LOON

THE LANGUAGE OF OBJECT NOTATION

Final Report

Kyle Hughes, Jack Ricci,
Chelci Houston-Borroughs, Niles Christensen, Habin Lee

December 2017

Contents

1 Introduction

Over the past decade, JavaScript Object Notation (JSON) has arguably become the format of choice for transferring data between web applications and services. With the rise of AJAX-powered sites, developers everywhere are using JSON to pass updates between client and server quickly & asynchronously. JSON has achieved its immense popularity in large part due to its flexibility and lightweight nature, but also due to its independence from any particular language. An application programmed in Java can send JSON data to a client running on a JavaScript engine, who can pass it along to a different application coded in C#, and so on and so forth. However, programs written in these languages generally utilize libraries to convert JSON data to native objects in order to manipulate the data. When the program outputs the modified JSON data, it must convert it back to JSON format from the created native objects.

LOON, the Language of Object Notation, provides a simple and efficient way to construct and manipulate JSON data for such transfers. Developers will be able to import large data sets and craft them into JSON without needing to import standard libraries or perform tedious string conversions. In addition to generating JSON format from other data formats, programmers will be able to employ LOON to operate on JSON data while maintaining valid JSON format during all iterations of the programming cycle. In other words, LOON eliminates the JSON-to-Native Object-to-JSON conversion process. This feature provides valuable debugging capabilities to developers, who will be able to log their output at any given point of their code and see if the JSON is being formatted properly. LOON is simple by nature. It resembles C-based languages in its support for standard data types such as int, float, char, boolean, and string. Arrays in LOON are dynamic and can hold any type. The language introduces two new types: Pair and JSON. These two types will be at the heart of most every piece of LOON source code. Where LOON truly separates itself is in its provision of operators, such as the "+=" operator for concatenation, for usage on values of the JSON and Pair types. This allows for more intuitive code to be written when working with JSON data.

2 LOON Tutorial

2.1 Setup

2.1.1 LLVM

We use LLVM version 5.0.0. Please make sure you have llvm 5.0.0 installed. symlink of llc is also required, similar to microc requirements.

2.2 Using LOON

2.2.1 Hello World

Use the provided script **./compile.sh loon.native filename.loon** to compile and run .loon file.

Below is the Hello World! code example in LOON. (helloWorld.loon)

```
void main() {
  printJSON("Hello, World!")
}
```

Input: ./compile.sh loon.native helloWorld.loon
Output: Hello, World!

Now, let's notice some things here.

1. the **main()** function is of **void** type, (not int), and LOON requires it.

2. second, **printJSON()** is the built-in function that will print stuff.

2.2.2 LOON Arrays

LOON Arrays are not type-restrictive. In other words, an array object may take any number of different types. Here's an example:

```
void main() {
  string name
    name = "Bob"
    char b1
    b1 = name[2]
    array arr
    arr = ["Jill", 24, name, b1]

    array arr2
    arr2 = [arr, 2, 3]
    printJSON(arr[0])
    printJSON(arr[1])
    printJSON(arr[2])
    printJSON(arr[3])
    printJSON(arr2[0][1])
}
```

Output: Jill 24 Bob b 24 (with newlines after each output)
Things to notice:
1. Characters can be accessed from string objects by array-indexing them,
2. Arrays are infinitely nest-able. You may assign and access any array of any depth.

3. Again, as a consequence, these arrays are completely type-blind.

2.2.3 Making a Pair

LOON's Pair type, contrary to our array type, enforces type consistency. Below you will find how to specify types for pair object and its syntax.

```
void main() {
  pair<int> p1
    p1 = <"Janet", 80000>
    int val
    val = *p
    printJSON(val)

    pair<string> p_str
    p_str = <"hello", "world">
    string w
    w = *p_str
    printJSON(w)

  pair<pair<int>> rp
    rp = <"nested", <"pairs", 5>>
    int nested_contents
    nested_contents = **rp
    printJSON(nested_contents)
}
```

Output: 80000 world 5

What to notice:

1. The types are enforced for the second item of the pair. (in the language of key-value pair, we can say that the specified types are for the 'value')
2. Dereferencing can be done in C style, with the dereferencing operator *.
3. Dereferencing will return the 'value' part of the pair.
4. use <type> to specify the type for the given pair.
5. The pair type can similarly be nested (as done in our array) and can be dereferenced as above (5 is printed).

2.2.4 the json type

This is the bread and butter of LOON.

```
void main() {
  json j
    j = {|"this": 5, "is": "my", "first": true, "json": "object"|}
    int x
    x = j["this"]
    printJSON(x)
    string y
    y = j["json"]
    printJSON(y)
}
```

Output: 5 object

what to note:

1. the 'pipe' character '|' distinguishes the json initialization.
2. As the json type suggests, all data within are key-value pairs where the value types are flexible.

3 Language Reference Manual

3.1 Types

3.2 JSON

An object of type json is formatted according to the official JSON standard. That is, any object of type json is the concatenation of:

1. An open brace character, **{** followed by a pipe character, |
2. Any number of pair objects, with the comma character spliced in between each instance of two consecutive pairs
3. A pipe character, |, followed by a closed brace character, **}**

Contents nested inside of the json object can be accessed through the key-value access notation described in section 3f. Objects of type json are initialized by two methods:

1. An open brace followed by two pipes followed by a closed brace after the $=$ operator. This is the default style of initializing a json object.
2. Entering any valid JSON object (defined above) after the $=$ operator. If an invalid JSON object is assigned as the initial value of an identifier of type json, the compiler will throw an error.

3.3 Pair

The pair type represents a key-value pair. The key will always be a String (as described below). The value can be an object of any type described in this language, except for an object of type pair. In totality, a valid pair object consists of the concatentation of the following:

1. An open carat, $<$
2. A string literal in quotation marks
3. A comma
4. An object of any valid type
5. A close carat, $>$

If two pairs are concatenated using the $+$ operator, the resulting object is of type json.
Pairs are declared using carat notation, where a valid LOON type name must be spliced between the carats. An example is:

```
pair<int> intPair
```

The compiler will throw an error on the following initialization miscues:

1. Type mismatch between the pair's declared value and the initialized value.
2. Attempting to assign the concatenation of two pair objects to an identifier of type pair, as the concatenation of two pair objects is of type json.

The value of a pair's key can be retrieved using the notation described in section 3f.

3.4 Int

A 32-bit two's complement integer. Standard mathematical operations will be implemented.

3.5 Char

An 8 bit integer. Can be used as an integer, but should typically be understood to represent an ASCII character.

3.6 Boolean

A 1-byte object that can have two values: True and False. Can use standard boolean operators.

3.7 Array

An array of any of the other types of values, including Array itself. JSON format allows for type flexibility within a single array, so LOON offers developers the ability to craft arrays holding values of any type. Array declaration consists of the array keyword followed by an identifier:

```
// Declare a new array identifier
array myArray
```

Array initialization occurs by assigning a constant list of values of any type to a previously declared array identifier:

```
// Initialize an array
array myArray
myArray = ["test", 4, "out"]
```

LOON supports passing identifiers, strings, characters, integers, and arrays in as values in array initialization:

```

// Initialize an array
array myArray
string testStr
testStr = "test"
myArray = [testStr, 4, [[6, "hurt"], "old"]]
```

The behavior of arrays is undefined when an object of type JSON or type pair is passed in as a value during initialization.
The contents of an array can be accessed by traditional array access notatation:

```

// Initialize an array
array myArray
myArray = ["test", 4, [[6, "hurt"], "old"]]

//Access the value at the 2nd index position
array nestedArr
nestedArr = myArray[2]  // Points to [[6, "hurt"], "old"]
printJSON(nestedArr[1]) // Prints "old"
```

Arrays in LOON are fixed-sized but content mutable; writes may occur at any position within the boundaries established by the initialized array.

```

// Initialize an array
array myArray
myArray = ["test", 4, [[6, "hurt"], "old"]]

// Modify value at zeroth index position
myArray[0] = 6 // myArray is now: [6, 4, [[6, "hurt"], "old"]]

myArray[2][2] = 7 // invalid write - behavior undefined
```

Passing in any expression type that is not an identifier or a literal will result in the compiler throwing an error for an illegal access attempt.

3.8 String

Strings are an immutable, array sequence of characters. To modify an existing string, it is necessary to create a new one that results from some use of legal string operations. The code snippet below details the declaration and manipulation of strings in LOON:

```
// Declare and initialize a string
string newLang
newLang = "LOON"
```

LOON allows for directly accessing a character from the contents of a string. The notation to do so is identical to that of array access:

```

// Initialize a string
string testStr
testStr = "test"

//Obtain the character located at the string's zeroth index position
char c
c = testStr[0]

printJSON(c) //Prints: 't'
```

However, writing a character to a specific index position within a string is not permitted within LOON.

LOON supports string concatentation using the '+' operator:

```

// Initialize a str
string testStr
testStr = "test " + " strcat" //testStr is now "test strcat"
```

String concatenation is only supported in the form of concatenation of two string constants. Behavior when attempting to concatenate two identifiers or one identifier and one string constant is undefined.

In its declaration, memory is allocated precisely according to the string's size. To expand the size of the string requires the allocation of a new string of the desired size, followed by copying the contents of the old string into the start of the newly allocated space in memory.

3.9 Lexical Conventions

3.10 Identifiers

LOON identifier refers to the name given to entities such as variables, functions, and objects. They give unique name to an entity to identify it during the execution of the program. You can choose any name for an identifier outside of the keywords.
For example:

```
int count
count = 0
String myString
myString = 'hello'
json result
```

Here *count*, *myString* and *result* are identifiers.

3.11 Keywords

The following are reserved words in LOON and cannot be used as identifiers to define variables or functions: *if, elseif, else, for, while, return, break, continue, int, float, char, boolean, string, array, json, pair.*

3.12 Literals

LOON literals refer to fixed values that are immutable during program execution. They can be of any of the primitive data types such as *integer, float, string, and boolean.*

1. Integer Literal
   An integer literal is a sequence of one or more integers from 0-9.
2. Float Literal
   A float literal has an integer part, decimal point, fractional part and exponential part.
3. String Literal
   String literals are sequences of characters enclosed in single quotes.
4. Boolean Literal
   Boolean literals are either *true* or *false*. If the user assigns a different value an error will be raised.
5. Pair Literal
   As described above
6. JSON Literal
   As described above

3.13 Comments

LOON allows for multiline/nested comments, as well as single-line comments. The table below summarizes the convention for both comment formats:

Comment Symbol	Description	Example
/* */	Multiline comments	/* This /* is legally */ commented */
//	Single-line comment	// This is a legal comment in LOON

3.14 Whitespace

The newline is significant in the LOON language; otherwise, whitespace is discarded.

3.15 Functions

3.15.1 Function Definitions

The general format used to define a function in LOON is as follows:

```
return_type function_name(parameter list ) {
   body of the function
}
```

Here are all the parts of a function in LOON -

1. Return Type
   The return type is the data type of the value the function returns. If the function does not return a value, users should use return type *void*.

2. Name
   This is the identifier for the function. The name paired with the parameters is the function signature.

3. Parameters
   Parameters act as placeholders in LOON. When a function is invoked, the user passes a value into the parameter. The parameter list refers to the type, order and number of parameters of a function. A function may also contain no parameters.

4. Body
   The body of a function contains a collection of statements that logically define what a function does.

3.15.2 Function Declaration

Functions in LOON are called by their identifiers. To call a function, pass the required parameters with the function name. If the function returns a value, you can store it in a variable.

3.16 Operators/Punctuation

Operator	Usage	Function
+	x + y (binary operator)	Depends on types added. When adding two Floats, or two Integers, or a Float and an Integer, it returns the sum of the two values. If both numbers added are Integers, then the expression evaluates to type Integer. Otherwise the expression evaluates to type Float. If both x and y are Arrays, then the result is a new Array that is y concatenated to the end of x. If x is an Array and y is an object of the same type as x stores, then the expression evaluates to a new array with all the values in x, and the value of y appended to its end. If either x or y is a String and the other is a character, then this concatenates the second value to the first and represents a String representation. If both x and y are Characters, then the expression evaluates to a String consisting of x first, and then y second. If both x and y are of type JSON, then it evaluates to a new object of type JSON that contains all the keys in both JSON objects. If adding 2 Pair objects, the expression evaluates to a JSON object containing both Pairs. If adding a Pair object to a JSON object, then the result is a JSON object containing all the Pairs from the JSON object as well as the Pair being added. If both x and y are Strings, then this is the concatenation operator.
-	x - y (binary operator)	Depends on types used on. If subtracting an Integer from an Integer, then this evaluates to an Integer representing the difference between the two values. If subtracting an Integer from a Float, a Float from an Integer, or a Float from a Float, then the expression evaluates to type Float, but is still the difference between the two values.
	- x (unary operator)	Valid when x is either an Integer or a Float. Returns the value of x * -1. Evaluated expression is of same type as x.

*	x * y (binary operator)	Depends on types used on. If multiplying an Integer by an Integer, then this evaluates to an Integer representing the product of the two values. If multiplying an integer by a Float, a Float by an Integer, or a Float by a Float, then the expression evaluates to type Float, but is still the product between the two values.
/	x / y (binary operator)	Depends on types used on. If dividing an integer by an integer, then this evaluates to an integer representing the result of dividing x by y, with the remainder discarded. If dividing an integer by a float, a float by an integer, or a float by a float, then the expression evaluates to type Float, and is x divided by y as a decimal as well as can be approximated.
[]	x[y]	Used to access values. There are a few possible valid combinations of types for x and y. If x is an Array and y is an Integer, this evaluates to the value that is stored at location y in x. Type will vary depending on what the Array stores. If x is a String and y is an Integer, this evaluates to the Character in location y of x. If x is a JSON object, and y is a String, then this expression evaluates to the value of the object in x with key y.
==	x == y	Evaluates to True if x is equal to y, and False otherwise.
!=	x != y	Evaluates to True if x is not equal to y, and False otherwise.
!	!x (unary operator)	Valid when x is a boolean. Evaluates to False if x is True and True if x is False.
*	*x (unary operator)	Valid when x is a pair. Evaluates to the value stored in the pair object.
>	x >y	Evaluates to True if x is greater than y. Valid for an combination of Integer, Float, and Char.
>=	x >= y	Evaluates to True if x is greater than or equal to y. Valid for an combination of Integer, Float, and Char.

<table>
<tr><td><</td><td>x <y</td><td>Evaluates to True if x is less than y. Valid for an combination of Integer, Float, and Char.</td></tr>
<tr><td><=</td><td>x <= y</td><td>Evaluates to True if x is less than or equal to y. Valid for an combination of Integer, Float, and Char.</td></tr>
<tr><td>&&</td><td>x && y</td><td>Logical and. Evaluates to True if both x and y are True, and False otherwise.</td></tr>
<tr><td>||</td><td>x || y</td><td>Logical or. Evaluates to True if either x or y are True, and False otherwise.</td></tr>
</table>

3.17 Syntax

3.18 Program Structure

In LOON, there is a single main function, designated void main(). This is the entry-point for the program. Other functions can then be called from within the body of this function, which can in turn call other functions.

3.19 Expressions

3.19.1 Declaration

Declaration of variables are achieved as follows. Type specification is mandatory for pair.

```
json myJSON
pair<int> intPair
string myName
array myIntArray
```

3.19.2 Assignment

Assignment : Assignment can be done using where lvalue is the variable and rvalue is the value.

```
myIntArray = [1, 2, 3, 4, 5]
```

3.19.3 Precedence

All operators follow the standard precedence rules. Every operation, apart from assignment (right-to-left associative), is left-to-right associative.

3.20 Statements

3.20.1 Expression Statements

Statements in within a line (not escaped) are treated as one statement

3.21 Loops

Loop Type	Usage	Function
While	while(x) {y}	If x is false, nothing happens. If x is true, then the block of code y is executed. Once y is finished, the loop is evaluated again. If x is now false, then the loop ends (breaks). However, if it is still true, then the entire process is repeated. The loop can carry on potentially infinitely if x never becomes false.

For	for(a ; b ; c) {y}	Upon encountering this loop, the command a is evaluated. A is only evaluated the first time through. Then, b is evaluated. If b is true, then the block of code y is evaluated. Once y is finished, c is evaluated. Then, b is evaluated again. If b is still true, then y is executed again. This process of y → c → b is repeated for as long as b is true when evaluated.

3.22 Scope

1. Single file scope
   Identifiers declared within a LOON file are either declared external to any function or inside of a function. Identifiers declared external to all functions within the file are accessible for all functions defined within the file. The identifiers persist from beginning to end of program runtime. Identifiers delcared within any function block or conditional block are accessible only within their block of declaration. They persist from the moment that they are declared to the moment that their block of declaration is no longer being executed. The only exception to this standard is that variables declared inside of a loop conditional definition persist until the loop's conclusion.

2. Multiple file scope
   Multiple file scope concerns the relationship between identifiers external to any function block in multiple files. An external identifer in one file is not accessible for methods in other files. As a result, there is no conflict between two external identifiers of the same name in two different files. The compiler will recognize that these are independent identifiers and treat them accordingly. Functions are utilized to pass data to and from external identifiers in different files. Functions native to one file are accessible to functions in another file by including the first file at the beginning of the second as defined in section 4a.

3.23 Input/Output

LOON contains the ability to print objects to standard output and read in text in the form of strings from standard input:

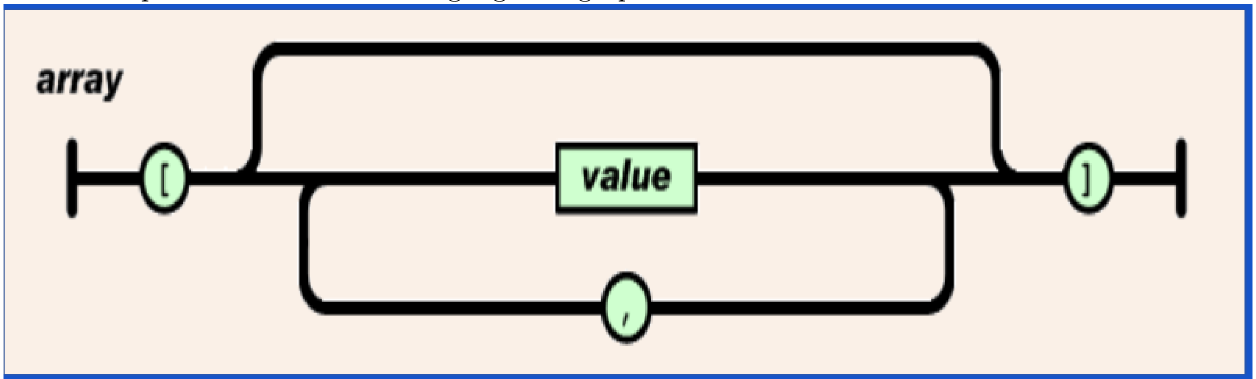
1. printJSON(AnyType x, ...)
   Prints each of the comma-delimited arguments to screen. Arguments can be of any of the following types and they will be printed with the proper format: string, char, int. Identifiers containing objects of the above types can also be printed using printJSON. The behavior of printJSON is undefined for passing in arrays, pairs, and json objects.

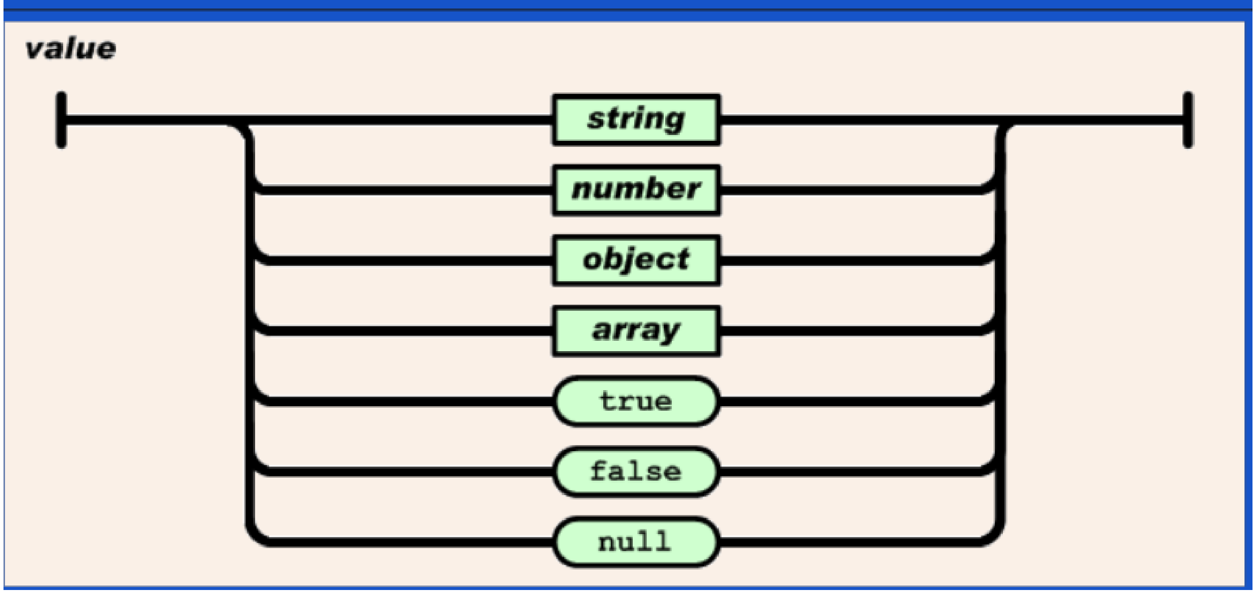
2. loon_scanf() reads directly from stdin. This function allows us to read input from stdin and store it in a character pointer for further manipulation.

4 Project Plan

4.1 Planning of

We started out by studying and specifying the actual JSON data type structure. We had to decide what the most important thing JSON offered and decided which features LOON can implement that can make json data manipulation easier. Naturally, www.json.org was a reference point for a lot of the specifics in the initial language design phase.





4.2 Team Roles

We assigned roles at the beginning of the project. However, as time progressed and people found niches that worked well, our roles gradually evolved. These descriptions represent what we eventually ended up doing rather than what we thought we would do at the beginning.

4.2.1 Jack Ricci: the Linguist

Worked across the translator stack, from improving the scanner and upgrading and heavily influencing the construction of the AST to handling a large percentage of the intermediate representation that was generated.

4.2.2 Niles C: Language Design, Tests, and Specialized Types

Took the lead on designing the semantics, types, and nuances of the language. Built the testing infrastructure and ensured that tests were kept up to date. Built the Pair and JSON types.

4.2.3 Kyle Hughes: PM

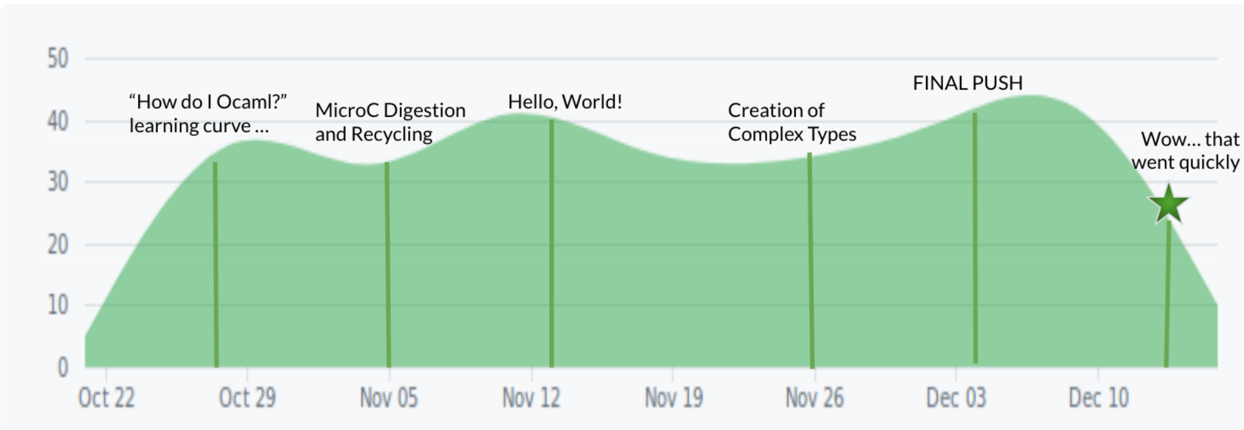
4.2.4 Chelci H:PM

Teamed up with Kyle to prepare some of the logistics of our semi-weekly meetings. Updated the README with next meeting deliverables. Teamed with Jack to work on parser updates, implementing arrays and scanf file input function. Added tests as necessary.

4.2.5 Habin Lee: Architecture

My job was to make sure all files compiled successfully by ensuring uniform development environment across all members. I was also to make sure the Makefile generalized to our purpose and make sure everything worked

4.3 Time Line

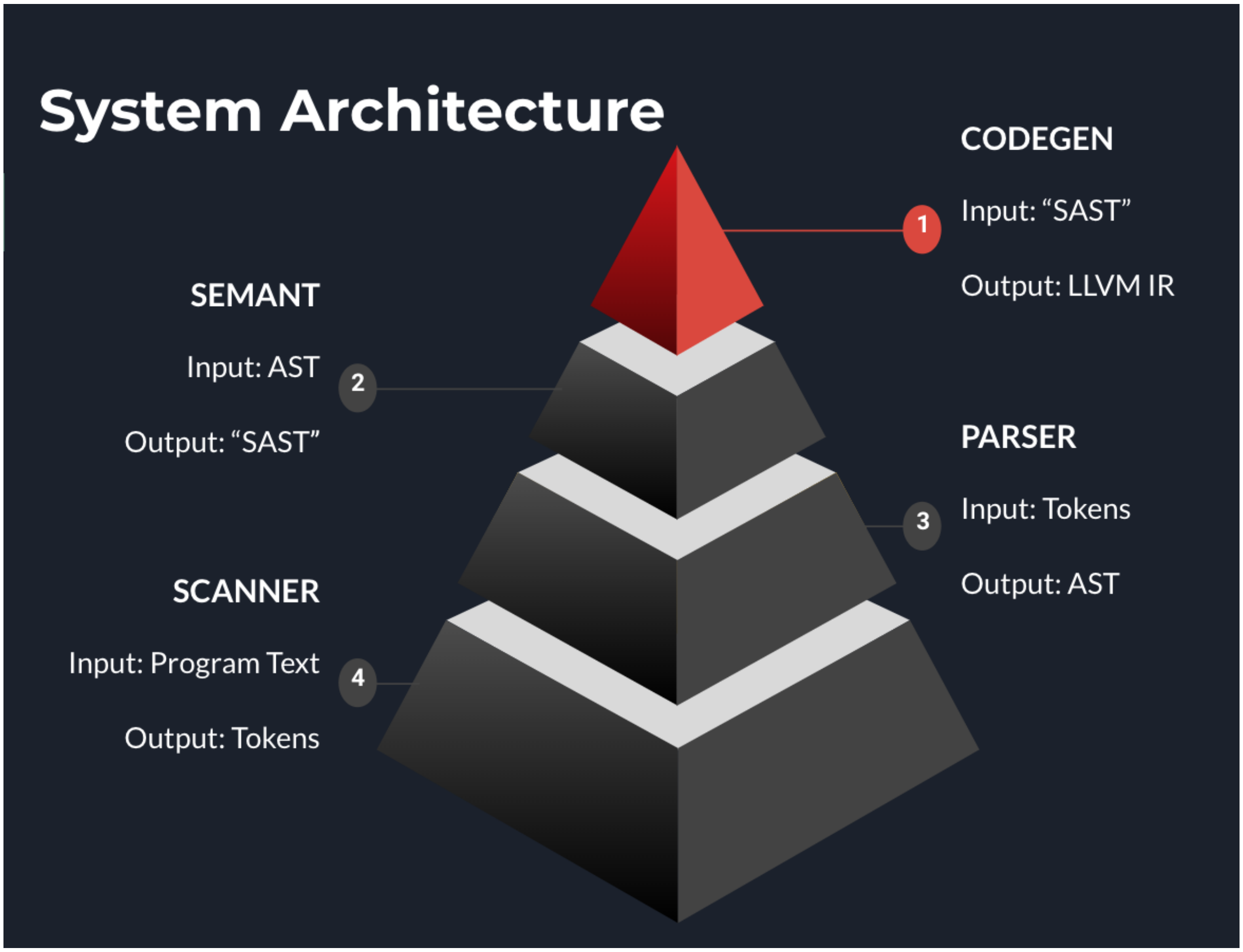


4.4 Software Development Environment

Our projects were done in individual machines with Github as our source control repository. Everyone made sure to use the same version of LLVM (5.0.0). We made different branches for each features/modules and completed them in individual chxunks and made pull requests to merge them into the master branch. Each pull requests were reviewed by all members of the team before being merged.

5 Architectural Design

5.1 Block Diagram of Translator Architecture



Our translator architecture was largely borrowed from microc and there is no notable difference to be discussed.

6 Test Plan

6.1 Test Suite Listing

```
==> fail-declare1.loon <==
int main() {
    int x
    int x
    return 0
}

==> fail-declare2.loon <==
void main() {
    x = 5
}
==> fail-declare3.loon <==
int main() {
    bool control
    int x
    x = 5
    control = false
    if (control) {
        printJSON("nah")
    } else {
        printJSON("yea")
        int x
        x = 5
    }
    int x
    x = 5
}
==> fail-followreturn.loon <==
int main(){
  int x
  x = 5
  return 0
  x = 6
}
==> fail-global1.loon <==
int c
int c
int main(){
  return 0
}
==> fail-illegaladd.loon <==
void main() {
  int x
  x = 5
  string y
  y = "this won't add"
  y + x
}
==> fail-illegalassign1.loon <==
void main(){
```

```
  string x
  x = 5
}
==> fail-illegalassign2.loon <==
void main(){
  bool x
  x = true
  bool y
  y = 0
}
==> fail-mainmissing.loon <==
void lol() {
  int x
  x = 5
}

==> fail-returntype1.loon <==
int main(){
  return "lol"
}
==> fail-returntype2.loon <==
void main() {
  return 0
}
==> fail-unmatched.loon <==
void loon() {
==> fail-voidglobal.loon <==
int c
void a
int main(){
  return 0
}
==> test-access-assign.loon <==
void main(){
  string test
  test = "hope"
  char c1
  c1 = test[0]

  array arr
  arr = ["merry" , 5, test , c1]
  printJSON(arr[0])
  arr[0] = "fur"
  int test2
  test2 = 5
  printJSON(arr[0])
  printJSON(arr[1])
  printJSON(arr[2])
  printJSON(arr[3])

}

==> test-access.loon <==
void main(){
```

```
  string testAcc
  testAcc = "dominant"
  char secChar
  secChar = testAcc[0]
  printJSON("Should print d character: ", secChar)
  printJSON("Should print t character: ", testAcc[7])
}


==> test-array-init.loon <==
void main(){
  string test
  test = "hope"
  char c1
  c1 = test[0]
  array arr
  arr = ["merry" , 5, test, c1]
  array arr2
  arr2 = [arr, 10]
  int test2
  test2 = 5
  printJSON(arr[0])
  printJSON(arr[1])
  printJSON(arr[2])
  printJSON(arr[3])
  printJSON(test[3])
}

==> test-array-nested-assignment.loon <==
void main(){
  array test
  test = [["frosty", 12], 5, "fresh"]
  string freshTest
  freshTest = test[2]
  printJSON("Result is: ", freshTest)
}

==> test-array-nested.loon <==
void main(){
  array test
  test = [["frosty", 12, ["ultimate"]], 5, "fresh"]
  printJSON(test[0][2][0])
}

==> test-dec-and-assign.loon <==
void main() {
    int x = 5
    printJSON(x)
}

==> test-deref.loon <==
void main(){
    pair<int> p
    p = <"hello", 50>
```

```
    int val
    val = *p
    printJSON(val)

    pair<string> ps
    ps = <"hello", "world!">
    string world
    world = *ps
    printJSON(world)

}

==> test-empty.loon <==
void main() {
}

==> test-for.loon <==
void main() {
    int i
    for (i = 0; i < 5; i = i + 1) {
        printJSON(i)
    }
}

==> test-func.loon <==
void f() {
  int x
  x = 2
  printJSON(x)
}

void main() {
  f()
}

==> test-func2.loon <==
string f(int a, string s) {
    printJSON(a)
    string str
    str = "funny"
    printJSON(s)
    return str
}

void main () {
    string test
    test = f(3, "hey")
    printJSON(test)
}

==> test-idprint.loon <==
void main(){
  string testAcc
  testAcc = "dominant "
```

```
  int test
  test = 5
  printJSON(testAcc, test, " over")
}

==> test-if.loon <==
void main() {
    bool control
    control = false
    if (control) {
        printJSON("this should not print")
    } else {
        printJSON("this should print")
    }
    control = true
    if (control) {
        printJSON("this should also print")
    } else {
        printJSON("this should also not print")
    }
    if (control) {
        printJSON("this should finally print")
    }
    if (!control) {
        printJSON("this should finally not")
    }
}

==> test-int-2.loon <==
void main() {
    int x
    x = 5
    printJSON(x)
}

==> test-int-3.loon <==
void main(){
  string test
  test = "fuck"

  string next
  next = " that"

  printJSON(test, next)
  int j
  j = 5
  printJSON(100)

  string strcat
  strcat = "new " + "concat"
  printJSON(strcat)
}
```

```
==> test-int.loon <==
void main() {
    int x
    x = 5
    printJSON(x)
}

==> test-nested-pair.loon <==
void main() {
    pair<pair<int>> rp
    rp = <"nested", <"pairs", 5>>
    int nested_contents
    nested_contents = **rp
    printJSON(nested_contents)
}

==> test-pair.loon <==
void main(){
    pair<int> p
    p = <"hello", 50>
}

==> test-pjson.loon <==
int main(){
  string testCat
  testCat = "The first str " + "the second str"
  printJSON(testCat)
  printJSON(2+2)
  return 0
}
==> test-print-2.loon <==
void main() {
    printJSON(5)
    printJSON(4)
}

==> test-print.loon <==
void main() {
    printJSON(5)
}

==> test-rec.loon <==
int fac(int n) {
    if (n == 1) {
        return 1
    }
    return n * fac(n - 1)
}

void main() {
    int x
    x = 5
    int rec_x
    rec_x = fac(x)
```

```
    printJSON(rec_x)
}

==> test-str-cat.loon <==
void main(){
  string testCat
  testCat = "The first str " + "the second str"
  printJSON(testCat)
}

==> test-while-2.loon <==
void main() {
  int x
  x = 3

  while (x > 1) {
    printJSON(1)

    x = x - 1

  }
}
==> test-while-3.loon <==
void main() {
  int x
  x = 3

  while (x > 1) {
    printJSON(1)

    x = x - 1

  }
}
==> test-while.loon <==
void main() {

    int x
    x = 1
    while (x < 3) {
        printJSON(2)
  x = x + 1
    }

    bool y
    y = true
    while (y) {
        printJSON(4)
        y = false
    }
}
```

6.2 Automation

```
#!/bin/sh

# Regression testing script for MicroC
# Step through a list of files
#  Compile, run, and check the output of each expected-to-work test
#  Compile and check the error of each expected-to-fail test

# Path to the LLVM interpreter
LLI="/usr/local/opt/llvm/bin/lli"

# Path to the LLVM compiler
LLC="/usr/local/opt/llvm/bin/llc"

# Path to the C compiler
CC="cc"

# Path to the microc compiler.  Usually "./microc.native"
# Try "_build/microc.native" if ocamlbuild was unable to create a symbolic link.
MICROC="./loon.native"
#MICROC="_build/microc.native"

# Set time limit for all operations
ulimit -t 30

globallog=testall.log
rm -f $globallog
error=0
globalerror=0

keep=0

Usage() {
    echo "Usage: testall.sh [options] [.mc files]"
    echo "-k    Keep intermediate files"
    echo "-h    Print this help"
    exit 1
}

SignalError() {
    if [ $error -eq 0 ] ; then
	echo "FAILED"
	error=1
    fi
    echo "  $1"
}

# Compare <outfile> <reffile> <difffile>
# Compares the outfile with reffile.  Differences, if any, written to difffile
Compare() {
    generatedfiles="$generatedfiles $3"
    echo diff -b $1 $2 ">" $3 1>&2
    diff -b "$1" "$2" > "$3" 2>&1 || {
```

```
  SignalError "$1 differs"
  echo "FAILED $1 differs from $2" 1>&2
    }
}

# Run <args>
# Report the command, run it, and report any errors
Run() {
    echo $* 1>&2
    eval $* || {
  SignalError "$1 failed on $*"
  return 1
    }
}

# RunFail <args>
# Report the command, run it, and expect an error
RunFail() {
    echo $* 1>&2
    eval $* && {
  SignalError "failed: $* did not report an error"
  return 1
    }
    return 0
}

Check() {
    error=0
    basename=`echo $1 | sed 's/.*\\///
                             s/.loon//'`
    reffile=`echo $1 | sed 's/.loon$//'`
    basedir="`echo $1 | sed 's/\/[^\/]*$//'`/."

    echo -n "$basename..."

    echo 1>&2
    echo "###### Testing $basename" 1>&2

    generatedfiles=""

    generatedfiles="$generatedfiles ${basename}.ll ${basename}.s ${basename}.exe ${basename}.out" &&
    Run "$MICROC" "<" "$1" ">" "${basename}.ll" &&
    Run "$LLC" "${basename}.ll" ">" "${basename}.s" &&
    Run "$CC" "-o" "${basename}.exe" "${basename}.s" "loon_scanf.o" &&
    ./${basename}.exe > ${basename}.out
    Compare ${basename}.out ${reffile}.out ${basename}.diff

    # Report the status and clean up the generated files

    if [ $error -eq 0 ] ; then
      if [ $keep -eq 0 ] ; then
          rm -f $generatedfiles
      fi
```

```
      echo "OK"
      echo "###### SUCCESS" 1>&2
    else
      echo "###### FAILED" 1>&2
      globalerror=$error
    fi
}

CheckFail() {
    error=0
    basename=`echo $1 | sed 's/.*\\///
                             s/.loon//'`
    reffile=`echo $1 | sed 's/.loon$//'`
    basedir="`echo $1 | sed 's/\/[^\/]*$//'`/."

    echo -n "$basename..."

    echo 1>&2
    echo "###### Testing $basename" 1>&2

    generatedfiles=""

    generatedfiles="$generatedfiles ${basename}.err ${basename}.diff" &&
    RunFail "$MICROC" "<" $1 "2>" "${basename}.err" ">>" $globallog &&
    Compare ${basename}.err ${reffile}.err ${basename}.diff

    # Report the status and clean up the generated files

    if [ $error -eq 0 ] ; then
  if [ $keep -eq 0 ] ; then
      rm -f $generatedfiles
  fi
  echo "OK"
  echo "###### SUCCESS" 1>&2
    else
  echo "###### FAILED" 1>&2
  globalerror=$error
    fi
}

while getopts kdpsh c; do
    case $c in
  k) # Keep intermediate files
      keep=1
      ;;
  h) # Help
      Usage
      ;;
    esac
done

shift `expr $OPTIND - 1`

LLIFail() {
```

```
  echo "Could not find the LLVM interpreter \"$LLI\"."
  echo "Check your LLVM installation and/or modify the LLI variable in
      testall.sh"
  exit 1
}

which "$LLI" >> $globallog || LLIFail

if [ ! -f loon_scanf.o ]
then
    echo "Could not find loon_scanf.o"
    echo "Try \"make loon_scanf.o\""
    exit 1
fi

if [ $# -ge 1 ]
then
    files=$@
else
    files="tests/test-*.loon tests/fail-*.loon"
fi

for file in $files
do
    case $file in
  *test-*)
      Check $file 2>> $globallog
      ;;
  *fail-*)
      CheckFail $file 2>> $globallog
      ;;
  *)
      echo "unknown file type $file"
      globalerror=1
      ;;
    esac
done

exit $globalerror
```

7 Lessons Learned

7.1 Jack Ricci

As advertised on day one of the course, the most difficult part of the project really had nothing to with tokenizing source code text, parsing the tokenized input into an abstract syntax tree, or generating intermediate representation code using the Ocaml Llvm bindings. Rather, the most challenging aspect of the project was building a group that could effectively agree upon a vision for the language and then implement that vision in programming the compiler.

With that in mind, this project was a tremendously humbling experience for me from a project management standpoint. I came to realize that without formulating a fairly detailed plan for what needs to be implemented, how it should be implemented, when each item should be implemented, and who should be implementing each item, a software project's potential to get derailed will increase drastically. I consistently failed to recognize the optimal way to handle the respective strengths and weaknesses of our group members and failed to identify the most natural order to implement deliverables. Furthermore, in every project of considerable size, there is an astoundingly high chance that obstacles will crop up throughout the course of the project that will force a change of plans in both what and how the project is executed. Perhaps the most important area in which I damaged our group's progress was in my inability to identify the correct counterreaction to each of the issues that arose over the course of the project. Our language could have been substantially more powerful and true to its original vision had I done a better job of adapting the language design once we realized that some of our original LRM goals would not be met. These will serve as some really strong higher-level project concepts for me to focus on during future software projects that I am involved in.

My advice to future students is two-fold:
Firstly, when you craft your LRM and the vision for your language, have both an ambitious version and a simple version of the language. This will allow you to initially aim for the idealized vision, but it will also allow the group to remain on the same page and still produce something powerful should you not be able to accomplish all your goals for the language (N.B.: you won't accomplish them all).
Lastly, the most fun part about this project is that it is truly a software engineering project. You're in a group, you need to coordinate and constantly keep others apprised of your additions to ensure continuity, you need to quickly understand the behavior of a new programming language, and you need to be able to utilize APIs that have limited online documentation. As a result, make sure that the team that you form consists almost entirely of strong coders. There is absolutely a huge difference between strong coders and strong students, and in order to meet the goals of your LRM, you will need to have a team in which each member can turn ideas into running code on their own. The coding challenges and design decisions in this project are incredibly enjoyable, but a certain level of commitment to learning Ocaml, Ocamllex, Ocamlyacc, and Llvm is certainly involved if you hope to build a strong compiler.

7.2 Niles Christensen

Trust that git and the test suite allow for radical changes to code, and know that if things go wrong you can always revert to a recent working state. Git is enormously powerful, and, especially when combined with an external service like github, it makes it very hard to cause any sort of lasting damage to your project. Remembering this is very liberating, and allows for a more aggressive sort of coding that is very good for rapid progress on large challenges (like building a compiler).

On that note, I would recommend learning about best practices when working with git. Agree on a tabs vs. spaces convention for your team and keep it consistent, and remember to NEVER PUSH TO MASTER. Pull requests are your friend.
Also, Professor Edwards mentions this, but make an automated testing suite and make it large. Include as many tests as possible and make sure never to commit anything, no matter how sure you are in it, until after you've made all your files from scratch and run all the tests. You'd be surprised what seemingly unrelated pieces of code can break.
Finally, and this is related to testing, anything repetitive can and should be automated. You'll have to do these things again and again, and you will stop doing them around the 50th time unless it's easy. You're lazier than you think, so take the time now when you have motivation to make it easier for the you during finals.

7.3 Kyle Hughes

As mentioned by Jack, I too would agree that the most significant takeaway from this project was the ability to function well as a group, which was especially interesting when presented with such large technical challenges over the course of the semester. I would qualify this project as one of the biggest challenges of my undergraduate career, but in the sense that it was the one that pushed me beyond my comfort zone the most (in the best way possible). For new students, I'd suggest that you persist throughout the semester with writing productive code while actively trying to break test cases that exist within an organically-evolving test suite. With this, I think it may be useful to explore development pathways beyond the structural foundation that Micro-C provides, because this may enable for the creation of more unique structures as you move forward with developing complex types in your language.

7.4 Chelci Erin Houston-Burroughs

Where do I even start? I want to make a pros vs cons list but the project lifecycle was way more complicated than that, such that perhaps a pro brought about a con and a con brought about a pro. For example I joined a group of really dope people with really dope skills that I felt like I semi-knew and would probably enjoy (hey some of them are my friends) but in that comfort came a general lack of holding each other accountable until the very end which inherently put way more pressure one everyone's lives than was necessary. So my first piece of advice is join a smaller group of people who perhaps don't know each other as well(there is less probability that folks will slack off in the comfort of having multiple team members pick up the work).Our core dynamic is where I feel most of the frustrations were arising from. Second piece of advice: volunteer to take lead on stuff you may or may not know how to do and take a chance to learn – but if you do this make sure you actually spend the time trying to figure how to work it out. I spent many a night with our TA to try to learn and contribute to some of the more complicated aspects of our project and you don't know how great it felt to actually make some semi complicated stuff actually work. Our TA Lizzie really enhanced my experience by showing me how to intelligently break things in order to debug more effectively. It was a pain in the ass, but I learned some cool functional programming tricks that maybe I can use to impress some technical interviewer with in the future. who knows? Last but certainly not least: ASK FOR CLARITY and SUM UP DELIVERABLES after EVERY MEETING. I feel like that's self explanatory so I think I'm done.

7.5 Habin Lee

This was by far the most difficult "programming" project I had ever done. There were a lot of aspects that made it hard – the actual materials and the design question of the features were difficult to visualize, the architecture of how LLVM and ocaml was difficult to grasp and above all, keeping in pace with the group was extremely difficult for me. All of these difficulties together made this project one of the most monolithically stacked, difficult-to-attack problem. At certain points, with certain amount of effort I found myself coming short to be able to keep up with the group's pace, and only be able to mind smaller features. That being said I would like to use this little sentence to apologize to my members for slowing down the team a few times. However, I did find this a great experience to learn from much talented developers and when to properly seek for help – our TA Lizzie was incredibly helpful in that frontier, and so were our group members. Also, I found that this project was the case where the available resources online were hardest to generalize to our use-case and so figuring out how to something without the usual help from Stack Overflow and Co. was nice, even if it were figuring out some nitty gritty oCaml specifics. As for the message to the future groups, I would say try to stay connected to the group as much as possible and get what's going on at any given moment. That way, you'll know what you know and what you don't know and know what to do next!

8 Appendix

```
==> Makefile <==
# Contributer: Habin
.PHONY : all
all:  loon.native loon_scanf.o

.PHONY : loon.native
loon.native :
  ocamlbuild -r -use-ocamlfind -pkgs llvm,llvm.analysis -cflags -w,+a-4 \
    loon.native

.PHONY : clean
clean :
  ocamlbuild -clean
  rm -rf testall.log *.diff loon scanner.ml parser.ml parser.mli
  rm -rf loon_scanf
  rm -rf *.cmx *.cmi *.cmo *.o *.s *.ll *.out *.exe

# semant.cmx goes before loon.cmx when it's there
OBJS = ast.cmx codegen.cmx parser.cmx scanner.cmx loon.cmx

loon: $(OBJS)
  ocamlfind ocamlopt -linkpkg -package llvm -package llvm.analysis $(OBJS)
      -o loon


scanner.ml : scanner.mll
  ocamllex scanner.mll

parser.ml parser.mli : parser.mly
  ocamlyacc parser.mly

%.cmo : %.ml
  ocamlc -c $<

%.cmi : %.mli
  ocamlc -c $<

%.cmx : %.ml
  ocamlfind ocamlopt -c -package llvm $<

#Tests scanf example
loon_scanf: loon_scanf.c

# Generated by ocamldep *.ml *.mli
ast.cmi :
codegen.cmo : ast.cmi
codegen.cmx : ast.cmi
loon.cmo : scanner.cmo parser.cmi codegen.cmo ast.cmi
loon.cmx : scanner.cmx parser.cmx codegen.cmx ast.cmi
parser.cmo : ast.cmi parser.cmi
parser.cmx : ast.cmi parser.cmi
parser.cmi : ast.cmi
```

```
scanner.cmo : parser.cmi
scanner.cmx : parser.cmx

==> scanner.mll <==
(* Ocamllex scanner for LOON
Authors:
Professor S. Edwards
J. Ricci
N.Christensen
*)

{ open Parser }

rule token = parse
  [' ' '\t' '\r'] { token lexbuf } (* Whitespace *)
  | '('                        {(*ignore(print_endline "Saw LPAREN") ;*)
     LPAREN }
  | ')'                        {(*ignore(print_endline "Saw RPAREN") ;*)
     RPAREN }
  | "{|"                       { OPEN_JSON }
  | "|}"                       { CLOSE_JSON }
  | ['\n']*'{'['\n']*          {(*ignore(print_endline "Saw LBRACE") ;*)
     LBRACE }
  | '}'['\n']*                 {(*ignore(print_endline "Saw RBRACE") ;*)
     RBRACE }
  | '['                        { (*ignore(print_endline "Saw LBRACKET") ;*)
     LBRACKET }
  | ']'                        {(*ignore(print_endline "Saw LBRACKET") ;*)
     RBRACKET }
  | ['\n']+                    { (*ignore(print_endline "Saw SEQ") ; *) SEQ
     }
  | ','                        { COMMA }
  | '+'                        { PLUS }
  | '-'                        { MINUS }
  | '*'                        { TIMES }
  | '/'                        { DIVIDE }
  | '='                        { ASSIGN }
  | ';'                        { SEMI }
  | ':'                        { COLON }
  | "=="                       { EQ }
  | "!="                       { NEQ }
  | '<'                        { LT }
  | "<="                       { LEQ }
  | ">"                        { GT }
  | ">="                       { GEQ }
  | "&&"                       { AND }
  | "||"                       { OR }
  | '|'                        { PIPE }
  | "!"                        { NOT }
  | "if"                       { IF }
  | "else"                     { ELSE }
  | "for"                      { FOR }
  | "while"                    { WHILE }
  | "return"                   { RETURN }
```

```
  | "char"                   {(*ignore(print_endline "Saw CHAR") ;*) CHAR }
  | "int"                    {(*ignore(print_endline "Saw INT") ;*) INT }
  | "bool"                 { BOOL }
  | "void"                   {(*ignore(print_endline "Saw VOID") ;*) VOID }
  | "json"                   { JSON }
  | "pair"                   { PAIR }
  | "string"               {(*ignore(print_endline "Saw STRING") ;*) STRING }
  | "array"                  { (*ignore(print_endline "Saw ARRAY") ;*) ARRAY }
  | "true"                   { TRUE }
  | "false"                  { FALSE }
  (* StringLit currently allows you to form strings over multiple lines*)
  | '\"' [^'\"']* '\"' as lxm    {(*ignore(print_endline "Saw STRINGLIT") ;*) STRINGLIT(String.sub lxm 1 (String.length lxm - 2)) }
  | ['0'-'9']+ as lxm        { LITERAL(int_of_string lxm) }
  | ['a'-'z' 'A'-'Z']['a'-'z' 'A'-'Z' '0'-'9' '_']* as lxm {(* ignore( print_endline "Saw ID") ;*) ID(lxm) }
  | eof { EOF }
  | _ as char { raise (Failure("illegal character " ^ Char.escaped char)) }

  and comment = parse
    "*/" { token lexbuf }
    | _    { comment lexbuf }


==> parser.mly <==
/* Ocamlyacc parser for LOON. Written by Niles, Jack, Chelci, and Habin */

%{
open Ast
%}


%token LPAREN RPAREN LBRACE RBRACE
%token LBRACKET RBRACKET
%token OPEN_JSON CLOSE_JSON
%token SEQ COMMA SEMI COLON PIPE
%token PLUS MINUS TIMES DIVIDE ASSIGN
%token EQ NEQ LT LEQ GT GEQ AND OR NOT TRUE FALSE
%token IF ELSE FOR WHILE RETURN

%token INT BOOL STRING VOID PAIR CHAR ARRAY JSON

%token <int> LITERAL
%token <char> CHARLIT
%token <string> STRINGLIT
%token <string> ID
%token EOF

%nonassoc NOELSE
```

```
%nonassoc ELSE
%right ASSIGN
%left OR
%left AND
%left EQ NEQ
%left LT GT LEQ GEQ
%left PLUS MINUS
%left TIMES DIVIDE
%right NOT NEG

%start program
%type <Ast.program> program

%%

program:
  decls EOF { $1 }

decls:
   /* nothing */ { [], [] }
 | decls vdecl { ($2 :: fst $1), snd $1 }
 | decls fdecl { fst $1, ($2 :: snd $1) }
/* | decls importDecl { fst $1, ($2 :: snd $1) } */

/* Import Standard library functions. LOOK UP RECURSIVE SCAN CALL*/
/*Need to define Import function, which uses file path and functionID to
return the an func_decl structure representing the desired function. Import(file_path, func_id)
can be located in a separate module, which we open at the top. A future optimization might
build a map of <file_name, list.String func_id> for easy access, rather than perform n fileOpenings
where n is the number of import statements */
/*importDecl:
    FROM STRINGLIT IMPORT ID SEQ { Import($2, $4) }
*/ /* Importing from libraries*/

fdecl:
  typ ID LPAREN formals_opt RPAREN LBRACE stmt_tuple RBRACE
    { { primitive = $1;
    fname = $2;
    formals = $4;
    locals = List.rev (fst $7);
    body = List.rev (snd $7) } }

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

formal_list:
    typ ID                   { [($1,$2)] }
  | formal_list COMMA typ ID { ($3,$4) :: $1 }

typ:
```

```
    INT { Int }
  | BOOL { Bool }
  | VOID { Void }
  | STRING { String}
  | PAIR LT typ GT { Pair $3 }
  | CHAR { Char }
  | ARRAY { Array }
  | JSON { Json }

/*vdecl_list:
    nothing     { [] }
  | vdecl_list vdecl { $2 :: $1 } */

vdecl:
   typ ID SEQ { ($1, $2) }

stmt_tuple:
    /* nothing - Make VDecls list and true statements list */  { ( [], [])
      }
  | stmt_tuple stmt { (fst $1, $2 :: (snd $1)) }
  | stmt_tuple vdecl { ($2 :: (fst $1), snd $1) }

stmt:
    expr SEQ { Expr $1 }
  | RETURN SEQ { Return Noexpr }
  | RETURN expr SEQ { Return $2 }
  | LBRACE stmt_tuple RBRACE { Block(List.rev (snd $2)) }
  | IF LPAREN expr RPAREN stmt %prec NOELSE { If($3, $5, Block([])) }
  | IF LPAREN expr RPAREN stmt ELSE stmt    { If($3, $5, $7) }
  | FOR LPAREN expr_opt SEMI expr SEMI expr_opt RPAREN stmt
     { For($3, $5, $7, $9) }
  | WHILE LPAREN expr RPAREN stmt { While($3, $5) }

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

expr:
    LITERAL          { Literal($1) }
  | CHARLIT      { CharLit($1) }
  | STRINGLIT        { StringLit($1) }
  | TRUE             { BoolLit(true) }
  | FALSE            { BoolLit(false) }
  | ID               { Id($1) }
  | LT expr COMMA expr GT { PairLit($2, $4) }
  | 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) }
  | MINUS expr %prec NEG { Unop(Neg, $2) }
  | NOT expr         { Unop(Not, $2) }
  | TIMES expr       { Unop(Deref, $2) }
  | ID access_list_opt ASSIGN expr   { Assign($1, List.rev $2, $4) }
  | ID LPAREN actuals_opt RPAREN { Call($1, $3) }
  | LPAREN expr RPAREN { $2 }
  | ID access_list { Access ($1, List.rev $2) }
  | LBRACKET actuals_opt RBRACKET {ArrayLit($2)}
  | OPEN_JSON pairs_opt CLOSE_JSON   { JsonLit($2) }

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

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

pairs_opt:
    /* nothing */ { [] }
  | pairs_list    { List.rev $1 }

pairs_list:
    json_pair                    { [$1] }
  | pairs_list COMMA json_pair  { $3 :: $1 }

json_pair:
    expr COLON expr              { ($1, $3) }

access_list_opt:
    /* nothing */ { [] }
  | access_list   { $1 }

access_list:
    LBRACKET expr RBRACKET { [ $2 ] }
  | access_list LBRACKET expr RBRACKET { $3 :: $1 }

==> ast.mli <==
(* LOON ast.mli. Written by Chelci, Jack, Niles, and Kyle *)
module L = Llvm

type op =
    (* numerical operators *)
    | Add | Sub | Mult | Div | Equal
    (* Relational operators *)
    | Neq | Less | Leq | Greater | Geq
    (* boolean operators *)
    | And | Or

type uop =
    | Neg | Not | Deref
```

```
type typ =
    | Int | Bool | Void | String | Pair of typ | Char | Array | Json

type bind = typ * string

type expr =
    | Literal of int
    | BoolLit of bool
    | CharLit of char
    | StringLit of string
    | PairLit of expr * expr
    | Id of string
    | Noexpr
    | Binop of expr * op * expr
    | Unop of uop * expr
  | Assign of string * expr list * expr
    | Call of string * expr list
  | Access of string * expr list
    | ArrayLit of expr list
    | JsonLit of (expr * expr) list

type stmt =
    | Block of stmt list
    | Expr of expr
    | If of expr * stmt * stmt
    | For of expr * expr * expr * stmt
    | While of expr * stmt
    | Return of expr

type func_decl = {
    primitive : typ;
    fname     : string;
    formals   : bind list;
    locals    : bind list;
    body    : stmt list;
}

type program = bind list * func_decl list

(** Wrapper around array value types *)
type val_type =
    | Val of L.lltype
    | Val_list of val_type list

val string_of_program : bind list * func_decl list -> string

val zero_of_typ : typ -> expr

(* Pretty-printing functions *)

val string_of_op : op -> string

val string_of_uop : uop -> string
```

```
val string_of_expr : expr -> string

val string_of_stmt : stmt -> string

val string_of_typ : typ -> string

val string_of_vdecl : bind -> string

val string_of_fdecl : func_decl -> string

val fmt_of_lltype : string -> string

val get_val_type : L.llcontext -> int list -> val_type -> L.lltype

val set_val_type :  L.llcontext -> val_type list -> val_type -> int list
    -> val_type list

==> ast.ml <==
(* LOON ast.ml. Written by Chelci, Jack, Niles, Kyle and Habin *)
module L = Llvm

type op =
  (* numerical operators *)
  | Add | Sub | Mult | Div | Equal
  (* Relational operators *)
  | Neq | Less | Leq | Greater | Geq
  (* boolean operators *)
  | And | Or

type uop =
    | Neg | Not | Deref

type typ =
    | Int | Bool | Void | String | Pair of typ | Char | Array | Json

type bind = typ * string

type expr =
    | Literal of int
    | BoolLit of bool
    | CharLit of char
    | StringLit of string
    | PairLit of expr * expr
    | Id of string
    | Noexpr
    | Binop of expr * op * expr
    | Unop of uop * expr
  | Assign of string * expr list * expr
    | Call of string * expr list
    | Access of string * expr list
    | ArrayLit of expr list
    | JsonLit of (expr * expr) list

```

```
type stmt =
    | Block of stmt list
    | Expr of expr
    | If of expr * stmt * stmt
    | For of expr * expr * expr * stmt
    | While of expr * stmt
    | Return of expr

type func_decl = {
    primitive : typ;
    fname     : string;
    formals   : bind list;
    locals    : bind list;
    body    : stmt list;
}

type program = bind list * func_decl list

(* Pretty-printing functions *)

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

let string_of_uop = function
    Neg -> "-"
  | Not -> "!"
  | Deref -> "*"

let rec string_of_expr = function
    Literal(l) -> string_of_int l
  | CharLit(c) -> Char.escaped c
  | StringLit(s) -> s
  | BoolLit(true) -> "true"
  | BoolLit(false) -> "false"
  | PairLit(k, v) -> string_of_expr k ^ ", " ^ string_of_expr v
  | Id(s) -> s
  | Binop(e1, o, e2) ->
      string_of_expr e1 ^ " " ^ string_of_op o ^ " " ^ string_of_expr e2
  | Unop(o, e) -> string_of_uop o ^ string_of_expr e
  | Assign(v, lst, e) -> ignore(v, lst, e);(*v ^ "[" ^ (List.map
     string_of_expr lst) ^ "]" ^ " = " ^ string_of_expr e *) "nah"
  | Access(id, indx_list) -> ignore(id, indx_list);(*id ^ "[" ^ (List.map
     string_of_epr indx_lst) ^ "]"*) "nah2"
```

```
  | Call(f, el) ->
      f ^ "(" ^ String.concat ", " (List.map string_of_expr el) ^ ")"
  | Noexpr -> ""
  | ArrayLit(l) -> "array: [" ^ String.concat ", " (List.map string_of_expr l) ^ "]"
  | JsonLit(l) ->
          let handle_tuples (first, second) = string_of_expr first ^ ", " ^ string_of_expr second in
          "array: [" ^ String.concat ", " (List.map handle_tuples l) ^ "]"

let rec string_of_stmt = function
    Block(stmts) ->
      "{\n" ^ String.concat "" (List.map string_of_stmt stmts) ^ "}\n"
  | Expr(expr) -> string_of_expr expr ^ "\n";
  | Return(expr) -> "return " ^ string_of_expr expr ^ ";\n";
  | If(e, s, Block([])) -> "if (" ^ string_of_expr e ^ ")\n" ^ string_of_stmt s
  | If(e, s1, s2) ->  "if (" ^ string_of_expr e ^ ")\n" ^
      string_of_stmt s1 ^ "else\n" ^ string_of_stmt s2
  | For(e1, e2, e3, s) ->
      "for (" ^ string_of_expr e1  ^ " ; " ^ string_of_expr e2 ^ " ; " ^
      string_of_expr e3  ^ ") " ^ string_of_stmt s
  | While(e, s) -> "while (" ^ string_of_expr e ^ ") " ^ string_of_stmt s

let string_of_typ = function
    Int -> "int"
  | Bool -> "bool"
  | Void -> "void"
  | String -> "string"
  | Array -> "array"
  | Pair _ -> "pair"
  | Char -> "char"
  | Json -> "json"

let string_of_vdecl (t, id) = string_of_typ t ^ " " ^ id ^ "\n"

let string_of_fdecl fdecl =
  string_of_typ fdecl.primitive ^ " " ^
  fdecl.fname ^ "(" ^ String.concat ", " (List.map snd fdecl.formals) ^
  ")\n{\n" ^
  String.concat "" (List.map string_of_vdecl fdecl.locals) ^
  String.concat "" (List.map string_of_stmt fdecl.body) ^
  "}\n"

let string_of_program (vars, funcs) =
  String.concat "" (List.map string_of_vdecl vars) ^ "\n" ^
  String.concat "\n" (List.map string_of_fdecl funcs)

(* Function to return the zero-value for each type *)
let zero_of_typ = function
    Int -> Literal(0)
    | Bool -> BoolLit(false)
    | String -> StringLit("")
    | Char -> CharLit(Char.chr 0)
```

```
    (* | Pair(p_type) -> PairLit(StringLit(""), Literal(0)) *)
    | Array -> ArrayLit([Literal(0)])
    | Json -> JsonLit([(StringLit(""), Literal(0))])
    |_ -> Literal(0)

let fmt_of_lltype =  function
    "i8*" -> "%s"
  | "i8" -> "%c"
  | "i32" -> "%d"
  | _ -> "%d"

(** Wrapper around array value types *)
type val_type =
    | Val of L.lltype
    | Val_list of val_type list

(** Check if array type is value or nested array *)
(*let is_val = function
  Val(v) -> ignore(v); true
  | Val_list(v_list) -> ignore(v_list); false
  | _ -> false *)

(** Get type of val at specified indx pos.
    indx_val: list of ints specifying indx pos
  returns lltype *)
let rec get_val_type context indx_list = function
    Val(v) -> (*ignore(print_endline("GET_VAL: Reached lltype: " ^ (L.string_of_lltype v)));*) v
  | Val_list(v_list) ->
    (* Get nth value, call function again on it *)
    if indx_list = [] then (
      ignore(print_endline("GET_VAL: Not accessing further - return i8***"));
      L.pointer_type(L.pointer_type(L.pointer_type (L.i8_type context))) )
    else (let this_indx = List.hd indx_list in
        let next_val = List.nth v_list this_indx in (*ignore(print_endline("GET_VAL: Array, call again - list size: " ^ (string_of_int (List.length indx_list))));*)
      get_val_type context (List.tl indx_list) next_val)

(** Set type of val at specified indx pos
    types_lst: previous list of types
    new_type: val_type of new type
    indxs_int_lst: list of index positions to scan*)
let rec set_val_type context types_lst new_type indxs_int_lst =
  let cur_indx = List.hd indxs_int_lst
  and rem_indxs = List.tl indxs_int_lst in

  (* Function that mapi calls to build new list*)
  let map_func i orig_elem_type =
    (* Match current indx and cannot index any further - replace this type *)
    if (i = cur_indx && rem_indxs = []) then(
    Val(L.pointer_type (match new_type with
```

```
        | Val v -> v
        | _ -> ignore(print_endline("SET_VAL_TYPE: Error: Matched with current index, and still have more indexing to do, but value is not indexable")); L.i32_type context) ))
    (* Match current indx and can index futher - call again on remaining indexes as orig_elem_types must be list *)
    else( if i = cur_indx then (
      let true_type = (match orig_elem_type with
        | Val_list nxt_lst -> nxt_lst
        | _ -> ignore(print_endline("SET_VAL_TYPE: Error: Matched with current index, and still have more indexing to do, but value is not indexable")); []) in
    Val_list(set_val_type context true_type new_type rem_indxs) )
    (* Otherwise no match on indx, so return previous value *)
    else orig_elem_type) in
  List.mapi map_func types_lst


(** read JSON value from a string *)
(*val from_string : ?buf:Bi_outbuf.t -> ?fname:string -> ?lnum:int -> string -> json *)
(** read JSON value from a file *)
(*val from_file : ?buf:Bi_outbuf.t -> ?fname:string -> ?lnum:int -> string -> json*)
(** read JSON value from channel *)
(*val from_channel : ?buf:Bi_outbuf.t -> ?fname:string -> ?lnum:int -> in_channel -> json
val from_string  : string     -> json
val from_file    : string     -> json
val from_channel : in_channel -> json
*)

==> semant.ml <==
(* LOON Compiler Semantic Checking *)
(* Authors: Kyle Hughes *)

open Ast

module StringMap = Map.Make(String)

let check (globals, functions) =

  (* Raise an exception if the given list has a duplicate *)
  let report_duplicate exceptf list =
      let rec helper = function
          n1 :: n2 :: _ when n1 = n2 -> raise (Failure (exceptf n1))
          | _ :: t -> helper t
          | [] -> ()
       in helper (List.sort compare list)
  in

  (* Raise an exception if a given binding is to a void type *)
  let check_not_void exceptf = function
      (Void, n) -> raise (Failure (exceptf n))
```

```
    | _ -> ()
in

(* Raise an exception of the given rvalue type cannot be assigned to the given lvalue type *)
let check_assign lvaluet rvaluet err =
    if lvaluet == rvaluet then lvaluet else raise err
in

(**** Checking Globals ****)
List.iter (check_not_void (fun n -> "illegal void global " ^ n)) globals;

report_duplicate (fun n -> "duplicate global " ^ n) (List.map snd globals);

(**** Checking Functions ****)
if List.mem "print" (List.map (fun fd -> fd.fname) functions)
then raise (Failure ("function print may not be defined")) else ();

if List.mem "printJSON" (List.map (fun fd -> fd.fname) functions)
then raise (Failure ("function printJSON may not be defined")) else ();

if List.mem "readJSON" (List.map (fun fd -> fd.fname) functions)
then raise (Failure ("function readJSON may not be defined")) else ();

(* CHecks for other LOON library functions here *)
report_duplicate (fun n -> "duplicate function " ^ n)
    (List.map (fun fd -> fd.fname) functions);

(* Function declaration for a named function *)
let built_in_decls =  List.fold_left (fun map (name, attr) -> StringMap.add
name attr map) StringMap.empty [
    ("printJSON", { primitive = Void; fname = "printJSON"; formals = []; locals = []; body = [] });
    ]
in

let function_decls = List.fold_left (fun m fd -> StringMap.add fd.fname fd m)
    built_in_decls functions
in

let function_decl s = try StringMap.find s function_decls
    with Not_found -> if s = "main" then raise (Failure ("main function must be defined"))
    else raise (Failure ("function " ^ s ^ " unrecognized!"))
in

let _ = function_decl "main" in

let check_function func =
```

```
      List.iter (check_not_void (fun n -> "illegal void formal " ^ n ^ " in " ^ func.fname)) func.formals;

      report_duplicate (fun n -> "duplicate formal " ^ n ^ " in " ^ func.fname)
        (List.map snd func.formals);

      List.iter (check_not_void (fun n -> "illegal void local " ^ n ^ " in " ^ func.fname)) func.locals;

      report_duplicate (fun n -> "duplicate local " ^ n ^ " in " ^ func.fname)
        (List.map snd func.locals);

    (* Variable types *)
    let symbols = List.fold_left (fun m (t, n) -> StringMap.add n t m)
        StringMap.empty (globals @ func.formals @ func.locals )
    in

    let type_of_identifier s =
        try StringMap.find s symbols
        with Not_found -> raise (Failure ("undeclared identifier " ^ s))
    in

    (* Return the type of an expression or throw an exception *)
    let rec expr = function
        Literal _ -> Int
      | BoolLit _ -> Bool
      | CharLit _ -> Char
      | StringLit _ -> String
      | PairLit (_, e) -> Pair (expr e)
      | JsonLit (x:xs)-> Json (List.fold_left(fun t e -> if (t == (expr e)) then t else
          Failure ("inconsistent JSON object")) (expr x) (x:xs))
      | ArrayLit _ -> Array
      | Id s -> type_of_identifier s
      | Binop(e1, op, e2) as e -> let t1 = expr  e1 and t2 = expr  e2 in
      begin match op with
        Add ->
          begin match t1, t2 with
          | Int, Int -> Int
          | Int, Float -> Float
          | String, String -> String (* Concatenation Operator *)
          | Array, Array -> Array
          | Pair, Pair -> Json
          | Pair, Json -> Json
          | _ -> raise (Failure ("illegal binary operator " ^
              string_of_typ t1 ^ " " ^ string_of_op op ^ " " ^
              string_of_typ t2 ^ " in " ^ string_of_expr e))
          end
       | Sub | Mult | Div ->
          begin match t1, t2 with
          | Int, Int -> Int
          | String, String -> String
```

```
      | _ -> raise (Failure ("illegal binary operator " ^
          string_of_typ t1 ^ " " ^ string_of_op op ^ " " ^
          string_of_typ t2 ^ " in " ^ string_of_expr e))
      end
    | Equal | Neq when t1 = t2 -> Bool
    | Less | Leq | Greater | Geq when t1 = Int && t2 = Int -> Bool
    | And | Or when t1 = Bool && t2 = Bool -> Bool
    | _ -> raise (Failure ("illegal binary operator " ^
      string_of_typ t1 ^ " " ^ string_of_op op ^ " " ^
      string_of_typ t2 ^ " in " ^ string_of_expr e))
  end
  | Unop(op, e) as ex -> let t = expr  e in
  begin match op with
    Neg when t = Int -> Int
    | Not when t = Bool -> Bool
    | _ -> raise (Failure ("illegal unary operator " ^ string_of_uop op ^
        string_of_typ t ^ " in " ^ string_of_expr ex))
  end
  | Noexpr -> Void
  | Assign(var, e) as ex -> let lt = type_of_identifier var
                            and rt = expr  e in
    check_assign lt rt (Failure ("illegal assignment " ^ string_of_typ lt ^
                 " = " ^ string_of_typ rt ^ " in " ^
                 string_of_expr ex))
  | Call(fname, actuals) as call -> let fd = function_decl fname in
      if fname = "printJSON" then
          let _ = List.iter (fun e -> ignore(expr e)) actuals in Void
      else
          if List.length actuals != List.length fd.formals
              then raise (Failure ("Expecting " ^ string_of_int (List.length fd.formals) ^
              "arguments in " ^ string_of_expr call))
          else
              let _ = List.iter2 (fun (ft, _) e -> let et = expr e in
                  ignore (check_assign ft et (Failure ("Illegal actual argument found "
                  ^ string_of_typ et ^ " expected " ^ string_of_typ ft ^ " in " ^ string_of_expr call))))
          fd.formals actuals
          in
      fd.primitive
in

let check_int_expr e = if expr e != Int
    then raise (Failure ("expected Int expression in " ^ string_of_expr e)) else ()
in

let check_bool_expr e = if expr e != Bool
    then raise (Failure ("expected Bool expression in " ^ string_of_expr e)) else ()
in
```

```


  let rec stmt in_loop = function
   Block sl -> let rec check_block = function
     [Return _ as s] -> stmt in_loop s
     | Return _ :: _ -> raise (Failure "nothing may follow a return")
     | Block sl :: ss -> check_block (sl @ ss)
     | s :: ss -> stmt in_loop s ; check_block ss
     | [] -> ()
    in check_block sl
   | Expr e -> ignore (expr e)
   | Return e -> let t = expr e in if t = func.primitive then () else
     raise (Failure ("return gives " ^ string_of_typ t ^ " expected "
       ^
            string_of_typ func.primitive ^ " in " ^
              string_of_expr e))
   | If(p, b1, b2) -> check_bool_expr p; stmt false b1; stmt false b2
   | For(e1, e2, e3, st) -> ignore (expr e1); check_bool_expr e2;
               ignore (expr e3); stmt true st
   | While(p, s) -> check_bool_expr p; stmt true s
  in

 stmt false (Block func.body)

 in
 List.iter check_function functions

==> codegen.ml <==
(* Based on the MicroC llvm. Modified by Niles, Jack, and Chelci *)

module L = Llvm
module A = Ast

(*Custom modules*)
(*module Asgn = Assign *)

module StringMap = Map.Make(String)

let translate (globals, functions) =
 let context = L.global_context () in
 let the_module = L.create_module context "LOON"
 and i32_t  = L.i32_type  context
 and i8_t   = L.i8_type   context
 and i1_t   = L.i1_type   context
 and void_t = L.void_type context in

 (* Define the array type *)
 let arr_type = L.pointer_type (L.pointer_type i8_t) in

 let rec ltype_of_typ = function
   A.Int -> (*ignore(print_endline("int gets called..."));*) i32_t
  | A.Char -> i8_t
  | A.Bool -> i1_t
```

```
    | A.String -> (*ignore(print_endline("str gets called..."));*) L.pointer_type i8_t
    | A.Array -> arr_type
    | A.Pair typ -> L.pointer_type (L.struct_type context [| L.pointer_type i8_t; ltype_of_typ typ |] )
    | A.Json -> arr_type
    | A.Void -> void_t in

  (* Declare each global variable; remember its value in a map *)
  let global_vars =
    let global_var m (t, n) =
      let init = L.const_int (ltype_of_typ t) 0
      in StringMap.add n (L.define_global n init the_module) m in
    List.fold_left global_var StringMap.empty globals in
(*  print_endline (string_of_bool (StringMap.mem "helloTest" global_vars)); *)

  (* Declare printf(), which the print built-in function will call *)
  let printf_t = L.var_arg_function_type i32_t [| L.pointer_type i8_t |] in
  let printf_func = L.declare_function "printf" printf_t the_module in

  (*Declare scanf(), which reads from stdin *)
  let loon_scanf_t = L.function_type (ltype_of_typ A.String) [| |] in
  let loon_scanf_func = L.declare_function "loon_scanf" loon_scanf_t the_module in


  (* Declare the built-in printbig() function *)
  let printbig_t = L.function_type i32_t [| i32_t |] in
  let printbig_func = L.declare_function "printbig" printbig_t the_module in

  (* Define each function (arguments and return type) so we can call it *)
  let function_decls =
    let function_decl m fdecl =
      let name = fdecl.A.fname
      and formal_types =
        Array.of_list (List.map (fun (t,_) -> ltype_of_typ t) fdecl.A.formals) in
      let ftype = L.function_type (ltype_of_typ fdecl.A.primitive) formal_types in
    StringMap.add name (L.define_function name ftype the_module, fdecl) m in
  List.fold_left function_decl StringMap.empty functions in

  (* Fill in the body of the given function *)
  let build_function_body fdecl =
    let (the_function, _) = StringMap.find fdecl.A.fname function_decls in
    let builder = L.builder_at_end context (L.entry_block the_function) in

    let format_str str_val = L.build_global_stringptr str_val "fmt" builder in

```

```
    (* Construct the function's "locals": formal arguments and locally
       declared variables.  Allocate each on the stack, initialize their
       value, if appropriate, and remember their values in the "locals" map *)
  (* Map of ids -> unmodified ocaml expression *)
  let id_vals_map = ref StringMap.empty in

    let local_vars =
      let add_formal m (t, n) p = L.set_value_name n p;
    let local = L.build_alloca (ltype_of_typ t) n builder in
    ignore (L.build_store p local builder);
    StringMap.add n local m in

      let add_local m (t, n) =
    let local_var = L.build_alloca (ltype_of_typ t) n builder in
    ignore(id_vals_map := StringMap.add n (A.zero_of_typ t) !id_vals_map ) ;
    StringMap.add n local_var m in

      let formals = List.fold_left2 add_formal StringMap.empty fdecl.A.formals
          (Array.to_list (L.params the_function)) in
      List.fold_left add_local formals fdecl.A.locals in

    (* Return the value for a variable or formal argument *)
    let lookup n = try StringMap.find n local_vars
                   with Not_found -> StringMap.find n global_vars

    (* Map with:
    key - Id name
    value - A.val_type list
    List contains value type for each element in a given array *)
    and arr_types_map = ref StringMap.empty
    (* id to list of types *)
    and json_types_map = ref StringMap.empty

    (* id to map of keys -> index *)
    and json_lookup_map = ref StringMap.empty in
    let add_arr_types id = function
        | A.Val_list types_list -> StringMap.add id types_list !arr_types_map
        | _ -> ignore(print_endline("ADD_ARR_TYPES: ERROR - Bad input to array types map")); StringMap.empty
    and get_arr_types id = StringMap.find id !arr_types_map
    and is_arr id = StringMap.mem id !arr_types_map
    and add_json_types id types_list = StringMap.add id types_list !json_types_map
    and add_json_keys id l =
        let new_string_map = StringMap.empty in
        let add_to_map (map, index) next =
            (StringMap.add next index map, index + 1)
        in
        let new_string_map = fst (List.fold_left add_to_map (new_string_map, 0) l) in
```

```
        json_lookup_map := StringMap.add id new_string_map !json_lookup_map

    (* Stack containing lists of value types for each array.
       Should only be one array's list of types on stack at any given time *)
    and arr_types_stack = ref (A.Val_list [])
    and json_types_stack = ref []
    and json_keys_stack = ref []
    and key_lookup = ref StringMap.empty
    and current_key = ref ""
    in

    (* Construct code for an expression; return its value *)
    let rec expr builder = function
        A.Literal i -> L.const_int i32_t i
      | A.BoolLit b -> L.const_int i1_t (if b then 1 else 0)
      | A.CharLit c -> L.const_int i8_t (int_of_char c)
      (* StringLit constructs a private address that points to the argument value's contents *)
      | A.StringLit s ->
            L.build_global_stringptr s "str" builder
      | A.PairLit (k, v) -> (* need to eval both k and v *)
            (* Evaluate both k and v *)
            let key_string = expr builder k and
            value = expr builder v in

            let key_string_as_string =
                match k with
                | A.StringLit s -> s
                | _ -> "error" in
            ignore(current_key:= key_string_as_string);

            (* Define our bespoke pair type *)
            let pair_type = L.struct_type context [| L.pointer_type i8_t; L.type_of value |] in

            (* Allocate an object of type pair *)
            let allocated_struct = L.build_alloca pair_type "pair" builder in

            let place_for_key =
                L.build_in_bounds_gep allocated_struct [| (L.const_int i32_t 0); (L.const_int i32_t 0) |] "key_addr" builder in

            ignore(L.build_store key_string place_for_key builder);

            let place_for_value =
                L.build_in_bounds_gep allocated_struct [| (L.const_int i32_t 0); (L.const_int i32_t 1) |] "val_addr" builder in

            ignore(L.build_store value place_for_value builder);
```

```

            allocated_struct

      | A.Noexpr -> L.const_int i32_t 0
      | A.ArrayLit l -> let arr_size = L.const_int i32_t (List.length l) in
            (* Allocate space for values and types*)
            let arr_space =  L.build_array_alloca (L.pointer_type i8_t) arr_size "arr" builder in

            (* Function to load each individual value *)
            let load_object (indx , temp_types_list) expr_val =
                let llvm_indx = [| L.const_int i32_t indx|]
                and llvm_expr = (expr builder expr_val) in

                (* Allocate space for given type and store*)
                let val_type = L.type_of llvm_expr in
                let stored_val = L.build_alloca val_type "arr_val" builder in
                ignore(L.build_store llvm_expr stored_val builder);

                (* Get pointer to the new value and cast it to i8 pointer *)
                let void_elem_ptr = L.build_bitcast stored_val (L.pointer_type i8_t) "cast_val" builder
                and arr_indx = L.build_in_bounds_gep arr_space llvm_indx "arr_pos" builder in

                (* Store the pointer to the value in the arr, return updated types list and next indx *)
                ignore(L.build_store void_elem_ptr arr_indx builder) ;

                (* If val is an array, get its list of types - else get val_type of primitive *)
                let indx_elem_type =
                    if val_type = arr_type then (

                (* Get stack of types *)
                let top_of_stack = (match !arr_types_stack with
                                        | A.Val_list ts -> ts
                                        | _ -> []) in

                (* Get top of stack and make tail new stack *)
                    let l_of_types = List.hd top_of_stack in
                    ignore(arr_types_stack := A.Val_list (List.tl top_of_stack));
                    l_of_types
                    ) else( A.Val (L.pointer_type val_type) ) in
                        match temp_types_list with
                        | A.Val_list lts -> (indx + 1, A.Val_list (indx_elem_type :: lts))
                        | _ -> (indx +1, A.Val_list [])  in

                let old_stack = (match !arr_types_stack with
```

```
                    | A.Val_list ts -> ts
                    | _ -> []) in
        let res_list = (match(snd (List.fold_left load_object (0, A.Val_list []) l)) with
          | A.Val_list f_list -> f_list
          | _ -> [] ) in
    ignore(arr_types_stack := A.Val_list (A.Val_list (List.rev res_list) :: old_stack) );
    arr_space
   | A.JsonLit l ->
      let rec unzip_keys_and_vals = function
        [] -> ([], [])
       | (k, v) :: rest ->
           let everything_else = unzip_keys_and_vals rest in
           (k :: (fst everything_else), v :: (snd everything_else))
      in
      let unzipped = unzip_keys_and_vals l in
      let keys = fst unzipped
      and vals = snd unzipped in

      let keys_as_strings = List.map (fun x -> match x with
        | A.StringLit s -> s
        | _ -> "Undefined") keys in

      let arr_size = L.const_int i32_t (List.length l) in
      (* Allocate space for values and types*)
      let arr_space =  L.build_array_alloca (L.pointer_type i8_t) arr_size "arr" builder in

      (* Function to load vals*)
      let load_object (indx , temp_types_list) expr_val =
        let llvm_indx = [| L.const_int i32_t indx|]
        (* HANDLE IDS HERE *)
        and llvm_expr = (expr builder expr_val) in

        (* Allocate space for given type and store*)
        let val_type = L.type_of llvm_expr in
        let stored_val = L.build_alloca val_type "arr_val" builder in
        ignore(L.build_store llvm_expr stored_val builder);

        (* Get pointer to the new value and cast it to i8 pointer *)
        let void_elem_ptr = L.build_bitcast stored_val (L.pointer_type i8_t) "cast_val" builder
        and arr_indx = L.build_in_bounds_gep arr_space llvm_indx "arr_pos" builder in

        (* Store the pointer to the value in the arr, return updated types list and next indx *)
        ignore(L.build_store void_elem_ptr arr_indx builder);

        let indx_elem_type = L.pointer_type val_type in
```

```
                (indx + 1, indx_elem_type :: temp_types_list)
            in

            (* Get the list of types for this array *)
            let res_list = snd (List.fold_left load_object (0, []) vals)
            in

            (* Push the list of types for this array onto stack and return
                address of this array literal *)
            ignore(json_types_stack := List.rev res_list);
            ignore(json_keys_stack := keys_as_strings);
            arr_space
      | A.Id s -> L.build_load (lookup s) s builder
      | A.Binop (e1, op, e2) ->
        let e1' = expr builder e1
        and e2' = expr builder e2
        (* Semantic checking ensures that two are of same type - can
           insert additional check for float/int conversions *)
        and check_expr_type e_1 e_2 = (match e_1 with
                A.StringLit s1 -> ignore(s1); "i8*"
              | A.Literal i -> ignore(i); "i32"
              | A.Id id ->
                    if StringMap.mem id !json_types_map then (
                        match e_2 with
                            A.Id id -> if StringMap.mem id !json_types_map
                                then ("json+json")
                                        else ("json+pair")
                          | _ -> "error"
                    )
                    else if StringMap.mem id !key_lookup then (
                    "pair+pair")
                    else
                    L.string_of_lltype (L.element_type (L.type_of (lookup
                        id)))
            (* Arrays will also go here *)
              | _ -> ignore(print_endline ("ERROR: CHECK_EXPR_TYPE:
                  Invalid operand") ); "null") in
        (match (check_expr_type e1 e2) with
            "i8*" ->
                let concat_str estr1 estr2 = (A.string_of_expr estr1) ^ (A
                    .string_of_expr estr2) in
                let m_op =
                    match op with
                        A.Add -> L.build_global_stringptr

                        (* Should never happen given semantically correct
                            tree
                        - Should still replace with L.NULL eventually *)
                    | _ ->
                        ignore(print_endline ("Not PLUS op which is a
                            problem..." ) );
                        L.build_global_stringptr
                in
```

```
                m_op (concat_str e1 e2) "str" builder
    (* Everything else is an int/float *)
          | "i32" ->
            (match op with
              (* Overload add to perform string concat*)
                A.Add     -> L.build_add
              | A.Sub     -> L.build_sub
              | A.Mult    -> L.build_mul
              | A.Div     -> L.build_sdiv
              | A.And     -> L.build_and
              | A.Or      -> L.build_or
              | A.Equal   -> L.build_icmp L.Icmp.Eq
              | A.Neq     -> L.build_icmp L.Icmp.Ne
              | A.Less    -> L.build_icmp L.Icmp.Slt
              | A.Leq     -> L.build_icmp L.Icmp.Sle
              | A.Greater -> L.build_icmp L.Icmp.Sgt
              | A.Geq     -> L.build_icmp L.Icmp.Sge
            ) e1' e2' "tmp" builder
          | "pair+pair" ->
                let get_val pair =
                    let pointer_to_value =
                        L.build_in_bounds_gep pair [| (L.const_int i32_t 0); (L.const_int i32_t 1) |] "val_addr" builder in

                    let return_value = L.build_load pointer_to_value "" builder in
                    return_value
                in

                let first_val = get_val e1'
                and second_val = get_val e2' in
                let first_id = match e1 with
                    A.Id id -> id
                  | _ -> "error"
                in
                let second_id = match e2 with
                    A.Id id -> id
                  | _ -> "error"
                in
                let first_key_string = StringMap.find first_id !key_lookup
                and second_key_string = StringMap.find second_id !key_lookup in

                let keys = [first_key_string; second_key_string]
                and vals = [first_val; second_val] in

                let arr_size = L.const_int i32_t (List.length keys) in (* Allocate space for values and types*)
                let arr_space =  L.build_array_alloca (L.pointer_type i8_t) arr_size "arr" builder in

                (* Function to load vals*)
                let load_object (indx , temp_types_list) llvm_expr =
```

```
          let llvm_indx = [| L.const_int i32_t indx|] in

          (* Allocate space for given type and store*)
          let val_type = L.type_of llvm_expr in
          let stored_val = L.build_alloca val_type "arr_val" builder in
          ignore(L.build_store llvm_expr stored_val builder);

          (* Get pointer to the new value and cast it to i8 pointer *)
          let void_elem_ptr = L.build_bitcast stored_val (L.pointer_type i8_t) "cast_val" builder
          and arr_indx = L.build_in_bounds_gep arr_space llvm_indx "arr_pos" builder in

          (* Store the pointer to the value in the arr, return updated types list and next indx *)
          ignore(L.build_store void_elem_ptr arr_indx builder);

          let indx_elem_type = L.pointer_type val_type in

          (indx + 1, indx_elem_type :: temp_types_list)
      in

      (* Get the list of types for this array *)
      let res_list = snd (List.fold_left load_object (0, []) vals)
      in

      (* Push the list of types for this array onto stack and return address of this array literal *)
      ignore(json_types_stack := List.rev res_list);
      ignore(json_keys_stack := keys);
      arr_space

| "json+pair" ->
      let get_val pair =
          let pointer_to_value =
              L.build_in_bounds_gep pair [| (L.const_int i32_t 0); (L.const_int i32_t 1) |] "val_addr" builder in

          let return_value = L.build_load pointer_to_value "" builder in
          return_value
      in

      let pair_val = get_val e2' in
      let pair_id = match e2 with
          A.Id id -> id
        | _ -> "error"
      and json_id = match e1 with
          A.Id id -> id
        | _ -> "error"
```

```
        in
        let key_string = StringMap.find pair_id !key_lookup in

        let json_lookup = StringMap.find json_id !json_lookup_map in
        let json_types = StringMap.find json_id !json_types_map in
        let fold_key_func key _ acc = key :: acc in

        let load_val acc key_string =
            let index = StringMap.find key_string json_lookup in

            (* Gets type to cast to *)
            let type_of_res = List.nth json_types index in
            (* Function to get element pointer to desired element in an array *)

            let llvm_index = L.const_int i32_t index in
            let elemptr = L.build_gep (L.build_load (lookup json_id) "" builder) [|llvm_index|] "" builder in
            let arr_val = L.build_load elemptr "" builder in
            let cast_val = L.build_bitcast arr_val type_of_res "cast" builder in
            L.build_load cast_val "" builder :: acc
        in
        let json_keys = StringMap.fold fold_key_func json_lookup [] in
        let json_vals = List.fold_left load_val [] json_keys in

        let keys = List.rev(key_string :: json_keys)

        and vals = List.rev(pair_val :: List.rev json_vals) in

        let arr_size = L.const_int i32_t (List.length keys) in
        (* Allocate space for values and types*)
        let arr_space =  L.build_array_alloca (L.pointer_type i8_t) arr_size "arr" builder in

        (* Function to load vals*)
        let load_object (indx , temp_types_list) llvm_expr =
            let llvm_indx = [| L.const_int i32_t indx|] in

            (* Allocate space for given type and store*)
            let val_type = L.type_of llvm_expr in
            let stored_val = L.build_alloca val_type "arr_val" builder in
            ignore(L.build_store llvm_expr stored_val builder);

            (* Get pointer to the new value and cast it to i8 pointer *)
            let void_elem_ptr = L.build_bitcast stored_val (L.pointer_type i8_t) "cast_val" builder
            and arr_indx = L.build_in_bounds_gep arr_space llvm_indx "arr_pos" builder in

```

```
                    (* Store the pointer to the value in the arr, return updated types list and next indx *)
                    ignore(L.build_store void_elem_ptr arr_indx builder);

                    let indx_elem_type = L.pointer_type val_type in

                    (indx + 1, indx_elem_type :: temp_types_list)
                in

                (* Get the list of types for this array *)
                let res_list = snd (List.fold_left load_object (0, []) vals)
                in

                (* Push the list of types for this array onto stack and return address of this array literal *)
                ignore(json_types_stack := List.rev res_list);
                ignore(json_keys_stack := keys);
                arr_space
          | _ -> ignore(print_endline ("NO SUITABLE BINARY OPERATIONS FOUND FOR LEFT OPERAND")); L.const_null i32_t)
      | A.Unop(op, e) ->
          let e' = expr builder e in
            (match op with
              A.Neg     -> L.build_neg e' "tmp" builder
            | A.Not     -> L.build_not e' "tmp" builder
            | A.Deref   ->
                let pointer_to_value =
                    L.build_in_bounds_gep e' [| (L.const_int i32_t 0); (L.const_int i32_t 1) |] "key_addr" builder in

                let return_value = L.build_load pointer_to_value "" builder in
                return_value
        )
      | A.Access(id, indx_lst) ->
        (* If id is a polymorphic array, cast pointer type accordingly*)
        if StringMap.mem id !json_types_map then (
            (* Get the first value in the indx_list as a string *)
            (* Assume that only primitives in json, never need more than first *)
            let index_string = A.string_of_expr (List.hd indx_lst)
            and types = StringMap.find id !json_types_map in
            let lookup_map = StringMap.find id !json_lookup_map in
            let index = StringMap.find index_string lookup_map in

            (* Gets type to cast to *)
            let type_of_res = List.nth types index in
            (* Function to get element pointer to desired element in an array *)

            let llvm_index = L.const_int i32_t index in
            let elemptr = L.build_gep (L.build_load (lookup id) "" builder) [|llvm_index|] "" builder in
```

```
            let arr_val = L.build_load elemptr "" builder in
            let cast_val = L.build_bitcast arr_val type_of_res "cast" builder in
            L.build_load cast_val "" builder)

        (* If id is a polymorphic array (it should be, this is really just a safety check), cast pointer type accordingly*)
        else (
            (* Function to get element pointer to desired element in an array *)
            let pos_finder prev_pos indx =
               (* Load Array value ( i8* ) *)
               let llv_of_indx = expr builder indx
               and load_of_orig = L.build_load prev_pos "" builder in

               (* Cast loaded i8* to an i8*** because it must be address of an array *)
               let cast_of_load = L.build_bitcast load_of_orig (L.pointer_type arr_type) "temp_cast" builder in

               (* Get address of element at desired index position *)
               L.build_gep (L.build_load cast_of_load "" builder) [|llv_of_indx|] "" builder in

            (* Get the initial value to index through *)
            let first_indx = List.hd indx_lst in
            let init_res = L.build_gep (L.build_load (lookup id) "" builder) [|(expr builder first_indx)|] "" builder in

            (* Fold list of index positions in order to get element pointer to final index position
            NOTE: If List.tl indx_list yields the empty list, the result of the below call is equal to init_res *)
            let final_pos = List.fold_left pos_finder init_res (List.tl indx_lst) in
            let arr_val = L.build_load final_pos "" builder in (*ignore(print_endline("ACCESS: Loaded value is: " ^ (L.string_of_llvalue arr_val) ^ " with type " ^ (L.string_of_lltype (L.type_of arr_val))));*)

            if StringMap.mem id !arr_types_map then
            (* First invoke function to map each expr in list to ocaml int *)
            (let expr_to_int e_i = match e_i with
                              A.Literal i -> i
                              | A.Id s ->  let oc_val = StringMap.find s !id_vals_map in
                     (match oc_val with
                       A.Literal s_i -> (*ignore(print_endline("Id is literal with val: " ^ (string_of_int s_i)));*) s_i
                                              | _ -> ignore(print_endline ("EXPR_TO_INT: Not a Literal - array index not
```

```
                                                    possible")); -1
                                            )
                             | _ -> ignore(print_endline("ACCESS: ERROR
                                : Bad type passed into access element")
                                ); 0 in

            let indxs_int_lst = List.map expr_to_int indx_lst in

            (* Gets type to cast the element pointer to *)
            let type_of_res = A.get_val_type context indxs_int_lst (A.
               Val_list(get_arr_types id)) in
            let cast_val = L.build_bitcast arr_val (type_of_res) "cast"
               builder in

            L.build_load cast_val "" builder) else arr_val)

      | A.Assign (s, lst, e) -> let e' = expr builder e in
            (* Check if you are assigning to array index position *)
            if(lst = []) then (

                (* No access assignment, simply load into id's address
                   space *)
        ignore(id_vals_map := StringMap.add s e !id_vals_map );
                ignore (L.build_store e' (lookup s) builder);
                (* In case of array assignment, pop the stack for list of
                   val_types and then add to map *)
                if L.element_type (L.type_of (lookup s)) = L.pointer_type
                   (L.pointer_type i8_t) then (
                ignore(
                  match e with
                    A.JsonLit _ ->
                        ignore(json_types_map := add_json_types s !
                           json_types_stack );
                        ignore(json_types_stack := []);
                        ignore(add_json_keys s !json_keys_stack);
                        ignore(json_keys_stack := []);
                  | A.Binop _ ->
                        ignore(json_types_map := add_json_types s !
                           json_types_stack );
                        ignore(json_types_stack := []);
                        ignore(add_json_keys s !json_keys_stack);
                        ignore(json_keys_stack := []);
                  | A.ArrayLit _ ->
                        let popped_val = match !arr_types_stack with
                        | A.Val_list lstv -> lstv
                        | _ -> [] in

                        ignore(arr_types_map := add_arr_types s (List.hd
                           popped_val) );
                        ignore(arr_types_stack := A.Val_list(List.tl
                           popped_val));
                  | _ -> ()
                );

```

```
                e') else (
                    (* If not a Json object or an array, check if it's a
                       pair (lit or id)
                    If it is, then add its key to key_lookup and return,
                       else just return *)
                    match e with
                      | A.PairLit _ ->
                        (ignore(key_lookup := StringMap.add s !current_key
                            !key_lookup); e')
                      | A.Id id -> ignore(print_endline id);
                        if StringMap.mem id !key_lookup then
                          (let key = StringMap.find id !key_lookup in
                              ignore(print_endline s);
                              ignore(print_endline key);
                              ignore(key_lookup := StringMap.add s key !
                                 key_lookup); e')
                        else
                              e'
                      | _ -> e'
                )

            ) else(
                (* Access assignment *)

                (* Function to get element pointer to desired element in
                   an array *)
                let pos_finder prev_pos indx =
                    let llv_of_indx = expr builder indx
                    and load_of_orig = L.build_load prev_pos "" builder in

                    let cast_of_load = L.build_bitcast load_of_orig (L.
                       pointer_type arr_type) "temp_cast" builder in
                    L.build_gep (L.build_load cast_of_load "" builder) [|
                       llv_of_indx|] "" builder in

                (* Get the initial value to index through *)
                let first_indx = List.hd lst in
                let init_res = L.build_gep (L.build_load (lookup s) ""
                   builder) [|(expr builder first_indx)|] "" builder in

                (* Fold list of index positions in order to get element
                   pointer to final index position
                NOTE: If List.tl indx_list yields the empty list, the
                   result of the below call is equal to init_res *)
                let final_pos = List.fold_left pos_finder init_res (List.
                   tl lst) in

                (* Turn the list of indx positions into a list of ints *)
                let expr_to_int e_i = match e_i with
                              A.Literal i -> i
                              | A.Id s ->  let oc_val = StringMap.find s
                                 !id_vals_map in
                      (match oc_val with
```

```
                        A.Literal s_i -> (*ignore(print_endline("Id is literal with val: " ^ (string_of_int s_i)));*) s_i
                                         | _ -> ignore(print_endline(" EXPR_TO_INT: Not a Literal - array index not possible ")); -1
                                         )
                            | _ -> ignore(print_endline("ACCESS: ERROR : Bad type passed into access element")); 0 in

                let indxs_int_lst = List.map expr_to_int lst

                 (* If new type is an array, pop types from stack - otherwise just wrap val type *)
                and new_type = let elt = L.type_of e' in
                               if elt = arr_type then(
                                  (*ignore(print_endline("ASSIGN: new value to store is array: " ^ (L.string_of_lltype elt)));*)
                                  let true_stack = match !arr_types_stack with
                                      | A.Val_list tlst -> tlst
                                      | _ -> [] in
                                  let new_types_list = List.hd true_stack in
                                  ignore(arr_types_stack := A.Val_list(List.tl true_stack));
                                  new_types_list)
                               else ((*ignore(print_endline("ASSIGN: new value to store is NOT array: " ^ (L.string_of_lltype elt))); *)
                                   A.Val(elt))

                 (* Get the current list of types for id *)
                 and types_lst = get_arr_types s in

                 (* Set the array value type at the specified position to the new value's type *)
                 let new_types_lst = A.set_val_type context types_lst new_type indxs_int_lst in

                 (* Add the updated list of types for this id to the map - then allocate space for type of the new value *)
                 ignore(arr_types_map := add_arr_types s (A.Val_list new_types_lst));

                 let alloc_new_val = L.build_alloca (A.get_val_type context [] new_type) "assign_acc_val" builder in

                 (* Store new value in allocated space, bitcast the pointer to it to void ptr *)
                 ignore(L.build_store e' alloc_new_val builder);
```

```
                let cast_new_val = L.build_bitcast alloc_new_val (L.pointer_type i8_t) "assign_tmp_cast" builder in

                (* Store cast value into the index position of the original array *)
                ignore(L.build_store cast_new_val final_pos builder); e'
            )

      | A.Call ("printJSON", lst) | A.Call ("printb", lst) -> let rec print_builder (fmt_str, lst_init) = (if (List.length lst_init)=0 then
                              (fmt_str ^ "\n", [] )
                              else (let x = List.hd lst_init in
                                  let str_new = match x with
                                  A.StringLit s1 -> ignore(s1); fmt_str ^ "%s"
                                  | A.CharLit c1 -> ignore(c1); fmt_str ^ "%c"
                                  | A.Literal i1 -> ignore(i1); fmt_str ^ "%d"
                                  | A.BoolLit b1 -> ignore(b1); fmt_str ^ "%d"
                                  | A.Id id1 -> let int_type = L.pointer_type (ltype_of_typ A.Int)
                                          and bool_type = L.pointer_type (ltype_of_typ A.Bool)
                                          and str_type = L.pointer_type (ltype_of_typ A.String)
                                          and char_type = L.pointer_type (ltype_of_typ A.Char) in
                                          (*and id_type = L.type_of (lookup id1) in *) (*ignore(print_endline ("ID_CATCH: Before match int_type: " ^ (L.string_of_lltype int_type) ));*)
                                          let type_match llt = if llt = str_type then
                                               "%s"
                                          else if llt = int_type then "%d" else if llt = char_type then ("%c")
                                          else if llt = bool_type then ("%d")
                                          else ((*ignore(print_endline ("ID_CATCH: Bad match...")); *) "Bad")
                                      in fmt_str ^ (type_match (L.type_of (lookup id1)))
                                  | A.Access (id, indx) -> if is_arr id then (

                                  (* Get proper type *)
                                  let expr_to_int e_i = match e_i with
```

```
                                    A.Literal i -> i
                                    | A.Id s ->  let oc_val = StringMap.find s !id_vals_map in
                      (match oc_val with
                        A.Literal s_i -> (*ignore(print_endline("Id is literal with val: " ^ (string_of_int s_i)));*)
                              s_i
                                              | _ -> ignore(print_endline("EXPR_TO_INT: Not a Literal - array index not possible")); -1
                                            )
                                    | _ -> ignore(print_endline("ACCESS: ERROR: Bad type passed into access element")); 0 in

                                let indxs_int_lst = List.map expr_to_int indx in

                                let llt_of_val = L.element_type (A.get_val_type context indxs_int_lst (A.Val_list(get_arr_types id))) in

                                fmt_str ^ (A.fmt_of_lltype (L.string_of_lltype llt_of_val)) ) else fmt_str ^ "%c"
                                | _ -> (*ignore(print_endline ("PRINT_BUILDER: Head is unknown type ..."));*) fmt_str ^ "BAD"

                                in let res = print_builder (str_new, (List.tl lst_init)) in
                                ((fst res), (expr builder x) :: (snd res) ) )) in
                            let full_args = print_builder ("", lst) in
                                L.build_call printf_func (Array.of_list ((format_str (fst full_args)) :: (snd full_args) )) "printf" builder
      | A.Call ("printbig", [e]) ->
    L.build_call printbig_func [| (expr builder e) |] "printbig" builder

      | A.Call ("loon_scanf", _ ) ->
          L.build_call loon_scanf_func [| |] "loon_scanf" builder

      (*| A.Call ("loon_scanf", [e]) ->failwith "why not? scanf"
    (*L.build_call scanf_func [| (expr builder e) |] "loon_scanf" builder *)*)
      | A.Call (f, act) ->
            let (fdef, fdecl) = StringMap.find f function_decls in
            let actuals = List.rev (List.map (expr builder) (List.rev act) ) in
            let result = (match fdecl.A.primitive with A.Void -> ""
                                              | _ -> f ^ "_result") in
         L.build_call fdef (Array.of_list actuals) result builder
```

```
    in


    (* Invoke "f builder" if the current block doesn't already
       have a terminal (e.g., a branch). *)
    let add_terminal builder f =
      match L.block_terminator (L.insertion_block builder) with
    Some _ -> ()
      | None -> ignore (f builder) in

    (* Build the code for the given statement; return the builder for
       the statement's successor *)
    let rec stmt builder = function
    A.Block sl -> List.fold_left stmt builder sl
      | A.Expr e -> ignore (expr builder e); builder
      | A.Return e -> ignore (match fdecl.A.primitive with
        A.Void -> L.build_ret_void builder
    | _ -> L.build_ret (expr builder e) builder); builder
      | A.If (predicate, then_stmt, else_stmt) ->
            let bool_val = expr builder predicate in
            let merge_bb = L.append_block context "merge" the_function in

            let then_bb = L.append_block context "then" the_function in
            add_terminal (stmt (L.builder_at_end context then_bb)
                then_stmt)
            (L.build_br merge_bb);

            let else_bb = L.append_block context "else" the_function in
            add_terminal (stmt (L.builder_at_end context else_bb)
                else_stmt)
            (L.build_br merge_bb);

            ignore (L.build_cond_br bool_val then_bb else_bb builder);
            L.builder_at_end context merge_bb

      | A.While (predicate, body) ->
            let pred_bb = L.append_block context "while" the_function in
            ignore (L.build_br pred_bb builder);

            let body_bb = L.append_block context "while_body" the_function
                in
            add_terminal (stmt (L.builder_at_end context body_bb) body)
            (L.build_br pred_bb);

            let pred_builder = L.builder_at_end context pred_bb in
            let bool_val = expr pred_builder predicate in

            let merge_bb = L.append_block context "merge" the_function in
            ignore (L.build_cond_br bool_val body_bb merge_bb pred_builder
                );
            L.builder_at_end context merge_bb

      | A.For (e1, e2, e3, body) -> stmt builder
```

```
            ( A.Block [A.Expr e1 ; A.While (e2, A.Block [body ; A.Expr e3
               ]) ] )
        in

    (* Build the code for each statement in the function *)
    let builder = stmt builder (A.Block fdecl.A.body) in

    (* Add a return if the last block falls off the end *)
    add_terminal builder (match fdecl.A.primitive with
        A.Void -> L.build_ret_void
      | t -> L.build_ret (L.const_int (ltype_of_typ t) 0))
  in
  List.iter build_function_body functions;
  the_module
```