

```
146                         ndigit = ndigit + xdigit;
147                 }
148
149                 return ndigit;
150         }
151
152         public int toDigit(char digit) {
153
154                 if (digit == '0') {
155                         return 0;
156                 } else if (digit == '1') {
157                 return 1;
158         } else if (digit == '2') {
159         return 2;
160 } else if (digit == '3') {
161 return 3;
162 } else if (digit == '4') {
163 return 4;
164 } else if (digit == '5') {
165 return 5;
166 } else if (digit == '6') {
167 return 6;
168 } else if (digit == '7') {
169 return 7;
170 } else if (digit == '8') {
171 return 8;
172 } else if (digit == '9') {
173 return 9;
174 }
175
176 return -1;
177 }
178
179
180 public class String copy(class String input) {
181
182         char[] newArray = this.copy_internal(input.string());
183         class String newString = new String(newArray);
184         return newString;
185 }
186
187 public int indexOf(char x) {
188
189         int i = 0;
190         for (; this.getChar(i) != x and this.getChar(i) != 0; i = i + 1) {
191         }
192
193         (* If the char was not found, return -1 *)
194         if (i == this.length()) {
```

```
195                 return -1;
196         }
197
198         return i;
199 }
200
201 public class String reverse() {
202
203         class String newString;
204
205         char[] temp = new char[this.length + 1];
206         int i = this.length;
207         for (; i > 0; i = i - 1) {
208
209                 temp[this.length - i] = this.getChar(i-1);
210         }
211         temp[this.length] = 0;
212         newString = new String(temp);
213         return newString;
214 }
215
216 public class String concat(class String temp) {
217
218         char[] temparray = new char[this.length() + temp.length() + 1];
219
220         (* Copy over the current string into a new char array *)
221         int i = 0;
222         for (; this.getChar(i) != 0; i = i + 1) {
223                 temparray[i] = this.getChar(i);
224         }
225
226         (* Append the new string *)
227         int j = 0;
228         for (; temp.getChar(j) != 0; j = j + 1) {
229                 temparray[i+j] = temp.getChar(j);
230         }
231
232         temparray[this.length() + temp.length()] = 0;
233         class String newString = new String(temparray);
234         return newString;
235 }
236
237 public bool compare(class String check) {
238
239         if (check.length != this.length) {
240                 return false;
241         }
242
243         int i = 0;
```

```
244
245         for (; i < check.length(); i = i + 1) {
246
247                 if (check.getChar(i) != this.getChar(i)) {
248                         return false;
249                 }
250         }
251
252         return true;
253 }
254
255 public bool contains(class String check) {
256
257
258         if (this.length < check.length) {
259                 return false;
260         } else if (this.compare(check)) {
261         return true;
262 } else {
263
264 int diff = this.length - check.length + 1;
265 int i;
266 int j;
267 for ( i = 0; i < diff; i = i + 1)
268
269 for ( j = 0; j < check.length; j = j + 1) {
270
271         if (this.getChar(i+j) != check.getChar(j)) {
272                 break;
273         }
274
275         if (j == check.length - 1) {
276                 return true;
277         }
278 }
279 }
280 return false;
281 }
282
283 public void free() {
284
285         delete(this.my_string);
286 }
287
288 }
289
290
291
292 class File {
```

```
293
294         private class String filePath;
295         private bool isWriteEnabled;
296         private int fd;
297
298         constructor(char[] path, bool isWriteEnabled) {
299
300                 this.filePath = new String(path);
301                 this.isWriteEnabled = isWriteEnabled;
302                 class String a = this.filePath;
303                 this.fd = this.openfile(a, this.isWriteEnabled);
304                 if (this.fd < 0) {
305                         print("open failed");
306                         exit(1);
307                 }
308         }
309
310         (* PRIVATE CLASSES ----------------------------------------   *)
311
312         private int openfile(class String path, bool isWriteEnabled) {
313
314                 if (isWriteEnabled) {
315                         (* 2 is the value for O_RDWR *)
316                         return open(path.string(), 2);
317                 }
318
319                 (* 0 is the value for O_RDONLY *)
320                 return open(path.string(), 0);
321         }
322
323         (* PUBLIC CLASSES -----------------------------------------   *)
324
325         public void closefile() {
326
327                 if (close(this.fd) < 0) {
328                         print("close failed");
329                 }
330         }
331
332         public char[] readfile(int bytes) {
333
334                 char[] buf = new char[bytes];
335
336                 int ret = read(this.fd, buf, bytes);
337
338                 if (ret < 0) {
339                         print("read failed");
340                 }
341
```

```
342                 return buf;
343         }
344
345         public int writefile(char[] buf, int offset) {
346
347                 class String temp = new String(buf);
348                 int err;
349                 (* seek to desired offset from beginning of file *)
350                 if (offset > 0) {
351                         err = lseek(this.fd, offset, 0);
352                 } else if (offset == -1) {
353                 err = lseek(this.fd, 0, 0);
354         } else {
355         (* Seek to the end of the file by default *)
356         err = lseek(this.fd, 0, 2);
357 }
358
359 if (err < 0) {
360         print("seek failed");
361 }
362
363 err = write(this.fd, temp.string(), temp.length());
364 if (err < 0) {
365         print("write failed");
366 }
367 return err;
368 }
369
370 }
```

utils.ml

```
(* Pretty Printer *)
open Ast
open Sast
open Parser
open Processor
open Yojson

let save file string =
let channel = open_out file in
output_string channel string;
close_out channel

let replace input output =
Str.global_replace (Str.regexp_string input) output

(* Print data types *)

let string_of_scope = function
Public          -> "public"
|          Private -> "private"

let string_of_primitive = function
Int_t                                                -> "int"
|          Float_t                                       -> "float"
|          Void_t                                              -> "void"
|          Bool_t                                               -> "bool"
|          Char_t                                               -> "char"
|          Objecttype(s)                                  -> "class " ^ s
|          ConstructorType                                  -> "constructor"
|           Null_t                                                -> "null"

let string_of_object = function
Datatype(Objecttype(s))         -> s
|          _ -> ""

let rec print_brackets = function
1 -> "[]"
|          a -> "[]" ^ print_brackets (a - 1)

let string_of_datatype = function
Arraytype(p, i)         -> (string_of_primitive p) ^ (print_brackets i)
|          Datatype(p)                -> (string_of_primitive p)
|           Any                         -> "Any"

(* Print expressions *)

let string_of_op = function
```

```
48  Add                         -> "+"
49  |         Sub                          -> "-"
50  |         Mult                -> "*"
51  |         Div                          -> "/"
52  |         Equal                -> "=="
53  |         Neq                          -> "!="
54  |         Less                -> "<"
55  |         Leq                          -> "<="
56  |         Greater               -> ">"
57  |         Geq                          -> ">="
58  |         And                          -> "and"
59  |         Not                          -> "not"
60  |         Or                          -> "or"
61  |         Mod                 -> "%"
62
63  let rec string_of_bracket_expr = function
64  []                                  -> ""
65  |         head :: tail         -> "[" ^ (string_of_expr head) ^ "]" ^
    ↪  (string_of_bracket_expr tail)
66  and string_of_array_primitive = function
67  []                                  -> ""
68  |   [last]                          -> (string_of_expr last)
69  |         head :: tail         -> (string_of_expr head) ^ ", " ^
    ↪  (string_of_array_primitive tail)
70  and string_of_expr = function
71  Int_Lit(i)                                -> string_of_int i
72  |         Boolean_Lit(b)                     -> if b then "true" else "false"
73  |         Float_Lit(f)                      -> string_of_float f
74  |         String_Lit(s)                      -> "\"" ^ (String.escaped s) ^ "\""
75  |         Char_Lit(c)                            -> Char.escaped c
76  |         This                                    -> "this"
77  |         Id(s)                                    -> s
78  |         Binop(e1, o, e2)               -> (string_of_expr e1) ^ " " ^ (string_of_op o)
    ↪  ^ " " ^ (string_of_expr e2)
79  |         Assign(e1, e2)                      -> (string_of_expr e1) ^ " = " ^
    ↪  (string_of_expr e2)
80  |         Noexpr                                    -> ""
81  |         ObjAccess(e1, e2)               -> (string_of_expr e1) ^ "." ^ (string_of_expr
    ↪  e2)
82  |         Call(f, el)                              -> f ^ "(" ^ String.concat ", "
    ↪  (List.map string_of_expr el) ^ ")"
83  |         ArrayPrimitive(el)               -> "|" ^ (string_of_array_primitive el) ^ "|"
84  |           Unop(op, e)                              -> (string_of_op op) ^ "(" ^
    ↪  string_of_expr e ^ ")"
85  |         Null                                    -> "null"
86  |   ArrayCreate(d, el)          -> "new " ^ string_of_datatype d ^ string_of_bracket_expr
    ↪  el
87  |   ArrayAccess(e, el)          -> (string_of_expr e) ^ (string_of_bracket_expr el)
```

```
88  |   ObjectCreate(s, el)         -> "new " ^ s ^ "(" ^ String.concat ", " (List.map
    ↪  string_of_expr el) ^ ")"
89  |         Delete(e)                                -> "delete (" ^ (string_of_expr e) ^
    ↪  ")"
90  ;;
91
92  let rec string_of_bracket_sexpr = function
93  []                                -> ""
94  |         head :: tail         -> "[" ^ (string_of_sexpr head) ^ "]" ^
    ↪  (string_of_bracket_sexpr tail)
95  and string_of_sarray_primitive = function
96  []                                -> ""
97  |   [last]                       -> (string_of_sexpr last)
98  |         head :: tail         -> (string_of_sexpr head) ^ ", " ^
    ↪  (string_of_sarray_primitive tail)
99  and string_of_sexpr = function
100 SInt_Lit(i)                                      -> string_of_int i
101 |         SBoolean_Lit(b)                            -> if b then "true" else "false"
102 |         SFloat_Lit(f)                             -> string_of_float f
103 |         SString_Lit(s)                             -> "\"" ^ (String.escaped s) ^
    ↪  "\""
104 |         SChar_Lit(c)                              -> Char.escaped c
105 |         SId(s, _)                                    -> s
106 |         SBinop(e1, o, e2, _)                -> (string_of_sexpr e1) ^ " " ^
    ↪  (string_of_op o) ^ " " ^ (string_of_sexpr e2)
107 |         SAssign(e1, e2, _)                        -> (string_of_sexpr e1) ^ " = " ^
    ↪  (string_of_sexpr e2)
108 |         SNoexpr                                                -> ""
109 |         SObjAccess(e1, e2, _)               -> (string_of_sexpr e1) ^ "." ^
    ↪  (string_of_sexpr e2)
110 |         SCall(f, el, _, _)                        -> f ^ "(" ^ String.concat ", "
    ↪  (List.map string_of_sexpr el) ^ ")"
111 |         SArrayPrimitive(el, _)                -> "|" ^ (string_of_sarray_primitive el) ^
    ↪  "|"
112 |          SUnop(op, e, _)                                -> (string_of_op op) ^ "(" ^
    ↪  string_of_sexpr e ^ ")"
113 |         SNull                                                -> "null"
114 |   SArrayCreate(d, el, _)          -> "new " ^ string_of_datatype d ^
    ↪  string_of_bracket_sexpr el
115 |   SArrayAccess(e, el, _)          -> (string_of_sexpr e) ^ (string_of_bracket_sexpr el)
116 |   SObjectCreate(s, el, _)         -> "new " ^ s ^ "(" ^ String.concat ", " (List.map
    ↪  string_of_sexpr el) ^ ")"
117 |         SDelete(e)                                        -> "delete (" ^
    ↪  (string_of_sexpr e) ^ ")"
118 ;;
119
120 let string_of_local_expr = function
121 Noexpr -> ""
122 |          e            -> " = " ^ string_of_expr e
```

```
123
124 (* Print statements *)
125
126 let rec string_of_stmt indent =
127 let indent_string = String.make indent '\t' in
128 let get_stmt_string = function
129
130 Block(stmts)                          ->
131 indent_string ^ "{\n" ^
132         String.concat "" (List.map (string_of_stmt (indent+1)) stmts) ^
133         indent_string ^ "}\n"
134
135 |         Expr(expr)                                  ->
136 indent_string ^ string_of_expr expr ^ ";\n";
137
138 |         Return(expr)                            ->
139 indent_string ^ "return " ^ string_of_expr expr ^ ";\n";
140
141 |         If(e, s, Block([Expr(Noexpr)]))          ->
142 indent_string ^ "if (" ^ string_of_expr e ^ ")\n" ^
143 (string_of_stmt (indent+1) s)
144
145 |         If(e, s1, s2)                            ->
146 indent_string ^ "if (" ^ string_of_expr e ^ ")\n" ^
147 string_of_stmt (indent+1) s1 ^
148 indent_string ^ "else\n" ^
149 string_of_stmt (indent+1) s2
150
151 |         For(e1, e2, e3, s)                   ->
152 indent_string ^ "for (" ^ string_of_expr e1  ^ " ; " ^ string_of_expr e2 ^ " ; " ^
    ↪  string_of_expr e3  ^ ")\n" ^
153 string_of_stmt (indent) s
154
155 |         While(e, s)                           ->
156 indent_string ^ "while (" ^ string_of_expr e ^ ")\n" ^
157 string_of_stmt (indent) s
158
159 |          Break                                        -> indent_string ^ "break;\n"
160 |          Continue                                  -> indent_string ^ "continue;\n"
161 |   Local(d, s, e)                           -> indent_string ^ string_of_datatype d ^ " "
    ↪  ^ s ^ string_of_local_expr e ^ ";\n"
162 in get_stmt_string
163
164 let string_of_local_sexpr = function
165 SNoexpr         -> ""
166 |          e                              -> " = " ^ string_of_sexpr e
167
168 let rec string_of_sstmt indent =
169 let indent_string = String.make indent '\t' in
```

```
170 let get_stmt_string = function
171
172 SBlock(stmts)                          ->
173 indent_string ^ "{\n" ^
174         String.concat "" (List.map (string_of_sstmt (indent+1)) stmts) ^
175         indent_string ^ "}\n"
176
177 |         SExpr(expr, _)                                  ->
178 indent_string ^ string_of_sexpr expr ^ ";\n";
179
180 |         SReturn(expr, _)                           ->
181 indent_string ^ "return " ^ string_of_sexpr expr ^ ";\n";
182
183 |         SIf(e, s, SBlock([SExpr(SNoexpr, _)]))         ->
184 indent_string ^ "if (" ^ string_of_sexpr e ^ ")\n" ^
185 (string_of_sstmt (indent+1) s)
186
187 |         SIf(e, s1, s2)                            ->
188 indent_string ^ "if (" ^ string_of_sexpr e ^ ")\n" ^
189 string_of_sstmt (indent+1) s1 ^
190 indent_string ^ "else\n" ^
191 string_of_sstmt (indent+1) s2
192
193 |         SFor(e1, e2, e3, s)                  ->
194 indent_string ^ "for (" ^ string_of_sexpr e1  ^ " ; " ^ string_of_sexpr e2 ^ " ; " ^
    ↪  string_of_sexpr e3  ^ ")\n" ^
195 string_of_sstmt (indent) s
196
197 |         SWhile(e, s)                           ->
198 indent_string ^ "while (" ^ string_of_sexpr e ^ ")\n" ^
199 string_of_sstmt (indent) s
200
201 |          SBreak                                        -> indent_string ^ "break;\n"
202 |          SContinue                                  -> indent_string ^ "continue;\n"
203 |   SLocal(d, s, e)                           -> indent_string ^ string_of_datatype d ^ " "
    ↪  ^ s ^ string_of_local_sexpr e ^ ";\n"
204 in get_stmt_string
205
206 (* Print Function *)
207
208 let string_of_fname = function
209 Constructor -> "constructor"
210 |        FName(s)        -> s
211
212 let string_of_formal = function
213 Formal(d, s) -> (string_of_datatype d) ^ " " ^ s
214 |          _                          -> ""
215
216 let string_of_formal_name = function
```

```
217 Formal(_, s) -> s
218 |          _ -> ""
219
220 let string_of_func_decl fdecl =
221 "" ^ (string_of_scope fdecl.scope) ^ " " ^ (string_of_datatype fdecl.returnType) ^ " " ^
    ↪  (string_of_fname fdecl.fname) ^ " " ^
222 (* Formals *)
223 "(" ^ String.concat "," (List.map string_of_formal fdecl.formals) ^ ") {\n" ^
224         (* body *)
225         String.concat "" (List.map (string_of_stmt 2) fdecl.body) ^
226         "\t}\n\n"
227
228 (* Class Printing *)
229
230 let string_of_extends = function
231 NoParent        -> ""
232 |          Parent(s)        -> "extends " ^ s ^ " "
233 let string_of_field = function
234 Field(s, d, id) -> (string_of_scope s) ^ " " ^ (string_of_datatype d) ^ " " ^ id ^ ";\n"
235
236 let string_of_cbody cbody =
237 String.concat "" (List.map (fun s -> "\t" ^ s) (List.map string_of_field cbody.fields)) ^
238 String.concat "" (List.map (fun s -> "\t" ^ s) (List.map string_of_func_decl
    ↪  cbody.constructors)) ^
239 String.concat "" (List.map (fun s -> "\t" ^ s) (List.map string_of_func_decl
    ↪  cbody.methods))
240
241 let string_of_class_decl cdecl =
242 "class " ^ cdecl.cname ^ " " ^ (string_of_extends cdecl.extends) ^ "{\n" ^
243         (string_of_cbody cdecl.cbody) ^
244         "}\n"
245
246 (* Include Printing *)
247
248 let rec string_of_include = function
249 Include(s) -> "include(" ^ s ^ ");\n"
250
251 (* Print whole program *)
252
253 let string_of_program = function
254 Program(includes, cdecls) ->
255 String.concat "" (List.map string_of_include includes) ^ "\n" ^
256 String.concat "\n" (List.map string_of_class_decl cdecls)
257
258 (* Print AST tree representation *)
259
260 let includes_tree includes =
261 `List (List.map (function Include s -> `String s) includes)
262
```

```
263 let map_fields_to_json fields =
264 `List (List.map (function Field(scope, datatype, s) ->
265 `Assoc [
266 ("name", `String s);
267 ("scope", `String (string_of_scope scope));
268 ("datatype", `String (string_of_datatype datatype));
269 ]) fields)
270
271 let map_formals_to_json formals =
272 `List (List.map (function Formal(d, s) -> `Assoc [
273 ("name", `String s);
274 ("datatype", `String (string_of_datatype d));
275 ]
276 | Many d -> `Assoc [("Many", `String (string_of_datatype d));]
277 ) formals)
278
279 let rec map_expr_to_json = function
280 Int_Lit(i)                              -> `Assoc [("int_lit", `Int i)]
281 |       Boolean_Lit(b)                      -> `Assoc [("bool_lit", `Bool b)]
282 |       Float_Lit(f)                       -> `Assoc [("float_lit", `Float f)]
283 |       String_Lit(s)                      -> `Assoc [("string_lit", `String s)]
284 |       Char_Lit(c)                            -> `Assoc [("char_lit", `String
    ↪  (Char.escaped c))]
285 |       This                                    -> `String "this"
286 |       Id(s)                                    -> `Assoc [("id", `String s)]
287 |       Binop(e1, o, e2)                -> `Assoc [("binop", `Assoc [("lhs",
    ↪  map_expr_to_json e1); ("op", `String (string_of_op o)); ("rhs", map_expr_to_json
    ↪  e2)])]
288 |       Assign(e1, e2)                          -> `Assoc [("assign", `Assoc [("lhs",
    ↪  map_expr_to_json e1); ("op", `String "="); ("rhs", map_expr_to_json e2)])]
289 |       Noexpr                                      -> `String "noexpr"
290 |       ObjAccess(e1, e2)                -> `Assoc [("objaccess", `Assoc [("lhs",
    ↪  map_expr_to_json e1); ("op", `String "."); ("rhs", map_expr_to_json e2)])]
291 |       Call(f, el)                                -> `Assoc [("call", `Assoc ([("name",
    ↪  `String f); ("params", `List (List.map map_expr_to_json el)); ]) )]
292 |       ArrayPrimitive(el)                -> `Assoc [("arrayprimitive", `List(List.map
    ↪  map_expr_to_json el))]
293 |        Unop(op, e)                              -> `Assoc [("Unop", `Assoc [("op",
    ↪  `String (string_of_op op)); ("operand", map_expr_to_json e)])]
294 |       Null                                      -> `String "null"
295 |   ArrayCreate(d, el)          -> `Assoc [("arraycreate", `Assoc [("datatype", `String
    ↪  (string_of_datatype d)); ("args", `List (List.map map_expr_to_json el))])]
296 |   ArrayAccess(e, el)          -> `Assoc [("arrayaccess", `Assoc [("array",
    ↪  map_expr_to_json e); ("args", `List (List.map map_expr_to_json el))])]
297 |   ObjectCreate(s, el)         -> `Assoc [("objectcreate", `Assoc [("type", `String s);
    ↪  ("args", `List (List.map map_expr_to_json el))])]
298 |        Delete(e)                              -> `Assoc [("delete", `Assoc
    ↪  [("expr", map_expr_to_json e)])]
299
```

```
300 let rec map_stmt_to_json = function
301 Block(stmts)                          -> `Assoc [("block", `List (List.map
    ↪  (map_stmt_to_json) stmts))]
302 |         Expr(expr)                                -> `Assoc [("expr", map_expr_to_json
    ↪  expr)]
303 |         Return(expr)                          -> `Assoc [("return", map_expr_to_json
    ↪  expr)]
304 |         If(e, s1, s2)                          -> `Assoc [("if", `Assoc [("cond",
    ↪  map_expr_to_json e); ("ifbody", map_stmt_to_json s1)]); ("else", map_stmt_to_json
    ↪  s2)]
305 |         For(e1, e2, e3, s)                -> `Assoc [("for", `Assoc [("init",
    ↪  map_expr_to_json e1); ("cond", map_expr_to_json e2); ("inc", map_expr_to_json e3);
    ↪  ("body", map_stmt_to_json s)])]
306 |         While(e, s)                          -> `Assoc [("while", `Assoc [("cond",
    ↪  map_expr_to_json e); ("body", map_stmt_to_json s)])]
307 |          Break                                        -> `String "break"
308 |          Continue                                  -> `String "continue"
309 |   Local(d, s, e)                          -> `Assoc [("local", `Assoc [("datatype",
    ↪  `String (string_of_datatype d)); ("name", `String s); ("val", map_expr_to_json e)])]
310
311 let map_methods_to_json methods =
312 `List (List.map (fun (fdecl:Ast.func_decl) ->
313 `Assoc [
314 ("name", `String (string_of_fname fdecl.fname));
315 ("scope", `String (string_of_scope fdecl.scope));
316 ("returnType", `String (string_of_datatype fdecl.returnType));
317 ("formals", map_formals_to_json fdecl.formals);
318 ("body", `List (List.map (map_stmt_to_json) fdecl.body));
319 ]) methods)
320
321
322 let cdecls_tree cdecls =
323 let map_cdecl_to_json cdecl =
324 `Assoc [
325 ("cname", `String cdecl.cname);
326 ("extends", `String (string_of_extends cdecl.extends));
327 ("fields", map_fields_to_json cdecl.cbody.fields);
328 ("methods", map_methods_to_json cdecl.cbody.methods);
329 ("constructors", map_methods_to_json cdecl.cbody.constructors)
330 ]
331 in
332 `List (List.map (map_cdecl_to_json) cdecls)
333
334 let print_tree = function
335 Program(includes, cdecls) ->
336 `Assoc [("program",
337 `Assoc([
338 ("includes", includes_tree includes);
339 ("classes", cdecls_tree cdecls)
```

```
340 ])
341 )]
342
343 (* Print SAST tree representation *)
344
345 let rec map_sexpr_to_json =
346 let datatype d = [("datatype", `String (string_of_datatype d))] in
347 function
348 SInt_Lit(i)                  -> `Assoc [("int_lit", `Assoc ([("val", `Int i)] @ (datatype
    ↪  (Datatype(Int_t))))]
349 |   SBoolean_Lit(b)              -> `Assoc [("bool_lit", `Assoc ([("val", `Bool b)] @
    ↪  (datatype (Datatype(Bool_t))))]
350 |   SFloat_Lit(f)                -> `Assoc [("float_lit", `Assoc ([("val", `Float f)]  @
    ↪  (datatype (Datatype(Float_t))))]
351 |   SString_Lit(s)               -> `Assoc [("string_lit", `Assoc ([("val", `String s)] @
    ↪  (datatype (Arraytype(Char_t, 1))))]
352 |   SChar_Lit(c)                 -> `Assoc [("char_lit", `Assoc ([("val", `String
    ↪  (Char.escaped c))] @ (datatype (Datatype(Char_t))))]
353 |   SId(s, d)                 -> `Assoc [("id", `Assoc ([("name", `String s)] @ (datatype
    ↪  d)))]
354 |   SBinop(e1, o, e2, d)     -> `Assoc [("binop", `Assoc ([("lhs", map_sexpr_to_json e1);
    ↪  ("op", `String (string_of_op o)); ("rhs", map_sexpr_to_json e2)] @ (datatype d)))]
355 |   SAssign(e1, e2, d)       -> `Assoc [("assign", `Assoc ([("lhs", map_sexpr_to_json e1);
    ↪  ("op", `String "="); ("rhs", map_sexpr_to_json e2)] @ (datatype d)))]
356 |   SNoexpr                    -> `Assoc [("noexpr", `Assoc (datatype
    ↪  (Datatype(Void_t))))]
357 |   SArrayCreate(t, el, d)  -> `Assoc [("arraycreate", `Assoc ([("datatype", `String
    ↪  (string_of_datatype d)); ("args", `List (List.map map_sexpr_to_json el))] @ (datatype
    ↪  d)))]
358 |   SArrayAccess(e, el, d)  -> `Assoc [("arrayaccess", `Assoc ([("array",
    ↪  map_sexpr_to_json e); ("args", `List (List.map map_sexpr_to_json el))] @ (datatype
    ↪  d)))]
359 |   SObjAccess(e1, e2, d)   -> `Assoc [("objaccess", `Assoc ([("lhs", map_sexpr_to_json
    ↪  e1); ("op", `String "."); ("rhs", map_sexpr_to_json e2)] @ (datatype d)))]
360 |   SCall(fname, el, d, i)  -> `Assoc [("call", `Assoc ([("name", `String fname);
    ↪  ("params", `List (List.map map_sexpr_to_json el)); ("index", `Int i) ] @ (datatype
    ↪  d)) )]
361 |   SObjectCreate(s, el, d) -> `Assoc [("objectcreate", `Assoc ([("type", `String s);
    ↪  ("args", `List (List.map map_sexpr_to_json el))] @ (datatype d)))]
362 |   SArrayPrimitive(el, d)  -> `Assoc [("arrayprimitive", `Assoc ([("expressions",
    ↪  `List(List.map map_sexpr_to_json el))] @ (datatype d)))]
363 |   SUnop(op, e, d)         -> `Assoc [("Unop", `Assoc ([("op", `String (string_of_op
    ↪  op)); ("operand", map_sexpr_to_json e)] @ (datatype d)))]
364 |   SNull                      -> `Assoc [("null", `Assoc (datatype
    ↪  (Datatype(Void_t))))]
365 |        SDelete(e)                                  -> `Assoc [("delete", `Assoc
    ↪  ([("expr", map_sexpr_to_json e)] @ (datatype (Datatype(Void_t))))]
366
367 let rec map_sstmt_to_json =
```

```
368 let datatype d = [("datatype", `String (string_of_datatype d))] in
369 function
370 SBlock sl                              -> `Assoc [("sblock", `List (List.map
    ↪  (map_sstmt_to_json) sl))]
371 |   SExpr(e, d)                         -> `Assoc [("sexpr", `Assoc ([("expr",
    ↪  map_sexpr_to_json e)] @ (datatype d)))]
372 |   SReturn(e, d)                          -> `Assoc [("sreturn", `Assoc ([("return",
    ↪  map_sexpr_to_json e)] @ (datatype d)))]
373 |   SIf (e, s1, s2)                       -> `Assoc [("sif", `Assoc [("cond",
    ↪  map_sexpr_to_json e); ("ifbody", map_sstmt_to_json s1)]); ("selse", map_sstmt_to_json
    ↪  s2)]
374 |   SFor (e1, e2, e3, s)                 -> `Assoc [("sfor", `Assoc [("init",
    ↪  map_sexpr_to_json e1); ("cond", map_sexpr_to_json e2); ("inc", map_sexpr_to_json e3);
    ↪  ("body", map_sstmt_to_json s)])]
375 |   SWhile (e, s)                          -> `Assoc [("swhile", `Assoc [("cond",
    ↪  map_sexpr_to_json e); ("body", map_sstmt_to_json s)])]
376 |   SBreak                                 -> `String "sbreak"
377 |   SContinue                              -> `String "scontinue"
378 |   SLocal(d, s, e)                        -> `Assoc [("slocal", `Assoc [("datatype",
    ↪  `String (string_of_datatype d)); ("name", `String s); ("val", map_sexpr_to_json e)])]
379
380 let string_of_func_type = function
381 User -> "user" | Reserved -> "reserved"
382
383 let map_sfdecl_to_json sfdecl =
384 `Assoc[("sfdecl", `Assoc[
385 ("sfname", `String (string_of_fname sfdecl.sfname));
386 ("sreturnType", `String (string_of_datatype sfdecl.sreturnType));
387 ("sformals", map_formals_to_json sfdecl.sformals);
388 ("sbody", `List (List.map (map_sstmt_to_json) sfdecl.sbody));
389 ("func_type", `String(string_of_func_type sfdecl.func_type));
390 ])]
391
392 let map_sfdecls_to_json sfdecls =
393 `List(List.map map_sfdecl_to_json sfdecls)
394
395 let map_scdecls_to_json scdecls =
396 `List(List.map (fun scdecl ->
397 `Assoc [("scdecl",
398 `Assoc[
399 ("scname", `String scdecl.scname);
400 ("sfields", map_fields_to_json scdecl.sfields);
401 ("sfuncs", map_sfdecls_to_json scdecl.sfuncs);
402 ])
403 ])
404 scdecls)
405
406 let map_sprogram_to_json sprogram =
407 `Assoc [("sprogram", `Assoc [
```

```
("classes", map_scdecls_to_json sprogram.classes);
("functions", map_sfdecls_to_json sprogram.functions);
("main", map_sfdecl_to_json sprogram.main);
("reserved", map_sfdecls_to_json sprogram.reserved);
])]

(* Print tokens *)

let string_of_token = function
LPAREN                                  -> "LPAREN"
|          RPAREN                                -> "RPAREN"
|          LBRACE                                -> "LBRACE"
|          RBRACE                                -> "RBRACE"
|          SEMI                                -> "SEMI"
|          COMMA                                -> "COMMA"
|          PLUS                                -> "PLUS"
|          MINUS                                -> "MINUS"
|          TIMES                                -> "TIMES"
|          DIVIDE                                -> "DIVIDE"
|          ASSIGN                                -> "ASSIGN"
|          EQ                                    -> "EQ"
|          NEQ                                    -> "NEQ"
|          LT                                    -> "LT"
|          LEQ                                    -> "LEQ"
|          GT                                    -> "GT"
|          GEQ                                    -> "GEQ"
|          AND                                    -> "AND"
|          OR                                    -> "OR"
|          NOT                                    -> "NOT"
|          DOT                                    -> "DOT"
|          LBRACKET                            -> "LBRACKET"
|          RBRACKET                            -> "RBRACKET"
|          BAR                                    -> "BAR"
|          IF                                    -> "IF"
|          ELSE                                -> "ELSE"
|          FOR                                    -> "FOR"
|          WHILE                                -> "WHILE"
|          RETURN                                -> "RETURN"
|          INT                                    -> "INT"
|          FLOAT                                -> "FLOAT"
|          BOOL                                -> "BOOL"
|          CHAR                                -> "CHAR"
|          VOID                                -> "VOID"
|          NULL                                -> "NULL"
|          TRUE                                -> "TRUE"
|          FALSE                                -> "FALSE"
|          CLASS                                -> "CLASS"
|          CONSTRUCTOR                        -> "CONSTRUCTOR"
|          PUBLIC                                -> "PUBLIC"
```

```
457 |          PRIVATE                                -> "PRIVATE"
458 |          EXTENDS                                -> "EXTENDS"
459 |          INCLUDE                                -> "INCLUDE"
460 |          THIS                                 -> "THIS"
461 |          BREAK                                 -> "BREAK"
462 |          CONTINUE                           -> "CONTINUE"
463 |    NEW                                      -> "NEW"
464 |          INT_LITERAL(i)                     -> "INT_LITERAL(" ^ string_of_int i ^ ")"
465 |          FLOAT_LITERAL(f)                -> "FLOAT_LITERAL(" ^ string_of_float f ^ ")"
466 |          CHAR_LITERAL(c)                    -> "CHAR_LITERAL(" ^ Char.escaped c ^ ")"
467 |          STRING_LITERAL(s)               -> "STRING_LITERAL(" ^ s ^ ")"
468 |          ID(s)                                  -> "ID(" ^ s ^ ")"
469 |          DELETE                                   -> "DELETE"
470 |          MODULO                                   -> "MODULO"
471 |           EOF                                          -> "EOF"
472
473 let string_of_token_no_id = function
474 LPAREN                                    -> "LPAREN"
475 |          RPAREN                                   -> "RPAREN"
476 |          LBRACE                                   -> "LBRACE"
477 |          RBRACE                                   -> "RBRACE"
478 |          SEMI                                   -> "SEMI"
479 |          COMMA                                  -> "COMMA"
480 |          PLUS                                   -> "PLUS"
481 |          MINUS                                  -> "MINUS"
482 |          TIMES                                  -> "TIMES"
483 |          DIVIDE                                   -> "DIVIDE"
484 |          ASSIGN                                   -> "ASSIGN"
485 |          EQ                                         -> "EQ"
486 |          NEQ                                          -> "NEQ"
487 |          LT                                         -> "LT"
488 |          LEQ                                          -> "LEQ"
489 |          GT                                         -> "GT"
490 |          GEQ                                          -> "GEQ"
491 |          AND                                          -> "AND"
492 |          OR                                         -> "OR"
493 |          NOT                                          -> "NOT"
494 |          DOT                                          -> "DOT"
495 |          LBRACKET                           -> "LBRACKET"
496 |          RBRACKET                           -> "RBRACKET"
497 |          BAR                                          -> "BAR"
498 |          IF                                         -> "IF"
499 |          ELSE                                     -> "ELSE"
500 |          FOR                                          -> "FOR"
501 |          WHILE                                    -> "WHILE"
502 |          RETURN                                   -> "RETURN"
503 |          INT                                          -> "INT"
504 |          FLOAT                                    -> "FLOAT"
505 |          BOOL                                   -> "BOOL"
```

```
506 |          CHAR                                -> "CHAR"
507 |          VOID                                -> "VOID"
508 |          NULL                                -> "NULL"
509 |          TRUE                                -> "TRUE"
510 |          FALSE                               -> "FALSE"
511 |          CLASS                               -> "CLASS"
512 |          CONSTRUCTOR                         -> "CONSTRUCTOR"
513 |          PUBLIC                              -> "PUBLIC"
514 |          PRIVATE                             -> "PRIVATE"
515 |          EXTENDS                             -> "EXTENDS"
516 |          INCLUDE                             -> "INCLUDE"
517 |          THIS                                -> "THIS"
518 |          BREAK                               -> "BREAK"
519 |          CONTINUE                          -> "CONTINUE"
520 |    NEW                                     -> "NEW"
521 |          INT_LITERAL(i)                    -> "INT_LITERAL"
522 |          FLOAT_LITERAL(f)                -> "FLOAT_LITERAL"
523 |          CHAR_LITERAL(c)                   -> "CHAR_LITERAL"
524 |          STRING_LITERAL(s)               -> "STRING_LITERAL"
525 |          ID(s)                                -> "ID"
526 |          DELETE                              -> "DELETE"
527 |          MODULO                              -> "MODULO"
528 |           EOF                                     -> "EOF"
529
530 let token_list_to_string_endl token_list =
531 let rec helper last_line_number = function
532 (token, curr)::tail ->
533 let line = curr.lineno in
534 (if line != last_line_number then "\n" ^ string_of_int line ^ ". " else " ") ^
535 string_of_token token ^ helper line tail
536 |          [] -> "\n"
537 in helper 0 token_list
538
539 let token_list_to_string token_list =
540 let rec helper = function
541 (token, line)::tail ->
542 string_of_token_no_id token ^ " " ^ helper tail
543 |          [] -> "\n"
544 in helper token_list
```

References

[1] http://www.gnu.org/software/gnu-c-manual/gnu-c-manual.html *The GNU C Reference Manual.*. N.p., n.d. Web. 26 Oct. 2015.

[2] https://docs.oracle.com/javase/specs/jls/se8/html/index.html *The Java Language Specification.* . N.p., n.d. Web. 26 Oct. 2015.

[3] Edwards, Stephen. *"Programming Language and Translators."* Lecture.

[4] "Control Flow Statements." *The Java Tutorials Learning the Java Language Language Basics* N.p., n.d. Web. 26 Oct. 2015.