BURGer

Bringing Undergraduate Residents Greater Evaluative Reasoning

—

Team Members:

Jordan Lee (jal2283)	Manager
Jacqueline Kong (jek2179)	Language Guru
Adrian Traviezo (aft2115)	Systems Architect
Ashley Nguyen (akn2121)	Tester

COMS W4115: Programming Languages and Translators
Fall 2017

Table of Contents

1. Introduction

BURGer is a general purpose programming language designed for convenient, intuitive use. By using a program structure similar to Python while adhering to conventions familiar to users of Java and C, programmers using BURGer can write strongly-typed code with the flexibility afforded by top-level code. It is intended to be lightweight and easily readable for users who are familiar with the syntax and types available in popular programming languages, by supporting common data types (ints, booleans, strings), control flow, and user-defined function behavior.

Like any good hamburger, BURGer is an amalgamation of different flavors and ideas from discrete sources with the intent to please a diverse user base. It integrates, for example, variable declarations inspired by C with the String type available in other languages, without requiring a main function. BURGer aims to bridge similar gaps between a variety of languages with a lightweight but powerful set of operators and features.

2. Language Tutorial

2.1 Environment Setup

Before writing a program in BURGer, you'll need to download the files associated with the language. They can be accessed on Github at the following link:
https://github.com/jacquelinekong/BURGer

BURGer was developed with LLVM.5.0.0 and OCaml 4.05.0, so in order to have your programs run correctly, please make sure those versions are installed on your machine.

2.2 Using the Compiler

Once you have the BURGer project files as well as OCaml and LLVM installed, you can navigate to the `BURGer/src` directory. Programs written in the BURGer language have a file extension of `.bun`. You can run the following commands:

- To build the compiler:

```
make
```

- To compile a .bun file:

```
./burgr-c.sh <name of .bun file>
```

- To compile **and** run a BURGer program:

```
./burgr.sh <name of .bun file>
```

Example of commands used to run a program in `usr/BURGER/src/hamburgers.bun`:

```
make
./burgr.sh hamburgers
```

2.3 Writing Programs

Writing programs in BURGer is very similar to writing programs in Java, C, and Python. Like Java and C, BURGer is strongly typed. Like Python, the programmer does not need to declare a main function and uses the `def` keyword to declare functions.

2.3.1 Hello World!

hello-word.bun:

```
print("Hello World!\n");
string s = "hello world";
println(s);
```

Output:

```
Hello World!
hello world
```

This program demonstrates the use of the built-in `print` and `println` functions, as well as variable initialization. In the first line of the code, the `print` function recognizes the `\n` in the string literal `"Hello World!\n"` as a newline character. In the second, the variable of type string, `s`, is assigned the string literal `"hello world"`. The `println` function is passed in the identifier `s`, prints `"hello world"`, and then appends a new line to it.

2.3.2 Variable Declaration and Initialization

test-assign.bun:

```
int x;
x = 2;
bool y = true;
int z;
bool zero;
println(x);
println(y);
println(z);
println(zero);
```

Output:
```
2
1
0
0
```

This program demonstrates a mix of approaches for variable declarations in BURGer. As shown in the first two lines of test-assign.bun, a variable can be declared on one line and then assigned a value on another line. Or, a value can be declared and instantiated to a value in the same line. Types of `bool` and `int` can be declared but never initialized by the programmer, and their value will return 0.

2.3.3 Functions

test-func1.bun:
```
def int add(int a, int b) {
      println("add numbers!");
      return a + b;
}
println(add(1, 3));
```

Output:
```
add numbers!
4
```

This program demonstrates function declaration in BURGer. A user-defined function is declared with the `def` keyword, followed by the return type of the function, its identifier (name), and a comma-separated list of arguments. The function is called at the end of test-func1.bun in order to execute it.

2.3.4 If/Else

test-if.bun:
```
if(true){
      int x;
      println("fff");
}
if(1 < 0){
      println("YES");
}
else{
      println("NO");
}
```

Output:
```
fff
NO
```

This is a typical use of control flow in BURGer. An `if` statement tests a boolean and executes the statement enclosed in its brackets if the boolean returns true. It can be followed by an `else` clause, which executes if the boolean is false, but it need not be.

2.3.5 Loops

test-while.bun:
```
int i;
i = 0;
while(i<3){
      println("yum");
      println("burgers");
      i = i+1;
}
```

Output:
```
yum
burgers
yum
burgers
yum
burgers
```

This is a typical use of a loop in BURGer. As long as the boolean indicated in parentheses evaluates to true at the end of the block within curly braces, that block will execute again.

2.3.6 Recursion

test-fib.bun:
```
def int fib(int x){
  if (x < 2) return 1;
  return fib(x-1) + fib(x-2);
}
println(fib(3));
```

Output:
```
3
```

This program demonstrates BURGer's support for recursion and calling functions within other functions by running the code corresponding to the algorithm for the Fibonacci sequence.

3. Language Reference Manual

3.1 Lexical elements

3.1.1 Identifiers

Identifiers are used for naming data types. Identifiers consist of any combination of alphanumeric characters. The first character of an identifier must be a letter.

3.1.2 Comments

Comments are denoted like so:

```
/* text
wow look at me
multiple lines
incredible */
```

3.1.3 Keywords, Symbols, and Operators

Keywords

`if`	`int`
`else`	`bool`
`while`	`string`
`def`	`true`
`return`	`false`
`null`	

Symbols and Operators

`+`	`/`	`<`	`==`	`&&`	`%`	`>=`
`-`	`%`	`>`	`.`	`||`	`!=`	`=`
`*`	`,`	`<=`	`( )`	`“ ”`	`!`	`;`
`{ }`						

3.1.4 Constants

3.1.4.1 Integer constants

An integer constant consists of one or more digits. Integers are optionally signed and default to positive if they are unsigned.

3.1.4.2 Boolean constants

Boolean constants can be `true` or `false`, which respectively correspond to logical true and false values.

3.1.5 Strings

A string is a sequence of characters enclosed in single or double quotes. BURGer strings are mutable and iterable.

3.2 Data Types

3.2.1 Primitive Types

There are three primitive data types: `int`, `bool`, and `null`. Numbers are denoted as an `int` type, which stores up to 4 bytes. `bool` holds a Boolean value of either `true` or `false`.
`null` is a type used for uninitialized variables.

3.2.2 Non-primitive Types

Strings, as discussed in Section 1.5, are a non-primitive type supported in BURGer.

3.3 Expression and Statement Syntax

3.3.1 Operators

3.3.1.1 Relational Operators

BURGer has the following relational operators:

<	less than
>	greater than
<=	less than or equal to
>=	greater than or equal to

3.3.1.2 Equality Operators

BURGer has the following equality operators:

==	equals
!=	not equals

3.3.1.3 Logical Operators

BURGer has the following logical operators:

`&`	and
`|`	or
`!`	not

3.3.1.4 Assignment Operator

The = binary operator sets the left operand equal to the right operand.

3.3.1.5 Arithmetic Operators

BURGer supports the standard arithmetic operations:

+	addition
-	subtraction
*	multiplication
/	division
%	modulo

3.3.1.6 Operator Precedence

The order of precedence for classes of operators is as follows, from the highest to the lowest: arithmetic, logical, assignment, equality/relational. Equality and relational operators have the same level of precedence. For arithmetic operations, multiplication and division take precedence over addition and subtraction.

Expressions contained within parentheses always take precedence. Otherwise, operators of equal precedence levels will take precedence from left to right.

3.3.2 Delimiters

3.3.2.1 Parentheses

BURGer uses parentheses to determine the operation precedence and for enclosing function calls.

3.3.2.2 Whitespace

BURGer uses whitespace to determine separate tokens. However, the amount of whitespace has no other bearing on the language.

3.3.2.3 Semicolons

BURGer uses semicolons to terminate statements.

3.3.3 Declaration and Initialization of Variables and Functions

3.3.3.1 Variable Declaration

Variables are declared with the desired type of the variable and the variable name. For example:

```
int x; bool y; string z;
```

3.3.3.2 Variable Initialization

`int x;`
`x = 42;`
	`int x = 42;`

Figure 3.3.3.2a	Figure 3.3.3.2b

Variables can be initialized in a separate assignment statement after being declared, like in Figure 3.3.3.2a. However, they can also be declared and initialized in the same statement, as shown in Figure 3.3.3.2b.

3.3.3.2 Functions

A function is declared with the `def` keyword, a function name, and a list of parameters between parentheses. The return type of the function and of its parameters are not specified on declaration, but a function that does return a value is assigned that return type. A function may be declared with curly braces specifying what the function does, like so:

```
def int adder (x, y) {
      return x + y;
};
```

3.3.4 Built-in Functions

BURGer uses the `print()` function to print a string literal, or an identifier that has been assigned a string literal, to the console. The `println()` has an identical functionality, but appends a new line character to it. These functions reside in stdlib.c, and are called from within codegen.ml using `L.declare_function`. The stdlib.c file is then linked into the BURGer environment during compilation.

3.3.5 Control Flow Expressions

3.3.5.1 if...else statements

BURGer uses the if else statements in a similar way to other languages. It's possible to nest these statements as well.

```
if( conditional expression ) { statement; } else { statement; }
```

3.3.5.2 while loops

BURGer performs `while` loops with the same syntax as Java.

```
while ( conditional expression ) { statement; }
```

3.3.5.3 return statements

BURGer uses the `return` statement to return values from a function. These values can be of any data type, including null.

3.4 Program Structure and Scope Rules

3.4.1 Program Structure

A BURGer program must be contained in a single source file with a `.bun` file extension.

3.4.2 Scope

BURGer is a statically scoped language. An object is not visible to any operations or declarations that have come before it - for example, a variable x_1 may not form part of the declaration of x_2 unless x_2 was previously formally declared. The scope of a global variable is the entire file, and the scope of a local variable is the block of code pertaining to the function in which is was initialized. Functions can only be defined in a global context; a function can be called, but not defined, within another function.

3.4.2.1 Globals

Global variables may be declared independently of any code block - as in, they do not need to exist within any function or as part of another declaration. By convention, they should be declared at the top of the file. These variables are visible from any point of the file.

3.4.2.2 Locals

Local variables are visible only from within the function where they are defined. If a helper function must access a variable declared in the outer function from which it is called, the variable must be passed into the helper function as a parameter.

3.5 Grammar

```
program → item_list EOF

item_list →
           /* nothing */
           | item_list item

item → stmt | fdecl
```

```
typ → INT | BOOL | STRING | NULL

/*** Statements ***/
stmt →
    expr SEMI
  | vdecl SEMI
  | RETURN expr SEMI
  | RETURN SEMI
  | LBRACE stmt_list RBRACE
  | IF LPAREN expr RPAREN stmt %prec NOELSE
  | IF LPAREN expr RPAREN stmt ELSE stmt
  | WHILE LPAREN expr RPAREN stmt

stmt_list →
    stmt
  | stmt_list stmt

/*** Expressions ***/
expr →
    MINUS expr
  | NOT expr
  | ID ASSIGN expr
  | expr PLUS   expr
  | expr MINUS  expr
  | expr TIMES  expr
  | expr DIVIDE expr
  | expr MOD      expr
  | expr EQ     expr
  | expr NEQ    expr
  | expr LT     expr
  | expr LEQ    expr
  | expr GT     expr
  | expr GEQ    expr
  | expr AND    expr
  | expr OR     expr
  | INTLIT
  | TRUE
  | FALSE
  | ID
  | STRINGLIT
  | LPAREN expr RPAREN
  | ID LPAREN actuals_opt RPAREN

/*** Variable Declarations ***/
```

```
vdecl →
   typ ID
 | typ ID ASSIGN expr

/*** Function Declarations ***/
fdecl →
  DEF typ ID LPAREN formals_opt RPAREN LBRACE stmt_list RBRACE

formals_opt →
    /* nothing */
  | formal_list

formal_list →
    typ ID
  | formal_list COMMA typ ID

actuals_opt →
    /* nothing */
  | actuals_list

actuals_list →
    expr
  | actuals_list COMMA expr
```

4. Project Plan

4.1 Planning Process

Our team met once a week after class on Mondays to work on the project and subsequently met with Freddy, our TA, on Tuesdays at the beginning of his office hours. Sometimes we scheduled extra meetings over the weekend if there was more work that we wanted to get done before our next meeting with Freddy. We would use our time on Mondays to set goals that we wanted to complete by our next meeting, work on tasks according to our project timeline, delegate tasks that we each needed to work on individually, and come up with questions that we could ask Freddy when we met with him the next day.

As soon as we solidified the members of our team, we created a Google Drive folder where we could store notes from all of our meetings, questions for Freddy, and all of our required documentation (the project proposal, language reference manual, etc.).

4.2 Specification Process

At first, we considered doing a poetry generation language or a language that would support server-based multiplayer games. However, after discussing these ideas with some of the TAs, we realized that our ideas were incredibly ambitious and would probably not be able to be completed within the timeframe of this class. We decided that we would try to build a text-based adventure game language but underestimated how long it would take to implement as many features as we had described in our Language Reference Manual. In the end, we scaled back to a general purpose programming language that combined Java-like syntax with Python-like top-level code.

4.3 Development Process

When we began code development, we would usually meet up and code together, especially since we were just starting to become familiar with OCaml. As we became more familiar with the language, we would delegate tasks to be done individually and come back together to discuss any issues that cropped up. Near the end of the process, we started pair programming and realized that this method of working helped increase our productivity significantly.

After we received feedback on our Language Reference Manual, we revised our grammar significantly. Then we worked on completing the front end of the compiler, mainly our scanner, parser, and AST. Once we successfully compiled Hello World, we began working on implementing function declarations, and then fleshed out other features such as operators, more robust printing functions, and control flow.

4.4 Testing Process

In the beginning of our testing process, we were focused on making sure our grammar was correct and unambiguous. We were able to check this using ocamlyacc to check for ambiguities and parsing particular strings using menhir. For example when we passed in the following tokens, we would receive the tree below which would be compared to the tree we created by hand.

```
ID LPAREN STRINGLIT RPAREN SEMI

ACCEPT
[program:
  [item_list:
    [item_list:
      [item:
        [stmt:
          [expr:
            [arith_expr:
              [arith_term:
                [atom:
                  ID
                  LPAREN
                  [actuals_opt:
                    [actuals_list:
                      [expr: [arith_expr: [arith_term: [atom:
STRINGLIT]]]]
                    ]
                  ]
                  RPAREN
                ]
              ]
            ]
          ]
          SEMI
        ]
      ]
    ]
```

Our test cases were written in BURGer and focused on both positive and negative testing. We had a separate folder that contained all of the test cases that could be executed by calling "make" and running "./testall.sh". Since we had a standard library in written in C, the testall script was modified to include linking the outside resources. Each of the positive test cases had a corresponding .out file which would be compared to the output of the .bun file using the print function. The negative test cases had a corresponding .err file that would come from checks created in semant.

4.5 Programming Style Guide

We all used the Atom text editor, which handily supported packages like language-ocaml and ocaml-indent. Those helped us more easily read syntax and use effective indentation.

Tab size: 2

Commenting convention:
All comments should be separated by a space from the opening and closing comment symbols. For example,
`/* comment */` or `(* comment *)`
Multi-line comments should be written as follows:

```
 /* beginning of comment    (* beginning of comment
  * more comment              * more comment
  * end of comment */   or    * end of comment *)
```

Parser rule convention:

```
   rule_name:
        pattern1 { action1 }
      | pattern2 { action2 }
```

The first pattern for each parser rule appears on the line below the rule name, tabbed and with two spaces before it. All patterns after that have a tab before the |, and a space before the pattern. Actions should align vertically with each other, meaning that for each parser rule, all actions will be left-aligned one space to the right of the lengthiest pattern in that rule.

When a rule accepts an empty string, the matching pattern should be denoted with a commented-out “nothing,” as follows:

```
   rule_name:
        /* nothing */ { [] }
      | pattern1      { action1 }
      | pattern2      { action2 }
```

AST type convention:

```
   type some_type =
        Name1 of type1
      | Name2 of type2
```

For AST types with multiple patterns, each name will begin with a capital letter. AST types with only one possible pattern can be written on one line, if desired.

OCaml coding convention:

Align Ocaml declarations and assignments as follows:

```
let example =
	(* example code *)
in
```

If a declaration is bound using the `in` keyword, it should align with the `let` keyword used to declare the function/variable. The body of functions should be indented with one tab. However, if the body of the assignment consists of only one line, it may occur on the same line as the declaration.

In some cases, the `in` may also occur on the same line as the body, for instance in a pattern-matching clause.

Use comments as necessary to explain specific portions of the code, and when raising exception messages, be as specific as possible.

4.6 Project Timeline

This was our projected timeline at the beginning of the project.

Dates	**Goals**
September 18-25	Assemble team members and develop language idea
September 26	Submit Proposal
October 16	Submit Language Reference Manual
November 7	Finish “Hello World”
November 21	Finalize language features (scanner, parser, AST) and add more to codegen, especially function declarations
November 28	Finish semantic checking and initial tests
December 5	Finish codegen and C standard library
December 12	Finish test suite
December 18-20	Present Final Presentation
December 20	Submit Final Report

4.7 Roles & Responsibilities

Team Member	**Role**
Jordan Lee	Manager
Jacqueline Kong	Language Guru
Adrian Traviezo	Systems Architect
Ashley Nguyen	Tester

The above table shows the assigned roles that we determined at the beginning of the project. Because of these assignments, we each did become more specialized in our respective areas. However, as we continued to develop our language, these roles became more fluid and our responsibilities were not necessarily restricted to our roles. This was especially evident after we started pair programming, which not only upped our productivity but also broadened our knowledge of different areas of our language.

4.8 Software Development Environment

We used LLVM version 5.0.0 and OCaml version 4.05.0. All of us had Mac OS X (different versions, however - Yosemite, Sierra, and High Sierra). We stored our project files in a git repository on GitHub and used Google Drive to store documentation (Proposal, LRM, meeting notes, etc.). Our text editor of choice was Atom, which supported some handy OCaml packages that assisted with syntax highlighting. This was especially helpful as we gradually learned to program effectively in OCaml.

4.9 Project Log

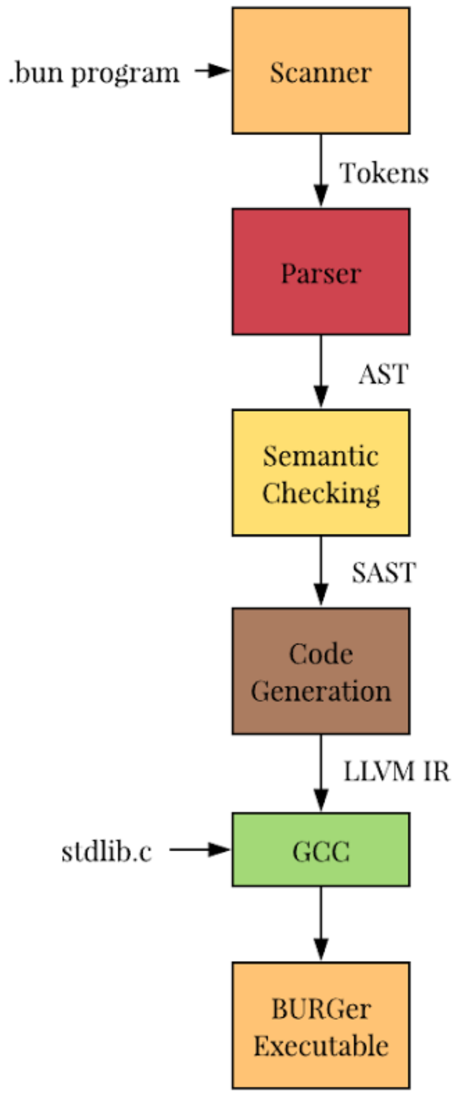
Dates	**Task**
September 18-25	Assembled team members and develop language idea
September 26	Submitted Proposal
October 15	Initial commit created in git repository
October 16	Submitted Language Reference Manual
October 27	Committed scanner and parser
November 10	Successfully compiled “Hello World”
November 25	Finalized language features (scanner, parser, AST) and flesh out codegen

November 28	Finish semantic checking
December 5	Finish codegen and C standard library
December 12	Finish test suite
December 18-20	Present Final Presentation
December 20	Submit Final Report

5. Architectural Design

5.1 Major Components

Here is a block diagram showing the major components of our compiler.



5.2 Interfaces Between Components

1. .bun program
    1.1. This is the source code for the program that is to be compiled in the BURGer language.
2. Scanner
    2.1. This component, defined in scanner.ml, handles the conversion of the input .bun file into a form that can be processed by the rest of the compilation process. The scanner generates a stream of tokens out of the text in the .bun file that will be parsed by the parser.
3. Parser
    3.1. This component, defined in parser.mly, processes the stream of tokens passed to it by the Scanner and creates an AST (Abstract Syntax Tree) as specified by the grammar rules written in it and the types in ast.ml.
4. Semantic Checker
    4.1. The semantic checker, located in semant.ml, opens the AST generated by the parser and ensures that the .bun file that is being processed contains a valid, semantically correct, program. It does so by performing type checking and raising errors when there are invalid parts of code that, for example, declare the same variable twice. After the AST is semantically checked, it is passed on to the code generation step.
5. Code Generation
    5.1. Once the .bun file is checked and it is verified that it contains a semantically correct program, the generated syntax tree is processed by codegen.ml. This file contains the code that converts types like ints and booleans to native LLVM primitives and builds LLVM blocks out of statements. LLVM Intermediate Representation is then generated to the burger.native file.
6. GCC
    6.1. This is the component that links the aforementioned files together. The GCC Compiler system is used to produce an output binary file and links the stdlib.c (Standard Library) file so that the built-in functions can be defined within the generated code.
7. BURGer Executable
    7.1. Once a .bun file successfully travels through the previous steps, a binary file is created by GCC and is ready to run. See section 2.2 for how to run this file.

5.3 Division of Labor

See Section 6.5: Division of Labor.

6. Test Plan

6.1 Source Code & their Target Language Programs

6.1.1 tests/test-hello.bun

```
print("Hello World!");
```

tests/test-hello.ll

```
; ModuleID = 'BURGer'
source_filename = "BURGer"

@fmt = private unnamed_addr constant [4 x i8] c"%d\0A\00"
@fmt.1 = private unnamed_addr constant [3 x i8] c"%d\00"
@str = private unnamed_addr constant [13 x i8] c"Hello World!\00"

declare i32 @print(i8*, ...)

declare i32 @printf(i8*, ...)

declare i32 @println(i8*, ...)

define i32 @main() {
entry:
  %print = call i32 (i8*, ...) @print(i8* getelementptr inbounds ([13
x i8], [13 x i8]* @str, i32 0, i32 0))
  ret i32 0
}
```

6.1.2 tests/test-assign1.bun

```
int x;
bool y;
x = 2;
y = true;
print("x = ");
println(x);
print("y = ");
println(y);
```

tests/test-assign2.ll

```
; ModuleID = 'BURGer'
source_filename = "BURGer"

@x = global i32 0
@y = global i1 false
@fmt = private unnamed_addr constant [4 x i8] c"%d\0A\00"
@fmt.1 = private unnamed_addr constant [3 x i8] c"%d\00"
@str = private unnamed_addr constant [5 x i8] c"x = \00"
@str.2 = private unnamed_addr constant [5 x i8] c"y = \00"

declare i32 @print(i8*, ...)

declare i32 @printf(i8*, ...)

declare i32 @println(i8*, ...)

define i32 @main() {
entry:
  store i32 2, i32* @x
  store i1 true, i1* @y
  %print = call i32 (i8*, ...) @print(i8* getelementptr inbounds ([5
x i8], [5 x i8]* @str, i32 0, i32 0))
  %x = load i32, i32* @x
  %print1 = call i32 (i8*, ...) @printf(i8* getelementptr inbounds
([4 x i8], [4 x i8]* @fmt, i32 0, i32 0), i32 %x)
  %print2 = call i32 (i8*, ...) @print(i8* getelementptr inbounds ([5
x i8], [5 x i8]* @str.2, i32 0, i32 0))
  %y = load i1, i1* @y
  %print3 = call i32 (i8*, ...) @printf(i8* getelementptr inbounds
([4 x i8], [4 x i8]* @fmt, i32 0, i32 0), i1 %y)
  ret i32 0
}
```

6.1.3 tests/test-fib.bun

```
def int fib(int x)
{
  if (x < 2) return 1;
  return fib(x-1) + fib(x-2);
}

println(fib(0));
println(fib(1));
println(fib(2));
println(fib(3));
println(fib(4));
println(fib(5));
println("Fibonacci test over!");
```

tests/test-fib.ll

```
; ModuleID = 'BURGer'
source_filename = "BURGer"

@fmt = private unnamed_addr constant [4 x i8] c"%d\0A\00"
@fmt.1 = private unnamed_addr constant [3 x i8] c"%d\00"
@str = private unnamed_addr constant [21 x i8] c"Fibonacci test
over!\00"
@fmt.2 = private unnamed_addr constant [4 x i8] c"%d\0A\00"
@fmt.3 = private unnamed_addr constant [3 x i8] c"%d\00"

declare i32 @print(i8*, ...)

declare i32 @printf(i8*, ...)

declare i32 @println(i8*, ...)

define i32 @main() {
entry:
  %fib_result = call i32 @fib(i32 0)
  %print = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([4
x i8], [4 x i8]* @fmt, i32 0, i32 0), i32 %fib_result)
  %fib_result1 = call i32 @fib(i32 1)
  %print2 = call i32 (i8*, ...) @printf(i8* getelementptr inbounds
([4 x i8], [4 x i8]* @fmt, i32 0, i32 0), i32 %fib_result1)
  %fib_result3 = call i32 @fib(i32 2)
  %print4 = call i32 (i8*, ...) @printf(i8* getelementptr inbounds
([4 x i8], [4 x i8]* @fmt, i32 0, i32 0), i32 %fib_result3)
  %fib_result5 = call i32 @fib(i32 3)
  %print6 = call i32 (i8*, ...) @printf(i8* getelementptr inbounds
([4 x i8], [4 x i8]* @fmt, i32 0, i32 0), i32 %fib_result5)
  %fib_result7 = call i32 @fib(i32 4)
  %print8 = call i32 (i8*, ...) @printf(i8* getelementptr inbounds
([4 x i8], [4 x i8]* @fmt, i32 0, i32 0), i32 %fib_result7)
  %fib_result9 = call i32 @fib(i32 5)
  %print10 = call i32 (i8*, ...) @printf(i8* getelementptr inbounds
([4 x i8], [4 x i8]* @fmt, i32 0, i32 0), i32 %fib_result9)
  %println = call i32 (i8*, ...) @println(i8* getelementptr inbounds
([21 x i8], [21 x i8]* @str, i32 0, i32 0))
  ret i32 0
}

define i32 @fib(i32 %x) {
entry:
  %x1 = alloca i32
  store i32 %x, i32* %x1
  %x2 = load i32, i32* %x1
  %tmp = icmp slt i32 %x2, 2
  br i1 %tmp, label %then, label %else

merge:                                            ; preds = %else
  %x3 = load i32, i32* %x1
  %tmp4 = sub i32 %x3, 1
  %fib_result = call i32 @fib(i32 %tmp4)
  %x5 = load i32, i32* %x1
  %tmp6 = sub i32 %x5, 2
  %fib_result7 = call i32 @fib(i32 %tmp6)
  %tmp8 = add i32 %fib_result, %fib_result7
  ret i32 %tmp8

then:                                             ; preds = %entry
  ret i32 1

else:                                             ; preds = %entry
  br label %merge
}
```

6.2 Test Suites Used

See Appendix, Section 8.12: Test Cases for the full list of test suites.

6.3 Significance of Chosen Test Cases

We tested for the syntax and the various features of our language and made sure that each new addition to our language would be compatible with the rest.

- **Assignment Operators**
  Assignment operators were used to test and make sure our primitive data types (`int`, `string`, `bool`, and `null`) were strongly typed as well as checking that variable initialization was possible.

- **Control Flow**
  Control flow statements were tested to make sure our if, if/else, and while statements worked. There were also negative test cases that would return an error if boolean values were not passed into the function.

- **Binary Operators**
  Binary operators were tested by comparing two values and making sure the correct response was returned by the file. Additionally, the arithmetic operators were tested by performing the calculations and making sure they matched with the expected output.

- **Unary Operators**
  Unary operators were tested by checking to see they followed their intended purposes by negating with (!) for logical expressions or (-) for numbers.

- **Built-in functions**
  Built-in functions such as print() and println() were also tested to make sure that print() would simply output a value, whereas println() would output a value appended with a new line. They were also checked in semant.ml so no one function could be defined using those names.

- **Comments**
  Comments were tested by adding them throughout the programs using the convention /* Testing Comments */ or (* Testing Comments *) to make sure it would not affect the code written

- **Top Level Code**
  Top level code was also checked by making sure programs were able to run without a main function and that a main function could not be declared in the file.

- **Function Calls**
  Function calls were tested to make sure the correct values were passed in and returned. The negative testing ensured that no null values would be passed into the function.

- **Static Scoping**
  Static scoping was tested by assigning variables globally and checking to make sure the correct output was given after it was changed in a function.

6.4 Test Automation

We used a script called `testall.sh` to automatically run all of our tests and create user-friendly output that would be easy to read. This was modified from the script used to test cases for MicroC.

The output when `./testall.sh` runs is as follows:

```
dyn-160-39-173-48:src jordanlee$ ./testall.sh
-n test-assign1...
OK
-n test-assign2...
OK
-n test-comm1...
OK
-n test-fib...
OK
-n test-func1...
```

```
OK
-n test-func2...
OK
-n test-func3...
OK
-n test-func4...
OK
-n test-func5...
OK
-n test-global1...
OK
-n test-hello...
OK
-n test-if2...
OK
-n test-if3...
OK
-n test-if4...
OK
-n test-if5...
OK
-n test-init1...
OK
-n test-init2...
OK
-n test-math1...
OK
-n test-math2...
OK
-n test-ops1...
OK
-n test-ops2...
OK
-n test-ops3...
OK
-n test-print...
OK
-n test-println...
OK
-n test-scope1...
OK
-n test-var1...
OK
-n test-var2...
OK
-n test-while1...
OK
-n test-while2...
OK
```

```
-n fail-assign1...
OK
-n fail-dup1...
OK
-n fail-expr1...
OK
-n fail-expr2...
OK
-n fail-func1...
OK
-n fail-func2...
OK
-n fail-func3...
OK
-n fail-func4...
OK
-n fail-func6...
OK
-n fail-func7...
OK
-n fail-func8...
OK
-n fail-if...
OK
-n fail-var1...
OK
-n fail-while1...
OK
```

6.5 Division of Labor

While BURGer's development team had clear roles that corresponded to our individual responsibilities, the bulk of the code was written and revised by every member of the group either simultaneously or within our own time. A few contributions that reflect the roles we set for ourselves, however, are listed below.

Jacqueline ensured that the grammar as defined in our parser was clear and consistent. Adrian integrated the stdlib.c file, which includes the built-in functions that comprise the standard library. Ashley, the assigned Tester, created most of the tests and set up the automated testing script. Jordan wrote the sample code used for our demo and contributed to the testing script as well, and managed the location and times of our group meetings. All members helped making the testing more rigorous and wrote some tests.

7. Lessons Learned

7.1 Jordan

Functional programming required me to learn a different, eye-opening way to solve problems. As someone so used to coding in imperative languages, it was hard for me to adjust to coding in a functional programming language like OCaml at first. However, I feel like I've come away with a better understanding of how to program more concisely. I also learned the importance of taking advantage of valuable resources such as the OCaml documentation and llvm.moe. Discovering and learning from such resources earlier on could have helped me in fulfilling my managerial role and setting goals for our team. By looking at previous final reports, I would have had a much better idea of what the scope of our project was and how to plan the work ahead of us.

7.2 Jacqueline

I had never thought too much about compilers before taking this class, and I have really enjoyed learning about the compiler pipeline and about functional programming in this course. Functional programming is a very different paradigm than I'm used to, and my CS Theory class didn't cover the Lambda Calculus, so this was definitely an eye-opening experience for me. I have learned a lot about solving problems functionally vs. imperatively. Additionally, the process of designing a language and implementing a compiler for it helped me learn a balance between preparing adequately to create a project and actually building the project. For instance, it's worth spending some time on setting up a workable development environment, but it might not be a good use of time to deliberate for too long on nitpicky design choices that will probably end up being altered anyway. I also realized the importance of having a meaningful test suite.

7.3 Adrian

Working on BURGer was perhaps the most educational group project I have experienced at Columbia. There were many smaller steps in learning to build the compiler that ended up teaching me more about the bigger picture. Learning how to program in OCaml made me a stronger programmer all around, since I learned to think about problems from a "functional" perspective. Understanding how to read assembly code became an invaluable asset in debugging. And, definitely, hands-on experience with things such as disambiguating grammars was extremely helpful when learning the material for this class, as well as for learning important topics about computer science in general.

However, the "bigger picture" I mentioned was probably even more valuable: by taking on a project so big and out of my comfort zone, I learned more about how to situate myself as part of a team and as part of a multi-step (and multi-domain) process. Because of BURGer, I gained experience with navigating the development of a complex product, and with learning to consider all the parts of that complex product both as discrete entities but also with respect to their importance to each other. I am therefore much more confident approaching not just compiler design, but projects that would require a similar holistic approach and group dynamics.

7.4 Ashley

Throughout this project and working on our language I learned a lot more about functional programming as well as working in a collaborative environment. Using OCaml and LLVM made the fixing issues a lot different since there is not as much documentation about it and I had to understand the entire process to debug the errors I was encountering. I also learned a lot about the importance of continuous integration testing since changing one thing could link to problems in other files that were never touched. It is definitely something I will be more aware of in the future as a way to fix errors as the arise making it easier to correct rather than having it possibly create even more problems as it continues. Using pair programming for the first time was also a very useful approach, having someone there to bounce ideas off of and help made things a lot more efficient than everyone working individually. From this I have a much better understanding about how to work with a group structure while learning a new language which I will be able to use in the future.

7.5 Advice for Future Teams

Perhaps the most obvious advice we could give here would be to start early. Even though our team had a good early start with defining our language (in terms of laying out our grammar, rules, and how we wanted to structure functions and statements), we could have done a little more research into how all the components of our compiler would integrate with each other earlier on. Do not expect to learn everything it will take to write your language from lectures; attending class and studying the slides and textbook will give you a good framework, but if you wait to read up on code generation until after the midterm, chances are you'll have some catching up to do later on. Make use of resources available to you, including MicroC and past projects (but approach everything with healthy criticism!), and try to really understand what's happening in the code holistically.

Another piece of advice we would like to give to future teams is to take time to read as much documentation about OCaml and LLVM as possible. This is paramount not only to learn about how to, say, use builders in LLVM or recursion in OCaml, but also to see whether there are functions that already exist in either that can cut the time you spend writing your own implementations. Making sure to be well-read on all the different steps of implementing a compiler is also extremely important—do not try to only be an expert on one aspect of the pipeline, as being knowledgeable about the entire process (from scanning to parsing to code generation to, possibly, a standard library) can help when making important design decisions.

8. Appendix with Code Listings

8.1 scanner.mll

```
(*
* Authors:
* Jordan Lee
* Jacqueline Kong
*)

{ open Parser }

let escape = '\\' ['\\' ''' '"' 'n' 'r' 't']
let ascii = ([' '-'!' '#'-'[' ']'-'~'])
let string = ('"') ( (ascii | escape)* as s) ('"')

rule token = parse
  [' ' '\t' '\r' '\n'] { token lexbuf }
  | "/*"                { comment lexbuf }
  | '('                 { LPAREN }
  | ')'                 { RPAREN }
  | ';'                 { SEMI }
  | string              { STRINGLIT(s) }
  | '{'                 { LBRACE }
  | '}'                 { RBRACE }
  | ','                 { COMMA }
  | '+'                 { PLUS }
  | '-'                 { MINUS }
  | '*'                 { TIMES }
  | '/'                 { DIVIDE }
  | '%'                 { MOD }
  | '='                 { ASSIGN }
  | "<"                 { LT }
  | ">"                 { GT }
  | "=="                { EQ }
  | "!="                { NEQ }
  | "<="                { LEQ }
  | ">="                { GEQ }
  | "&&"                { AND }
  | "||"                { OR }
  | "!"                 { NOT }
  | "if"                { IF }
  | "else"              { ELSE }
  | "while"             { WHILE }
  | "true"              { TRUE }
  | "false"             { FALSE }
  | "int"              { INT }
  | "string"           { STRING }
  | "bool"             { BOOL }
  | "null"             { NULL }
  | "return"           { RETURN }
  | "def"              { DEF }
  | ['0'-'9']+ as lxm  { INTLIT(int_of_string lxm) }
  | ['a'-'z' 'A'-'Z']['a'-'z' 'A'-'Z' '0'-'9' '_']*
               as lxm { ID(lxm) }
  | eof                { EOF }

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

8.2 parser.mly

```
/* Parser for BURGer Programming Language
 * PLT Fall 2017
 * Authors:
 * Jacqueline Kong
 * Jordan Lee
 * Adrian Traviezo
 * Ashley Nguyen */

%{ open Ast %}

%token <string> ID
%token <string> STRINGLIT
%token <int> INTLIT

%token LPAREN RPAREN SEMI LBRACE RBRACE COMMA
%token PLUS MINUS TIMES DIVIDE ASSIGN MOD
%token LT GT LEQ GEQ EQ NEQ
%token AND OR NOT
%token IF ELSE NOELSE
%token TRUE FALSE
%token INT STRING BOOL NULL
%token WHILE DEF RETURN
%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 MOD
%right NOT

%start program
%type <Ast.program> program

%%

/*** Top Level ***/

program:
   item_list EOF { List.rev $1 }

item_list:
   /*nothing */ { [] }
  | item_list item { ($2 :: $1) }

item:
    stmt      { Stmt($1) }
  | fdecl     { Function($1) }

typ:
    INT                    { Int }
  | BOOL                   { Bool }
  | STRING                 { String }
  | NULL                   { Null }

/*** Statements ***/

stmt:
    expr SEMI                                 { Expr($1) }
  | vdecl SEMI                                { $1 }
  | RETURN expr SEMI                          { Return($2) }
  | RETURN SEMI                               { Return(NoExpr) }
  | LBRACE stmt_list RBRACE                   { Block(List.rev $2) }
  | IF LPAREN expr RPAREN stmt %prec NOELSE { If($3, $5, Block([])) }
  | IF LPAREN expr RPAREN stmt ELSE stmt    { If($3, $5, $7) }
  | WHILE LPAREN expr RPAREN stmt           { While($3, $5) }

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

/*** Expressions ***/

expr:
    MINUS expr                        { Unop(Neg, $2) }
  | NOT expr                          { Unop(Not, $2) }
  | ID ASSIGN expr                    { Assign($1, $3) }
  | 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 MOD    expr                  { Binop($1, Mod,   $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) }
  | INTLIT                            { IntLit($1) }
  | TRUE                              { BoolLit(true) }
  | FALSE                             { BoolLit(false) }
  | ID                                { Id($1) }
  | STRINGLIT                         { StringLit($1) }
  | LPAREN expr RPAREN                { $2 }
  | ID LPAREN actuals_opt RPAREN      { Call($1, $3) }

/*** Variable Declarations ***/

vdecl:
   typ ID             { VDecl($1, $2) }
 | typ ID ASSIGN expr { VAssign(($1, $2), $4) }

/*** Function Declarations ***/

fdecl:
  DEF typ ID LPAREN formals_opt RPAREN LBRACE stmt_list RBRACE
    { { typ = $2;
        fname = $3;
        formals = $5;
        body = List.rev $8;
      } }

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

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

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

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

8.3 ast.ml

```
(* Abstract Syntax Tree
 * BURGer Programming Language
 * PLT Fall 2017
 * Authors:
 * Jacqueline Kong
 * Jordan Lee
 * Adrian Traviezo
 * Ashley Nguyen *)

(*** Syntax Types ***)

type op = Add | Sub | Mult | Div | Equal | Neq | Less | Leq | Greater
| Geq |
          And | Or | Mod

type uop = Neg | Not

type expr =
    Id of string
  | Call of string * expr list
  | IntLit of int
  | BoolLit of bool
  | StringLit of string
  | Assign of string * expr
  | Binop of expr * op * expr
  | Unop of uop * expr
  | NoExpr

type typ = Int | Bool | String | Null | Array of typ * expr

type bind = typ * string

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

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

type item =
      Stmt of stmt
    | Function of func_decl

type program = item list

(*** Functions for Printing ***)

let string_of_typ = function
    Int -> "int"
  | Bool -> "bool"
  | Null -> "null"
  | String -> "string"

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

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

let rec string_of_expr = function
    IntLit(l) -> string_of_int l
  | BoolLit(true) -> "true"
  | BoolLit(false) -> "false"
  | StringLit(s) -> s
  | 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, e) -> v ^ " = " ^ string_of_expr e
  | Call(f, el) ->
   f ^ "(" ^ String.concat ", " (List.map string_of_expr el) ^ ")"
  | NoExpr -> ""

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
  | While(e, s) -> "while (" ^ string_of_expr e ^ ") " ^
string_of_stmt s
```

8.4 semant.ml

```
(* Semantic Checker for the BURGer Programming Language
   PLT Fall 2017
   Authors:
   Adrian Traviezo
   Jacqueline Kong
   Ashley Nguyen
   Jordan Lee
*)

open Ast

module StringMap = Map.Make(String)

(* Semantic checking of a program. Returns void if successful,
   throws an exception if something is wrong.
   Check each global variable, then check each function *)

let check_program program =

  (* 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

  (* figure out which items are statements and make a list of
statements *)
  let stmt_list =
    let stmts_as_items =
      List.filter (fun x -> match x with
        Ast.Stmt(x) -> true
        | _ -> false) program
    in List.map (fun x -> match x with
        Ast.Stmt(x) -> x
        | _ -> failwith "stmt casting didn't work") stmts_as_items
  in

  (* after you figure out which items are statements, you need to go
through the statements
     and figure out which ones contain the variable declarations and
     variable decl+assignment statements *)
  let globals =
    let global_list = List.filter (fun x -> match x with
        Ast.VDecl(x) -> true
      | Ast.VAssign(x, _) -> true
      | _ -> false) stmt_list
    in List.map (fun x -> match x with
        Ast.VDecl(x) -> x
      | Ast.VAssign(x, _) -> x
      | _ -> failwith "not turned into global") global_list
  in

  let functions =
      let functions_as_items = List.filter (fun x -> match x with
          Ast.Function(x) -> true
        | _ -> false) program
      in
        let all_functions_as_items = functions_as_items
        in List.map (fun x -> match x with
            Ast.Function(x) -> x
          | _ -> failwith "function casting didn't work")
all_functions_as_items
  in

  (* let function_locals =
    let get_locals_from_fbody fdecl =
      let get_vdecl locals_list stmt = match stmt with
```

```
          Ast.VDecl(typ, string) -> (typ, string) :: locals_list
          | _ -> locals_list
      in
      List.fold_left get_vdecl [] fdecl.Ast.body
    in List.fold_left get_locals_from_fbody (List.hd functions)
(List.tl functions)
  in *)

  let symbols = List.fold_left (fun var_map (varType, varName) ->
StringMap.add varName varType var_map)
    StringMap.empty (globals)
  in

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

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

  let built_in_decls = StringMap.add "println"
      { typ = Null; fname = "println"; formals = []; body = [] }
      (StringMap.singleton "print"
    { typ = Null; fname = "print"; formals = [];
      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 -> raise (Failure ("unrecognized function " ^
s))
  in

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

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

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

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

    report_duplicate (fun n -> "duplicate function " ^ n)
      (List.map (fun fd -> fd.fname) functions);

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

    List.iter (check_not_void (fun n -> "illegal null formal " ^ n ^
      " in " ^ func.fname)) 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); *)
  in

  let rec expr = function
      IntLit _ -> Int
    | BoolLit _ -> Bool
    | StringLit _ -> String
    | Id s -> type_of_identifier s
    | 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))
    | Binop(e1, op, e2) as e -> let t1 = expr e1 and t2 = expr e2 in
    (match op with
        Add | Sub | Mult | Div | Mod when t1 = Int && t2 = Int -> Int
        | 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))
      )
    | Call(fname, actuals) as call -> let fd = function_decl fname in
      if (fname = "print" || fname = "println")
             then
                 let _ = List.iter (fun e -> ignore(expr e)) actuals
in Null
      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
           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 e))))
             fd.formals actuals;
         fd.typ)
    | Unop(op, e) as ex -> let t = expr e in
      (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)))
      | NoExpr -> Null
  in

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

    let rec check_stmt s = match s with
      Block sl -> let rec check_block = function
         [Return _ as x] -> check_stmt x
       | Return _ :: _ -> raise (Failure "nothing may follow a
return")
       | Block sl :: ss -> check_block (sl @ ss)
       | x :: ss -> check_stmt x ; check_block ss
       | [] -> ()
      in check_block sl
        | VDecl _ -> ()
        | VAssign ((typ, string), e) -> ignore (expr (Assign(string,
e)))
        | Expr e -> ignore (expr e)
        | If(p, b1, b2) -> check_bool_expr p; check_stmt b1;
check_stmt b2
        | While(p, s) -> check_bool_expr p; check_stmt s

  in

  (* Check for assignments and duplicate vdecls *)
  List.iter check_function functions;
  List.iter check_stmt stmt_list;
  (* List.iter stmt stmt_list; *)
  report_duplicate (fun n -> "Duplicate assignment for " ^ n)
(List.map snd globals);
```

8.5 codegen.ml

```
(*
 * Code generation for BURGer Programming Language
 * Authors:
 * Jordan Lee
 * Jacqueline Kong
 * Adrian Traviezo
 * Ashley Nguyen
 *)

module L = Llvm
module A = Ast

module StringMap = Map.Make(String)

let translate (program) =
  let context = L.global_context () in
    let the_module = L.create_module context "BURGer"
    and i8_t = L.i8_type context
    and str_t = L.pointer_type (L.i8_type context)
    and null_t = L.void_type context
    and i1_t   = L.i1_type context
    and i32_t  = L.i32_type context in

  (* types of variables in BURGer*)
  let ltype_of_typ = function
      A.String -> str_t
    | A.Null -> null_t
    | A.Int -> i32_t
    | A.Bool -> i1_t
  in

  (* isolate list of items that match as statements and then form a
list of statements *)
  let stmt_list =
    let stmts_as_items =
      List.filter (fun x -> match x with
        A.Stmt(x) -> true
        | _ -> false) program
    in List.map (fun x -> match x with
        A.Stmt(x) -> x
        | _ -> failwith "stmt casting didn't work") stmts_as_items
  in

  (*after you figure out which items are statements, you need to go
through the statements
    and figure out which ones contain the variable declarations *)
  let globals =
    let global_list = List.filter (fun x -> match x with
        A.VDecl(x) -> true
      | A.VAssign(x, _) -> true
      | _ -> false) stmt_list
    in List.map (fun x -> match x with
        A.VDecl(x) -> x
      | A.VAssign(x, _) -> x
      | _ -> failwith "not turned into global") global_list
  in

  (* isolate list of statements that are NOT variable declarations *)
  let not_globals_list = List.filter (fun x -> match x with
    A.VDecl(x) -> false
  | _ -> true) stmt_list in

  (* from list of items in program, form list of functions from items
and
  build the main function *)
  let functions =
    (* generating the hidden main function *)
    let fdecl_main = A.Function({
           typ = A.Int;
           fname = "main";
           formals = [];
           body = List.rev(A.Return(A.IntLit(0)) ::
List.rev(not_globals_list))
         })
    in
    (* filtering out items that match as functions *)
      let functions_as_items = List.filter (fun x -> match x with
          A.Function(x) -> true
        | _ -> false) program
      in
    let all_functions_as_items = fdecl_main :: functions_as_items
    in List.map (fun x -> match x with
        A.Function(x) -> x
      | _ -> failwith "function casting didn't work")
all_functions_as_items
  in

  (* Store the global variables in a string map *)
  let global_vars =
    let global_var map (t, n) =
      if (ltype_of_typ t = str_t)
      then (
        let init = L.const_null str_t in
        StringMap.add n (L.define_global n init the_module) map
      )
      else (
        let init = L.const_int (ltype_of_typ t) 0
        in StringMap.add n (L.define_global n init the_module) map
      )
    in
    List.fold_left global_var StringMap.empty globals in

  (* printf() declaration *)
  let print_t = L.var_arg_function_type i32_t [| L.pointer_type i8_t
|] in
  let print_func = L.declare_function "print" print_t the_module in

  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

  let println_t = L.var_arg_function_type i32_t [| L.pointer_type
i8_t |] in
  let println_func = L.declare_function "println" println_t
the_module in

  (* Define each function (arguments and return type) so we can call
it *)
  let function_decls =
    let function_decl map 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.typ)
formal_types in
      StringMap.add name (L.define_function name ftype the_module,
fdecl) map
    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 int_format_str_ln = L.build_global_stringptr "%d\n" "fmt"
builder in
    let int_format_str = L.build_global_stringptr "%d" "fmt" builder
in

    let local_vars =
      let add_formal var_map (formal_type, formal_name) param =
L.set_value_name formal_name param;
           let local = L.build_alloca (ltype_of_typ formal_type)
formal_name builder in
           ignore (L.build_store param local builder);
        StringMap.add formal_name local var_map
      in

      let add_local map (formal_type, formal_name) =
        let local_var = L.build_alloca (ltype_of_typ formal_type)
formal_name builder in
        StringMap.add formal_name local_var map
      in

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

      let function_locals =
        let get_locals_from_fbody function_body =
          let get_vdecl locals_list stmt = match stmt with
              A.VDecl(typ, string) -> (typ, string) :: locals_list
            | A.VAssign((typ, string), _) -> if (fdecl.A.fname =
"main")
                then
                  locals_list
                else
```

```
                   (typ, string) :: locals_list
               | _ -> locals_list
           in
           List.fold_left get_vdecl [] function_body
         in get_locals_from_fbody fdecl.A.body
       in List.fold_left add_local formals function_locals
     in

   let lookup n = try StringMap.find n local_vars
                  with Not_found -> StringMap.find n global_vars
   in

   (* generate code for different kinds of expressions *)
   let rec expr builder = function
     A.StringLit s -> L.build_global_stringptr s "str" builder
   | A.IntLit i -> L.const_int i32_t i
   | A.BoolLit b -> L.const_int i1_t (if b then 1 else 0)
   | A.NoExpr -> L.const_int i32_t 0
   | 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 in
         (match op with
          A.Add     -> L.build_add
         | A.Sub     -> L.build_sub
         | A.Mult    -> L.build_mul
        | A.Div     -> L.build_sdiv
        | A.Mod     -> L.build_srem
         | 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
   | A.Unop(op, e) ->
        let e' = expr builder e in
         (match op with
           A.Neg     -> L.build_neg
        | A.Not     -> L.build_not)
        e' "tmp" builder
   | A.Assign (s, e) -> let e' = expr builder e in
     ignore (L.build_store e' (lookup s) builder); e'
   | A.Call ("print", [s]) ->
     let test = s in (match s with
           A.StringLit test -> L.build_call print_func [| (expr
builder s) |] "print" builder
```

```
        | A.Id test ->
          let ptr_32 = L.pointer_type i32_t
          and ptr_bool = L.pointer_type i1_t in
          let test_type = L.type_of (lookup test) in
          if ((test_type = ptr_32) || (test_type = ptr_bool)) then
            L.build_call printf_func [| int_format_str ; (expr
builder s) |] "printf" builder
          else L.build_call print_func [| (expr builder s) |] "print"
builder
        | _ -> L.build_call printf_func [| int_format_str ; (expr
builder s) |] "printf" builder
      )
  | A.Call("println", [s]) ->
    let test = s in (match s with
          A.StringLit test -> L.build_call println_func [| (expr
builder s) |] "println" builder
        | A.Id test ->
          let ptr_32 = L.pointer_type i32_t
          and ptr_bool = L.pointer_type i1_t in
          let test_type = L.type_of (lookup test) in
          if ((test_type = ptr_32) || (test_type = ptr_bool)) then
            L.build_call printf_func [| int_format_str_ln ; (expr
builder s) |] "print" builder
          else L.build_call println_func [| (expr builder s) |]
"println" builder
        | _ -> L.build_call printf_func [| int_format_str_ln ; (expr
builder s) |] "print" 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.typ with A.Null -> ""
                                          | _ -> 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

    (* generate code for different kinds of statements *)
    let rec stmt builder = function
      A.Block sl -> List.fold_left stmt builder sl
    | A.Expr e -> ignore(expr builder e); builder
    | A.VDecl (typ, string) -> builder
    | A.VAssign ((typ, string), e) -> ignore(expr builder
(A.Assign(string, e))); builder
    | A.Return e -> ignore (match fdecl.A.typ with
        A.Null -> 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
  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.typ with
        A.Null -> 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
```

8.6 burger.ml

```
(* Top-Level of the BURGer Programming Language
* Author:
*   Jacqueline Kong
*)

let _ =
  let lexbuf = Lexing.from_channel stdin in
  let ast = Parser.program Scanner.token lexbuf in
  Semant.check_program ast;
  let m = Codegen.translate ast in
  Llvm_analysis.assert_valid_module m;
  print_string (Llvm.string_of_llmodule m)
```

8.7 stdlib.c

```
/* Standard Library for BURGer Programming Language
PLT Fall 2017
Includes 2 print functions (print and println)
Author: Adrian Traviezo */

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


// Prints given string and accepts escape characters
void print(char *s){
  char c;
  int i = 0;
  while (i < strlen(s)){
    c = s[i];
    if (c == '\\'){
      if (s[i+1] == 'n')
        printf("\n");
      if (s[i+1] == 't')
        printf("\t");
      i++;
    }
    else if (c == '\\' && s[i+1] == 't'){
      printf("\t");
      i++;
    }
    else
      printf("%c", c);
    i++;
  }
}

// as above, but with a newline at end
void println(char *s){
  char c;
  int i = 0;
  while (i < strlen(s)){
    c = s[i];
    if (c == '\\'){
      if (s[i+1] == 'n')
        printf("\n");
      if (s[i+1] == 't')
        printf("\t");
      i++;
    }
    else
      printf("%c", c);
    i++;
  }
  printf("\n");
}
```

8.8 demo.bun (sample code)

```
/* Author: Jordan Lee */


def int countFactors(int num) {
  int current_num = num;
  int count = 1;
  int number_of_factors = 1;

  while (count <= current_num) {
    if (current_num % count == 0 && count != 1) {
      count = count + 1;
      number_of_factors = number_of_factors + 1;
    } else {
        count = count + 1;
    }
  }
  return number_of_factors;
}
```

```
def null printFactor(int count, int current_num) {
  print("This is a factor of ");
  print(current_num);
  print(": ");
  println(count);
  count = count + 1;
}

def bool checkPrime(int num) {
  int current_num = num;
  int count = 1;
  bool isPrime = true;

  while (count < current_num) {
    if (current_num % count == 0 && count != 1) {
      printFactor(count, current_num);
      count = count + 1;
      isPrime = false;
    } else {
        count = count + 1;
    }
    if (count == current_num) {
      return isPrime;
    }
  }
  return isPrime;
}

def null printResults(bool check, int num) {
  if (check == false) {
    print("CONCLUSION: ");
    print(num);
    print(" has ");
    print(countFactors(num));
    println(" factors including itself and 1. It is not a prime
number!");
  } else {
    print("CONCLUSION: ");
    print(num);
    println(" is a prime number!");
  }
}

printResults(checkPrime(120), 120);
printResults(checkPrime(57), 57);
```

8.9 Makefile

```
# Make sure ocamlbuild can find opam-managed packages: first run
#
# eval `opam config env`

# Easiest way to build: using ocamlbuild, which in turn uses
ocamlfind
# Authors: Jordan Lee and Jacqueline Kong

.PHONY : all
all : clean burger.native

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

# "make clean" removes all generated files

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

.PHONY : parser
parser :
	ocamlyacc -v parser.mly

.PHONY : menhir
menhir:
	menhir --interpret --interpret-show-cst parser.mly

# More detailed: build using ocamlc/ocamlopt + ocamlfind to locate
LLVM

OBJS = ast.cmx codegen.cmx parser.cmx semant.cmx scanner.cmx
burger.cmx

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

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 $<

ast.cmo :
ast.cmx :
codegen.cmo : ast.cmo
codegen.cmx : ast.cmx
microc.cmo : semant.cmo scanner.cmo parser.cmi codegen.cmo ast.cmo
microc.cmx : semant.cmx scanner.cmx parser.cmx codegen.cmx ast.cmx
parser.cmo : ast.cmo parser.cmi
parser.cmx : ast.cmx parser.cmi
scanner.cmo : parser.cmi
scanner.cmx : parser.cmx
semant.cmo : ast.cmo
semant.cmx : ast.cmx
parser.cmi : ast.cmo
```

8.10 burgr.sh (compiling + runtime script)

```
#!/bin/bash
# usage: "./burgr.sh file-name"
# do not append with .bun
# Authors: Jordan Lee and Jacqueline Kong

echo "------compiling object file------"
gcc -c -Wall stdlib.c
echo "------running burger.native------"
./burger.native < $1.bun > $1.ll
echo "------llc command------"
llc $1.ll
echo "------linking files------"
gcc -o $1 $1.s stdlib.o
echo "------running executable------"
./$1
```

8.11 burgr-c.sh (compiling script)

```
#!/bin/bash
# usage: "./burgr.sh file-name"
# do not append with .bun
# Author: Jordan Lee

echo "------compiling object file------"
gcc -c -Wall stdlib.c
echo "------running burger.native------"
./burger.native < $1.bun > $1.ll
echo "------llc command------"
llc $1.ll
```

8.12 testall.sh (automated testing script)

```
#!/bin/sh

# Regression testing script for BURGer(Modified from 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
# Authors: Ashley Nguyen and Jordan Lee

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

# Path to the LLVM compiler
LLC="llc"

# Path to the C compiler
CC="gcc"

# Path to the microc compiler.  Usually "./microc.native"
# Try "_build/microc.native" if ocamlbuild was unable to create a
symbolic link.
BURGER="./burger.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] [.bun 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/.bun//'`
    reffile=`echo $1 | sed 's/.bun$//'`
    basedir="`echo $1 | sed 's/\/[^\/]*$//'`/."

    echo -n "$basename..."

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

    generatedfiles=""

    generatedfiles="$generatedfiles ${basename}.ll ${basename}.out"
&&
    Run "$CC" "-c -Wall" "stdlib.c" &&
    Run "$BURGER" "<" $1 ">" "${basename}.ll" &&
    Run "$LLC" "${basename}.ll" &&
   Run "$CC" "-o" "${basename}" "${basename}.s" "stdlib.o" &&
   Run "./${basename}" > "${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/.bun//'`
    reffile=`echo $1 | sed 's/.bun$//'`
    basedir="`echo $1 | sed 's/\/[^\/]*$//'`/."

    echo -n "$basename..."
    echo 1>&2
    echo "###### Testing $basename" 1>&2

    generatedfiles=""

    generatedfiles="$generatedfiles ${basename}.err ${basename}.diff"
&&
    RunFail "$BURGER" "<" $1 "2>" "${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 [ $# -ge 1 ]
```

```
then
    files=$@
else
    files="tests/test-*.bun tests/fail-*.bun"
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
```

8.12 Test Cases

fail-assign1.bun

```
int x;
x = true;
```

Fatal error: exception Failure("illegal assignment int = bool in x = true")

fail-dup1.bun

```
int x;
int x;
```

Fatal error: exception Failure("Duplicate assignment for x")

fail-expr1.bun

```
bool a = true;
bool b;
a = true;
b = false;
bool c = a+b;
```

Fatal error: exception Failure("illegal binary operator bool + bool in a + b")

fail-expr2.bun

```
int a;
a = 4;
bool b = true;

if(a == b){
  print("wrong");
}
```

Fatal error: exception Failure("illegal binary operator int == bool in a == b")

fail-func1.bun

```
def int check(int a){
  return a;
}

def int check(int b){
  return b;
}
```

Fatal error: exception Failure("duplicate function check")

fail-func2.bun

```
def int check(int a, int b){
     return a;
}
check(1, 2, 3);
```

Fatal error: exception Failure("expecting 2 arguments in check(1, 2, 3)")

fail-func3.bun

```
def int check(int a, int a){
     return a;
}
```

Fatal error: exception Failure("duplicate formal a in check")

fail-func4.bun

```
def null print(){
     print("hi");
}
```

Fatal error: exception Failure("function print may not be defined")

fail-func6.bun

```
def null main(){
     print("hi");
}
```

Fatal error: exception Failure("function main may not be defined")

fail-func7.bun

```
def null println(){
     print("wtf");
}
```

Fatal error: exception Failure("function println may not be defined")

fail-func8.bun

```
def null what(null x){
     print("wtf");
}
```

Fatal error: exception Failure("illegal null formal x in what")

fail-if.bun

```
if(43){
     print("fail");
}
```

Fatal error: exception Failure("expected Boolean expression in 43")

fail-var1.bun

```
int x;
print(x);
int x;
```

Fatal error: exception Failure("Duplicate declaration or assignment for x")

fail-while1.bun

```
while(41){
     print("what");
}
```

Fatal error: exception Failure("expected Boolean expression in 41")

test-assign1.bun

```
int x = 2;
bool y = true;
print("x = ");
println(x);
print("y = ");
println(y);
```

```
x = 2
y = 1
```

test-assign2.bun

```
string butterfly = "monarch";
bool pretty = true;
print(butterfly);
println(" butterflies are pretty if this prints 1: ");
println(pretty);
```

```
monarch butterflies are pretty if this prints 1:
1
```

test-comm1.bun

```
def null test(){
      /*commenting stuff*/
      print("ya\n");
}
test();
```

```
ya
```

test-fib.bun

```
def int fib(int x)
{
  if (x < 2) return 1;
  return fib(x-1) + fib(x-2);
}

println(fib(0));
println(fib(1));
println(fib(2));
println(fib(3));
println(fib(4));
println(fib(5));
println("Fibonacci test over!");
```

```
1
1
2
3
5
8
Fibonacci test over!
```

test-func1.bun

```
def int add(int a, int b) {
      println("add numbers!");
      return a + b;
}
println(add(1, 3));
```

```
add numbers!
4
```

test-func2.bun

```
def null testPrint(int a, bool c, string d){
      println(a);
      println(c);
      println(d);
}
testPrint(4, false, "hello");
```

```
4
0
hello
```

test-func3.bun

```
def null add(int a, int b){
      int r;
      r = a+b;
      println(r);

      r = a-b;
      println(r);

      r = a*b;
      println(r);

      if(a>b){
            println("works");
      }
      if(a<b){
            println("wrong");
      }
      if(a>=b){
            println("yes");
      }
      if(a<=b){
            println("no");
      }

      if(a!=b){
       println("yas");
      }

}

add(46, 32);
```

```
78
14
1472
works
yes
yas
```

test-func4.bun

```
def string ret1(string s){
      return s;
}

string t;
t = ret1("omg");
println(t);

def int ret4(int i){
      return i;
}
int a;
a = ret4(5);
println(a);
```

```
omg
5
```

test-func5.bun

```
def int check(int a){
      return a;
      int x;
}

print(check(5));
```

```
5
```

test-global1.bun

```
int a;
int b;
a = 2;
b = 4;

def int inc(int a){
      a = a + 1;
      return a;
}

println(a);
println(b);
a = inc(a);
b = inc(b);
println(a);
println(b);
```

```
2
4
3
5
```

test-hello.bun

```
print("Hello World!");
```

```
Hello World!
```

test-if2.bun

```
if(true){
      int x;
      int y;
      println("fff");
      }

if (1<3){
      println("YES");
      println("HI");
}
else {
        println("NO");
}
```

fff
YES
HI

test-if3.bun

```
if(true){
        println("hi");
        if(true){
                println("idk");
        }
        if(false){
                print("what");
        }
}
```

hi
idk

test-if4.bun

```
if(true){
        print("what");
}
else{
        print("wtf");
}
```

what

test-if5.bun

```
if(false){
        print("no");
}
else if(true){
        print("yes");
}
else {
        println("why");
}

if(false){
        print("no");
}
else if(false){
     print("yes");
}
else {
     println("why");
}
```

yeswhy

test-init1.bun

```
int a;
print(a);
```

0

test-init2.bun

```
bool a;
print(a);
```

0

test-math1.bun

```
int a;
a = 2*7+1;
println(a);


int b;
b = 2*(7+1);
println(b);

int c;
c = a+b;
println(c);
```

15

16

31

test-math2.bun

```
def int math(int a, int b, int c){
     int r = a*(b+c);
     return r;
}
int x = math(3, 2, 4);
print(x);
```

```
18
```

test-ops1.bun

```
print(1+2);
print(2-1);
print(3/1);
print(15%4);
println(2*2);
```

```
31334
```

test-ops2.bun

```
println(1==1);
print(1==2);
print(1<=1);
print(1<=2);
print(2<=1);
print(2<2);
print(2<1);
print(1<2);
print(2>2);
print(2>1);
print(1>2);
print(1>=2);
print(2>=2);
print(1!=2);
println(2>=1);
```

```
1
01100010100111
```

test-ops3.bun

```
print(true);
print(false);
print(true && true);
print(true && false);
print(false && true);
print(false && false);
print(true || true);
print(true || false);
print(false || true);
print(false || false);
print(!true);
print(!false);
```

```
101000111001
```

test-print.bun

```
print("Hello World!");
print("5+3");
print("!@#$%^&*()-=_+,./<>?~``[]{}|:;\t");
print(5*6);
print("\n");
```

```
Hello World!5+3!@#$%^&*()-=_+,./<>?~``[]{}|:;	30
```

test-println.bun

```
println(3);
println("hellooo");
println("sooooooooo hmm");
println(true);
```

```
3
hellooo
sooooooooo hmm
1
```

test-scope1.bun

```
int b;
b = 5;
def int foo(){
      int a;
      a = b+5;
      return a;
}

def int bar(){
      int b;
      b = 2;
      return foo();
}

println(foo());
println(bar());
```

```
10
10
```

test-var1.bun

```
string bucket = "water";
println(bucket);
```

water

test-var2.bun

```
int x = 5;
print(x);
```

5

test-while1.bun

```
int i;
i = 0;
while(i<5){
      println("yum");
      println("burgers");
      i = i+1;
}
```

yum
burgers
yum
burgers
yum
burgers
yum
burgers
yum
burgers

test-while2.bun

```
def int idk(int a){
      int i = 0;
      int j = 0;
      while(i<a){
            i = i+1;
            j = j+2;
      }
      return j;
}
print(idk(5));
```

10