The RAPID Programming Language

Nate Brennand (nsb2142)  Ben Edelstein (bie2103),  Brendan Fish (bjf2127),  
Dan Schlosser (drs2126),  Brian Shin (ds2791)

December 17, 2014
1. Introduction

With increased demand in the public and private sector for cloud-connected mobile and web applications has come a rising need for web servers to maintain state across multiple devices and users. Development of web servers is complex, however. Building a web server using modern web server packages requires learning a server-side programming language, and then integrating a web server package and implementing required methods. Furthermore, local testing and development of these servers is excessively complex, as they have numerous dependencies and are difficult to build.

RAPID is a programming language intended specifically for the rapid development of modern web APIs. Using RAPID, developers can quickly build REST API routes using familiar paradigms in object oriented programming. This abstracts away much of the boiler plate code that developers typically write when building an API server.

0.1 1.1 Why RAPID?

The name RAPID represents the goal of the language: making API server development quick. Also, it’s a recursive acronym for RAPID Application Programmer Interface Dialect.

2. Tutorial

Variables

```java
int x = 5;
string b = "hello world";
boolean f = false;
float pi = 3.14;
```

Casting

**Float <-> Int**

```java
float f = 7.5
int i = 3
float f = float(i) // f == 3.0
int i = int(f)    // i == 7
```

**Boolean Casting**

Boolean casting is accomplished using the ? operator. All primitive types and lists can be cast to boolean.
int i = 7;
i? // true

String s = "Rapid Rocks";
s? // true

String e = "";
e? // false

Lists

list<int> a = [1,2,3,4];
list<string> b = ["hello", "world"];

Comments

// This is a single line comment

/*
This is a multi-line
comment
*/

Simple function example

func gcd(int p, int q) int {
    while (q != 0) {
        int temp = q;
        q = p % q;
        p = temp;
    }
    return p;
}

Simple GCD Server

func gcd(int p, int q) int {
    while (q != 0) {
        int temp = q;
        q = p % q;
        p = temp;
    }
    return p;
}
namespace gcd {
    param int a {
        param int b {
            http (int a, int b) int {
                int res = gcd(a, b);
                return res;
            };
        }
    }
}

Here, the namespace represents an http route, and the params represent inputs with that route. For example, sending a get request to `http://hostname:8080/gcd/15/20` would return 5.

Simple Object Oriented Programming

class User {
    int age;
    string name = "Stephen";
    optional int height;

    instance my {
        func is_old() boolean {
            return (my.age >= 30);
        }
        func make_older() {
            my.age = my.age + 1;
        }
    }
}

User bob = new User(age=29);
println(bob.age)

Instance methods are called by using dot notation (`bob.is_old()`) and from within an object by using the instance name before the dot (`my.is_old()`).
Our LRM is provided here as originally submitted. Please note that this version of the LRM does not describe all features implemented in this version of RAPID, rather an LRM for the language we sought to make.

RAPID Language Reference Manual

Coms W 4115

Ben Edelstein, Brian Shin, Brendon Fish, Dan Schlosser, Nate Brennand

1. Introduction

With increased demand in the public and private sector for cloud-connected mobile and web applications has come a rising need for web servers to maintain state across multiple devices and users. Development of web servers is complex, however. Building a web server using modern web server packages requires learning a server-side programming language, and then integrating a web server package and implementing required methods. Furthermore, local testing and development of these servers is excessively complex, as they have numerous dependencies and are difficult to build.

RAPID is a programming language intended specifically for the rapid development of modern web APIs. Using RAPID, developers can quickly build a database-backed REST API server that guarantees JSON shapes in responses. RAPID is object oriented and database-backed, meaning that classes represent an SQL table, and upon instantiation objects are automatically saved in the database. This abstracts away much of the boiler plate code that developers typically write when building an API server.

1.1 Why RAPID?

The name RAPID represents the goal of the language: making API server development quick. Also, it’s a recursive acronym for RAPID Application Programmer Interface Dialect.

1.2 RAPID Programs

There are two types of RAPID programs, servers and scripts. If a program contains an HTTP method, it is a server, otherwise it is a script. (See more in later subsubsections).
2. Lexical Conventions

2.1 Identifiers

Identifiers must start with a letter or an underscore, followed by any combination of letters, numbers, and underscores.

Valid Identifiers:

abc, abc_def, a___1, __a__, _1, ABC

Invalid Identifiers:

123, abc-def, 1abc, ab\ cd

2.2 Keywords

The following identifiers are keywords in RAPID, and are reserved. They can not be used for any other purpose.

if, else, for, in, while, switch, case, default, fallthrough, http, func, json, class, namespace, param, true, false, new, optional, unsafe, instance, and, or

2.3 Literals

Integer literals  Integer literals may be declared using digits.

int x = 5

Float literals  Float literals are declared as an integer part, a decimal point, and a fraction part, all of which are mandatory. The integer part may not start with a zero, unless it is only a zero (for floats less than 1.0), in which case it is still required. There may not be any whitespace between these three parts.

// Valid float literals:
float x = 15.0
float y = 0.25

// Invalid float literals:
float z = .5
float w = 10.
float v = 1 . 4
**String literals**  String literals are declared using double quotes. Special characters may be declared using the `\` escape character.

```cpp
string a = "hello"
string b = " \"world\"\n"
```

**Boolean literals**  Boolean literals are declared using the `true` and `false` keywords.

```cpp
boolean t = true
boolean f = false
```

**List literals**  List literals may be declared between square brackets, with comma-separated values.

```cpp
list<int> a = [1,2,3,4]
list<string> b = ["hello", "world"]
```

**Dictionary literals**  Dictionary literals may be declared as comma-separated key value pairs between braces, with a colon separating the key and value. Whitespace is ignored.

```cpp
dict<string, int> a = {"hello": 4, "world": 5}
dict<string, int> b = {
    "a": 42,
    "b": 27
}
```

### 2.4 Comments

There are two types of comments in RAPID: single line comments, and block comments. Single line comments are preceded by `//` and block comments begin with `/*` and end with `*/`

```cpp
// This is a single line comment

/*
This is a multi-line
comment
/*/ 

/* They may be nested */
*/
```

### 2.5 Operators
<table>
<thead>
<tr>
<th>Operator</th>
<th>Use</th>
<th>Associativity</th>
<th>Types</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>Addition</td>
<td>left</td>
<td>int, float</td>
</tr>
<tr>
<td>*</td>
<td>Multiplication</td>
<td>left</td>
<td>int, float</td>
</tr>
<tr>
<td>/</td>
<td>Division</td>
<td>left</td>
<td>int, float</td>
</tr>
<tr>
<td>-</td>
<td>Subtraction</td>
<td>left</td>
<td>int, float</td>
</tr>
<tr>
<td>%</td>
<td>Modulus</td>
<td>left</td>
<td>int</td>
</tr>
<tr>
<td>=</td>
<td>Assignment</td>
<td>non-associative</td>
<td>All</td>
</tr>
<tr>
<td>==</td>
<td>Equal to</td>
<td>non-associative</td>
<td>All</td>
</tr>
<tr>
<td>!=</td>
<td>Not equal to</td>
<td>non-associative</td>
<td>All</td>
</tr>
<tr>
<td>&gt;</td>
<td>Greater than</td>
<td>non-associative</td>
<td>int, float</td>
</tr>
<tr>
<td>&lt;</td>
<td>Less than</td>
<td>non-associative</td>
<td>int, float</td>
</tr>
<tr>
<td>&gt;=</td>
<td>Greater than or equal to</td>
<td>non-associative</td>
<td>int, float</td>
</tr>
<tr>
<td>&lt;=</td>
<td>Less than or equal to</td>
<td>non-associative</td>
<td>int, float</td>
</tr>
<tr>
<td>and</td>
<td>Logical And</td>
<td>non-associative</td>
<td>bool</td>
</tr>
<tr>
<td>or</td>
<td>Logical Or</td>
<td>non-associative</td>
<td>bool</td>
</tr>
</tbody>
</table>

3. Types

3.1 Static Typing

RAPID is a statically typed language; variables must be explicitly typed upon declaration. Variables can be cast to other types (see Casting).

3.2 Primitive Types

**null**  In RAPID, the `null` keyword represents an uninitialized value. Any type in rapid may take on `null` if it hasn’t been initialized, or otherwise doesn’t exist. The `null` keyword represents a value, but null values are still typed. Null variables of different types may not be assigned to each other, and may not be compared. Null variables of the same type are equal. All variables will `null` value are equal to the keyword `null`.

```rapid
int x
int y = null
string s
boolean eq = (x == y) and (x == null) and (s == null)
x == s  // not valid RAPID
x = s  // not valid RAPID
```

Null values of any type may not be used in operations together. If they are, the program will exit prematurely, or the HTTP server will return a 500 error.

```rapid
int x       // null
```
int y = x + 2 // not allowed, the program exits or the request returns 500.
list<int> a = [1, 2, 3, 4]
a[x] // not allowed, the program exits or the request returns 500.

**Booleans**  Boolean values are defined by the `true` and `false` keywords. Because they are their own type, non-boolean values must be cast to `boolean` in order to be used in logical expressions.

For example:

```rapid
!(3+5)? // valid RAPID
!(3+5) // not valid RAPID
```

The `?` is a an operator on all primitive types that evaluates to the “truthiness” of that value.

**Integers**  Integers are preceded by the type `int`, and represent an 8 byte, signed integer. Integers can be declared and initialized later, or initialized inline. Uninitialized integers are null.

```rapid
int i // null
int i = 5 // 5
```

Integers are copied by value.

```rapid
int a = 1
int b = a
a = 2
printf("%d, %d", a, b) // 2 1
```

**Floating Point Numbers**  Floating point numbers are preceded by the type `float`, and represent IEEE-754 64-bit floating-point numbers.. They can be declared and initialized later, or initialized inline.

```rapid
float i // null
float j = 3.14 // 3.14
```

**Strings**  Strings in RAPID are mutable, and declared with the `string` type, and have the default value of the empty string. String literals are declared using double quotes, and special characters may be escaped using the `\` character. Strings may be indexed using square brackets. Because there is no Character type in RAPID, single characters are strings of length 1. Multiline strings may be declared using triple double quotes. Newlines are preserved and quotes do not need to be escaped, but they may not be nested. Strings are pass by value.
string s // null
string character = "c" // c
string s = "He is \"Batman\"" // He called himself "Batman"
string c = s[0] // H
string multi = ""

Did you hear?

"" // multi[0] => "\n"

3.3 Non-Primitive Types

List  The list type is a zero-indexed array that expands to fit its contents. The type of the contents must be provided within angle brackets in the type signature. RAPID list literals may be declared using square brackets, and values may be accessed or set using square brackets. Uninitialized lists default to the empty list. Lists are pass by reference.

/* List declaration */

list< /* type */ > /* id */ = [ /* expression */, /* expression */, ...

]  

list<int> empty // []
list<int> numbers = [1,2,3,42]
numbers[3] // 42
numbers[1] = 5 // [1,5,3,42]

Dictionary  The dict type is similar to an object, but its key set is mutable. The type of the key and value must be provided within angle brackets in the type signature. Only primitive types may be used as keys. Keys may be added, set, and accessed using square brackets. RAPID dict literals may be declared as comma-separated key:value pairs surrounded by braces. Uninitialized dictionaries default to the empty dictionary. Dictionaries are pass by reference.

/* Dictionary declaration */

dict< /* type */ , /* type */ > /* id */ = {
  /* expression:string */ : /* expression */,
  /* expression:string */ : /* expression */,
  ...
}
Object  The object type is a generic, non-primitive, dictionary-backed type that has attributes for instance variables and functions. Accessing instance variables or functions can be done with dot notation. Objects may not be declared anonymously; they must be declared as instances of classes. Objects have immutable key sets, so variables and functions may not be added or removed, although their values may be changed. For more on classes and instantiation, see Classes.

json  The json type is shares qualities of a dictionary and of an Object. Every json type is directly connected to an Object class that is user-defined. They have keys and values like dictionaries, but have the strict requirements of shape like objects do. Every property of a class is a mandatory key on the corresponding json object, and properties that have default values on objects have default values in json. Unlike objects, however, json objects do not have methods associated with them, and instances do not represent rows in the database. Each class declaration defines an Object type and a json object type, and only json objects that are associated with classes may be instantiated.

For example, if we previously defined a User object with three instance variables username, full_name, and password (all strings), then we may declare a json User like so:

/* JSON Object initialization */

json< /* id:classname */ > /* id */ = json< /* id:classname */ >(
  key = /* expression */,
  key = /* expression */,
  ...
  key = /* expression */
)

json<User> steve = json<User>(
  username="sedwards",
  full_name="Stephen Edwards",
  password="easypeasy"
)

Errors  Errors in RAPID are not thrown and caught, rather they are returned directly by unsafe functions (see Functions). Errors contain a string message, which can be dot-accessed, an integer error code that conforms with the HTTP/1.1 standard, and an optional string name.
For example, to declare a custom error:

```plaintext
error e = error(message="There was an error with that Request.",
                  code=400,
                  name="RequestError")
```

Unsafe operations return an error as the last return value:

```plaintext
dict<string, int> d = {"foo": 4, "bar": 5}
int val, error e = d['baz']
if (!e) {
    printf("%d\n", e.code) // 500
    printf("%s\n", e.message) // Key error when accessing "baz" on 'd'.
    printf("%s\n", e.name) // KeyError
}
```

Many standard library classes and built-in objects define errors pertinent to their functions, to which an error instance may be compared.

```plaintext
dict<string, int> d = {"foo": 4, "bar": 5}
int val, error e = d['baz']
if (!e) {
    printf("%s", e == dict.KeyError) // true
}
```

**Stacking** Unsafe functions (like list and dictionary access) may exist in the same expression. If unsafe functions return successfully, the error that is returned is consumed (ignored), and the return value is taken. If an unsafe function returns an error, the expression evaluation short-circuits, and the value of the expression is null and the error that is returned by the failed function call.

```plaintext
dict<string, list<int>> d = {"foo": [4,5], "bar": [1,2,3]}
int val, error e = d['foo'][2] // List index out of bounds...
printf("%d", val) // null
printf("%s", e.name) // IndexError
printf("%t", e == list.IndexError) // true
```

```plaintext
val, e = d['baz'][0] // No such key, short circuit
printf("%d", val) // null
printf("%s", e.name) // KeyError
printf("%t", e == dict.KeyError) // true
```

More generally, if a subexpression of an expression is unsafe, it is presumed to be successful and the return value of the subexpression is used in the evaluation of the larger expression, unless the unsafe expression evaluates to an error, in which case evaluation of the large expression short-circuits, and the value of the large expression is null, /* sub-expressions error */.
Predefined Responses

Unsafe functions may also choose to return a predefined response, which is a predefined literal that will be cast to a generic error object at compile time. See Functions for more details.

All predefined errors exist in the root scope and are named according to their status code, `e<code>`.

`e404` is the error, `error("Not Found", 404, "NotFound")` (message, code, name). All the below errors are predefined as such.

<table>
<thead>
<tr>
<th>Response</th>
<th>Message</th>
<th>Code</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>e100</td>
<td>Continue</td>
<td>100</td>
<td>Continue</td>
</tr>
<tr>
<td>e200</td>
<td>OK</td>
<td>200</td>
<td>OK</td>
</tr>
<tr>
<td>e201</td>
<td>Created</td>
<td>201</td>
<td>Created</td>
</tr>
<tr>
<td>e301</td>
<td>Moved Permanently</td>
<td>301</td>
<td>MovedPermanently</td>
</tr>
<tr>
<td>e302</td>
<td>Found</td>
<td>302</td>
<td>Found</td>
</tr>
<tr>
<td>e304</td>
<td>Not Modified</td>
<td>304</td>
<td>NotModified</td>
</tr>
<tr>
<td>e400</td>
<td>Bad Request</td>
<td>400</td>
<td>BadRequest</td>
</tr>
<tr>
<td>e401</td>
<td>Unauthorized</td>
<td>401</td>
<td>Unauthorized</td>
</tr>
<tr>
<td>e403</td>
<td>Forbidden</td>
<td>403</td>
<td>Forbidden</td>
</tr>
<tr>
<td>e404</td>
<td>Not Found</td>
<td>404</td>
<td>NotFound</td>
</tr>
<tr>
<td>e405</td>
<td>Method Not Allowed</td>
<td>405</td>
<td>MethodNotAllowed</td>
</tr>
<tr>
<td>e410</td>
<td>Gone</td>
<td>410</td>
<td>Gone</td>
</tr>
<tr>
<td>e413</td>
<td>Request Entity Too Large</td>
<td>413</td>
<td>RequestEntityTooLarge</td>
</tr>
<tr>
<td>e414</td>
<td>Request-URI Too Long</td>
<td>414</td>
<td>RequestURITooLong</td>
</tr>
<tr>
<td>e417</td>
<td>Expectation Failed</td>
<td>417</td>
<td>ExpectationFailed</td>
</tr>
<tr>
<td>e500</td>
<td>Internal Server Error</td>
<td>500</td>
<td>InternalServerError</td>
</tr>
<tr>
<td>e501</td>
<td>Not Implemented</td>
<td>501</td>
<td>NotImplemented</td>
</tr>
<tr>
<td>e502</td>
<td>Bad Gateway</td>
<td>502</td>
<td>BadGateway</td>
</tr>
<tr>
<td>e503</td>
<td>Service Unavailable</td>
<td>503</td>
<td>ServiceUnavailable</td>
</tr>
<tr>
<td>e504</td>
<td>Gateway Timeout</td>
<td>504</td>
<td>GatewayTimeout</td>
</tr>
</tbody>
</table>

Functions

Functions are first class objects, and may be passed around as variables (see Functions)

3.4 Casting

Integers and Floats

Casting between float and int can be done using the `float()` and `int()` keyword functions. Floats are floored when they are cast to int. Additionally, integers are cast to floats if floats and integers are used together in a binary operator.

```plaintext
float f = 7.5
int i = 3
float f = float(i) // f == 3.0
int i = int(f)    // i == 7
```
When an int and a float are involved in a binary operator, the integer will be cast to a float implicitly.

```java
float f = 7.5 + 10  // 17.5
boolean eq = 4.0 == 4 // true
```

**Booleans** Any value may be cast to boolean using the ? operator.

See the following table for the result of using the ? operator on various types:

<table>
<thead>
<tr>
<th>Type</th>
<th>true</th>
<th>false</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>boolean</td>
<td>true?</td>
<td>false?</td>
<td>Booleans retain their value</td>
</tr>
<tr>
<td>int</td>
<td>1?, -1?</td>
<td>0?</td>
<td>0 is false, other ints are true</td>
</tr>
<tr>
<td>float</td>
<td>1.0?, -1.0?</td>
<td>0.0?</td>
<td>0.0 is false, other floats are true</td>
</tr>
<tr>
<td>null</td>
<td>-</td>
<td>null?</td>
<td>null is false</td>
</tr>
<tr>
<td>list</td>
<td>[0]?, [false]?</td>
<td>[]?</td>
<td>Empty lists are false</td>
</tr>
<tr>
<td>dict</td>
<td>{&quot;key&quot;:false}?</td>
<td>{}?</td>
<td>Empty dicts are false</td>
</tr>
<tr>
<td>json</td>
<td>json&lt;Obj&gt;()?</td>
<td>-</td>
<td>JSON objects are true</td>
</tr>
<tr>
<td>object</td>
<td>Obj()?</td>
<td>-</td>
<td>Objects are true</td>
</tr>
</tbody>
</table>

4. Database Backing

4.1 Classes

RAPID classes are backed by a PostgreSQL database. Classes are defined using the `class` keyword, and represent an SQL table. Instance variables (variables declared directly within the `class` block) represent columns for the table. Instances of a class represent rows in SQL. By default, columns are not nullable, but this may be overwritten using the `optional` keyword. If the assignment syntax is used, a default value will be given.

```java
class /* id */ {
    /* declaration */
    /* declaration */
    ...
    /* declaration */
}
```

Take the following example:

```java
class User {
    string username
    optional string full_name
```
int age = 18
string password
}

In this example, the "User" table has four columns: `username`, `full_name`, `age`, and `password`. The `full_name` column may be omitted in the instantiation, and if `age` is omitted, it will take the value 18.

**Instance Methods** Instances of objects may have methods that operate on their instances variables. Using the `instance` keyword, a block may be created in which instance methods may be defined:

```plaintext
class /* id:classname */ {
    instance /* id:selfname */ {
        /* declaration */
        /* declaration */
        ...
        /* declaration */
    }
}
```

The identifier specified after the `instance` keyword will represent the instance in all functions defined inside the `instance` block.

Instance methods may not be `http` routes. The `.` is syntactic sugar for calling instance methods.

For example:

```plaintext
class User {
    string first_name
    string last_name

    instance my {
        func get_full_name() string {
            return my.first_name + " " + my.last_name
        }
    }
}

User obama = User(first_name="Barrak", last_name="Obama")
printf("%s", obama.get_full_name()) // Barrak Obama
```

**Instantiation** New instances of a class may be declared using the `new` keyword. The `new` keyword is followed by the name of the class and a pair of parenthesis, in which a JSON User literal (described
more in-depth in the next subsubsection) may be passed to declare instance variables. Once a user
defined object is created, it will be database backed. Any changes to the object will trigger an
update to the database backed copy. Every object will have an ID attribute generated (this is a
reserved attribute that cannot be used). This is a unique identifier to the object.

```
User bob = new User(
    username="burgerbob",
    full_name="Bob Belcher",
    password="burgersrock",
    age=42
)
```

**Deletion**

All objects have an instance method, `delete()`, defined that will delete the database record. There
is no return value for the `delete` call.

```
json<User> bob_json = json<User>(
    username="burgerbob",
    full_name="Bob Belcher",
    password="burgersrock",
    age=42
)
```

This `json<User>` object does not represent a row in the database, and will be deallocated
when it leaves scope.

It may be passed into an instantiation statement for a User object, to be persisted:

```
User bob, error e = new User(bob_json)
```

**4.2 Querying**

Objects may be queried from the database using the `get` function, which is automatically defined
on all classes. `get` is an unsafe function which will return a list of objects as well as an error.

The following example queries all User objects from the database:

```
Tweet[] tweets = Tweet.get()
```
A optional filter parameter can be set to limit the responses returned by Get(). The filter value should be a dictionary of attributes and values. Any non-existent attributes in the dictionary will be logged as a warning and ignored.

```java
// returns all tweets by burgerbob.
Tweet[] tweets, error e = Tweet.get(filter={
    username="burgerbob"
})
```

A optional ID parameter can be set to return a single Object. This can be combined with filter if desired.

In the case that the object is not found, the returned object will be null and the error will be a non-null value.

```java
// returns all tweets by burgerbob.
Tweet t, error e = Tweet.get(ID="123abc")
```

### 4.3 Updates

### 5. Functions

#### 5.1 Declaration

Functions in RAPID are first-class objects, but may not be declared anonymously. Functions are declared using the `func` keyword. The arguments (within parenthesis), return type (after the parenthesis, but before the braces), and the body of the function (within the braces) must be declared explicitly. Return types may include multiple types separated by commas, or may be omitted for void functions.

Return values are specified using the `return` keyword, which must be followed by an expression to be returned for functions that have declared return types. If the return type is omitted, the function is void, and the result of calling it may not be assigned to a value. Void functions may use the `return` keyword by itself to exit prematurely from the function.

Unsafe functions may not be void, because they must return errors.

```java
return /* expression */
```

The arguments must be in order namespace arguments, then formal arguments. Arguments may be given a literal default value, using an equal sign, but all arguments with default values must follow arguments without default values.

```java
[unsafe] func /* id */ ( /* namespace args */ /* formal args */ ) /* return type*/ {
    // statements
}
```
For example:

```go
func sum(int a, int b=1) int {
    return a + b
}
sum(5) //

Or:

func printInt(int a) {
    printf("%d", a)
}
```

### 5.2 Unsafe Functions

If a function performs actions that may be unsafe, it must be preceded by the keyword `unsafe`. Unsafe functions return unsafe expressions, which is denoted by the presence of an `error`-typed second value that is returned.

```go
unsafe func access(dict<string, int> d, string key) int {
    int val, error error = d[key]
    return val, error
}
```

Notice that the return type remains `int`, although an error is also returned. For more on unsafe expressions, see Expressions.

Unsafe functions may also return a `error`, which are integer literals that will be cast to a generic `error` object at compile time. See Status Code Definitions for a complete list of error codes that may be declared as anonymous errors.

```go
/* Default dict accessing:
* If there is a KeyError, return 0 with a 400 Not Found error
*/
unsafe func access(dict<string, int> d, string key) int {
    int val, error error = d[key]
    if (error == dict.KeyError) {
        return 0, e400
    }
    return val, e200
}
```
6. Routing

One of the core features of RAPID is it’s ability to easily define routes for a REST API server.

6.1 Declaring Routes

Routes may be declared like functions, but substituting the `http` keyword for the `func` keyword. Routes specify a REST API endpoint, it’s parameters, it’s response, and any side effects.

Like functions, routes take namespace arguments, and then other formal arguments. Unlike functions, however, routes may also take a single request body argument that of a `json<Obj>` type. It will be read from the request body and interpreted as JSON.

```
http /* id */ ( /* namespace args */ /* formal args */ /* request body args */) {
    // statements
}
```

Routes are unsafe by default, and therefore must include `error` in their return types. This may be an anonymous error (see Functions).

For example, the following route echos the URL parameter that it is passed.

```
http echo(string foo) string, error {
    return foo, e200
}
```

The name of the function will be the name of the route. Therefore, in the preceding example, a GET request to `/echo?foo=Dog` will return "Dog".

6.2 Path context

The endpoint associated with each route is determined by the combination of one or more blocks associated with it and the name of the route itself. There is a one-to-one mapping from any route to a series of accessors on class instances.

**Classes** Classes provide path context. Class names are put to lowercase, and appended to path context. The following example defines a route named `add` inside a class called `Math`.

```
class Math {
    http add(int a, int b) int {
        return a + b, e200
    }
}
```
A GET request to /math/add?a=3&b=4 will return 7.

Similarly, the following code will print 7:

```java
math = Math()
int sum, error = math.add(3,4)
printf("%d", sum)
```

**Namespaces** Sometimes, functions or routes should be grouped together for organization purposes, rather than any functional purpose. The `namespace` keyword defines a named block of functions that has the namespace name appended to the path context for those functions.

```java
/* Namespace declaration */
namespace /* id */ {
    // statements
}
```

class Math {
    namespace ops {
        http add(int a, int b) int { return a + b, e200 }
        http sub(int a, int b) int { return a - b, e200 }
    }
    namespace convert {
        func ft_to_in(float feet) float { return feet*12, e200 }
    }
}

This defines routes at /math/ops/add and /math/ops/sub, and functions at Math.ops.add, Math.ops.sub, and Math.convert.ft_to_in.

A GET request to /math/ops/add?a=3&b=4 will return 7.

**Parameters** Variable URL parameters may be defined similar to namespaces, using a named block with the `param` keyword. The `param` keyword is followed by a type and an identifier.

Any function or route defined within a `param` block must take the parameters defined by the `param` blocks in order from inside to out.

```java
param /* type */ /* id */ { /* id */ }
```

For example:
class Math {
    param int a {
        param int b {
            http add(int a, int b) int { return a + b, e200 }
        }
        http square(int a) int { return a*a, e200 }
    }
}

A GET request to /math/5/7/add will return 12, and a GET request to /math/5/square will return 25. A GET request to /math/5/7/add?a=4 will return a 400 HTTP error. The following code snipped will print 12 then 25:

```rapid
math = Math()
int sum, error = math.add(5,7)
printf("%d", sum)
int sqr, error = math.square(5)
printf("%d", sqr)
```

7. Syntax

7.1 Program Structure

A valid RAPID program is a series of valid statements. If the program contains any http blocks, it will be interpreted as a restful web API, and will run a HTTP web server on localhost:5000.

7.2 Expressions

Expressions are series of operators and operands that may be evaluated to a value and type. Any subexpressions are evaluated from left to right, and side effects of evaluations occur by the time the evaluation is complete. Type checking on operations occur in compile time.

Literals  Literals may be of type string, integer, float, boolean, dict, or list. See Lexical Conventions for more information.

Identifiers  Identifiers could be primitive types, lists, dictionaries, objects, JSON objects, functions, classes, or errors.

If an identifier represents a primitive type, list, dictionary, object, JSON object, or error, it may be reused once per block.

For example, in the following example, the variable a changes value three times.
Identifiers are tied to the scope that they are declared in. The following example prints 3, then 5, then 3:

```rapid
int a = 3
if (true) {
    printf("%d", a) // 'a' is from the parent scope.
    int a = 5
    printf("%d", a) // 'a' is from the local scope.
}
printf("%d", a) // the 'a' from within the block does not leave it
```

### Binary Operators

Binary operators have two operands, one on the left side, and one on the right.

```rapid
/* expression */ /* bin-op */ /* expression */
```

In the case of multiple consecutive binary operations without parenthesis, the association of the binary operator is followed (see Operators).

### Parenthesized Expressions

Parenthesis may be used to alter the order of operand evaluation.

### 7.3 Statements

#### Assignments

Assignments have a lvalue, and an expression, separated by an equal sign. Possible lvalues include identifiers, accessors (either list, dict, or object), a declaration, or another assignment:

```rapid
/* lvalue */ = /* expression */
```

Examples include:

```rapid
a = b
int i = 7
j = square(i)
k = 5 * b
```
**Declarations**  A declaration may be the declaration of a variable, an assignment, or the declaration of a function, route, class, namespace, or param.

**Variable Declaration**  A variable declaration consists of a type and an id. A variable declared in a scope block is accessible at every line following the line of its declaration.

/* type */ /* id */

**Function Declaration**  The declaration of a function is a valid statement (see Functions). Functions defined in a scope are accessible from anywhere in that scope. Functions may call each other mututally independant of definition order.

**Route Declaration**  The declaration of a class is a valid statement (see Routing). Like functions, routes declared in a scope are accessible from anywhere in that scope.

**Class Declaration**  The declaration of a class is a valid statement (see Classes). Like functions, classes declared in a scope are accessible from anywhere in that scope.

**Namespace or Parameter Declaration**  The declaration of a namespace or parameter is a valid statement (see Path Context). Like functions, namespaces or parameters declared in a scope are accessible from anywhere in that scope.

**Function call**  A function call is an identifier of a declared function and a set of parenthesis containing the comma-separated arguments. There may not be a space between the identifier and the open parenthesis. Function arguments may be referenced by name, independent of whether or not the argument has a default value. When arguments are referenced in the function call by name, they may be rearranged, but may not be placed before arguments that are not referenced by name.

```rapid
func sub(int a=2, int b=1) int { return a - b }
int x = sub()    // 1
int y = sub(4, 2) // 2
int z = sub(b=5, a=2) // 3
int w = sub(7, b=3) // 4
int f = sub(a=3, 2) // Not valid RAPID
```

**Control flow**

**If**  If the expression between the parenthesis of an if statement evaluates to true, then the statements within the body are executed. Note that non-boolean values will not be cast to boolean, and will result in a compile-time error.
if (/* expression */) { /* statements */ }

**If-else**  An if statement may be immediately followed by an else statement, in which case the block of code within the braces after the else keyword will be executed if the if’s expression evaluates to false.

if (/* expression */) {
    // statements
} else {
    // statements
}

**Else-if**  An if statement may be followed by an else if statement, in which case the second if statement will be evaluated if and only if the first if statement evaluates to false. The body of the else if is executed if the second if statement is evaluated, and evaluates to true. An else if statement may be followed by another else if statement, or an else statement.

if (/* expression */) {
    // statements
} else if (/* expression */) {
    // statements
} else if (/* expression */ ) {
    // statements
} else {
    // statements
}

**Switch**  A switch statement includes an expression, which is evaluated and then compared in order to a series of one or more case expressions. If the expressions are equal, the body of the case statement that matches will be executed, and then the switch statement will short circuit. The fallthrough keyword may be used to avoid this short circuit, continuing to compare the switch expression with subsequent case expressions.

The default statement may be included after all case statements, and will be executed if it is reached. This can be thought of as a case whose expression always equals that of the switch. Observe the syntax below:

switch (/* expression */) {

While loops  While loops contain an expression and a body. If the expression evaluates to `true`, the body will be executed. Afterwards, the expression will be evaluated again, and the process repeats. Like `if` statements, `while` statements must have expressions that evaluate to a boolean in order to compile.

```c
while (/* expression */) {
    // statements
}
```

For loops  A `for` loop may be used to iterate over a list. The syntax is:

```c
for (/* type */ /* id */ in /* list expr */) {
    // statements
}
```

For example:

```c
list<int> my_list = [1,2,3,4,5]
for (int num in my_list) {
    printf("%d ", num)
}
// 1 2 3 4 5
```

The `range()` function in the standard library may be used to generate lists of sequential integers.

```c
for (int num in range(1,6)) {
    printf("%d ", num)
}
// 1 2 3 4 5
```
Return statements  A return statement may be used to exit a function, optionally passing the value of an expression as the return value of the function.

return /* optional expression */

For example:

func int add(int x, int y) int {
    return x + y
}
printf("%d", add(3,4))
// 7

Break Statements  A break statement can be used to exit a loop prematurely.

while (/* expression */) {
    break
}

In the case of nested loops, the break statement only breaks the loop in which it is stated.

while (/* expression */) {
    while (/* expression */) {
        break /* only breaks inner loop */
    }
}

8. Built-in Functions

8.1 length()

func length(string s) int
func length(list<T> l) int
func length(dict<T,S> d) int
func length(json<T> j) int

Returns the length of the argument. For strings, this is the number of characters in the string, for lists, this is the number of elements in the list. For dictionaries, this is the number of keys, for JSON objects, this is the number of keys.

Examples:
length("hello") // 5
length([0,1,2,3]) // 4
length({"a":0, "b":null, "c": False, "d": ""}) // 4

Taking the length of a null value will return null

8.2 range()

func range(int stop) int[]
func range(int start, int stop[, int step=1]) int[]

Returns a list of integers $r$ where $r[i] = start + step*i$ where $i>=0$ and while $r[i] < stop$. If start is omitted, it defaults to 0. If step is omitted, it defaults to 1.

Step may be negative, in which case $r[i] > stop$

Examples:

range(5) // [1,2,3,4,5]
rangereg(4,5) // [4]
rangereg(3,7,2) // [3,5]
rangereg(10,4,-2) // [10,8,6]

8.3 Output Functions

There are four methods of outputing from the server. They all accept a format string as the first argument and optional additional arguments for format strings.

Print Functions

• printf(String formatStr, [values])

printf does not include a newline at the end of the output. The output is directed to STDOUT.

Logging Functions

• log.info(String formatStr, [values])
• log.warn(String formatStr, [values])
• log.error(String formatStr, [values])

All output is preceded by a timestamp. The logging functions print with a newline at the end. The output is directed to STDERR.

The method name being called precedes the message in all caps.
log.info("Hello, %s", "world")
// 2009/11/10 23:00:00 INFO: Hello, world

9. Standard Library

9.1 string

string.is_empty()

func is_empty() boolean

Returns a boolean value of whether the string on which it is called is of length 0 or null.

Examples:

string a = "dog"
string b = ""

a.is_empty() // false
b.is_empty() // true

string.substring()

unsafe func substring(start, stop) string

Returns the substring of a string at the given indexes. The start and stop indexes are inclusive and exclusive respectively. Providing improper indexes will cause the function to throw an error (both must be 0 <= i <= length(s))

string a = "catdog"

string sub, error e = a.substring(1,4) // "atd", null
string sub, error e = a.substring(3,99) // null, error
string sub, error e = a.substring(50,99) // null, error

Get (c = string[i]) Strings may be indexed using brackets. Inside the brackets must be a zero-indexed integer. Getting is unsafe, and returns string, error.

Examples:

string a = "catdog"

printf("%s", a[3]) // prints d
Set (string[i] = s)  After indexing, an assignment may occur, to set a value of the list. Setting is **unsafe** due to the possibility of index errors, and returns **string, error**.

Examples:

```plaintext
string a = "catdog"
a[2], error e = "p"
if (!e?) {
    printf("%s", a)
}
// prints capdog
```

**Iterate (c in string)** Strings may be iterated over in for loops. Each element is returned in order.

Examples:

```plaintext
string a = "catdog"
for (string c in a) {
    printf("%s", c)
}
// prints catdog
```

**Slice (string[i:j])** Strings may be sliced into substrings using slicing with brackets. Slicing is **unsafe**, and returns **string, error**. Note that unlike for list.substring, the second index of a slice may be larger than the length of the string.

```plaintext
string a = "catdog"
a[1:4]  // "atd"
a[3:99]  // "dog"
a[50:99]  // error
```

9.2 list

list.is_empty()

**func is_empty() boolean**

Returns whether the list on which it is called is empty.

```plaintext
list<int> a = []
list<int> b = [3,4]
```
a.is_empty()  // false
b.is_empty()  // true

list.append()

func append(T elem) list<T>

Appends the argument to the end of a list. The list is returned which allows for chaining of append calls but the function has side effects and does not need to be used in an assignment.

list<int> a = []
a.append(7)  // [7]
a.append(3)  // [7,3]

list.pop()

unsafe func pop() T

Removes the last element in a list, and returns it. If the list is empty, an error is returned.

list<int> a = [3,4]
a.pop()     // 4
a.pop()     // 3
a.pop()     // error

list.push()

func push(T elem) list<T>

list.concat()

func concat(list<T> l) list<T>

list.reverse()

func reverse() list<T>

Reverses the list on which it is called and returns the reversed list.
```c
list<int> a = [1,2,3,4,5]

a.reverse()  // [5,4,3,2,1]

list.copy()  Copies by the list value.

func copy() list<T>

Returns a copy by value of the list on which it is called

list<int> a = [1,2,3,4,5]  // [1,2,3,4,5]

Get (list[i])  Lists may be indexed using brackets. Inside the brackets must be a zero-indexed integer. Getting is unsafe, and for a list<T> returns T, error

Examples:

list<int> a, error E = [1,2,3,4]
printf("%d", a[2])

Set (list[i] = j)  After indexing, an assignment may occur, to set a value of the list. Setting is unsafe, and for a list<T> returns T, error.

Examples:

list<int> a = [1,2,3,4]
a[2], error e = 5
if (!e?) {
    for (int i in a) {
        printf("%d", i)
    }
}
// prints 1254

Iterate (j in list)  Lists may be iterated over in for loops. Each element is returned in order.

Examples:

list<int> a = [1,2,3,4]
for (int i in a) {
    printf("%d", i)
}
// prints 1234
```
Slice (list[i:j])  Lists may be sliced into sub-lists using slicing with brackets. Slicing is **unsafe**, and for a `list<T>` returns `list<T>`, _error_.

```cpp
class list<T> {
public:
    // Slicing
    list<T> slice(int i, int j);

    // Access
    T& operator[](int i);  // Returns a reference to the T at index i
    T* data() const;      // Returns a pointer to the first element

    // Iterators
    iterator begin() const { return _impl->begin(); }
    iterator end() const { return _impl->end(); }
    const_iterator begin() const { return _impl->begin(); }
    const_iterator end() const { return _impl->end(); }

private:
    _impl_type _impl;
}
```

```cpp
list<int> a = [1,2,3,4]
list<int> b, error e = a[2:4]
if (!e?) {
    for (int i in a) {
        printf("%d", i)
    }
}
// prints 234
```

### 9.3 dict

**dict.is_empty()**

```cpp
func is_empty() boolean
```

Returns a boolean value of whether the dictionary on which it is called is empty.

```cpp
dict<string, string> d = {"Dog" : "cat"}
dict<string, string> e = {}

d.is_empty()   // false
e.is_empty()   // true
```

**dict.has_key()**

```cpp
func has_key(T key) boolean
```

Returns a boolean value corresponding to whether the dictionary on which it is called contains argument as a key.

```cpp
dict<string, string> d = {"Dog" : "cat"}

d.has_key("Dog")   // true
d.has_key("Cow")   // false
```

**dict.insert()**

```cpp
func insert(T key, S value)
```
Inserts the arguments as a key, value pair in the dictionary on which it is called.

dict<string, string> d = {"Dog" : "cat"}

d.insert("Cow" : "Pig") // {"Dog" : "cat", "Cow" : "Pig"}

dict.remove()

unsafe func remove(T key)

Removes the value for the key given in the argument from the dictionary on which the function is called.

dict<string, string> d = {"Dog" : "cat", "Cow" : "Pig"}

d.remove("Dog") // {"Cow" : "Pig"}

dict.keys()

func keys() list<T>

Returns a list of all keys in the dictionary on which it is called. The type of the returned list is that of the type of the keys in the dictionary.

dict<string, string> d = {"Dog" : "cat", "Cow" : "Pig"}

d.keys() // ["Dog", "Cow"]

dict.values()

func values() list<S>

Returns a list of all values for the keyset in the dictionary on which it is called. The type of the returned list is that of the type of the values in the dictionary.

dict<string, string> d = {"Dog" : "cat", "Cow" : "Pig"}

d.values() // ["Cat", "Pig"]
Get (dict[k]) Lists may be indexed using brackets. Inside the brackets must be a key in the
dictionary. Getting is unsafe, and for a dict<S,T> returns T, error.

Examples:

```c
const dict<string, int> d = {"a":1, "b":2}
const int v, error e = d["a"]
if (!e?) {  
    printf("%d", v)  
}
// prints 1
```

Set (dict[k] = v) After indexing, an assignment may occur, to set a value of the list.

Examples:

```c
const dict<string, int> d = {"a":1, "b":2}
d["a"] = 5  
printf("%d", d["a"])  
// prints 5
```

Iterate (j in dict) Lists may be iterated over in for loops. Each element is returned in order.

Examples:

```c
const dict<string, int> d = {"a":1, "b":2}
for (int k, v in d) {  
    printf("%s:%s, ", k, v)  
}
// prints a:1, b:2,
```

9.4 error

error.message

string message

Returns the value of the error message on for the error object on which it is called.

```c
const error e = error(message="There was an error with that Request.",  
    code=400,  
    name="RequestError")

const string e.message = e.message  
// "There was an error with that Request."  
```
error.code

int code

Returns the value of the error message on for the error object on which it is called.

error e = error(message="There was an error with that Request.",
    code=400,
    name="RequestError")

e.code  // 400

error.name

string name

Returns the value of the error message on for the error object on which it is called.

error e = error(message="There was an error with that Request.",
    code=400,
    name="RequestError")

e.name  // "RequestError"

10. Program Execution

RAPID programs compile to a Go executable which is a platform specific binary. Statements will
be executed in order of their declaration. If a RAPID program contains an http route, then running
the executable will start a server on localhost:5000 after all statements are executed.

Flags  There are several flags to customize the runtime of the app.

- -D <pg_url> : a string of “:@/”. This is used to connect to the postgres server
- -L <filename> : log output will be appended to the specified file, defaults to server.log
- -P <port> : alters the port the service will run on, defaults to 80
- -H : prints all the options available
- -V : verbosely logs every HTTP request and the return values.
4. Project Plan

Our workflow for the compiler consisted of an initial version with output statements completed by the architect. The intention was that group members would then add their language features to the compiler through the entire pipeline. Parsing properly, semantically checking, and then generating code that compiled and ran. There was test infrastructure in place throughout the entire project to guarantee that none of the added features were broken by subsequent commits.

We worked in an agile workflow of completing a feature through one step of the compiler before asking for feedback from the remainder of the group through the “Pull Request” feature of Github. This worked very well to prevent members from getting too far off track during their implementation.
In our LRM, we set out to create an extremely ambitious language. Throughout the course of our project, we adjusted our goals, removing or modifying language features as we saw fit. The evolution of RAPID can be seen through the examination of several features that existed in one form in our original LRM, and exist in a different form in our current version of RAPID.

4.1 Errors

In our LRM, we sought out to create a system of errors that included what we called unsafe functions. Unsafe functions would be labeled as unsafe, and then returned their return type as well as an Error. Operations like dictionary accesses, list accesses, database operations, and the like would all be unsafe.

Here is an example unsafe function, from our LRM.

```go
unsafe func access(dict<string, int> d, string key) int {
    int val, error error = d[key]
    return val, error
}
```

Implementing Errors as desired was out of scope for us, and so we do not allow unsafe functions. The implications of this have been felt in other language features as well. We wanted to allow users to return `null`, `Error` to throw an error, no matter the return type. To do this, we had to make all of our objects nullable. Although we do not allow unsafe functions, we do allow multiple return types for functions, as well as nullable primitives to facilitate errors.

4.1.2 Database backing

One of the features we hoped to implement in RAPID was the ability to persist instances of classes in a postgres database. Then, using API calls, users could interact with their data over an API written in RAPID. This code would have been primarily written in Go however, and because our team was relatively inexperienced in Go, what resources we could have spent implementing database backing were instead spent on code generation.

4.1.3 Arguments to functions over HTTP

Our LRM originally portrayed three different ways to pass data to a route: as parameters within the URL path for the request, as query string arguments in the request, and as JSON POSTed in the request body. Because of this, we defined our `http` functions to take in URL path params using `param` blocks, query string arguments in the parenthesis of the function args, and finally JSON as the final parameter. The following defines code block from our LRM describes our intention to do this.
http /* id */ ( /* path args */ /* formal args */ /* request body args */) {
    // statements
}

However, because we were unable to facilitate arguments from the query string or the response body, including the path args in the parenthesis of the function is superfluous. Despite this, that is how we have implemented http functions in our current version of RAPID.

### 4.1.4 Summary

In summary, our language has evolved and simplified greatly over the semester. We are proud of RAPID, and it has been exciting to observe how the language has changed over the course of its development.
5. Architecture

5.1 Overview

Flow chart of compiler architecture:

The compilation of Rapid to go is broken down into 5 stages:

1. Scanning
2. Parsing
3. Translating
4. Semantic Checking/Type Checking
5. Code Generation

5.2 Scanning

The scanner is written in ocamllex and takes a .rapid text file and converts it into a stream of tokens. The tokens represent keywords, types, and identifiers in the rapid program.

5.3 Parsing

The parser uses ocamlyacc to convert the tokens it receives from the scanner into and abstract syntax tree (AST). The parser builds the AST as a list of statements, functions, classes and http trees. Statements hold the kind of statement and associated expressions. Functions hold a their name, a list of statements representing the variable declarations for the arguments, a list of statements representing the body and a list of statements. A class declaration has a list of it’s members (Attributes or Functions). Finally an Http is a recursive tree here each no leaf node of the tree is a namespace block or a param block. The leaves of the tree are http functions.
5.4 Translation

Translation gets the AST from the parser in the form of a tuple of AST Statements, AST Functions, AST HTTP trees and AST Classes. Translation takes the AST and translates it to an initial semantic abstract syntax tree (SAST). This SAST has the same structure as the AST. At this step all literals are rewritten to typed expressions.

5.5 Semantic Checking

After translation the SAST is scanned for variable declarations, function declarations, and class declarations. When one of these is found it is added to the symbol table, function table, or class respectively. This pass must be done before semantic checking because functions and class do not have to be declared before being used. Once these tables are built the SAST is walked once more to check types. Expressions are recursively rewritten to typed expressions and then typed checked. In binary operations ints are marked for casting to float if they are found in a binary operator expression expecting a float. Functions declarations are scanned for a return statement of the proper type. Statements are check to make sure their expressions are of the correct type. Class instantiations and function calls are check for the proper number and type of arguments they are passed. Once all this type checking and rewriting is done the SAST is passed to the code generator.

5.6 Code Generation

The code generator takes the SAST from the Semantic Checker and generates go code. Because go has a main function and RAPID does not, we first pull out all the var declarations and write the go declarations before main. This is because these variables are global in our language. Now we recurse through each part of the SAST (statements, functions, classes, and http blocks) and write each part of them to code. Because our language supports null values and go does not we use go pointers to represent all of our types. This means that to generate functioning code statements have to be written in two parts. First, temp variables are created in go and set to value of on part of the expression. Then these temp values are used to evaluated the expression. For example rapid code:

```rapid
int x = 5;
```

Would generate the following go code:

```go
//declarations pulled out
x Int //Int is type we defined as pointer to int.
i Int
//int x = 5;
tmp = 5
x = &tmp
//int i = x;
```
tmp = *x
i = &tmp
6. Test Plan and Scripts

6.1 Representative Programs

gcd_server.rapid

```rapid
func gcd(int p, int q) int {
    while (q != 0) {
        int temp = q;
        q = p % q;
        p = temp;
    }
    return p;
}

namespace gcd {
    param int a {
        param int b {
            http (int a, int b) int {
                int res = gcd(a, b);
                return res;
            }
        }
    }
}
```

```go
package main

import (
    "fmt"
    "github.com/julienschmidt/httprouter"
    "log"
    "net/http"
)

var _ = fmt.Printf
var _ = http.StatusOK
var _ = log.Fatal
var _ = httprouter.CleanPath
```

func HTTPgcdab(w http.ResponseWriter, r *http.Request, XXX httprouter.Params) {
    var res Int

    tmp_8860581127662105769 := XXX.ByName("b")
b := StringToInt(&tmp_8860581127662105769)
    _ = b
    tmp_8489902106938695875 := XXX.ByName("a")
a := StringToInt(&tmp_8489902106938695875)
    _ = a
tmp_3192161587922682755 := *a
tmp_3651816005641979265 := *b
    res = gcd(&tmp_3192161587922682755, &tmp_3651816005641979265)

    tmp_6419075023523112259 := *res
    w.Write([]byte(*IntToString(&tmp_6419075023523112259)))
    return
}

func main() {
    router := httprouter.New()
    router.GET(`/gcd/:a/:b/`, HTTPgcdab)
    log.Fatal(http.ListenAndServe(`:8080`, router))
}

func gcd(p Int, q Int) Int {
    for {
        tmp_7985179176134664640 := *q
        tmp_4968362049263200281 := 0
        tmp_2152367932835595560 := *&tmp_7985179176134664640 != *&tmp_4968362049263200281
        if !(*(&tmp_2152367932835595560)) {
            break
        }
    }
    var temp Int
    tmp_6575234515618809058 := *q
    temp = &tmp_6575234515618809058

    tmp_8175353689487733453 := *p
    tmp_7976305469002585692 := *q
    tmp_1330295997328933460 := *&tmp_8175353689487733453 % *&tmp_7976305469002585692
    q = &tmp_1330295997328933460

    tmp_8211254607639216761 := *temp
```rapid
p = &tmp_8211254607639216761
}

tmp_286320284431439979 := *p
return &tmp_286320284431439979
}

oop.rapid

class User {
    int age;
    string name = "Stephen";
    optional int height;

    instance my {
        func is_old() boolean {
            return (my.age >= 30);
        }
        func make_older() {
            my.age = my.age + 1;
        }
    }
}

User stephen = new User(age=29);
println(stephen.age);
stephen.height = 73;
println(stephen.height);

if (stephen.is_old()) {
    println("Stephen is old");
} else {
    println("Stephen is young");
}
stephen.make_older();
if (stephen.is_old()) {
    println("Stephen is old");
}
```

44
package main

import (
    "fmt"
    "github.com/julienschmidt/httprouter"
    "log"
    "net/http"
)

var _ = fmt.Printf
var _ = http.StatusOK
var _ = log.Fatal
var _ = httprouter.CleanPath

type User struct {
    height Int
    name String
    age Int
}

func main() {
    tmp_7985179176134664640 := 29
    tmp_4968362049263200281 := "Stephen"
    stephen := User{
        age: &tmp_7985179176134664640,
        height: nil,
        name: &tmp_4968362049263200281,
    }

    _ = stephen
    tmp_6575234515618809058 := &stephen
    tmp_2152367932835595560 := (*tmp_6575234515618809058).age

    println(*tmp_2152367932835595560)
    tmp_8175353689487733453 := 73
    tmp_7976305469002585692 := &stephen
    (*tmp_7976305469002585692).height = &tmp_8175353689487733453

    tmp_8211254607639216761 := &stephen
    tmp_1330295997928933460 := (*tmp_8211254607639216761).height
println(*tmp_1330295997928933460)

tmp_3192161587922682755 := &stephen

if *((*tmp_3192161587922682755).User__is_old()) {
    tmp_3651816005641979265 := "Stephen is old"
    println(*&tmp_3651816005641979265)
} else {
    tmp_286320284431439979 := "Stephen is young"
    println(*&tmp_286320284431439979)
}

tmp_6419075023523112259 := &stephen

(*tmp_6419075023523112259).User__make_older()

tmp_8860581127662105769 := &stephen

if *((*tmp_8860581127662105769).User__is_old()) {
    tmp_8489902106938695875 := "Stephen is old"
    println(*&tmp_8489902106938695875)
}

}

func (my *User) User__make_older() {

tmp_2861391897282324475 := &my
    tmp_5447851695057989743 := (*tmp_2861391897282324475).age

    tmp_5134325039177141690 := 1
    tmp_3910187507312597662 := *tmp_5447851695057989743 + *tmp_5134325039177141690
    tmp_628303324353860165 := &my
        (*tmp_628303324353860165).age = &tmp_3910187507312597662
}

func (my *User) User__is_old() Bool {

tmp_3602583590299280641 := &my
    tmp_3452337755216291150 := (*tmp_3602583590299280641).age

    tmp_5008436610057606955 := 30
    tmp_1025476952255935310 := *tmp_3452337755216291150 >= *tmp_5008436610057606955
    return &tmp_1025476952255935310
6.2 Test Suites

The RAPID project utilized a complex test suite including multiple levels of integration testing. Tests in the /parser directory would only run the parser, tests in the /sast directory would invoke the parser and the semantic checker, and tests in the /compiler directory would do full compilation and output checking.

6.3 Why and How

The test suite was used to ensure no regressions occurred in the test suite and also as a means for guiding development. Most group members practiced Test Driven Development (TDD) in which they wrote failing tests and then wrote code in the compiler to fix those failing tests.

The reason we had multiple levels of testing is so that members of the team could submit pull requests which would implement features only partially in the pipeline (implement only the parsing support, for example.)

6.4 Automation

Our test suite ran for every pull request on our project using a tool called Travis CI, which would add the result of the test suite next to the pull request itself on the Github interface. This insured that we would not inadvertently commit broken code. In addition, the full test suite was run whenever the compiler was built using the Make command.

6.5 Testing Roles

Each member of the team was responsible for writing and fully testing their own code through all the pipeline layers.
7. Lessons Learned

7.1 Nate Brennand

Failure to stick to a strict weekly team meeting made it difficult to get consistent work done by the group at large. We made a plan that 1-2 members would write out the first section of the compiler pipeline. This worked well, but it should have been completed at an earlier date to give others more time to ramp up with the project. The late ramp up in work on the project created issues with code generation because we chose a target language that only one group member had experience with. In retrospect, we could (and should) have transitioned to a more common language, Java, that we all had familiarity with.

7.2 Ben Edelstein

As this was the first large team project I have worked on, I probably learned as much from my teammates as I did from the subject matter. Using a continuous integration platform as well as enforcing code reviews seemed excessive at the beginning of the project, but I quickly learned that these actually made it much easier to develop. Writing separate tests for the different stages of the compiler was also very helpful since it made it easy to introduce a new feature on stage at a time.

7.3 Brendon Fish

I would advise other teams that having strict early code review helped make the latter stages of the project easier. And Having a testing framework working from day one was critical to generating bug free code, especially when merges with conflicts were happening frequently. If I could do it again I think we should have created smaller milestones so that starting the project wasn’t so daunting. We set milestones, like functions work, but I think we would have been better off with breaking this up into smaller tasks. This would have helped us start work earlier and allowed for smaller merges and hopefully less conflicts.

7.4 Dan Schlosser

Our LRM was too ambitious. As the language guru for RAPID, it was my job to make sure we were keeping true to our vision of the language, so I felt the conflict between what we wanted to implement and what we could feasibly implement very often. Narrowing our scope would have allowed us set better goals that would be more achievable within the span of one semester.

Throughout working on this project I learned a lot about both OCaml and Go. I appreciate the experience of learning two new languages at once, but in retrospect, choosing Go as our destination language was not ideal. Because of how Go’s primitives work, there were a lot of compromises that we had to make in order to have reasonable output at the end of our project.
Code review and a robust testing frameworks were very important in catching bugs and regressions in the code. Our team did not have or enforce milestones for the project, leading development to be very compressed towards the end of the project. Translating code from RAPID (with nullable primitives) to Go (without nullable primitives) proved to be more difficult than expected, which slowed down code generation.
8. Appendix

parser

{%
    open Ast
    open Datatypes
%

%token SEMI LPAREN RPAREN LBRACE RBRACE COMMA
%token LBRACKET RBRACKET LIST
%token PLUS MINUS TIMES DIVIDE ASSIGN CASTBOOL
%token EQ NEQ LT LEQ GT GEQ AND OR MOD
%token RETURN IF ELSE FOR WHILE FUNC IN
%token CLASS NEW ACCESS OPTIONAL INSTANCE
%token HTTP PARAM NAMESPACE
%token <string> ID TYPE STRING_LIT
%token <int> INT_VAL
%token <float> FLOAT_LIT
%token <bool> BOOL_LIT
%token NULL
%token EOF

%nonassoc NOELSE
%nonassoc ELSE

%right ASSIGN
%left LT GT LEQ GEQ EQ NEQ AND OR
%left PLUS MINUS
%left TIMES DIVIDE MOD
%left ACCESS
%left CASTBOOL

%start program
%type <Ast.program> program

%%%% /* Parser Rules */

primtype:
    | TYPE { string_to_t $1 }
    | LIST LT primtype GT { ListType $3 }

50
/* todo: add arrays, dicts to primtype */

anytype:
  | ID    { string_to_t $1 }
  | primtype  { $1 }

/* Base level expressions of a program:
 * TODO: Classes */
program:
  | /* nothing */ { [], [], [], [] }
  | program stmt {
    let (statements, classes, functions, http_tree) = $1 in
    ($2 :: statements), classes, functions, http_tree }
  | program class_decl {
    let (statements, classes, functions, http_tree) = $1 in
    statements, ($2 :: classes), functions, http_tree }
  | program func_decl {
    let (statements, classes, functions, http_tree) = $1 in
    statements, classes, ($2 :: functions), http_tree }
  | program http_type_block {
    let (statements, classes, functions, http_tree) = $1 in
    statements, classes, functions, ($2 :: http_tree) }

/* TODO: allow user defined types */
datatype_list:
  | datatype_list COMMA primtype { $3 :: $1 }
  | primtype  { [$1] }
  | /* nothing */ { [] }

return_type:
  /* TODO: allow user defined types */
  | datatype_list { List.rev $1 }

/*var declarations can now be done inline*/
func_decl:
  // func w/ return types
  | FUNC ID LPAREN arguments RPAREN return_type LBRACE fstmt_list RBRACE
  | {
    fname = $2;
    args = $4;
    return = $6;
    body = List.rev $8
  }
/* TODO: unsafe functions */

func_decl_list:
  | /* nothing */ { [] }  
  | func_decl_list func_decl { $2 :: $1 }

arguments:
  | /* nothing */ { [] }  
  | formal_list { List.rev $1 }

formal_list:
  /* TODO: allow user defined types */
  | primtype ID { [($1, $2, None)] }  
  | primtype ID ASSIGN lit { [($1, $2, Some($4))] }  
  | formal_list COMMA primtype ID { ($3, $4, None) :: $1 }  
  | formal_list COMMA primtype ID ASSIGN lit { ($3, $4, Some($6)) :: $1 }  

/* a tuple here of (primtype, ID, optional expr) expr is the optional assign */
var_decl:
  | primtype ID { ($1 , $2, None) }  
  | primtype ID ASSIGN expr { ($1, $2, Some($4)) }

user_def_decl:
  | ID ID { ($1, $2, None) }  
  | ID ID ASSIGN expr { ($1, $2, Some($4)) }

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

ret_expr_list:
  | expr {[$1]}  
  | ret_expr_list COMMA expr {$3 :: $1}  
  | { [] }

func_stmt:
  | RETURN ret_expr_list SEMI { Return( List.rev $2) }  
  | stmt { FStmt($1) }

id_list:
  | id_list COMMA primtype ID { VDecl($3, $4, None) :: $1 }
RAPID

| id_list COMMA ID { ID($3) :: $1 } |
| ID {[[ID($1)]]} |
| primtype ID {{VDecl($1, $2, None)}} |

call: |
| ID LPAREN expression_list_opt RPAREN { (None, $1, $3) } |
| expr ACCESS ID LPAREN expression_list_opt RPAREN { (Some($1), $3, $5) } |

call: |
| fcall {FuncCall([], $1)} |
| LPAREN id_list RPAREN ASSIGN fcall { FuncCall(List.rev $2, $5) } |

lhs: |
| ID { LhsId($1) } |
| expr ACCESS ID { LhsAcc($1, $3) } |

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

stmt: |
| var_decl SEMI { VarDecl $1 } |
| user_def_decl SEMI { UserDefDecl $1 } |
| func_call SEMI { $1 } |
| lhs ASSIGN expr SEMI { Assign($1, $3) } |
| http_type_block { HttpTree $1 } |
| FOR LPAREN anytype ID IN expr RPAREN LBRACE stmt_list RBRACE { For($3, $4, $6, List.rev $9) } |
| IF LPAREN expr RPAREN LBRACE stmt_list RBRACE %prec NOELSE { If($3, List.rev $6, []) } |
| IF LPAREN expr RPAREN LBRACE stmt_list RBRACE ELSE LBRACE stmt_list RBRACE { If($3, List.rev $6, List.rev $10) } |
| WHILE LPAREN expr RPAREN LBRACE stmt_list RBRACE { While($3, List.rev $6) } |

typed_param_list: |
| /* nothing */ { [] } |
| TYPE ID { [(Datatypes.string_to_t $1, $2, None)] } |
| typed_param_list COMMA TYPE ID { (Datatypes.string_to_t $3, $4, None) :: $1 } |

http_tree_list: |
| {} |
| http_type_block { [$1] } |
http_type_block:
  | PARAM primtype ID LBRACE http_tree_list RBRACE
  |   { Param($2, $3, $5) }
  | NAMESPACE ID LBRACE http_tree_list RBRACE
  |   { Namespace($2, $4) }
  | HTTP ID LPAREN typed_param_list RPAREN primtype LBRACE fstmt_list RBRACE
  |   { Endpoint($2, $4, $6, $8) }
  | HTTP LPAREN typed_param_list RPAREN primtype LBRACE fstmt_list RBRACE
  |   { Endpoint("", $3, $5, $7) }

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

lit:
  | INT_VAL { IntLit $1 }
  | BOOL_LIT { BoolLit $1 }
  | STRING_LIT { StringLit $1 }
  | FLOAT_LIT { FloatLit $1 }
  | NULL { Nullxpr }

expr:
  | lit { $1 }
  | ID { Id $1 }
  | expr PLUS expr { Binop($1, Add, $3) }
  | expr MINUS expr { Binop($1, Sub, $3) }
  | expr TIMES expr { Binop($1, Mult, $3) }
  | expr DIVIDE expr { Binop($1, Div, $3) }
  | expr EQ expr { Binop($1, Equal, $3) }
  | expr NEQ expr { Binop($1, Neq, $3) }
  | expr LT expr { Binop($1, Less, $3) }
  | expr LEQ expr { Binop($1, Leq, $3) }
  | expr GT expr { Binop($1, Greater, $3) }
  | expr GEQ expr { Binop($1, Geq, $3) }
  | expr AND expr { Binop($1, And, $3) }
  | expr OR expr { Binop($1, Or, $3) }
  | expr MOD expr { Binop($1, Mod, $3) }
  | expr CASTBOOL { CastBool $1 }
  | primtype LPAREN expr RPAREN { Cast($1, $3) }
  | fcall { Call $1 }
  | LPAREN expr RPAREN { $2 }
  | NEW ID LPAREN actuals_list_opt RPAREN { UserDefInst($2, $4)}
expr ACCESS ID { Access($1, $3) }
| LBRACKET expression_list_opt RBRACKET { ListLit $2 }
| expr LBRACKET expr RBRACKET { ListAccess($1, $3) }

instance_block:
| INSTANCE ID LBRACE func_decl_list RBRACE { InstanceBlock($2, $4) }

instance_block_opt:
| /* nothing */ { None }
| instance_block { Some($1) }

expression_list:
| expression_list_internal { List.rev $1 }

expression_list_opt:
| /* nothing */ { [] }
| expression_list { $1 }

expression_list_internal:
| expr { [$1] }
| expression_list_internal COMMA expr { $3 :: $1 }

actuals_list:
| /* nothing */ { [] }
| actuals_list_internal { List.rev $1 }

actuals_list_opt:
| /* nothing */ { [] }
| actuals_list { $1 }

actuals_list_internal:
/* TODO: allow user defined types */
| ID ASSIGN expr { [Actual($1, $3)] }
| actuals_list COMMA ID ASSIGN expr { Actual($3, $5) :: $1 }

attr_decl:
| primtype ID { NonOption($1, $2, None) }
/* we limit the default values to literals */
| primtype ID ASSIGN lit { NonOption($1, $2, Some($4)) }
| OPTIONAL primtype ID { Optional($2, $3) }

member_list:
| /* nothing */ { [] }
| member_list attr_decl SEMI { Attr($2):: $1 }
| member_list func_decl { ClassFunc($2):: $1 }

class_decl:
| CLASS ID LBRACE member_list instance_block_opt member_list RBRACE
| { $2, List.rev ($6 @ $4), $5 }

%%

scanner
{
  open Parser;;
  let string_to_bool = function
  | "false" -> false
  | "true" -> true
  | _ -> false
}

let decdigit = ['0'-'9']

let floating = ',.' decdigit+
| decdigit+ ',.' decdigit*
| decdigit+ (',.' decdigit*)? 'e' '‐'? decdigit+
| ',.' decdigit+ 'e' '‐'? decdigit+

rule token = parse
| [', ' \t \r \n] { token lexbuf } (* Whitespace *)
| "/*" { comment lexbuf } | "//" { comment2 lexbuf } (* Comments *)

(* blocks *)
| "(" { LPAREN }
| ")" { RPAREN }
| "{" { LBRACE }
(* operators *)
| '+'   { PLUS } |
| '-'   { MINUS } |
| '*'   { TIMES } |
| '/'   { DIVIDE } |
| "and" { AND } |
| "or"  { OR } |
| "⅓"  { MOD } |

(* comparisons *)
| "=="  { EQ } |
| "!="  { NEQ } |
| "<"   { LT } |
| "<="  { LEQ } |
| ">"   { GT } |
| ">="  { GEQ } |

(* control structures *)
| "if"   { IF } |
| "else" { ELSE } |
| "for"  { FOR } |
| "in"   { IN } |
| "while" { WHILE } |

(* primatives *)
| '='    { ASSIGN } |

(* Casting operators *)
| '?'    { CASTBOOL } |
| "true" | "false" as bool_val { BOOL_LIT( string_to_bool bool_val ) } |
| "boolean" | "int" | "float" | "string" as prim { TYPE prim } |
Rapid

| "dict" { DICT } |
| "list" { LIST } |

(* functions *)
| "func" { FUNC } |
| "return" { RETURN } |

(* classes *)
| "class" { CLASS } |
| "." { ACCESS } |
| "new" { NEW } |
| "optional" { OPTIONAL } |
| "instance" { INSTANCE } |

(* http related *)
| "http" { HTTP } |
| "param" { PARAM } |
| "namespace" { NAMESPACE } |

(* switch statements *)
(*
| "switch" { SWITCH } |
| "case" { CASE } |
| "default" { DEFAULT } |
| "fallthrough" { FALLTHROUGH } |
*)

(* literals *)
| \[0-9]+ as lxm { INT_VAL(int_of_string lxm) } |
| "", ([""\""]* as str )"," { STRING_LIT str } |
| floating as lit { FLOAT_LIT(float_of_string lit) } |
| "null" { NULL } |

(* ID's *)
| .'a'-.'z' .'A'-.'Z'\[.'a'-.'z' .'A'-.'Z' .'0'-.'9' \'_\]* as lxm { ID lxm } |

| eof { EOF } |
| _ as char { raise (Failure("illegal character " ^ Char.escaped char)) } |

and comment = parse

58
let rec string_of_t = function
  | Int -> "int"
  | Bool -> "bool"
  | String -> "string"
  | Float -> "float"
  | ListType(s) -> sprintf "list<%s>" (string_of_t s)
  | UserDef(s) -> sprintf "(USER_DEF %s)" s
  | Void -> "void"
  | Multi -> "multi return"
  | InfiniteArgs -> "InfiniteArgs"
  | AnyList -> "AnyList"

let bin_op_s = function
  | Add -> "+
  | Sub -> "-"
  | Mult -> "*"
  | Div -> "/"
  | Equal -> "=="
  | Neq -> "!="
  | Less -> "<"
  | Leq -> "<="
  | Greater -> ">
  | Geq -> ">="
  | Qmark -> "?"
(* Converts expressions to strings *)
let rec expr_s = function
| IntLit l -> sprintf "(Int Literal (%d))" l
| Id s -> sprintf "(Id %s)" s
| Binop(e1, o, e2) -> sprintf "(Binop (%s) %s (%s))"
  (expr_s e1)
  (bin_op_s o)
  (expr_s e2)
| Call f -> fcall_s f
| BoolLit b -> sprintf "(Bool literal %b)" b
| StringLit s -> sprintf "(String literal %s)" s
| FloatLit f -> sprintf "(Float literal %f)" f
| CastBool e -> sprintf "(Cast (%s) to boolean)" (expr_s e)
| Cast(t, i) -> sprintf "(Cast (%s) to %s)" (expr_s i) (string_of_t t)
| ListLit l -> sprintf "(List literal [%s])"
  (String.concat "", " " (List.map expr_s l))
| UserDefInst(id, actls) -> sprintf "(INSTANTIATE new UserDef %s(%s))"
  id
  ("\n" ^ (String.concat "", \n" " (List.map actual_s actls)))
| Access(e, mem) -> sprintf "(ACCESS %s.%s)" (expr_s e) mem
| ListAccess(xpr_l, xpr_r) -> sprintf "(List access %s at index %s)"
  (expr_s xpr_l)
  (expr_s xpr_r)
| Noexpr -> "( NOEXPR )"
| Nullxpr -> "(Null)"
and fcall_s = function
| (None, f, es) -> sprintf "(Call (%s) with (%s))"
  f
  (concat "", " " (List.map (fun e -> sprintf "(%s)" (expr_s e)) es))
| (Some(xpr), f, es) -> sprintf "(Call %s.(%s) with (%s))"
  (expr_s xpr)
  f
  (concat "", " " (List.map (fun e -> sprintf "(%s)" (expr_s e)) es))
and actual_s = function
| Actual(id, e) -> sprintf "(ACTUAL: %s=%s)" id (expr_s e)
and lhs_s = function
| LhsId(id) -> id
| LhsAcc(xpr, id) -> sprintf "(%s.%s)" (expr_s xpr) id
let string_of_vdecl (t, nm, e) = sprintf "%s %s %s"
    (string_of_t t)
    nm
    (match e with
     | Some xpr -> sprintf "= %s" (expr_s xpr)
     | None     -> "(Not assigned)"
    )

let string_of_user_def_decl (cls, nm, e) = sprintf "%s %s %s"
    cls
    nm
    (match e with
     | Some xpr -> sprintf "= %s" (expr_s xpr)
     | None     -> "(Not assigned)"
    )

(* Prettyprint statements *)
let func_lvalue_s = function
    | ID(i) -> i
    | VDecl(t, id, x) -> string_of_vdecl (t,id,x)

let rec stmt_s = function
    | Assign(lhs, e) -> sprintf "(Assign %s (%s))"
      (lhs_s lhs)
      (expr_s e)
    | If(e, s1, [] ) -> sprintf "(If (%s) -> (%s))"
      (expr_s e)
      (concat "\n" (List.map stmt_s s1))
    | If(e, s1, s2 ) -> sprintf "(If (%s) -> (%s))(Else -> (%s))"
      (expr_s e)
      (concat "\n" (List.map stmt_s s1))
      (concat "\n" (List.map stmt_s s2))
    | For(t, id, xpr, stmt_l) -> sprintf "(For (%s %s in %s)\n{(%s)})"
      (string_of_t t)
      id
      (expr_s xpr)
      (String.concat "\n" (List.map stmt_s stmt_l))
    | While(e, s) -> sprintf "(While (%s)\n\{(%s)})"
      (expr_s e)
      (concat "\n" (List.map stmt_s s))
    | VarDecl vd -> sprintf "(VarDecl (%s))"
      (string_of_vdecl vd)
    | UserDefDecl ud -> sprintf "(UserDefDecl (%s))"
      (string_of_user_def_decl ud)
    | FuncCall(s,f) -> sprintf "(FuncCall(%s = %s))"
      (concat ", " (List.map func_lvalue_s s))
(fcall_s f)

and stmt_s = function
 | Return e -> sprintf "(Return (%s))\n"
   (concat ", " (List.map expr_s e))
 | FStmt s -> stmt_s s

let rec http_tree_s = function
 | Param(t, id, tree) -> sprintf "(param (%s %s)\n\t%s)\n"
   (string_of_t t)
   id
   (indent_block (String.concat "\n" (List.map http_tree_s tree)))
 | Namespace(id, tree) -> sprintf "(namespace (%s)\n\t%s)\n"
   id
   (indent_block (String.concat "\n" (List.map http_tree_s tree)))
 | Endpoint(id, args, ret_t, body) -> sprintf "(HTTP %s(%s)%s\n\t%s)\n"
   id
   (String.concat "," (List.map string_of_vdecl args))
   (string_of_t ret_t)
   (indent_block(String.concat "\n" (List.map stmt_s body)))

let func_decl_s f = sprintf "{\nfname = "%s"
args = [%s]
	body = [%s]\n}" f.fname
   (concat ", " (List.map string_of_vdecl f.args))
   (concat ",\n" (List.map stmt_s f.body))

let attr_s = function
 | NonOption(t, id, Some(xpr)) -> sprintf "(ATTR: %s of %s = %s)"
   id
   (string_of_t t)
   (expr_s xpr)
 | NonOption(t, id, None) -> sprintf "(ATTR: %s of %s)"
   id
   (string_of_t t)
 | Optional(t, id) -> sprintf "(ATTR: OPTIONAL %s of %s)"
   id
   (string_of_t t)

let member_s = function
 | Attr a -> attr_s a
 | ClassFunc f -> func_decl_s f

let instance_block_s = function
Some(InstanceBlock(id, fns)) -> sprintf "(INSTANCE %s {\n	%s}"
  id
  (concat "\n\t" (List.map func_decl_s fns))
| None -> "(NO INSTANCE BLOCK)"

let class_s (name, members, instance_block) =
  sprintf "(CLASS %s:\n%s\n%s)"
  name
  (instance_block_s instance_block)
  (concat "\n" (List.map (fun a -> "\t" ^ a) (List.map member_s members)))

let program_s (stmts, classes, funcs, http_tree) = sprintf
  "classes:{\n%s\n}
statements:{\n%s\n}
functions:
%sHTTP tree:
%s"
  (concat "\n" (List.rev (List.map class_s classes)))
  (concat "\n" (List.rev (List.map stmt_s stmts)))
  (concat "\n" (List.rev (List.map func_decl_s funcs)))
  (concat "\n" (List.rev (List.map http_tree_s http_tree)))

datatypes

(* AST type for datatypes
 * Primative types and a placeholder for userdefined types *)
type var_type =
  | Int
  | String
  | Bool
  | Float
  | UserDef of string
  | ListType of var_type
  | Void (*Used for funcions with no args or no rets*)
  | Multi(*Used for função with multiple rets*)
  | InfiniteArgs
  | AnyList

(* Converts a string to a datatype *)
let string_to_t = function
  | "boolean" -> Bool
  | "int" -> Int
  | "float" -> Float
| "string" -> String
| c -> UserDef(c)

```

generate

open Format
open Sast
open Sast_printer
open Datatypes

exception UnsupportedSemanticExpressionType of string
exception UnsupportedSemanticStatementType
exception UnsupportedStringExprType
exception UnsupportedIntExprType
exception UnsupportedFloatExprType
exception UnsupportedOutputType
exception UnsupportedSExprType of string
exception UnsupportedBoolExprType
exception UnsupportedListExprType
exception UnsupportedDeclType of string
exception UnsupportedDatatypeErr
exception InvalidClassInstantiation
exception StringExpressionsRequired
exception InvalidUserDefExpr

module StringMap = Map.Make(String)

let need_dereference_funcs = let sm = StringMap.empty in
  let sm = StringMap.add "append" true sm in
  let sm = StringMap.add "len" true sm in
  let sm = StringMap.add "println" true sm in
  let sm = StringMap.add "printf" true sm in
  sm

let rand_var_gen _ = "tmp_" ^ Int64.to_string (Random.int64 Int64.max_int)

let rec go_type_from_type = function
  | Int -> "Int"
  | Float -> "Float"
  | Bool -> "Bool"
  | String -> "String"
  | ListType(t) ->
    sprintf "%sList" (go_type_from_type t)
```
RAPID

| UserDef(name) -> name |
| _ -> raise UnsupportedDatatypeErr |

and go_tmp_type_from_type = function
| ListType(t) ->
|   sprintf "[]%s" (go_type_from_type t) |
| other_t -> go_type_from_type other_t |

let go_type_from_sexpr = function
| SExprInt _ -> go_type_from_type Int |
| SExprFloat _ -> go_type_from_type Float |
| SExprBool _ -> go_type_from_type Bool |
| SExprString _ -> go_type_from_type String |
| x -> raise(UnsupportedSemanticExpressionType(Sast_printer.sexpr_s x)) |

(* must return a direct reference to a string *)
let get_string_literal_from_sexpr = function
| SExprString(SStringExprLit s) -> sprintf "(""s")" s |
| SExprString(SStringVar id) -> sprintf "%s" id |
| _ -> raise StringExpressionsRequired |

let op_to_code o = Ast_printer.bin_op_s o

(* top level function for generating sexprs * returns a tuple of (setup code, reference code) *)
let rec sexpr_to_code = function
| SExprInt i -> int_expr_to_code i |
| SExprString s -> string_expr_to_code s |
| SExprFloat f -> float_expr_to_code f |
| SExprBool b -> bool_expr_to_code b |
| SCallTyped(t, c) -> func_expr_to_code c |
| SExprList l -> list_expr_to_code l |
| SExprUserDef u -> user_def_expr_to_code u |
| NullExpr -> "", "nil" |
| s -> raise(UnsupportedSExprType(Sast_printer.sexpr_s s)) |

(* returns a reference to a string *)
and string_expr_to_code = function
| SStringExprLit s ->
|   let tmp_var = rand_var_gen () in |
|   sprintf "%s := \"%s\"" tmp_var s, |
|   sprintf "&%s" tmp_var |
| SStringVar id ->
|   let tmp_var = rand_var_gen () in |
|   sprintf "%s := *%s" tmp_var id, |
|   sprintf "%s" tmp_var |
| SStringAcc(ud_xpr, attr) -> ud_access_to_code ud_xpr attr |
| SStringCast c -> cast_to_code String c |
| SStringNull -> "", "nil" |
| _ -> raise UnsupportedStringExprType |

(* returns a reference to an integer *)
and int_expr_to_code = function |
| SIntExprLit i -> |
| let tmp_var = rand_var_gen () in |
| sprintf "%s := %d" tmp_var i, |
| sprintf "&%s" tmp_var |
| SIntVar id -> |
| let tmp_var = rand_var_gen () in |
| sprintf "%s := *%s" tmp_var id, |
| sprintf "&%s" tmp_var |
| SIntBinOp(lhs, o, rhs) -> bin_op_to_code lhs o rhs |
| SIntAcc(ud_xpr, attr) -> ud_access_to_code ud_xpr attr |
| SIntNull -> "", "nil" |
| SIntCast c -> cast_to_code Int c |
| _ -> raise UnsupportedIntExprType |

(* returns a reference to a float *)
and float_expr_to_code = function |
| SFloatExprLit f -> |
| let tmp_var = rand_var_gen () in |
| sprintf "%s := %f" tmp_var f, |
| sprintf "&%s" tmp_var |
| SFloatVar id -> |
| let tmp_var = rand_var_gen () in |
| sprintf "%s := *%s" tmp_var id, |
| sprintf "&%s" tmp_var |
| SFloatBinOp(rhs, o, lhs) -> bin_op_to_code rhs o lhs |
| SFloatNull -> "", "nil" |
| SFloatAcc(ud_xpr, attr) -> ud_access_to_code ud_xpr attr |
| SFloatCast c -> cast_to_code Float c |
| _ -> raise UnsupportedFloatExprType |

(* returns a reference to a boolean *)
and bool_expr_to_code = function |
| SBoolExprLit b -> |
| let tmp_var = rand_var_gen () in |
| sprintf "%s := %b" tmp_var b, |
| sprintf "&%s" tmp_var |
| SBoolVar id -> |
| let tmp_var = rand_var_gen () in |
| sprintf "%s := *%s" tmp_var id, |
| sprintf "&%s" tmp_var |
| SBoolAcc(ud_xpr, attr) -> ud_access_to_code ud_xpr attr |
| SBboolCast c -> cast_to_code Bool c |
| SBboolBinOp(lhs, o, rhs) -> bin_op_to_code lhs o rhs |
| SBboolNull -> "", "nil" |
| _ -> raise UnsupportedBoolExprType |

(* takes the destination type and the expression and creates the cast statement *)

and cast_to_code t xpr =
   let src_type = go_type_from_sexpr xpr in
   let dest_type = go_type_from_type t in
   let setup, ref = sexpr_to_code xpr in
   let cast = sprintf "%sTo%s(%s)" src_type dest_type ref in
   setup, cast

and func_expr_to_code = function
   | SFCall(None, id, arg_xrps) ->
      let (tmps, refs) = (list_of_sexpr_to_code "" arg_xrps) in
      let s = if StringMap.mem id (need_dereference_funcs) then "*
      else "" in
      let refs = List.map (fun str -> s ^ str ) refs in
      let tmps, call = if StringMap.mem id (need_dereference_funcs) then
         let tmp = rand_var_gen () in
         let dots = if id = "append" then "..." else "" in
         tmps @ [(sprintf "\n%s := %s(%s%s)" tmp id
           (String.concat "\n" refs) dots )] , "+" - tmp
      else
         tmps ,sprintf "%s(%s)" id (String.concat "\n" refs) in
         (String.concat "\n" tmps), call
   | SFCall(Some(xpr), id, arg_xrps) ->
      let setup, ref = sexpr_to_code xpr in
      let (tmps, refs) = (list_of_sexpr_to_code "" arg_xrps) in
      let call = sprintf "%s.%s(%s)" ref id (String.concat "\n" refs) in
      (String.concat "\n" tmps ^ \n setup^ \n ), call

(* Helper function that turns a list of expressions into code *)

and list_of_sexpr_to_code deref_string xpr_l =
   let trans = List.map sexpr_to_code xpr_l in
   let setups = List.map (fun (s, _) -> s) trans in
   let refs = List.map (fun (_, r) -> deref_string^r) trans in
   setups, refs

and bin_op_to_code lhs o rhs =
   let setup1, lefts = sexpr_to_code lhs in
   let setup2, rights = sexpr_to_code rhs in
   let os = op_to_code o in
   let tmp_var = (rand_var_gen ()) in
   let new_tmps = sprintf "%s := *%s %s *%s" tmp_var lefts (op_to_code o) rights in
   (setup1 ^ \n setup2 ^ \n new_tmps) , (sprintf "%s" tmp_var)
and list_expr_to_code = function
  | SListExprLit(Some(t), l) ->
    let tmp_var = rand_var_gen () in
    let setups, refs = list_of_sexpr_to_code "\n" l in
    let setups = String.concat "\n" setups in
    let list_setup = sprintf "\n%s := %s{%s}
" tmp_var
      (go_tmp_type_from_type t)
      (String.concat " , " refs) in
    setups ^ list_setup,
    sprintf "&%s" tmp_var
  | SListVar(t, id) ->
    let tmp_var = rand_var_gen () in
    sprintf "%s := %s" tmp_var id, tmp_var
  | SListAccess(xpr_l, xpr_r) ->
    let setup_l, ref_l = sexpr_to_code xpr_l in
    let setup_r, ref_r = sexpr_to_code xpr_r in
    sprintf "%s
%s" setup_l setup_r,
    sprintf "(*%s)[*%s]" ref_l ref_r
  | _ -> raise UnsupportedListExprType
and user_def_expr_to_code = function
  | SUserDefInst(UserDef(class_id), act_list) ->
    let expand (attr, xpr) =
      let setup, ref = sexpr_to_code xpr in
      setup, sprintf "%s: %s," attr ref in
    let trans = List.map expand act_list in
    (String.concat "\n" (List.map fst trans),
     sprintf "%s{\n%s
}%s\n" class_id (String.concat "\n" (List.map snd trans)))
  | SUserDefVar(_, id) ->
    let tmp_var = rand_var_gen () in
    sprintf "%s := &%s" tmp_var id, sprintf "(*%s)" tmp_var
  | SUserDefNull _ ->
    "", "nil"
  | _ -> raise(InvalidUserDefExpr)
and ud_access_to_code ud_expr attr_id =
  let tmp_var = rand_var_gen () in
  let setup, ref = user_def_expr_to_code ud_expr in
  sprintf "%s\n%s := %s.%s\n" setup tmp_var ref attr_id,
  tmp_var
let sassign_to_code = function
  | (SLhsId id, xpr) ->
let setup, ref = sexpr_to_code xpr in
    sprintf "\%s\n\%s = \%s\n" setup id ref
| (SLhsAcc(e, mem), xpr) ->
    let setup, ref = sexpr_to_code xpr in
    let lhs_setup, lhs_ref = sexpr_to_code e in
    sprintf "\%s\n\%s.%s = \%s\n" (setup ^ "\n" ^ lhs_setup) lhs_ref mem ref
| a -> raise(UnsupportedSemanticExpressionType(
    sprintf "Assignment expression not yet supported -> \%s"
    (svar_assign_s a)))

let sreturn_to_code xprs =
    let (tmps, refs) = list_of_sexpr_to_code "" xprs in
    sprintf "\%s\nreturn \%s" (String.concat "\n" tmps) (String.concat ", " refs)

let decls_from_lv = function
    | SFuncDecl(t, (id, _)) -> sprintf "var \%s \%s" id (go_type_from_type t)
    | _ -> ""

let get_ids = function
    | SFuncDecl(_, (id, _)) -> id
    | SFuncTypedId (_, id) -> id

let lv_to_code lv =
    String.concat ", " (List.map get_ids lv)

let sfunccall_to_code lv c =
    let lhs = lv_to_code lv in
    match c with
    | SFCall(None, id, arg_xprs) ->
        let (tmps, refs) = list_of_sexpr_to_code "" arg_xprs in
        let tmps = String.concat "\n" tmps in
        let s, id = if StringMap.mem id (need_dereference_funcs) then ",", "" ^ id else ",", id in
        let refs = List.map (fun str -> s ^ str) refs in
        let refs = if s = "Println" then
            String.sub (List.hd refs) 1
            ((String.length (List.hd refs)) - 1) :: (List.tl refs)
        else refs in
        let refs = String.concat "," refs in
        if lhs = "" then sprintf "\%s\n\%s\( \%s \)" tmps id refs
        else sprintf "\%s\n\%s\( \%s \)\" tmps lhs_ref mem ref
    | SFCall(Some(xpr), id, arg_xprs) ->
        let setup, call = func_expr_to_code c in
        if lhs = "" then sprintf "\%s\n\%s" setup call

else sprintf "%s
%s := %s" setups id attribs

let class_instantiate_to_code class_id (id, inst_xpr) =
  let setups, attrs = user_def_expr_to_code inst_xpr in
  sprintf "%s
%s := %s
_ = %s" setups id attrs id

let rec grab_decls = function
  | SDecl(t, (id, _)) :: tl ->
    sprintf "var %s %s" id (go_type_from_type t) :: grab_decls tl
  | SFuncCall(lv, _) :: tl ->
    (String.concat "\n" (List.map decls_from_lv lv)) :: grab_decls tl
  | _ :: tl -> grab_decls tl
  | [] -> []

let rec control_code b expr stmts =
  let tmps, exprs = sexpr_to_code expr in
  let body = String.concat "\n" (List.map sast_to_code stmts) in
  let decls = String.concat "\n" (grab_decls stmts) in
  match b with
  | IF -> sprintf "%s
if *(%s){%s
%s}" tmps exprs decls body
  | WHILE -> sprintf "for{%s
if !(*(%s)){
break
}%s
%s}
" tmps exprs decls body

and sast_to_code = function
  | SDecl(_, (id, xpr)) -> sassign_to_code (SLhsId id, xpr)
  | SAssign (lhs, xpr) -> sassign_to_code (lhs, xpr)
  | SReturn xprs -> sreturn_to_code xprs
  | SFuncCall (lv, c) -> sfunccall_to_code lv c
  | SUUserDefDecl(class_id, (id, SExprUserDef(xpr))) ->
    class_instantiate_to_code class_id (id, xpr)
  | SUUserDefDecl(class_id, (id, NullExpr)) ->
    sprintf "var %s %s
_ = %s" id class_id id
  | SIf(expr, stmts) -> (control_code IF expr stmts) ^ "\n"
  | SWhile (expr, stmts) -> control_code WHILE expr stmts
  | SIfElse(expr, stmts, estmts) ->
    sprintf "%selse{%s
%s}
" (control_code IF expr stmts)
    (String.concat "\n" (grab_decls estmts))
    (String.concat "\n" (List.map sast_to_code estmts))
  | SFor(t, id, xpr, stmts) -> sfor_to_code t id xpr stmts
  | _ -> raise(UnsupportedSemanticStatementType)
and sfor_to_code t id xpr stmts =
  let body = (List.map sast_to_code stmts) in
  let s_tmp, s_ref = sexpr_to_code xpr in
  sprintf "%s
for _, %s := range *%s {
%s
%s
}"
    s_tmp
    id
    s_ref
    (String.concat "\n" (grab_decls stmts))
    (String.concat "\n" body)

let arg_to_code = function
  | SDecl(t, (id, _)) -> sprintf "%s %s"
    id
    (go_type_from_type t)

let defaults_to_code = function
  | SDecl(_, (id, xpr)) -> if xpr = NullExpr then ""
    else sprintf "if %s == nil \n%s\n"
    id
    (sassign_to_code (SLhsId id, xpr))

let func_to_code f =
  let (id, selfref_opt, args, rets, body) = f in
  sprintf "func %s%s( %s ) (%s){
%s
%s
%s
}"
    (match selfref_opt with
      | Some(SelfRef(class_id, id)) -> sprintf "(%s *%s) " id class_id
      | None -> "")
    id
    (String.concat "," (List.map arg_to_code args))
    (String.concat "," (List.map go_type_from_type rets))
    (String.concat "\n" (List.map defaults_to_code args))
    (String.concat "\n" (grab_decls body))
    (String.concat "\n" (List.map sast_to_code body))

let class_def_to_code (class_id, attr_list) =
  let attr_to_code_attr = function
    | SNonOption(t, id, _) | SOptional(t, id) ->
      sprintf "%s %s" id (go_type_from_type t) in
  let attrs = List.map attr_to_code_attr attr_list in
  sprintf "type %s struct{%n%s\n}" class_id (String.concat "\n" attrs)

(* rewrites returns as writes to the connection *)
let http_fstmt_to_code = function
| SReturn(xpr :: _) ->
| let setup, ref = cast_to_code String xpr in
| sprintf "%s\n\nWrite([]byte(*%s))\nreturn"
| setup
| ref
| s -> sast_to_code s

let strip_route_re = Str.regexp "[:/]"
let strip_path s = Str.global_replace strip_route_re "" s

let generate_route_registrations routes =
| let routes = List.map (fun (r, _, _, _) ->
| let fname = strip_path r in
| sprintf "router.GET("%s", HTTP%s)"
| fname
| if regs = ""
| then ""
| else "router := httprouter.New()\n\nlog.Fatal(http.ListenAndServe("":8080", router))\n"

let endpoint_to_code (path, args, ret_type, body) =
| let decl_vars = grab_decls body in
| (* grabs the parameters from the request and instantiates the variables *)
| let grab_param (t, name, default) =
| let setup = if default = NullExpr then
| let tmp = rand_var_gen () in
| sprintf "%s := XXX.ByName("%s")\n%s := StringTo%s(&%s)"
| tmp name name (go_type_from_type t) tmp
| else
| let xpr_setup, ref = sexpr_to_code default in
| sprintf "%s\n%s := &%s" xpr_setup name ref
| let func_start = sprintf "func HTTP%s(w http.ResponseWriter, r *http.Request, "
| (strip_path path) in
| let func_end = sprintf "%Xhttprouter.Params){%n\n\n}n"
| ((String.concat "\n" decl_vars) ~ "\n\n" ~
| (String.concat "\n" (List.map grab_param args)))
| (String.concat "\n" (List.map http_fstmt_to_code body)) in
| func_start ~ func_end

let skeleton decls http_funcs classes main fns router =
| let packages = ("fmt", "fmt.Printf") ::
| ("net/http", "http.StatusOK") ::
| ("log", "log.Fatal") ::
let build_prog sast =  
  let (stmts, classes, funcs, route_list) = sast in
  let decls = String.concat "\n" (grab_decls stmts) in
  let code_lines = List.map sast_to_code stmts in
  let stmt_code = String.concat "\n" code_lines in
  let func_code = String.concat "\n\n" (List.map func_to_code funcs) in
  let class_struct_defs = String.concat "\n\n" (List.map class_def_to_code classes) in
  let http_funcs = String.concat "\n" (List.map endpoint_to_code route_list) in
  let router_reg = generate_route_registrations route_list in
  skeleton decls http_funcs class_struct_defs stmt_code func_code router_reg

let translate ast =  
  let sast = Semantic_check.sast_from_ast ast in
  let code = Generate.build_prog sast in
  code

let _ =  
  let action = if Array.length Sys.argv > 1 then
    List.assoc Sys.argv.(1) [
      ("-a", Ast);
      ("-s", Sast);
    ]
("-c", Compile);
]
else Compile in (* Assume compiling *)
let lexbuf = Lexing.from_channel stdin in
let ast = try
Parser.program Scanner.token lexbuf
with except ->
  let curr = lexbuf.Lexing.lex_curr_p in
  let line = curr.Lexing.pos_lnum in
  let col = curr.Lexing.pos_cnum in
  let tok = Lexing.lexeme lexbuf in
  raise (SyntaxError (line, col, tok))
in
let lexbuf = Lexing.from_channel stdin in
match action with
| Ast -> print_string (Ast_printer.program_s ast)
| Sast -> let sast = Semantic_check.sast_from_ast ast in
          print_string (Sast_printer.string_of_sast sast)
| Compile -> let code = translate ast in
            print_string code

sast_helper

open Sast
open Datatypes

exception UnsupportedSexprTypeClassification
exception UnsupportedAssignExpr
exception UnsupportedDeclStmt
exception UnsupportedSattr
exception UnsupportedSactual
exception ExistingSymbolErr
exception ExistingClassErr
exception ExistingActualErr
exception ExistingAttributeErr of string
exception MissingSymbolTablesErr
exception VariableNotDefinedErr of string
exception ClassNotDefinedErr of string
exception AttributeNotDefinedErr of string
exception MissingActualErr of string
exception ExistingFuncErr
exception ExistingRouteErr
exception BadFunctionId
exception CannotFindFunctionIDForArgTypes
exception CannotFindFunctionIDForReturnType
exception CannotFindFunctionIDForReturnTypeList

(* Maps a function to a expr option if it is defined, otherwise return NullExpr *)
let expr_option_map func = function
| Some o -> func o
| _ -> NullExpr

(* returns the the possible types a binary op can have*)
let get_op_types = function
| Ast.Greater | Ast.Leq | Ast.Geq ->
[Int; Float]
| Ast.Equal | Ast.Neq ->
[Bool; Int; Float; String]
| Ast.And | Ast.Or ->
[(Bool)]
| Ast.Mod ->
[(Int)]

module StringMap = Map.Make(String)

let empty_function_table = StringMap.empty

(*add a (id -> typelist*typelist into the function table*)
let add_func ft id arg_ts ret_ts =
  if StringMap.mem id ft
    then raise ExistingFuncErr
  else
    let v = (arg_ts, ret_ts) in
    StringMap.add id v ft

let default_ft ft =
  let ft = add_func ft "append"
  [(ListType(AnyList), NullExpr); (ListType(AnyList), NullExpr)]
  [ListType(AnyList)] in
  let ft = add_func ft "printf"
  [(String, NullExpr); (InfiniteArgs, NullExpr)]
  [] in
  let ft = add_func ft "len"
  [ListType(AnyList), NullExpr] [Int] in
  let ft = add_func ft "println"
  [(InfiniteArgs, NullExpr)]
let empty_symbol_table = StringMap.empty
let symbol_table_list = StringMap.empty :: []

let empty_actuals_table = StringMap.empty
let empty_attribute_table = StringMap.empty
let class_table = StringMap.empty

(* inserts a (symbol -> type) into the top level scope *)
let add_sym t id = function
  | current_scope :: scope_list ->
    if StringMap.mem id current_scope
    then raise ExistingSymbolErr
    else (StringMap.add id t current_scope) :: scope_list
  | _ -> raise MissingSymbolTablesErr

(* retrieves a type in the top level scope that it is found *)
let rec get_type id = function
  | current_scope :: scope_list ->
    if StringMap.mem id current_scope
    then StringMap.find id current_scope
    else get_type id scope_list
  | _ -> raise(VariableNotDefinedErr(Format.sprintf "%s is not defined" id))

let get_return_type id ft =
  if StringMap.mem id ft
  then let (_, ret_t) = StringMap.find id ft in
    match ret_t with
    | [] -> Void
    | t :: [] -> t
    | t_list -> Multi
    else raise CannotFindFunctionIDForReturnType

let get_return_type_list id ft =
  if StringMap.mem id ft
  then let (_, retl) = StringMap.find id ft in retl
  else raise CannotFindFunctionIDForReturnTypeList

let get_arg_types id ft =
  if StringMap.mem id ft
  then let (_, tlist) = StringMap.find id ft in tlist
  else raise CannotFindFunctionIDForReturnTypeList
then let (arg_ts, _) = StringMap.find id ft in arg_ts
else raise CannotFindFunctionIDForArgTypes

(* adds a new empty symbol table for use in the new scope *)
let new_scope sym_tbl = empty_symbol_table :: sym_tbl

(* pops the last added scope removing all scoped variables. *)
let pop_scope = function
  | current_scope :: scope_list -> scope_list
  | [] -> raise MissingSymbolTablesErr

(* adds a new entry to the class table *)
let new_class class_id attr_tbl class_tbl =
  if StringMap.mem class_id class_tbl
  then raise ExistingClassErr
  else (StringMap.add class_id attr_tbl class_tbl)

(* adds a new attribute on the class called class_id *)
let rec add_attrs attr_tbl = function
  | SNonOption(t, attr_id, Some(default)) :: tl ->
    add_attrs (insert_attr attr_id attr_tbl (t, false, default)) tl
  | SNonOption(t, attr_id, None) :: tl ->
    add_attrs (insert_attr attr_id attr_tbl (t, true, NullExpr)) tl
  | SOptional(t, attr_id) :: tl ->
    add_attrs (insert_attr attr_id attr_tbl (t, false, NullExpr)) tl
  | [] -> attr_tbl
  | _ -> raise UnsupportedSattr
and insert_attr attr_id attr_tbl triple =
  if StringMap.mem attr_id attr_tbl
    then raise(ExistingAttributeErr(attr_id))
  else StringMap.add attr_id triple attr_tbl

(* Get the table of attributes for a specific class *)
let get_attr_table class_id class_tbl =
  if StringMap.mem class_id class_tbl
    then StringMap.find class_id class_tbl
  else raise (ClassNotDefinedErr
    (Format.sprintf "%s is not a class" class_id))

(* Get attribute, as (type, required, default) *)
let get_attr class_id class_tbl id =
let attr_tbl = get_attr_table class_id class_tbl in
  if StringMap.mem id attr_tbl
    then StringMap.find id attr_tbl
    else raise(AttributeNotDefinedErr(Format.sprintf
      "%s is not an attribute on the class %s"
      id class_id))

(* gets the type of the attribute called id on the class called class_id *)
let get_attr_type class_id class_tbl id =
  let (t, _, _) = get_attr class_id class_tbl id in
  t

(* Add the actuals into actl_tbl from an actuals list, verifying uniqueness *)
let rec add_actls actl_tbl = function
  | (key, xpr) :: tl ->
    if StringMap.mem key actl_tbl
      then raise ExistingActualErr
      else add_actls (StringMap.add key xpr actl_tbl) tl
  | [] -> actl_tbl
  | _   -> raise UnsupportedSactual

(* Get the type of an actual given *)
let get_actl_type actl_tbl key =
  if StringMap.pem key actl_tbl
    then StringMap.find key actl_tbl
    else raise(MissingActualErr(
      Format.sprintf "Argument %s is missing" key))

(* returns the type of a typed sexpr *)
let rec sexpr_to_t expected_t = function
  | SExprInt _ -> Int
  | SExprFloat _ -> Float
  | SExprBool _ -> Bool
  | SExprString _ -> String
  | SExprUserDef(SUserDefInst(s, _) | SUserDefVar(s, _) | SUserDefNull(s)) -> s
  | SExprAccess _ -> expected_t
  | NullExpr -> expected_t
  | SCallTyped(t, _) -> t
  | UntypedNullExpr -> expected_t
  | SExprList l -> (match l with
    | SListExprLit(Some(t), _) -> t
    | SListExprLit(None, _) -> expected_t
    | SListVar(t, _) -> ListType t
let empty_route_table = StringMap.empty

let add_route route rt =
  if StringMap.mem route rt then
    raise ExistingRouteErr
  else
    StringMap.add route "" rt

let rec sexpr_s = function
  | SExprInt i -> int_expr_s i
  | SExprString s -> string_expr_s s
  | SExprFloat s -> float_expr_s s
  | SExprBool b -> bool_expr_s b
  | SExprUserDef u -> user_def_expr_s u
  | SCallTyped (t, c) -> scall_typed_s (t, c)
  | SExprAccess (e, m) -> raise(UntypedAccess("Accesses must be rewritten with type information"))
  | SExprList l -> list_expr l
  | SId _ -> raise(UntypedVariableReference("Variable references must be rewritten with type information"))
  | NullExpr -> "(NULL EXPR)"
  | UntypedNullExpr -> "(HARD NULL EXPR)"
| _ -> raise UnsupportedSexpr

and string_expr_s = function
| SStringExprLit s -> sprintf "(String Lit: %s)" s
| SStringVar id -> sprintf "(String Var: %s)" id
| SStringCast xpr -> sprintf "String Cast (%s)" (sexpr_s xpr)
| SStringAcc(v, mem) -> sprintf "(String Access: %s.%s)"
  (user_def_expr_s v) mem
| SStringNull -> "(String NULL)"

and int_expr_s = function
| SIntExprLit i -> sprintf "(Int Lit: %d)" i
| SIntVar id -> sprintf "(Int Var: %s)" id
| SIntCast e -> sprintf "(Cast (%s) to int)" (sexpr_s e)
| SIntBinOp(lhs, o, rhs) -> sprintf "(%s %s %s)"
  (sexpr_s lhs) (Ast_printer.bin_op_s o) (sexpr_s rhs)
| SIntAcc(v, mem) -> sprintf "(Int Access: %s.%s)"
  (user_def_expr_s v) mem
| SIntNull -> "(Int NULL)"

and float_expr_s = function
| SFloatExprLit f -> sprintf "(Lit %f)" f
| SFloatVar id -> sprintf "(Float Var %s)" id
| SFloatAcc(v, mem) -> sprintf "(Float Access: %s.%s)"
  (user_def_expr_s v) mem
| SFloatCast e -> sprintf "(Cast (%s) to float)" (sexpr_s e)
| SFloatBinOp(lhs, o, rhs) -> sprintf "(%s %s %s)"
  (sexpr_s lhs) (Ast_printer.bin_op_s o) (sexpr_s rhs)
| SFloatNull -> "(Float NULL)"

and bool_expr_s = function
| SBoolExprLit b -> sprintf "(Bool lit: %b)" b
| SBoolVar id -> sprintf "(Bool Var: %s)" id
| SBoolCast e -> sprintf "(Cast (%s) to boolean)" (sexpr_s e)
| SBoolBinOp(lhs, o, rhs) -> sprintf "(%s %s %s)"
  (sexpr_s lhs) (Ast_printer.bin_op_s o) (sexpr_s rhs)
| SBoolAcc(v, mem) -> sprintf "(Bool Access: %s.%s)"
  (user_def_expr_s v) mem
| SBoolNull -> "(Bool NULL)"

and scall_typed_s = function
| (t, SFCall(None, id, args)) -> sprintf
  "(Call %s) args = %s returns = %s"
  id
  (String.concat ", " (List.map sexpr_s args))
  (Ast_printer.string_of_t t t)
| (t, SFCall(Some(xpr), id, args)) -> sprintf
  "(Call %s.%s) args = %s returns = %s"
  (sexpr_s xpr)
  id
(String.concat ",
" (List.map sexpr_s args))
(Ast_printer.string_of_t t)

and sactual_s = function
  | (k,v) -> sprintf "(ACTUAL: %s=%s)" k (sexpr_s v)

and user_def_expr_s = function
  | SUUserDefInst(UserDef cls, sactls) ->
    sprintf "(INSTANTIATE new UserDef %s(
	%s\n" cls
    (String.concat ",
	" (List.map sactual_s sactls))
  | SUUserDefVar(UserDef cls, id) -> sprintf "(UserDef %s %s)" cls id
  | SUUserDefAcc(UserDef cls, var, mem) ->
    sprintf "(UserDef %s Access: %s.%s)"
    cls (user_def_expr_s var) mem
  | SUUserDefNull(UserDef cls) ->
    sprintf "(UserDef %s NULL)" cls

and sattr_s = function
  | SNonOption(t, id, Some(xpr)) ->
    sprintf \n\t(ATTR %s of %s = %s)
    id
    (Ast_printer.string_of_t t)
    (sexpr_s xpr)
  | SNonOption(t, id, None) ->
    sprintf \n\t(ATTR %s of %s NO_DEFAULT)
    id
    (Ast_printer.string_of_t t)
  | SOptional(t, id) ->
    sprintf \n\t(ATTR OPTIONAL %s of %s)
    id
    (Ast_printer.string_of_t t)
  | _ -> raise UnsupportedSattr

and list_expr = function
  | SListExprLit(Some(t), l) ->
    sprintf "(Lit <%s> %s )"
    (Ast_printer.string_of_t t)
    (String.concat ", " (List.map sexpr_s l))
  | SListExprLit(None, l) ->
    sprintf "(Lit <None> %s )"
    (String.concat ", " (List.map sexpr_s l))
  | SListAccess(xpr_l, xpr_r) ->
    sprintf "(List access %s at index %s)"
    (sexpr_s xpr_l)
    (sexpr_s xpr_r)
  | SListVar(t, id) ->
    sprintf "(List Var <%s> %s)"
    id
    (Ast_printer.string_of_t t)
  | SListNull -> "(List NULL)"

let slhs_s = function
  | SLhsId id -> id
  | SLhsAcc (xpr, mem) ->
    sprintf "%s.%s" (sexpr_s xpr) mem

let svar_assign_s (lhs, xpr) =
  sprintf "(Assign (%s) to %s)" (slhs_s lhs) (sexpr_s xpr)
let svar_decl_s t (id, xpr) =  
    sprintf "(Declare %s (%s) to %s)" id (Ast_printer.string_of_t t) (sexpr_s xpr)

let suser_def_decl_s cls (id, xpr) =  
    sprintf "(Declare USERDEF %s (%s) to %s)" id cls (sexpr_s xpr)

let lv_s = function  
| SFuncDecl(t, (id, _)) -> sprintf "%s %s" (Ast_printer.string_of_t t) id  
| SFuncTypedId(_, id) -> id  
| _ -> raise UnsupportedSOutput

let sfcall_s = function  
| SFCall(None, id, args) -> sprintf "((FCall %s) args = %s)" id (String.concat "", " (List.map sexpr_s args))  
| SFCall(Some(xpr), id, args) -> sprintf "((FCall %s.%s) args = %s)" (sexpr_s xpr) id (String.concat "", " (List.map sexpr_s args))

let rec semantic_stmt_s = function  
| SAssign (lhs, xpr) -> svar_assign_s (lhs, xpr) ^ "\n"  
| SDecl(t, vd) -> svar_decl_s t vd ^ "\n"  
| SUserDefDecl(cls, vd) -> suser_def_decl_s cls vd ^ "\n"  
| SReturn s -> sprintf("Return(%s)\n") (String.concat "", " (List.map sexpr_s s))  
| SFuncCall(lv, sfc) -> sprintf "Assign(%s) to %s" (String.concat "", " (List.map lv_s lv)) (sfcall_s sfc)  
| SFor(t, string_id, xpr, stmts) ->  
    sprintf "(For %s %s in %s {\n%s\n})" (Ast_printer.string_of_t t) string_id (sexpr_s xpr) (String.concat "\n" (List.map semantic_stmt_s stmts))  
| SWhile(expr, stmts) -> sprintf "(While(%s){\n%s\n})\n" (sexpr_s expr) (String.concat "\n" (List.map semantic_stmt_s stmts))  
| _ -> "Unsupported statement"

let semantic_func_s f =  
    let (id, selfref_opt, args, rets, body) = f in
let args_strings = (List.map semantic_stmt_s args) in
let ret_strings = (List.map Ast_printer.string_of_t rets) in
let body_strings = (List.map semantic_stmt_s body) in
sprintf "(func %s%s(%s) %s
%{
 %s 
})"
  (match selfref_opt with
   | Some(SelfRef(class_id, id)) -> sprintf "(%s %s) " class_id id
   | None -> "")
  id
  (String.concat "," args_strings)
  (String.concat ", " ret_strings)
  (String.concat "\n" body_strings)

let semantic_class_s (classname, sattrs) =
  let actl_strings = String.concat "" (List.map sattr_s sattrs) in
  sprintf "(Class %s %s)" classname actl_strings

let route_s (path, args, ret_type, body) =
  let rec arg_to_s = ( function
    | (t, id, xpr) :: tl -> ((sprintf "%s %s = %s"
                                (Ast_printer.string_of_t t)
                                id
                                (sexpr_s xpr)) :: arg_to_s tl)
    | _ -> []) in
  sprintf "(HTTP: %s (%s) (%s)

{
%s
})
" path
  (String.concat ", " (arg_to_s args))
  (Ast_printer.string_of_t ret_type)
  (String.concat ", "
   (List.map semantic_stmt_s body))

let string_of_sast sast =
  let (stmts, classes, funcs, routes) = sast in
  let stmt_strings = List.map semantic_stmt_s stmts in
  let class_strings = List.map semantic_class_s classes in
  let func_strings = List.map semantic_func_s funcs in
  let route_strings = List.map route_s routes in
  String.concat "" (stmt_strings @ class_strings @ func_strings @ route_strings)

  semantic_check

open Sast
open Sast_helper
open Sast_printer
open Datatypes

83
exception RepeatDeclarationErr of string
exception InvalidTypeDeclarationErr of string
exception UncaughtCompareErr of string
exception UnsupportedStatementTypeErr of string
exception UndeclaredVarErr of string
exception InvalidTypeReassignErr of string
exception InvalidTypeErr of string
exception MissingRequiredArgument of string
exception UnsupportedExpressionType
exception UnsupportedSexpr
exception UnsupportedDatatypeErr
exception StringDatatypeRequiredErr
exception InvalidTypeMemberAccess
exception InvalidArgErr
exception InvalidArgOrder
exception InvalidReturnTypeErr
exception NoReturnErr
exception ReturnTypeMismatchErr
exception SfuncIdNotReWritten
exception TooFewArgsErr
exception TooManyArgsErr
exception InvalidBinaryOp
exception AccessOnNonUserDef
exception AmbiguousContextErr of string
exception NotPrintTypeErr
exception ClassAttrInClassErr
exception UserDefinedTypNeeded
exception UnusedParamArgument

type allowed_types = AllTypes | NumberTypes

(* Takes a type and a typed sexpr and confirms it is the proper type *)
let check_t_sexpr expected_t xpr =
  let found_t = sexpr_to_t expected_t xpr in
  if found_t = expected_t
    then ()
  else raise(InvalidTypeErr(Format.sprintf "Expected %s expression, found %s"
     (Ast_printer.string_of_t expected_t)
     (Ast_printer.string_of_t found_t)))

let is_not_default x = (x = NullExpr)
let check_print_arg = function
  | SExprUserDef(_) | SExprList(SListVar(_,_)) -> raise NotPrintTypeErr
  | _ -> ()

(*takes a list of args as SDecl(t, xpr) and list of params as sexprs
 Checks the type and if there is some default args not entered, fill them with
 NullExpr*)

let rec check_arg_types lt = function
  | ((ListType AnyList, _) :: tl ), (param :: pl) ->
    let t = sexpr_to_t lt param in
    let r = match param with
      | SExprList _ ->
        if lt = Void or t = lt
          then param :: check_arg_types lt (tl, pl)
          else raise InvalidArgErr
      | _ -> raise InvalidArgErr in
    r
  | ((InfiniteArgs, _) :: tl ), (param :: pl) ->
    let () = check_print_arg param in
    param :: check_arg_types lt ([((InfiniteArgs, NullExpr)], pl)
  | ((InfiniteArgs, _ ) :: tl), ([]) -> []
  (*| ((InfiniteArgs, _) :: tl ), (param :: pl) -> *)
  | (((t, _)::tl),(param :: pl)) ->
    let () = check_t_sexpr t param in
    param :: check_arg_types lt (tl, pl)
  | (((_, xpr) :: tl), ([])) -> if (is_not_default xpr)
    then raise TooFewArgsErr
    (*This is the case where the user didn't enter some optional args*)
    else NullExpr :: check_arg_types lt (tl, [])
  | ([], (param :: pl)) -> raise TooManyArgsErr
  | ([],[]) -> []

let get_cast_side = function
  | (Int, Float) -> Left
  | (Float, Int) -> Right
  | (l, r) when l = r -> Neither
  | _ -> raise BinOpTypeMismatchErr

(* Check that for a given attribute, it either exists in the actuals, or
* it if it isn't place a default value if one exists (or raise if no default
* value exists). *)

let check_attr sactuals_table = function
  | (name, (t, true, NullExpr)) ->
    if StringMap.mem name sactuals_table
then let expr = StringMap.find name sactuals_table in
let () = check_t_sexpr t expr in
(name, expr)
else raise(MissingRequiredArgument
(Format.sprintf "Argument %s is missing" name))
| (name, (t, false, xpr)) ->
  if StringMap.mem name sactuals_table
  then let expr = StringMap.find name sactuals_table in
  let () = check_t_sexpr t expr in
  (name, expr)
  else (name, xpr)

(* Check that all of the actuals in the instantiation are valid. *)
let check_user_def_inst ct t sactls =
  (* build a table from the explicit actuals *)
  let sactuals_table = add_actls empty_actuals_table sactls in
  let attr_table = get_attr_table t ct in
  let checked_sactuals = List.map
    (check_attr sactuals_table)
    (StringMap.bindings attr_table) in
  SUserDefInst (UserDef t, checked_sactuals)

(* Takes a symbol table and sexpr and rewrites variable references to be typed *)
let rec rewrite_sexpr st ct ft ?t = function
  | SId id -> (match get_type id st with
    | Int -> SExprInt(SIntVar id)
    | String -> SExprString(SStringVar id)
    | Float -> SExprFloat(SFloatVar id)
    | Bool -> SExprBool(SBoolVar id)
    | UserDef cls -> SExprUserDef(SUserDefVar ((UserDef cls), id))
    | ListType(ty) -> SExprList(SListVar(ty, id))
    | _ -> raise UnsupportedDatatypeErr)
  | SExprBool(SBoolCast e) ->
    SExprBool(SBoolCast(rewrite_cast st ct ft e AllTypes))
  | SExprInt(SIntCast e) ->
    SExprInt(SIntCast(rewrite_cast st ct ft e NumberTypes))
  | SExprFloat(SFloatCast e) ->
    SExprFloat(SFloatCast(rewrite_cast st ct ft e NumberTypes))
  | SExprString(SStringCast e) ->
    SExprString(SStringCast(rewrite_cast st ct ft e AllTypes))
  | SCall(c) -> (match c with
    | SFCall(Some(xpr), fn_id, xprs) ->
      let xpr = rewrite_sexpr st ct ft xpr in
      let class_id = (match sexpr_to_t Void xpr with

let id = (class_id ^ "__" ^ fn_id) in
let xprs = (List.map (rewrite_sexpr st ct ft) xprs) in
let xprs = check_arg_types Void ((get_arg_types id ft), xprs) in
let rt = get_return_type id ft in
let rt = match rt with
  | ListType(AnyList) -> sexpr_to_t Void (List.hd xprs)
  | _ -> rt in
SCallTyped(rt, SFCall(Some(xpr), id, xprs))
| SFCall(None, id, xprs) ->
  let xprs = (List.map (rewrite_sexpr st ct ft) xprs) in
  let xprs = check_arg_types Void ((get_arg_types id ft), xprs) in
  let rt = get_return_type id ft in
  let rt = match rt with
    | ListType(AnyList) -> sexpr_to_t Void (List.hd xprs)
    | _ -> rt in
SCallTyped(rt, SFCall(None, id, xprs))
| SExprList(SListExprLit(None, untyped_l)) ->
  rewrite_sexpr_list st ct ft untyped_l t
| SExprList(SListAccess(xpr_l, xpr_r)) ->
  let rewritten_l = rewrite_sexpr st ct ft xpr_l in
  let rewritten_r = rewrite_sexpr st ct ft xpr_r in
  (* Verify that index is an int *)
  let () = check_t_sexpr Int rewritten_r in
  SExprList(SListAccess(rewritten_l, rewritten_r))
| SBinop (lhs, o, rhs) ->
  let lhs = rewrite_sexpr st ct ft lhs in
  let rhs = rewrite_sexpr st ct ft rhs in
  let lt = sexpr_to_t Void lhs in
  let rt = sexpr_to_t Void rhs in
  let possible_ts = get_op_types o in
  if (List.mem rt possible_ts) && (List.mem lt possible_ts) then
    match o with
      let lhs, rhs = binop_cast_floats lhs rhs (get_cast_side (lt, rt)) in
      SExprBool(SBoolBinOp(lhs, o, rhs))(*bool exprs allow casting *)
    | Ast.And | Ast.Or ->
      if(rt = lt && rt = Bool)
      then SExprBool(SBoolBinOp(lhs, o, rhs))
      else raise BinOpTypeMismatchErr
    | _ ->
      if(rt = lt) then match lt with
      | Int -> SExprInt(SIntBinOp(lhs, o, rhs))
      | Float -> SExprFloat(SFloatBinOp(lhs, o, rhs))
      else
        let lhs, rhs = binop_cast_floats lhs rhs (get_cast_side (lt, rt)) in
        SExprFloat(SFloatBinOp(lhs, o, rhs))
  else
    raise BinOpTypeMismatchErr

87
else raise InvalidBinaryOp

| SExprUserDef udf -> (match udf with
| | SUserDefInst(UserDef t, saactls) ->
| | let rewritten_sactls = List.map (rewrite_sactl st ct ft) saactls in
| | let expr = check_user_def_inst ct t rewritten_sactls in
| | SExprUserDef(expr)
| | _ -> SExprUserDef udf)
| | SExprAccess(xpr, mem) ->
| | let rewritten_sexpr = rewrite_sexpr st ct ft xpr in
| | let cls = match rewritten_sexpr with
| | | SExprUserDef(
| | | | SUserDefInst(UserDef s, _)
| | | | SUserDefVar(UserDef s, _)
| | | | SUserDefNull(UserDef s)) -> s
| | | _ -> raise InvalidTypeMemberAccess in
| | let class_var_expr = (match rewritten_sexpr with
| | | SExprUserDef(xpr) -> xpr
| | | _ -> raise UserDefinedTypNeeded) in
| | let t = get_attr_type cls ct mem in
| | (match t with
| | | Bool -> SExprBool(SBoolAcc(class_var_expr, mem))
| | | Int -> SExprInt(SIntAcc(class_var_expr, mem))
| | | Float -> SExprFloat(SFloatAcc(class_var_expr, mem))
| | | String -> SExprString(SStringAcc(class_var_expr, mem))
| | | _ -> raise ClassAttrInClassErr)
| | _ -> xpr

and rewrite_sactl st ct ft = function
| | (name, xpr) -> (name, rewrite_sexpr st ct ft xpr)

and rewrite_cast st ct ft xpr t_opt =
| let xpr = rewrite_sexpr st ct ft xpr in
| let t = sexpr_to_t Void xpr in
| (match t with
| | (AllTypes, (Int | Float | String | Bool)) -> xpr
| | (NumberTypes, (Int | Float)) -> xpr
| | _ -> raise(InvalidTypeErr(Format.sprintf
| "Cast cannot use %s expression" (Ast_printer.string_of_t t)))
| | xpr -> xpr

and rewrite_sexpr_list st ct ft untyped_l = function
| | Some(ListType(child_type) as ty) ->
let typed_sexprs = List.map (rewrite_sexpr st ct ft ~t:child_type ) untyped_l in
let _ = List.map (fun child ->
  let actual_type = sexpr_to_t child_type child in
  if actual_type <> child_type
  then
    raise(InvalidTypeErr(Format.sprintf
      "Actual type %s did not match declared child type %s"
      (Ast_printer.string_of_t actual_type)
      (Ast_printer.string_of_t child_type)))
  ) typed_sexprs in
SExprList(SListExprLit(Some(ty), typed_sexprs))
| None -> raise(AmbiguousContextErr("Type must be passed in for lists"))

(* typechecks a sexpr *)
let rewrite_sexpr_to_t st ct ft xpr t =
  let typed_xpr = rewrite_sexpr st ct ft xpr in
  let () = check_t_sexpr t typed_xpr in
  typed_xpr

(* checks that an assignment has the proper types *)
let check_var_assign_use st id xpr =
  let var_t = (get_type id st) in
  let () = check_t_sexpr var_t xpr in
  st

(*Check that the return statement has expressions with the right types*)
let rec check_return_types = function
  | (xpr :: s),(t :: types) -> let () = (check_t_sexpr t xpr) in
    check_return_types (s, types)
  (*To few vals returned*)
  | ([],(t::types)) -> raise InvalidReturnTypeErr
  (*to many vals returned*)
  | (xpr :: s),([]) -> raise InvalidReturnTypeErr
  | [],[] -> ()

(*Scan all stmts in a function for returns then check the types*)
let rec check_returns r = function
  | SReturn(s) :: tl-> let _ = check_return_types (s, r) in
    SReturn(s) :: check_returns r tl
  | hd :: tl -> hd :: check_returns r tl
  | [] -> []

(*takes an sfunc_lval list * var_type list, gotten the return type list
in the function table this checks if the left hand side vars or var
decls are the same types as the return types. *)

let rec check_lv_types = function
    | (SFuncTypedId(t, _) :: tl), (expected_t :: types) -> if t = expected_t
        then check_lv_types (tl, types)
        else raise ReturnTypeMismatchErr
    | (SFuncDecl(t, _) :: tl), (expected_t :: types) -> if t = expected_t
        then check_lv_types (tl, types)
        else raise ReturnTypeMismatchErr
    | (SFuncId(i) :: tl), _ -> raise SfuncIdNotReWritten
    | [], (t::types) -> raise ReturnTypeMismatchErr
    | (s :: tl), [] -> raise ReturnTypeMismatchErr
    | [],[] -> ()

(*rewrite so ids have a type*)

let rewrite_lv st = function
    | SFuncId(i) -> SFuncTypedId((get_type i st), i)
    | SFuncDecl(t, sv) -> SFuncDecl(t, sv)

(*adds any var decls on the left hand side of a function statement to the symbol table.*)

let rec scope_lv st = function
    | SFuncDecl(t, (id, _)) :: tl -> let st = (add_sym t id st) in
        scope_lv st tl
    | SFuncId(i) :: tl -> scope_lv st tl
    | SFuncTypedId(_, _) :: tl -> scope_lv st tl
    | [] -> st

(*
Adds all var decls in a stmt list to the scope and returns the new scope
This does not do any type checking, and ignores the optional expression
*)

let rec add_to_scope st = function
    | SDecl(t, (id, xpr)) :: tl ->
        let st = add_sym t id st in
        add_to_scope st tl
    | SFuncCall (lv, _) :: tl ->
        let st = scope_lv st lv in
        add_to_scope st tl
    | _ :: tl -> add_to_scope st tl
    | [] -> st

(* Processes an unsafe SAST and returns a type checked SAST *)

let rec var_analysis st ct ft = function
    | SDecl(t, (id, xpr)) :: tl ->
        let expr = rewrite_sexpr st ct ft xpr ~t:t in
        let st = add_sym t id st in
let () = check_t_sexpr t expr in
  SDecl(t, (id, expr)) :: var_analysis st ct ft tl
| SAssign(SLhsId(id), xpr) :: tl ->
  let expr = rewrite_sexpr st ct ft xpr in
  let st = check_var_assign_use st id expr in
  SAssign(SLhsId id, expr) :: (var_analysis st ct ft tl)
| SAssign(SLhsAcc(x, mem), xpr) :: tl ->
  let x = rewrite_sexpr st ct ft x in
  let xpr = rewrite_sexpr st ct ft xpr in
  let lhs_class_id = (match sexpr_to_t Void x with
    | UserDef u -> u
    | _ -> raise AccessOnNonUserDef) in
  let lhs_t = get_attr_type lhs_class_id ct mem in
  let () = check_t_sexpr lhs_t xpr in
  SAssign(SLhsAcc(x, mem), xpr) :: (var_analysis st ct ft tl)

(* Return stmts are xpr lists, translate all the expressions here *)
| SReturn(s) :: tl -> let xprs = List.map (rewrite_sexpr st ct ft) s in
  SReturn(xprs) :: (var_analysis st ct ft tl)
| SFuncCall (lv, SFCall(xpr, id, xprs)) :: tl ->
  let xpr, class_id__ = (
    match xpr with
    | Some(e) ->
      let e = rewrite_sexpr st ct ft e in
      let c_id__ = (match sexpr_to_t Void e with
        | UserDef u -> u ^ "__"
        | _ -> raise AccessOnNonUserDef)
    | None -> None, ""
  ) in
  let id = (class_id__ ^ id) in
  let lv = (List.map (rewrite_lv st) lv) in
  let check_lv ft id = function
    | [] -> () (*ignoring return types so foo(); is always a valid stmt*)
    | _ -> check_lv_types (lv, (get_return_type_list id ft)) in
  let () = check_lv ft id lv in
  let xprs = (List.map (rewrite_sexpr st ct ft) xprs) in
  let xprs = check_arg_types Void ((get_arg_types id ft), xprs) in
  let st = scope_lv st lv in
  SFuncCall(lv, SFCall(xpr, id, xprs)) :: (var_analysis st ct ft tl)
| SUserDefDecl(cls, (id, xpr)) :: tl ->
  let checked_expr = rewrite_sexpr st ct ft xpr in
  let t = UserDef cls in
let st = add_sym t id st in
let () = check_t_sexpr t checked_expr in
SUserDefDecl(cls, (id, checked_expr)) :: var_analysis st ct ft tl
| SIf(xpr, stmts) :: tl ->
  let expr = rewrite_sexpr st ct ft xpr in
  let () = check_t_sexpr Bool expr in
  let new_scope = new_scope st in
  let stmts = var_analysis new_scope ct ft stmts in
  SIf(expr, stmts) :: (var_analysis st ct ft tl)
| SIfElse(xpr, stmts, estmts) :: tl ->
  let expr = rewrite_sexpr st ct ft xpr in
  let () = check_t_sexpr Bool expr in
  let if_scope = new_scope st in
  let stmts = var_analysis if_scope ct ft stmts in
  let else_scope = new_scope st in
  let estmts = var_analysis else_scope ct ft estmts in
  SIfElse(expr, stmts, estmts) :: (var_analysis st ct ft tl)
| SWhile(xpr, stmts) :: tl ->
  let expr = rewrite_sexpr st ct ft xpr in
  let () = check_t_sexpr Bool expr in
  let stmts = var_analysis (new_scope st) ct ft stmts in
  SWhile(expr, stmts) :: var_analysis st ct ft tl
| SFor (t, string_id, xpr, stmts) :: tl ->
  let scoped_st = new_scope st in
  let scoped_st = add_sym t string_id scoped_st in
  let xpr = rewrite_sexpr scoped_st ct ft xpr ~t:(ListType t) in
  let for_body = var_analysis scoped_st ct ft stmts in
  SFor(t, string_id, xpr, for_body) :: (var_analysis st ct ft tl)
| [] -> []

(*Called when we see an arg with default val, all the rest must have defaults*)
let rec check_default_args = function
  | SDecl(t, (id, xpr)) :: tl -> if is_not_default xpr
    then raise InvalidArgOrder
    else
      SDecl(t, (id, xpr)) :: check_default_args tl
  | _ :: _ -> raise InvalidArgErr
  | [] -> []

(*Checks to make sure args with default vals come at the end fo the arg list*)
let rec check_arg_order = function
  | SDecl(t, (id, xpr)) :: tl -> if is_not_default xpr
    then SDecl(t, (id, xpr)) :: check_arg_order tl
    else
  | [] -> []
let check_for_return body = let last_stmt = List.hd (List.rev body) in match last_stmt with | SReturn(s) -> () | _ -> raise NoReturnErr

let rec check_funcs st ct ft = function | (fname, class_opt, args, rets, body) :: tl -> let _ = check_arg_order args in let scoped_st = new_scope st in let targs = var_analysis scoped_st ct ft args in (*args are added to function scope*) let scoped_st = add_to_scope scoped_st args in (* Add the reference to self to the symbol table, if there is one *) let scoped_st = (match class_opt with | Some(SelfRef (class_id, id)) -> add_sym (UserDef class_id) id scoped_st | None -> scoped_st) in (*typecheck the body and rewrite vars to have type*) let tbody = var_analysis scoped_st ct ft body in (*check the return type matches the return statement*) let _ = check_returns rets tbody in (*if no return types then don’t worry, else find a return stmt*) if rets = [] then (fname, class_opt, targs, rets, tbody) :: check_funcs st ct ft tl else let () = check_for_return tbody in let qq = (fname, class_opt, targs, rets, tbody) :: check_funcs st ct ft tl in qq | [] -> []

let rec build_function_table ft = function | (fname, class_opt, args, rets, body) :: tl -> let args_to_type = function | SDecl(t, (id, xpr)) -> (t, xpr) | _ -> raise InvalidArgErr in let arg_ts = List.map args_to_type args in let ft = (add_func ft fname arg_ts rets) in build_function_table ft tl | [] -> ft
let rec class_analysis class_tbl = function
| (class_id, attrs) :: tl ->
  let attr_tbl = addattrs empty_attribute_table attrs in
  let class_tbl = new_class class_id attr_tbl class_tbl in
  let lst, class_tbl = class_analysis class_tbl tl in
  ((class_id, attrs) :: lst), class_tbl
| [] -> [], class_tbl

let gen_class_stmts stmts =
  let sclasses, sclass_fns = translate_classes [] [] stmts in
  let (checked_sclasses, ct) = class_analysis class_table sclasses in
  checked_sclasses, ct, sclass_fns

let rec populate_http_symbol_table st = function
| (t, id, xpr) :: tl -> populate_http_symbol_table (add_sym t id st) tl
| [] -> st

let rec validate_http_tree path params rt ctx = function
| SParam(t, id, tree) :: tl ->
  let rest, rt = validate_http_tree path params rt ctx tl in
  let params = (t, id) :: params in
  let path = Format.sprintf "%s/:%s" path id in
  let rt = add_route path rt in
  let sub_tree, rt = validate_http_tree path params rt ctx tree in
  (rest @ sub_tree), rt
| SNamespace(name, tree) :: tl ->
  let rest, rt = validate_http_tree path params rt ctx tl in
  let path = Format.sprintf "%s/%s" path name in
  let rt = add_route path rt in
  let sub_tree, rt = validate_http_tree path params rt ctx tree in
  (rest @ sub_tree), rt
| SEndpoint(name, args, ret_t, body) :: tl ->
  (* takes arguments and path params and confirms they all exist *)
  let rec check_args = (function (* args, required args *)
    (* http arguments must be unpacked *)
    | ((a_t, id, _) :: a_tl), (req :: req_tl) ->
      if (a_t, id) = req then check_args (a_tl, req_tl)
      else raise UnusedParamArgument
    | x, [] -> ()
  )
let () = check_args (args, params) in
let rest, rt = validate_http_tree path params rt ctx tl in
let path = Format.sprintf "%s/%s" path name in
let st, ct, ft = ctx in
let st = populate_http_symbol_table st args in
let body = var_analysis st ct ft body in
(path, args, ret_t, body) :: rest, rt

(* TODO *)
let flatten_tree tree = []

(*The order of the checking and building of symbol tables may need to change
to allow functions to be Hoisted
NOTE: route_list is of type: "route"
*)
let gen_semantic_program stmts classes funcs h_tree =
  (* build an unsafe semantic AST *)
  let s_stmts = List.map translate_statement stmts in
  let s_http_tree = translate_http_tree h_tree in
  let s_funcs = List.map (translate_function None) funcs in
  let checked_classes, ct, sclass_funcs = gen_class_stmts classes in
  let s_funcs = sclass_funcs @ s_funcs in
  let dft = default_ft empty_function_table in
  let ft = build_function_table dft s_funcs in
  (* typecheck and reclassify all variable usage *)
  let checked_stmts = var_analysis symbol_table_list ct ft s_stmts in
  (*Add all the var decls to the global scope*)
  let st = add_to_scope symbol_table_list s_stmts in
  (*typecheck all the functions (including args and returns)*)
  let ctx = (new_scope symbol_table_list, ct, ft) in
  let checked_funcs = check_funcs st ct ft s_funcs in
  let route_list, _ = validate_http_tree "" [] empty_route_table ctx s_http_tree in
  (checked_stmts, checked_classes, checked_funcs, route_list)

let sast_from_ast ast =
  (* ignore functions for now *)
  let (stmts, classes, funcs, h_tree) = ast in
  let stmts = List.rev stmts in
  gen_semantic_program stmts classes funcs h_tree
exception SyntaxError of int * int * string;;

let lexbuf = Lexing.from_channel stdin in
let ast = try
  Parser.program Scanner.token lexbuf
with except ->
  let curr = lexbuf.Lexing.lex_curr_p in
  let line = curr.Lexing.pos_lnum in
  let col = curr.Lexing.pos_cnum in
  let tok = Lexing.lexeme lexbuf in
  raise (SyntaxError (line, col, tok))
in
let program_str = Ast_printer.program_s ast in
  print_endline program_str;;

print_endline "TEST_SUCCESS";;

test_sast

let lexbuf = Lexing.from_channel stdin in
let ast = Parser.program Scanner.token lexbuf in
let () = print_endline "\nAST" in
let () = print_endline (Ast_printer.program_s ast) in
let sast = Semantic_check.sast_from_ast ast in
let () = print_endline "\nSAST" in
let program_str = Sast_printer.string_of_sast sast in
  print_endline program_str;;

print_endline "TEST_SUCCESS";;

translate

open Sast
open Sast_printer
open Sast_helper
open Datatypes
exception UnsupportedStatementTypeErr of string
exception UnsupportedOutputType of string
exception UnsupportedExpressionType
exception InvalidStringExprType
exception InvalidBoolExprType
exception UnsupportedDeclType
exception InvalidIntExprType
exception InvalidFloatExprType

(* Convert AST.exp to SAST.sexpr *)
let rec translate_expr = function
  (* TODO: a ton more types here, also support recursive expressions *)
  | Ast.IntLit i -> SExprInt(SIntExprLit i)
  | Ast.StringLit s -> SExprString(SStringExprLit s)
  | Ast.FloatLit f -> SExprFloat(SFloatExprLit f)
  | Ast.BoolLit b -> SExprBool(SBoolExprLit b)
  | Ast.ListLit l ->
    let sexpr_list = List.map translate_expr l in
    SExprList(SListExprLit(None, sexpr_list))
  | Ast.ListAccess(xpr_l, xpr_r) ->
    let sxpr_l = translate_expr xpr_l in
    let sxpr_r = translate_expr xpr_r in
    SExprList(SListAccess(sxpr_l, sxpr_r))
  | Ast.CastBool c -> SExprBool(SBoolCast (translate_expr c))
  | Ast.Cast(t, xpr) -> translate_cast xpr t
  | Ast.UserDefInst(nm, actls) -> translate_user_def_inst nm actls
  | Ast.Access(e, mem) -> translate_access e mem
  (* we put a placeholder with the ID in and check after and reclassify *)
  | Ast.Id id -> SId id
  | Ast.Call c -> translate_call c
  | Ast.Binop(lhs, o, rhs) -> Sast.SBinop(translate_expr lhs, o, translate_expr rhs)
  | Ast_Null xpr -> UntypedNullExpr
  | _ -> raise UnsupportedExpressionType

and translate_cast xpr = function
  | Int -> SExprInt(SIntCast(translate_expr xpr))
  | Float -> SExprFloat(SFloatCast(translate_expr xpr))
  | Bool -> SExprBool(SBoolCast(translate_expr xpr))
  | String -> SExprString(SStringCast(translate_expr xpr))

and translate_user_def_inst class_id actls =
  SExprUserDef(SUserDefInst(UserDef class_id, (List.map translate_actual actls)))
and translate_actual = function
  | Ast.Actual(nm, xpr) -> (nm, (translate_expr xpr))
and translate_access xpr mem =
SExprAccess((translate_expr xpr), mem)

and translate_call = function
| (None, id, exprs) -> SCall(SFCall
  (None, id, (List.map translate_expr exprs)))
| (Some(xpr), id, exprs) -> SCall(SFCall
  (Some(translate_expr xpr), id, (List.map translate_expr exprs)))

and translate_lhs = function
| Ast.LhsId(id) -> SLhsId id
| Ast.LhsAcc(xpr, id) -> SLhsAcc (translate_expr xpr, id)

let translate_assign id xpr = match translate_expr xpr with
| SExprInt _ -> (id, xpr)
| SExprString _ -> (id, xpr)
| SExprBool _ -> (id, xpr)
| SExprFloat _ -> (id, xpr)
| SExprList _ -> (id, xpr)
| SId _ -> (id, xpr)
| _ -> raise UnsupportedExpressionType

let translate_decl = function
| (Int | String | Float | Bool) as t, id, i_xpr_opt ->
  SDecl(t, (id, expr_option_map translate_expr i_xpr_opt))
| ListType t, id, i_xpr_opt ->
  SDecl(ListType t, (id, expr_option_map translate_expr i_xpr_opt))
| t, _, _ -> raise(UnsupportedStatementTypeErr (Ast_printer.string_of_t t))
| _ -> raise UnsupportedDeclType

let translate_user_def_decl = function
| class_id, id, xpr ->
  SUserDefDecl(class_id, (id, (expr_option_map translate_expr xpr)))

let translate_vars = function
| Ast.ID(s) -> SFuncId(s)
| Ast.VDecl(vd) ->
  match translate_decl vd with
  | SDecl(t, (id, xpr)) -> SFuncDecl(t,(id, xpr))(*xpr will be None*)

let translate_fcall = function
| (Some(xpr), id, exprs) ->
  let sexprs = (List.map translate_expr exprs) in
  SFCall(Some(translate_expr xpr), id, sexprs)
| (None, id, exprs) ->
  let sexprs = (List.map translate_expr exprs) in
  SFCall(None, id, sexprs)
let rec translate_statement = function
    | Ast.VarDecl vd -> translate_decl vd
    | Ast.Assign(lhs, xpr) -> SAssign(translate_lhs lhs, translate_expr xpr)
    | Ast.UserDefDecl udd -> translate_user_def_decl udd
    | Ast.FuncCall(vl, fc) -> let sfc = translate_fcall fc in
        SFuncCall((List.map translate_vars vl), sfc)
    | Ast.If(expr, ifstmts, []) ->
        let ifs = List.map translate_statement ifstmts in
        SIf(translate_expr expr, ifs)
    | Ast.If(expr, ifstmts, else_stmts) ->
        let ifs = List.map translate_statement ifstmts in
        let es = List.map translate_statement else_stmts in
        SIfElse(translate_expr expr, ifs, es)
    | Ast.While(expr, stmts) ->
        SWhile(translate_expr expr, (List.map translate_statement stmts))
    | Ast.For(t, string_id, xpr, stmts) ->
        let s_xpr = translate_expr xpr in
        let s_stmts = List.map translate_statement stmts in
        SFor(t, string_id, s_xpr, s_stmts)
    | _ -> raise(UnsupportedStatementTypeErr "type unknown")

let translate_attr = function
    | Ast.NonOption (t, name, Some(xpr)) ->
        SNonOption(t, name, Some(translate_expr xpr))
    | Ast.NonOption (t, name, None) -> SNonOption(t, name, None)
    | Ast.Optional (t, name) -> SOptional(t, name)

let translate_fstatement = function
    | Ast.FStmt stmt -> translate_statement stmt
    | Ast.Return expr -> SReturn(List.map translate_expr expr)

let translate_function class_opt (f : Ast.func_decl) =
    (f.fname, class_opt, (List.map translate_decl f.args), f.return, (List.map translate_fstatement f.body))

let rec translate_http_tree = function
    | Ast.Param(t, id, tree) :: tl ->
        SParam(t, id, translate_http_tree tree) :: translate_http_tree tl
    | Ast.Namespace(name, tree ) :: tl ->
        SNamespace(name, translate_http_tree tree) :: translate_http_tree tl
let rec translate_members class_id sattrs sclass_funcs = function
| (Ast.Attr a) :: tl -> translate_members class_id ((translate_attr a) :: sattrs) sclass_funcs tl
| (Ast.ClassFunc f) :: tl -> let func_name = (class_id ^ "__" ^ f.fname) in let sclass_func = (translate_function None
({
  fname = func_name;
  args = f.args;
  return = f.return;
  body = f.body
})) in translate_members class_id sattrs (sclass_func :: sclass_funcs) tl
| [] -> (sattrs, sclass_funcs)

let translate_instance_fn class_id id (f: Ast.func_decl) = let func_name = (class_id ^ "__" ^ f.fname) in (translate_function (Some (SelfRef (class_id, id)))
({
  fname = func_name;
  args = f.args;
  return = f.return;
  body = f.body
}))
let translate_instance_block class_id (id, fns) = (List.map (translate_instance_fn class_id id) fns)
let rec translate_classes sclasses sclass_funcs = function
| (name, members, Some(Ast.InstanceBlock (id, fns))) :: tl -> let sattrs, sclass_fns = translate_members name [] [] members in let sinst_fns = translate_instance_block name (id, fns) in translate_classes
((name, sattrs) :: sclasses)
(sinst_fns @ sclass_fns @ sclass_funcs)
(name, members, None) :: tl ->
  let sattrs, sclass_fns = translate_members name [] [] members in
  translate_classes
    ((name, sattrs) :: sclasses)
    (sclass_fns @ sclass_funcs)
  tl
| [] -> sclasses, sclass_funcs

append

list<int> a = [1,2,3];
list<int> b = [3,4];
a = append(a, b);

for(int i in a)
{
  println(i);
}
println(len(a));

binops_auto_float_cast

float a = 3 + 5.5;
float b = 6.5 - 5;
println(a);
println(b);
println(a > 1);
int c = int(5.5 + 3);
println(c % 3);
println(c);
println(foo(b, c));
func foo(float f, int b) float{
  float a = f + b;
  println(a);
return a/b;

bool_binop

int a = 5;
int b = 2;

println(a == b);
println(a != b);
println(a > b);
println(a < b);
println(a >= b);
println(a <= b);
println((a == b) or (b != a));

string s = "a";
println("a" == s);
println(foo(b, a == b));

func foo(int a, boolean b, int c = 2) boolean{
    return a == c and b;
}

float c = 3.14;
int i = 3;
println (i < c);
println(a + b - 3 * 2);
println(3 + c * 1.1);

cast_bool

int i = 4;
printf("%t\n", i);

boolean b = false;
printf("%t\n", b);

string s = "";
printf("%t\n", s);
s = "hi";
printf("%t\n", s?);

float f = 3.14;
printf("%t\n", f?);
f = 0.0;
printf("%t\n", f?);

foo(f);

func foo(boolean i) {
println(i);
}

casts_everywhere

println(int(3.14));
println(string(3.14));
println(boolean(3.14));
println(float(3.14));
println(string(true));

class_literals_in_for_loops

class User {
    string name;
    int age = 21;
    optional float balance;
}

User a = new User(name="tester", age=20, balance=3.14);
println(a.balance);

User b = new User(name="tester", balance=3.14);
println(a.age);

User c = new User(name="tester");
println(c.balance?);

for (User u in [a, b, c]) {
  println(u.age);
}

decl_all

int a = 2;
float b = 3.14;
boolean c = true;
string d = "hello world";

int w = a;
float x = b;
boolean y = c;
string z = d;

println(w);
println(x);
println(y);
println(z);

decl_and_print_floats

float pi = 3.14;
float x = pi;
println(pi);
println(x);

decl_bool

boolean b = true;
b = false;
printf("%t
", b);

dcl_class_mult_attr

class User {
    string name;
    int age = 20;
    optional float balance;
}

User a = new User(name="tester", age=20, balance=3.14);
println(a.balance);

User b = new User(name="tester", balance=3.14);
println(a.age);

User c = new User(name="tester");
println(c.balance);

println(a.age + b.age + c.age);

User d;
d = new User(name="tester2");

fail_print_null_value

int x;
println(x);

int y = x;
println(y);

for_loop
list<int> my_list = [1, 13, 5];

for (int num in my_list) {
    int n = 7;
    printf("%d ", num);
    printf("%d ", n);
}

for (int num in [5, 6, 7]) {
    int n = 7;
    printf("%d ", num);
    printf("%d ", n);
}

formatted_print

string phrase = "I like %s\n";
string name = "STRFKR";
printf(phrase, name);

name = "Flume";
printf(phrase, name);

func

int a = foo(1, "hi");

println(a);

func foo(int a, string b = "hi") int {
    println(b);
    return a;
}

func_all

int x;
string z = "wow";
string y;
(x, y) = foo();

println(x); // 1
println(y); //wow

blank(1);

func foo() int, string{
    return boo(1), z;
}

func boo(int i, string s = "def") int {
    (int j, string q) = too(i, s);
    println(j);
    println(q);
    println(s);
    println(i);
    return i;
}

func too(int n, string a) int, string{
    println(n);
    println(a);
    return 2, "b";
}

func blank(int i){
    println(i);
    return;
}

    func_call_default_args

int b = 1;

b = foo(foo(2));

println(b);

func foo(int a, string b = "hi") int {
    println(b);
    return a;
}

    func_call_with_func_call_arg
int b = 1;

b = foo(foo(b, "hi"), "bye");

println(b);

func foo(int a, string b = "hi") int {
    println(b);
    return a;
}

func multireturn

(int x, int y) = foo(1, 3);

println(x);
println(y);

func foo(int x, int y) int, int{
    return y, x;
}

gcd

println(gcd(85125,25545));

func gcd(int p, int q) int {
    while (q != 0)
    {
        int temp = q;
        q = p % q;
        p = temp;
    }
    return p;
}

hello_world

println("Hello world");

instance_blocks
class User {
    int age;
    string name = "Default User";

    instance self {
        func set_age(int a) {
            self.age = a;
        }
        func get_name() string {
            return self.name;
        }
    }
}

User bob = new User(age=10);
println(bob.age);
bob.set_age(15);
println(bob.age);
println(bob.name);
bob.name = "Bob Burger";
string msg = bob.get_name();
println(msg);

    int_float_casting

int i = 4;
float f = float(i);

float d = 3.14;
int j = int(d);

println(f);
println(j);

    lists

list<string> x;
list<int> y = [];
list<int> z = [1, 2, 3];
list<list<int>> a = [
    z,
    [1, 2]
list<int> b = [1,2];

func foo(list<int> q)
{
    println(q[0]);
}

foo(b);

println(a[0][0]);
println(a[1][1]);

oop

class User {
    int age;
    string name = "Stephen";
    optional int height;

    instance my {
        func is_old() boolean {
            return (my.age >= 30);
        }
        func make_older() {
            my.age = my.age + 1;
        }
    }
}

User stephen = new User(age=29);
println(stephen.age);
stephen.height = 73;
println(stephen.height);

if (stephen.is_old()) {
    println("Stephen is old");
} else {
    println("Stephen is young");
}
stephen.make_older();
if (stephen.is_old()) {
    println("Stephen is old");
}
output_with_var

int x = 2;
string y = "hi";
println(x, y);

println_int_lit

int x = 2;
println(2);

while_count

int i = 0;
while(i < 3)
{
    println(i+1);
    if(i + 1 == 2)
    {
        println("two");
    }
    else
    {
        println("not two");
    }
    i = i + 1;
}

gcd

func gcd(int p, int q) int {
    while (q != 0) {
        int temp = q;
        q = p % q;
        p = temp;
    }
    return p;
}
func gcd(int p, int q) int {
    while (q != 0) {
        int temp = q;
        q = p % q;
        p = temp;
    }
    return p;
}

namespace gcd {
    param int a {
        param int b {
            http (int a, int b) int {
                int res = gcd(a, b);
                return res;
            }
        }
    }
}

println("Hello world");
http () string {
    return "hello world";
}

http hello_world

param string name {
    http hello(string name) string {
        return name;
    }
}

http plus(int i) int {
    return 2;
}

math
class Math {
    param int a {
        param int b {
            http add(int a, int b) int {
                return a + b
            }
        }
        http square(int a) int {
            return a*a
        }
    }
}

binary_expression
int a = 2 + b - c + 4;
i = a == b;
b = a > b;
c = a and b;
d = b or c;
bool_binop

int a = 5;
int b = 2;

println(a == b);
println(a != b);
println(a > b);
println(a < b);
println(a >= b);
println(a <= b);
println((a == b) or (b == a));

call_funcs
too();

func too(){
}

// now w/ return type
func foo() int {
(x, int y, int z) = boo();
return 5;
}

// now w/ return type tuple
func boo() int, int, int {
int a = 2;
int b = 3;
return a, b, 4;
}

cast_bool

boolean b = true?;
b = 3?;
b = [3];
b = [];

class_decl_with_optional
class User {
    optional int age;
    string name;
    string username;
}

class functions

my_object.my_function();
a.b(1, "world");

comments

/*
 * I can do a block comment
 */

// or a small comment

control_flow

boolean b = false;
int q = 9 * 9 - 3;
while(b)
{
    int a = 1;
    int b = 3;
    if((b-1) > a)
    {
        b = b + a;
    }
    else
    {
        b = 4 - a * 2;
    }
}

decl_bool
boolean b = true;
b = false;

int c = 5;

b = c?;

decl_class

// all valid
class Hello {
    string username;
}

class Empty {
}

fail_cast_bool

boolean b = true;
b = ?false;

fail_class_decl

// No class name
class {
}

fail_class_function_keyword_name

optional.my_func();

fail_class_invalid_attr_type
/ Integer is not a primitive type
class User {
    Integer hi;
}
ail_dangling_expr

1+1

fail_instance_block_missing_id

class User {
    int age;
    instance {
        func get_age() int {
            return self.age;
        }
    }
}

fail_instance_block_outside_class

instance self {
    func get_age() int {
        return self.age;
    }
}

fail_instantiate_missing_comma

class User {
    string username;
    string full_name;
    string password;
    int age;
}

User bob = new User(
    username="burgerbob
class User {
    string username;
    string full_name;
    string password;
    int age;
}

User bob = new User(
    username=,
    full_name="Bob Belcher",
    password="burgersrock",
    full_name="Sam Sour",
    age=42
);

fail_instantiate_missing_value

// missing class keyword
User {
    string hi;
}

fail_lists

list<list> x = [1,];

fail_main

main() {
}
fail_member_double_dot

abc..hello

fail_missing_optional_attr_name

class User {
    optional int;
    string username;
}

fail_missing_type_optional

class User {
    optional age;
    string username;
}

fail_return

// should fail because it is not in a function block
return x;

for_loop

for (int num in my_list) {
    printf("%d ", num);
    printf("%d ", num);
}

func

func foo(){
}
// now w/ return type
def func foo() int {
  
}

// now w/ return type tuple
def func foo() int, int, int {
  
}

func_call_larg

int a = foo();

func_default_value_args

def func foo(int x, int a = 0) {
  x = a;
}

func_default_with_null

def func foo(int x, int a = 0, int n = null) {
  x = a;
}

func_multireturn

def func foo(string s, int x = 2, int a) int, string{
  a = 3;
  //return a, s;
}

func_with_body

func foo(int a){
  int x = 2;
  a = a + 2;
}

func_with_param
func foo(int a) {
}

func foo(int a, int b, string c) {
}

func_with_return

func useless(int x) int {
    return x;
}

function_inside_class

class User {
    int age;
    string name;

    func add(int a, int b) int {
        return a + b;
    }
}

http

namespace test {
    param int test_number {
        http info(int date) string {
            return 0;
        }
    }
}

param int test_number {
    http info(int date) string {
        return 0;
    }
    http info(int date) string {

return 0;
}

http info(int date) string {
    return 0;
}

namespace test2 {
    param int test_number {
        http info(int date) string {
            return 0;
        }
    }
    param int test_number {
        http info(int date) string {
            return 0;
        }
    }
    param int test_number {
        http info(int date) string {
            return 0;
        }
    }
    param int test_number {
        http info(int date) string {
            return 0;
        }
    }
    param int test_number {
        http info(int date) string {
            return 0;
        }
    }
    param int test_number {
        http info(int date) string {
            return 0;
        }
    }
    param int test_number {
        http info(int date) string {
            return 0;
        }
    }
}

http () int {
    return 0;
}

instance_blocks
class User {
  int age;
  instance self {
    func get_age() int {
      return self.age;
    }
  }
}

instantiate_class

User bob = new User(
  username="burgerbob",
  full_name="Bob Belcher",
  password="burgersrock",
  age=42
);

int_and_float_casting

float f = 5.3;
int x = int(f);

float d = 3.3;
int y = float(d);

literals

string s = "hello world";

string b = "~~ strings work ~~";

member_access

int a = abc.def;
abc.def = 5;

print_hello

println("hello world");
int x = 1;

func foo(){
   printf("%d", x);
}

func print_name(string name) {
   printf("hi %s", name);
}

string name = "hunter42";
printf("%s, what's your password", name);
println("*******");

string dog = "dog";

float x = 3.14;
y = 0.;
z = .20;
a = 2e0;
b = 2234e340;

access_optional_attrs

class User {
    optional int age;
    string username;
}

User bob = new User(username="bobiscool");

int x = bob.age;

binops

int a = 2 + 3 + 3;
int c = 2;
c = c + c - a;
boolean b = c == 4.4;
b = a < c;
b = "afda" == "Afda";
b = a == c and b;

float f = 3 + 3.3 + 2.2 + 3;

call_default_args

func foo(int a, int b, string s = "", string e = null) int{
    foo(1, 2);
    foo(1, 2, "3");
    foo(1, 3, "123", "adf");
    return 1;
}

int i = foo(2, 3);
i = foo(1, 2, "3");
i = foo(1, 3, "123", "adf");

call_funcs
int n = foo(); // this is vdecl stmt with a funcall expr.
n = 2;
boo(); //one statement

func too(int b) string{
    return "s";
}

// now w/ return type
func foo() int {
    int x;
    (x, int y, string z) = boo(); //Also one statement, boo not an expr.
    return 5;
}

// now w/ return type tuple
func boo() int, int, string{
    int a = 2;
    int b = 3;
    return a, foo(), too(b);
}

call_multireturns_args

int i = 3;
int b;
b = 3;

(int x, string s) = foo( b, "a");
x = 5;

func foo(int a, string b) int, string{ return a, "s"; }
cast_bool

int i = 4;
boolean b = i?;

call_flow_scoping

int a = 0;
while(a == 0)
{
    boolean a = false;

    if(a)
    {
        float a = 02.3;
    }
    else
    {
        a = true;
        float b = 3.3;
        if(3.33 > b/2)
        {
            println(b);
        }
    }
}

decl_class_mult_attr

class User {
    string name;
    int age;
    float balance;
}

decl_mult_class

class User {
    string name;
    int age;
    float balance;
}
class OtherUser {
    string name;
    int age;
    float balance;
}

dcl_null_val

int x;

dcl_float

float pi = 3.14;

float pi2 = pi;

pi = pi2;

fail_assign_order

x = 2;
int x = 3;

fail_binop_assign_wrong_type

int a;

a = 3 == 2;

fail_binop_type_mismatch

int c = 3 * "asd";
string s = "Afda" + "asdfa";

func foo(int a) int {
    return;
}

int i = 3;
int b;
b = 3;

(int x, i) = foo(b, "a");

func foo(int a, string b) int, string{ return a, "s"; }
func bar() int{ return 1; }

func foo(int a, int b, int i, string c = "1231", string d = null) {}
foo(1, 2);

func foo(int a, int b, int i, string c = "1231", string d = null) {}
foo(1, 2, 3, 4, "5");

float f = float("fail");
string s = "hello";
int i = int(s);

fail_decl_class_repeat_attr

class User {
    string name;
    int age;
    float balance;
    float age;
}

fail_decl_repeat_class

class User {
    string name;
    int age;
    float balance;
}

class User {
    string name;
    int age;
    float balance;
}

fail_float_use

float pi = 3.14;
int pi2 = pi;
list<int> my_list = [1, 13, 5];

for (string not_a_string in my_list) {
    printf("%d ", num);
    printf("%d ", num);
}

printf("%d ", num);

list<int> my_list = [1, 13, 5];

for (int num in my_list) {
    printf("%d ", num);
    printf("%d ", num);
}

printf("%d ", num);

int i = 3;
int b;
b = 3;

int x = foo(foo(1, b), "a");

func foo(int a, string b) int{ return a; }

func foo(int x = 2, int a) {
a = 3;
}

func foo(int a= "ada") {
int x;
}
fail_func_no_return

```go
func foo(int a, int b = 0) int{
    int x = 2;
    x = 3;
}
```

fail_func_nodefault_arg_after_default

```go
func foo(string s, int x = 2, int a) {
    a = 3;
}
```

fail_func_with_cast

```go
func foo() int{
    int i = 1;
    return i?;
}
```

fail_func_wrong_return_types

```go
func foo(int a, int b = 0) int, string{
    return a, b;
}
```

fail_instance_block_access_wrong_type

```go
class User {
    int age;
    instance self {
        func get_age() string {
            string s = self.age;
            return s;
        }
    }
}
```

fail_instantiate_duplicate_args
class User {
    string username;
    string full_name;
    string password;
    int age;
}

User bob = new User(
    username="burgerbob",
    full_name="Bob Belcher",
    password="burgersrock",
    full_name="Sam Sour",
    age=42
);

fail_instantiate_missing_args

class User {
    string username;
    string full_name;
    string password;
    int age;
}

User bob = new User(
    username="burgerbob",
    password="burgersrock",
    age=42
);

fail_instantiate_type_mismatch

class User {
    string username;
    string full_name;
    string password;
    int age;
}
User bob = new User(
    username="burgerbob",
    full_name=7,
    password="burgersrock",
    age=42
);

fail_invalid_float

float a = e3;

fail_lists_inner_wrong_type

list<boolean> x = [True];
lst<int[][]> y = [x];

fail_lists_mixed_types

list<int> x = [1, 2, "3"];

fail_member_functions_arg_types

class User {
    int age;
    string name;

    func add(int a, int b) int {
        return a + b;
    }
}

User bob = new User(age=42, name="Bob Burger");
string hello = "hello";
int three = bob.add(hello,2);

fail_member_functions_no_function
class User {
    int age;
    string name;
}

User bob = new User(age=42, name="Bob Burger");
int three = bob.add(1,2);

    fail_member_functions_return_type

class User {
    int age;
    string name;

    func add(int a, int b) int {
        return a + b;
    }
}

User bob = new User(age=42, name="Bob Burger");
string three = bob.add(1,2);

    fail_misnamed_param_argument

param int test1 {
    http info(int test2) string {
        return 0;
    }
}

    fail_missing_param_arg

param int test_number {
    http info() string {
        return 0;
    }
}
int c;
(int b, c) = boo();

// now w/ return type tuple
func boo() int, int, int {
    int a = 2;
    int b = 3;
    return a, b, 4;
}

(int a, string b) = foo();
func foo() int, int{
}

int a = 5;
int b = a.hello;

printf(1);

int x = 5;
x = 4;
x = "no types yet";
string full_name;
string password;
int age;
}

User bob = new User(
    username="burgerbob",
    full_name="Bob Belcher",
    password="burgersrock",
    age=42
);

bob = new User(
    username=7000,
    full_name="Not Bob Belcher",
    password="notburgersrock",
    age=41
);

fail_repeat_http_route

namespace test {
    http info(int date) string {
        return "hi";
    }
}

namespace test {
    http info(int date) string {
        return "hi";
    }
}

fail_repeat_int_decl

int x = 5;
int x = 1;
x = 2;

fail_return_type_mismatch
int b = boo();

// now w/ return type tuple
func boo() int, int, int {
    int a = 2;
    int b = 3;
    return a, b, 4;
}

    fail_string_assign_to_int

string ten = 10;

    fail_too_many_defaultargs

func foo(int a, int b, int i, string c = "1231", string d = null) {}
foo(1, 2, 3, "4", "5", 3);

    fail_type_mismatch_optional_attr

class User {
    optional int age;
    string username;
}
User bob = new User(username="bobiscool");
string s = bob.age;

    fail_undecl_in_lhs_call

func foo(int a) int, int { return 1, 2; }
int x = 0;
(x, b) = foo(x);

    fail_undeclared
x = 2;

fail_undeclared_func_body

func foo(int a){
    b = 2;
}

fail_var_reassign_to_wrong_type

int x = 2;
string y = "hi";
x = y;

fail_wrong_arg_type

func foo(int a){
    string x = a;
}

func
c

func too(){
}

// now w/ return type
func foo() int {
    return 5;
}

// now w/ return type tuple
func boo() int, int, int {
    int a = 2;
    int b = 3;
    return a, b, 4;
}
func foo(int a) {
    return;
}

func foo(int a) {
    string x;
    int aksdjf;
}

int i = 3;
int b;
b = 3;

b = foo(i, "232");
int x = foo(foo(1, "!"), "a");

func foo(int a, string b) int { return a; }

int i = 3;
int b;
b = 3;

foo(foo(i, "b"), "a");

func foo(int a, string b) int { return a; }

func foo(int x, int a = 0, int n = null) {
    x = a;
}

func foo(int a, string b) int { return a; }
func foo(string s, int a, int x = 2) {
    a = 3;
    int i = x;
}

default_value_args

func foo(int a = 0) {
    int x;
}

with_body

func foo(int a) {
    int x = 2;
    a = x;
}

with_cast

func foo() boolean {
    int i = 1;
    return i?;
}

with_globalvar

int b;

func foo(int a) {
    b = 2;
}

with_param

func foo(int a) {
    int x = a;
}

func boo(int a, int b, string c) {
    a = b;
}
func useless(int x) int{
    return x;
}

handle_null_var

int x;
printf("res: \%d", x);
int y = x;

instantiate_using_defaults

class User {
    string username = "burgerbob";
    int age = 42;
}

User bob = new User();

instance_blocks

class User {
    int age;
    string name = "Default User";

    instance self {
        func set_age(int a) {
            self.age = a;
        }
        func get_name() string {
            return self.name;
        }
    }
}


User bob = new User(age=10);
string n = bob.get_name();
bob.set_age(15);
bob.name = "Bob Burger";

instantiate_class

class User {
    string username;
    string full_name;
    string password;
    int age;
}

User bob = new User(
    username="burgerbob",
    full_name="Bob Belcher",
    password="burgersrock",
    age=42
);

instantiate_class_optional_attrs

class User {
    optional int age;
    string username;
}

User bob = new User(username="bobiscool");

int_and_float_casting

float d = 3.3;
int y = int(d);

int_decl

int a;
int x = 5;
x = 2;
member_access

class User {
    int age;
    string name;
}

User u = new User(age = 10 , name = "Sarah");

string n = u.name;
int a = u.age;

member_functions

class User {
    int age;
    string name;

    func add(int a, int b) int {
        return a + b;
    }
}

User bob = new User(age=42, name="Bob Burger");
int three = bob.add(1,2);

reinstantiate_class

class User {
    string username;
    string full_name;
    string password;
    int age;
}

User bob = new User(
    username="burgerbob",
    full_name="Bob Belcher",
    password="burgersrock",
    age=42
)
bob = new User(
    username="notburgerbob",
    full_name="Not Bob Belcher",
    password="notburgersrock",
    age=41
);

test_all_casts

string s = string(7);
s = string(3.14);
s = string(3.14);
s = string(false);

int i = int(7);
i = int(3.14);

boolean b = 7?;
b = boolean(7);
b = 3.14?;
b = boolean(3.14);

string s2 = string(b);

var_declarations

int a = 1;
int b = 2;
int c = 3;

var_reassign_to_var

int x = 2;
int y = 3;
x = y;