The *simpliCty* Compiler

Course: COMS 4115
Professor: Dr. Stephen A. Edwards
TA: Graham Gobieski
Date: Thursday August 11, 2016

Alphabetical Author List:

- Rui Gu, rg2970

- Adam Hadar, anh2130

- Zachary Moffitt, znm2104

- Suzanna Schmeelk, ss4648

Table of Contents

1. Introduction (by Team)

The simpliCty language is a simplified version of C, which contains a subset of C grammar with a strict type system and uses LLVM as the backend to produce bytecode. One of the major distinctions SimpliCty has from ANSI C is that pointers are not supported in the front end. Our language operates best with programs that need to be Turing complete and can be defined using strict type casting and only stack-based memory management, which decreases runtime errors. The C language domain is for number crunching and embedded systems. Our programs will operate in the same domain as C with the exception that our compiler supports only a limited features of C. The following sub-sections further explain the main points of our compiler.

1.1 C Grammar Subset

Primitive types are supported (i.e., int, float, bool, char). Types implicitly cast up (bool to int, char to int, int to float), but casting down is not supported. Structs and multidimensional arrays are supported (but structs cannot contain structs). Our compiler supports C control-flow statements: ‘while’, ‘for’, ‘if-else’, ‘return’, ‘break’, and ‘continue.’ Function calls and recursion are supported. Nested functions are not allowed. The language supports both globally scoped variables, and locally (within a function) scoped variables. The language implements memory on the stack, and does not support heap memory allocations.

1.2 Compiler Implementation

We built our own compiler frontend using OCaml. The generated AST will be hooked with LLVM APIs to construction LLVM IR. We then utilize LLVM to generate machine dependent assembly code. The compiler solely relies on the stack for memory management. Variables exist only within the scope of the function where the variables are created which requires the developer to handle the return value. If a variable is declared without a value assigned to it, is is assigned the value zero. By definition; when the function ends, all the allocated memory will be reclaimed.

2. Language Tutorial (by Team)

This section explains how to use our language. First we give a short explanation for our language and then we provide some example programs.

2.1. Source Code Tutorial

SimpliCty is wildly similar to C, in that a program is made up of definitions of either functions or variables. The first function run after compilation is the function called “main” in the program. Function can be “called” within other functions, and this is actually the only way the a function other than main will process its code. Variables are declared so the user can write programs that modify the values of those variables in order to reach some end goal.
Functions are declared by specifying the type (if any) that they return, the set of inputs (if any) that the function requires to run, and the set of instructions that are processed when the function is called. The main function can return an input, but its value will be ignored by the computer.
Variables are declared by specifying their type, name, and an optional initial value. Variables that are defined within a function can only be accessed within that same function (no other function can read or modify their values) unless they are passed as parameters to other functions. Parameters are normally passed by value to other functions, which means that a copy of that variable is generated for that function. Arrays can be either passed by value (a copy is generated), or passed by reference (the original array will be accessed).

2.2. Source Code Examples

This source code examples show features of our language.

2.2.1. Example: Hello World

The following three examples are three different ways to print “Hello World.” Example 1

2.2.1.1. Hello World with Chars

```
void main()                                /* control flow always begins at main() */
{
    char h = ‘h’;
    char e = ‘e’;
    char l = ‘l’;
    char o = ‘o’;
    char s = ‘ ’;
    char w = ‘w’;
    char r = ‘r’;
    char d = ‘d’;
    char n = ‘\n’;
    putchar(h);
    putchar(e);
    putchar(l);
    putchar(l);
    putchar(o);
    putchar(s);
    putchar(w);
    putchar(o);
    putchar(r);
    putchar(l);
    putchar(d);
    putchar(n);
    return;
}
```

```
Output:
> ./simplicty helloworld1.sct
hello world
>
```

2.2.1.1. Hello World with Arrays

```
int main()
{
    char[12] hello = {|‘h’,’e’,’l’,’l’,’o’,’ ’,’w’,’o’,’r’,’l’,’d’,’\n’|};
    int i;
    for(i = 0; i < 12; i++)
    {
        putchar(hello[i]);
    }
    return 0;
}
```

```
Output:
> ./simplicty helloworld2.sct
hello world
>
```

2.2.1.1. Hello World with Strings

```
void prints(string str)
{
    int ptr;
    while(str[ptr] != ‘\0’)
        putchar(str[ptr++]);
    putchar(‘\n’);
    return;
}

int main()
{
    string hello = “hello world”;
    prints(hello);
    return 0;

}
```

```
Output:
> ./simplicty helloworld3.sct
hello world
>
```

2.2.2. Example: *Fibonacci.sct*

```
extern prints(string s);                        /* includes a way to print vars */
extern printi(int i);
int main ()                                     /* control flow begins at main */
{
    int n;                                      /* must init var on separate lines */
    int first = 0;
    int second = 1;
    int next;
    int c;
    string inputReq = “Enter number of terms:”;
    prints(inputReq);
    n = getchar();
    prints(“The first”);
    printi(n);
    prints(”terms of Fibonacci:”);
    printi(first);
    printi(second);
    for (i = 0; i < n; i++)
    {
        next = first + second;
        first = second;
        second = next;
        printi(next);
    }
    return 0;
}
```

```
Output:
> ./simplicty fibonacci.sct > fib.ll
> lli fib.ll
Enter number of terms:
>8
The first
8
terms of Fibonacci:
0
1
1
2
3
5
8
11
>
```

3. Language Reference Manual

The simpliCity language is a simplified version of C which was developed by Kernighan and Ritchie. The simpliCity language consists of a subset of C grammar.

3.1 Lexical Conventions

3.1.1. Identifiers

An identifier is any sequence of alphanumeric characters, where the first character must be alphabetic. The `_` character is the only non-alphanumeric symbol accepted in an identifier, and it is read as an alphabetic character, but it cannot be the first character for an identifier.

3.1.2. Comments

Comments are introduced with the `/*` character string, and terminate with the `*/` character string. All characters within these indicators are ignored. Comments do not nest.

3.1.3. Whitespace

Whitespace is ignored by the compiler; any combination of whitespace characters will be interpreted as one whitespace character.

3.1.4. Keywords

The following identifier are reserved for various uses, and cannot be used in any other way than how they are specified (later in this manual):

- `int`
- `char`
- `float`
- `string`
- `bool`
- `true`
- `false`
- `void`
- `struct`
- `main`
- `return`
- `break`
- `continue`
- `if`
- `else`
- `while`
- `for`
- `putchar`

- getchar

3.2. Constants

3.2.1. Integer constants

An integer constant is a sequence of digits. It is taken to be a decimal number. It can have a + or – character preceding it to indicate its sign, but is assumed positive if not specified.

3.2.2. Character constants

A character constant is 1 or 2 characters enclosed in single quotes, ` and ′. To represent a single quote character as a character constant, it must be preceded by a backslash, e.g. \′. The backslash character is used as an escape for several other special character constants, as shown in this table:

Backslash	\\
Single quote	\′
New line	\n
End of string / null byte	\0

3.2.3. Floating constants

A floating constant contains an integer part, a fractional part, and an exponent part. The integer part recognized as a series of decimal numbers. It is followed by either the fractional, exponent part, or the decimal point (the . character). If the fractional part follows the integer part, the decimal point must separate them. The fractional part can also be optionally followed by the exponent part. The exponent part must begin with the alphabetic character e, and is then followed by an optional sign (+ or –) followed by a series of decimal numbers.

Many of the parts of a floating constant are optional, but there are some specific rules with excluding parts. The integer part can be written alone, but it must be followed by the decimal point to be considered a floating constant. The fractional parts can be written alone, but it must be preceded by the decimal point to be considered a floating constant. If the exponent part is present, either/both of the integer part and the fractional part must be present. The entire constant may be preceded by (+ or –) to indicate sign, but it is assumed positive if none are present.

3.2.4. Strings

A string is a set of characters surrounded by double quotes, `“` and `”`. A string is considered in the back end as an array of characters, which is held in memory as a contiguous block of data. To represent the double-quote character within a string, it must be preceded with the escape character as specified for character constants, e.g. `\”`. The other special characters specified there should be written in the same method (however the escape sequence for the single quote is not need/supported for strings).

3.3. Objects, types, and conversion

3.3.1. Fundamental types

SimpliCty supports four fundamental types - integers, characters, single-precision floating-point numbers, and booleans:

- characters (from here on labelled `char`), are representative of the ASCII character set; for stack alignment we transform all `char` to `int`.
- integers (`int`) are 32 bit (4 byte) numbers.
- single-precision numbers (`float`) are 32 bit (4 byte) numbers represented with 24 bits of precision, 8 bits for an exponent, and 1 bit for a sign.
- booleans (`bool`) are 1 bit representations of true or false statements. The reserved keywords `true` and `false` are its two possible values.

3.3.2. Derived types

There are three types which can be constructed from the fundamental types:

- arrays - a set of objects that are all the same type. A string is a type of array (an array of `char`) which has the special label `string`.
- structures - a set of objects that may not be all the same type
- functions - a subroutine that returns an object of a specific type

These derived types are generally recursive. There can be an array of arrays, an array of structures, a structure of arrays, a structures of structures, and a function can return an array or a structure. Functions cannot return functions.

3.3.3. ‘lvalues’

‘lvalues’ are expressions that refer to objects. An identifier is an lvalue. When code is being parsed, expressions like `a = b` or `a = 3` will interpret `a` as being an lvalue. `b` is also an lvalue in this example.

3.3.4. Casting and Conversions

SimpliCty is a strongly typed language, and so it does very little type conversion. Any attempt to perform a direct type casting will result in a compiler error. There is built in casting for two exceptions for this rule to make the mathematical operators work across `char`, `int` and `float`. The built in casting exception are `char` to `int` and `int` to `float`. A cast between `char` and `int` and a `int` and a `float` does not affect the variable size. They are all represented by a 4-byte value.

3.4. Expressions

Precedence of expressions is represented in this manual in the order of their section numbers. For example, all expressions from subsection 3 (unary operators) will always take precedence over any expression in subsection 4 (multiplicative operators), or subsection 5 (additive operators). It will always be ignored before expressions from subsection 2 (primary expressions), however.
Expressions defined within the same subsection have undefined precedence over each other. The compiler will arbitrarily compute them, in whatever order.

3.4.1. Syntax notation

Syntactic categories are indicated by text in *the font Cambria, and in italics*. Literal words or symbols are written in `Courier New, with a light gray background`The subscript characters $_{opt}$ mark a category that is optional.

3.4.2. Primary expressions

Primary expressions group left-to-right.

a. *identifier*
An identifier is a primary expression, but only if it has been properly declared. Upon declaration, its type must be specified. However, if the type of the identifier is “array of type T”, then the value of the identifier expression is an internal pointer to the first object in the array. The user will not be able to manipulate the address directly, as pointers are not available to be manipulated.

a. *constant*
Decimal, character, boolean, or floating point constants are all primary expressions. It’s type is `int` for decimal integers, `char` for character, `float` for floating point, and `bool` for boolean.

a. *string*
A string is a primary expression, whose type is ‘array of `char`’.

a. `(` *expression* `)`

Parenthesized expressions are primary expressions who would be evaluated identically to the same expression without parentheses. This is useful to avoid ambiguity in writing expressions.

a. *primary-lvalue* `[` *expression* `]`
This expression defines a reference to a specific value within an array. It is a valid primary expression when the left expression resolves to a variable of type array, and when the right expression (within brackets) resolves to an `int` which is not larger than the size of the array.

a. *primary-lvalue* `.` *member-of-structure*
This expression is a valid primary expression when the lvalue is referring to a structure, and the member of the structure referred exists within that structure.

3.4.3. Unary operators

Unary operations group right-to-left.

a. `-` *expression*
The “mathematical negation” expression results in the negative of the given expression of the same type. It only operates on `int`, `char`, and `float`. The output of this operation is the same type as the inner expression.

a. `!` *expression*
The “logical negation” expression results in `true` if the expression resolves to 0, and `false` if the expression resolves to a non-zero value. It only operates on `bool`. The output of this operation is a `bool`.

a. `++` *lvalue-expression*
`--` *lvalue-expression*
The “precrement” expressions increment or decrement by 1 the object referred to by the lvalue, as long as it is of type `int` or `char`. The final expression returns this incremented value, of the same type as the lvalue object.

a. *lvalue-expression* `++`
lvalue-expression `--`
The “postcrement” expressions operates similarly to “precrement” but they return the value referred to by the object before the operation occurs.

3.4.5. Multiplicative operators

Both multiplicative operators and the following subsection (additive operators) group left-to-right, in order to emulate math in common usage.

a. *expression* `*` *expression*

The `*` operator expresses multiplication. Both expressions on either side must be of the same valid type, `int` or `float`. The result of this operation is the same type as its two operands.

a. *expression* `/` *expression*

   The `/` operator expresses division. Its type requirements are the same as multiplication. Note that on `int/int` division, this operation throws away the remainder to keep its output `int`.

a. *expression* `%` *expression*

   The `%` operator expresses the modulo operation. Its type requirements are the same as the other multiplicative operators.

3.4.6. Additive operators

a. *expression* `+` *expression*

   The `+` operator expresses addition. Its type requirements are the same as the multiplicative operators; both operands must be of the same type (both `int`, both `char`, or both `float`), and it returns a value of the same type as its operands.

a. *expression* `–` *expression*

   The `–` operator expresses subtraction. Its type requirements are the same as addition and the multiplicative operators.

3.4.7. Relational operators

Relational operators (as well as the two following section ‘Equality’ and ‘Comparison’) resolve its two operands into an output that is a `bool`. These three sections all group left-to-right.

a. *expression* `<` *expression*
b. *expression* `<=` *expression*
c. *expression* `>=` *expression*
d. *expression* `>` *expression*

   These operators each evaluate to `true` if the relation is true, and `false` otherwise. Both operands must be of the same type. Types `int`, `char`, `float`, and `bool` are accepted. The output of these operations is a `bool`.

3.4.8. Equality operators

a. *expression* `==` *expression*
b. *expression* `!=` *expression*

   These operators are equivalent in practice to the relational operators, but they always have lower precedence.

3.4.9. Comparison operators

SimpliCty does not support bitwise operations. As such, the comparison operators do not need to be a double character.

a. *expression* `&&` *expression*
   Returns `true` if both operands are nonzero, otherwise `false`. If the first expression evaluates to `false`, the second expression is not evaluated. All primitive types are accepted as operands, but a `bool` is outputted.
a. *expression* `||` *expression*
   Returns `true` if both or either operands are nonzero, otherwise `false`. If the first expression evaluates to `true`, the second expression is not evaluated. All primitive types are accepted as operands, but a `bool` is outputted.

3.4.10. Assignment operators

Assignment operations group right-to-left. They all require an lvalue as the left operand. The value returned is the value that is placed in the object referred to by the lvalue. The object referred to by the lvalue must be declared before it can be assigned.

a. *lvalue* `=` *expression*
   Both the object referred to by the lvalue, and the evaluated expression, must have the same type.
a. *lvalue* `+=` *expression*
b. *lvalue* `-=` *expression*
c. *lvalue* `*=` *expression*
d. *lvalue* `/=` *expression*
e. *lvalue* `%=` *expression*
   The behavior of these four operations resolve to (for example `+=`) `lvalue = lvalue + expression`. The mathematical operation detailed in the symbol evaluates that operation on the object referred to by the lvalue and the expression, and then that value is assigned to the lvalue. The types of of the lvalue and the expression must match, as detailed in section 4 and 5.

3.5. Declarations

Declarations are used within functions to declare instances of designated type. Declarations have the following form

declaration:

decl-specifier identifier `;`

decl-specifier identifier `;`

The declarator list contains all the identifiers waiting to be declared. For decl-specifier, only one specifier is allowed for each declaration.

decl-specifiers:

type-specifier

type-specifier arr-dimensions

Note that a variable cannot be declared and assigned a value to in the same statement.

3.5.1. Type specifiers

The only types that can be specified are the primitive types, the special character array `string`, and data structures.

type-specifier:

`int`

`char`

`float`

`bool`

`string`

struct-specifier

3.5.2. Declarators

A declarator is a variable name, and optionally a definition of an array. A declarator list must exist for all types, except for data structures, where they are optional.

declarator:

identifier

declarator `[` *constant* `]`

`(` *declarator* `)`

Together with the associated type specifiers, each declarator yields an instance of the indicated type. A declarator with the form of *declarator* `[` *constant* `]` indicates we are declaring an instance of *array*, with size *constant*. If the type-specifier was `string`, the size of the array does not need to be specified. An array may be constructed from one of the primitive types, from a structure, or from another array (to generate a multidimensional array).

3.5.3. Structure Declarations

A data structure specifies a new composite type, which is composed of one or more primitive types, array, or other previously specified data structures.

struct-specifier:

`struct` `{` *type-decl-list* `}`

`struct` *identifier* { *type-decl-list* }

The *type-decl-list* is a sequence of type declarations for the members of the structure. *type-declaration* is just normal declaration with *type-specifier* and *declarator*.

As stated above, the declarator list is optional for structure declaration. A structure declaration can be specified for one or multiple variables in one statement or, alternatively, just the type can be declared, with variables of that type declared later.

Note that the first instance of the struct specifier, `struct` { *type-decl-list* }, requires a declarator list, while the second does not.

SimpliCty does not allow self-referential structures. The declaration for structures is otherwise similar to the way one declares a variable. However, the *declarations* within the *type-decl-list* should always have names as well.

type-decl-list:

type-declaration

type-declaration type-decl-list

3.6. Statements

3.6.1. Expression statement

An expression statement has the form:

expression `;`

Expression statements are assignments or function calls.

3.6.2. Compound statement

Write multiple expression statements, that will be evaluated one after the other, like this:

{ *stat-decl-list* }

stat-decl-list will be more clearly defined in the section “Program definitions” but for now it will be defined as:

stat-decl-list:

statement statement-list

statement

3.6.3. Conditional statement

The two types of conditional statements are:

`if` (*expression*) *statement*

`if` (*expression*) *statement* `else` *statement*

If the expression surrounded by parentheses evaluates to `true` (it must be a `bool`), then the first statement is evaluated.
In the second instance of a conditional, the second statement is evaluated if the expression evaluated to `false`. Never are both statements evaluated.

3.6.4. While statement

The while statement is expressed as such:

`while` `(` *expression* `)` *statement*

The expression is evaluated - if it evaluates to a `true` then the statement is evaluated. The expression must be a `bool`. Control flow then jumps back to the expression and re-evaluated. This process is repeated until the expression evaluates to `false`.

3.6.5. For statement

The for statement is expressed thusly:

`for` `(` *expression1* `;` *expression2* `;` *expression3* `)` *statement*

But this statement is equivalent to:

```
expression1 ;
while ( expression2 )
{
        statement
        expression3 ;
}
```

There must be an expression in each of the three positions.

3.6.6. Break statement

The statement

`break;`

Makes the latest `while` or `for` statement terminate prematurely. Control flow moves to the statement following the terminated statement.

3.6.5. Continue statement

The statement

`continue;`

Can only be used on `while` or `for` statements, in order to prematurely jump back to the evaluation of the potentially `false` expression.

3.6.8. Return statement

The statements

`return` `;`

`return` *expression* `;`

Move control flow back to the caller of the function within which these statements have been expressed. In the first type of statement no value is returned. In the second, the expression must evaluate to the type specified in the function definition. Returning arrays or structures are defined.

3.6.9. Putchar statement

The statement

`putchar` `(`*expression* `)` `;`

Prints the ASCII representation of the expression to the command line. The expression must resolve to a `char` to `int`. Comes from C.

3.6.10. Getchar statement

The statement

`getchar` `(` `)` `;`

Pulls the next character types by the user in the command line until the a new line character or null terminator is received. This statement outputs a `char`. Comes from C.

3.7. Program definitions

A simpliCty program consists of series of external definitions. An external definition is either a function definition or a global data structure definition.

program:

external-definition program

external-definition

external-definition:

function-definition

struct-definition

3.7.1. Function definition

A function definition is defined as

function-definition:

type-specifier function-declarator function-body

Where the type specifier details what the function returns, the function body details the statements that will be evaluated when the function is called, and the function declaration

function-declarator:

declarator `(` *parameter-list*$_{opt}$ `)`

`main` `(` `)`

Details the name of the function (*declarator*), and the series of parameters that are passed to the function as inputs. The special case of the function declaration, with the keyword `main`, is used to define the entry point for control flow, if any. It is equivalent to the “main” function keyword in the regular C Language. `main` does not have any argument inputs.

A parameter list is defined as

parameter-list:

identifier `,` *parameter-list*

identifier

Since the parameter list is optional, a function does not necessarily need any inputs.

A function body has the form

function-body:

type-decl-list function-statement

function-body:

`{` *stat-decl-list* `}`

stat-decl-list:

statement-declaration stat-decl-list

statement-declaration

statement-declaration:

statement

declaration

stat-decl-list was earlier defined as only a list of statements. Its full definition allows it to be a list, of arbitrary length, of either statements or declarations. At least one of the statements in a function must be a “return” statement, as the function needs to at some point return control flow to its parent function (or terminate the program).

Once the specially defined function `main` “returns”, a program terminates.

3.7.2. Structure definition

A struct definition is defined as

struct-definition:

struct-specifier `;`

A structure is defined here in the same manner as previously defined in the declarations section, although it does not allow for the declaration of variables of this new data structure type. A structure may be defined in this scope when it needs to be defined as an input or a return value for one or multiple functions. A structure defined within a function will be undefined (as out of scope) once the function returns control flow.

3.8. Scope and Preprocessor Rules

Variables declared within a function are undefined when the function returns. Structures defined within a function become undefined when the function returns. If the user wants a structure to be available to more than one function, it must be defined at the level of other functions (as specified in the previous section, “Program definitions”).

3.8.1. File inclusion

The source text of a simpliCty program does not need to be all in one file. There are special preprocessor rules that allow the importing of contents of an external file at the start of the file, before compilation.

`#include "`*filename*`"`

Where the *filename* is a relative path to the external file.

3.9. Context Free Grammar Summary

3.9.1. **Expressions**

expression:

primary
`(` *expression* `)`
`-` *expression*
`!` *expression*
`++` *lvalue*
`--` *lvalue*
lvalue `++`
lvalue `--`

expression `*` *expression*
expression `/` *expression*
expression `%` *expression*

expression `+` *expression*

expression `-` *expression*

expression `<` *expression*

expression `<=` *expression*

expression `>=` *expression*

expression `>` *expression*

expression `==` *expression*

expression `!=` *expression*

expression `&&` *expression*

expression `||` *expression*

lvalue `=` *expression*

lvalue `+=` *expression*

lvalue `*=` *expression*

lvalue `/=` *expression*

lvalue `%=` *expression*

expression `#` *expression*

primary:

lvalue

int-constant

float-constant

char-constant

string-constant

lvalue:

identifier

identifier `(` *expression-list*$_{opt}$ `)`

lvalue `[` *expression* `]`

lvalue `.` *lvalue*

expression-list:

expression `,` *expression-list*

expression

3.9.2. **Declarations**

declaration:

type-specifier varname-list

struct-specifier varname-list

type-specifier:

`int`

`char`

`float`

`bool`

`string`

`struct` *identifier*

struct-specifier:

`struct` `{` *declaration-list* `}`

`struct` *identifier* `{` *declaration-list* `}`

varname-list:

varname `,` *varname-list*

varname

varname:

identifier

`(` *varname* `)`

varname `[` *int-constant* `]`

declaration-list:

declaration `;` *declaration-list*

declaration `;`

3.9.3. **Statements**

statement:

expression `;`

declaration

`{` *statement-list* `}`

`if` `(` *expression* `)` *statement*

`if` `(` *expression* `)` *statement* `else` *statement*

`while` `(` *expression* `)` *statement*

`for` `(` *expression* `;` *expression* `;` *expression* `)` *statement*

`break` `;`

`continue` `;`

`return` `;`

`return` *expression* `;`

`print` *expression* `;`

`scan` `;`

statement-list:

statement statement-list

statement

3.9.4. **Function Definitions**

program:

module-definition program

module-definition

module-definition:

function-definition

struct-definition

function-definition:

type-specifier identifier (*parameter-list$_{opt}$*) *statement*

parameter-list:

parameter , *parameter-list*

parameter

parameter:

type-specifier identifier

type-specifier identifier [ ]

struct-definition:

struct-specifier ;

3.9.5. **Preprocessor**

`extern` "*filename*"

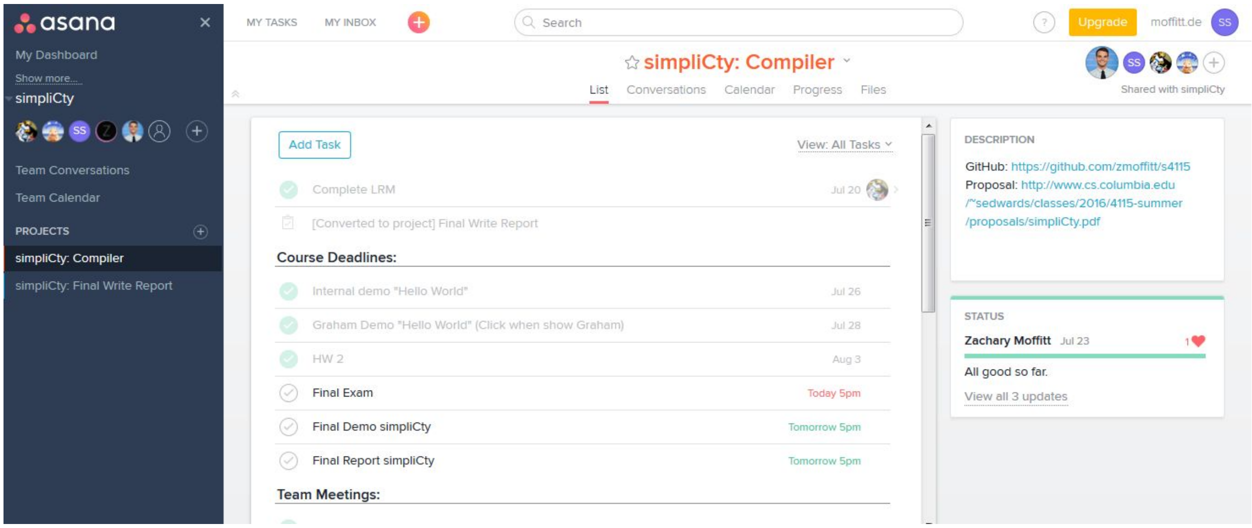
4. Project Plan (by Manager)

4.1. Process: Planning, Specification, Development and Testing

We spent an intensive four weeks implementing the compiler.

4.1.1 Project Planning

We spent many hours on project planning. First, the team and the project manager went through the LRM and divided-up the compiler features and subfeatures in Asana.   We chose Asana (https://asana.com/) as the project management tool becuase of its distributed framework, deadline and notification system and excell spreadsheet export features. The screenshot below shows our Asana projects (both compiler and final write-up).



Complete LRM Jul 20

[Converted to project] Final Write Report

Course Deadlines:

Internal demo "Hello World" Jul 26

Graham Demo "Hello World" (Click when show Graham) Jul 28

HW 2 Aug 3

Final Exam Today 5pm

Final Demo simpliCty Tomorrow 5pm

Final Report simpliCty Tomorrow 5pm

Team Meetings:

DESCRIPTION

GitHub: https://github.com/zmoffitt/s4115

Proposal: http://www.cs.columbia.edu/~sedwards/classes/2016/4115-summer/proposals/simpliCty.pdf

STATUS

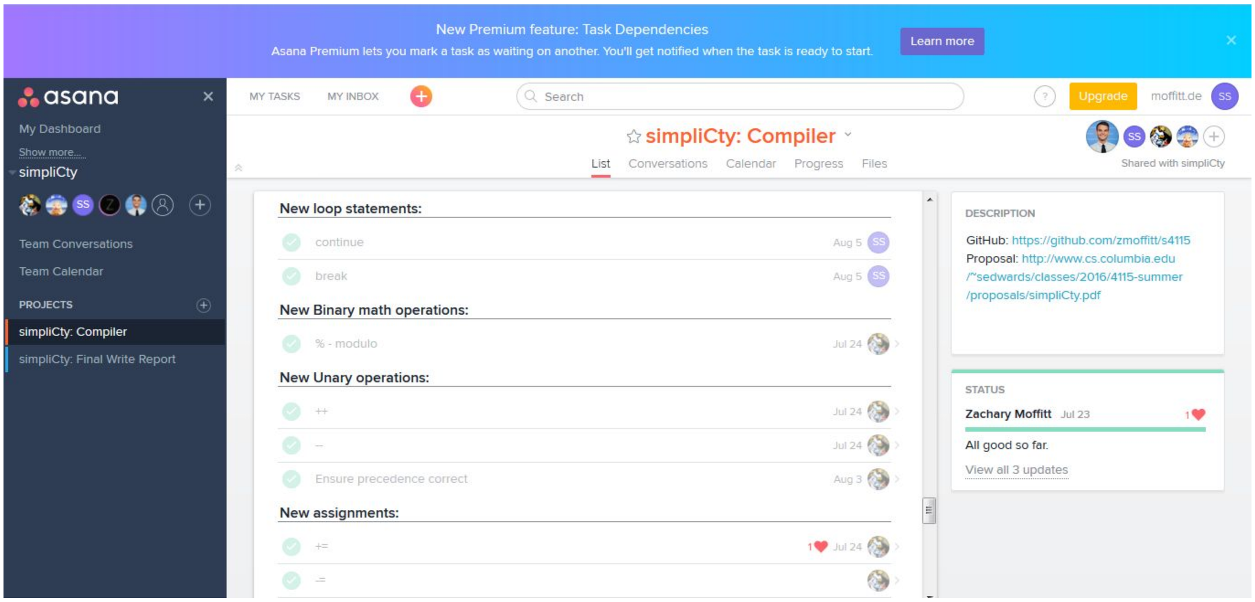
Zachary Moffitt Jul 23 1

All good so far.

View all 3 updates

4.1.2 Project Specification

We used Asana to divide the project up into specification



New Premium feature: Task Dependencies

Asana Premium lets you mark a task as waiting on another. You'll get notified when the task is ready to start.

New loop statements:

continue Aug 5

break Aug 5

New Binary math operations:

% - modulo Jul 24

New Unary operations:

++ Jul 24

-- Jul 24

Ensure precedence correct Aug 3

New assignments:

+= 1 Jul 24

-=

DESCRIPTION

GitHub: https://github.com/zmoffitt/s4115

Proposal: http://www.cs.columbia.edu/~sedwards/classes/2016/4115-summer/proposals/simpliCty.pdf

STATUS

Zachary Moffitt Jul 23 1

All good so far.

View all 3 updates

4.1.3 Project Testing

We ran regression tests with over 150 test cases. Our regression testing keeps track of the number of tests run and the number of tests that passed as well as failed.

4.2. One-page Programming Style Guide used by the Team

We followed the programming style used by the standard OCaml libraries. We used the following series of styles throughout the writing of our compiler.

1. First, we used meaningful names.
2. Second, we used standard indententing for both long expressions as well as match expressions.
3. Third, we placed comments above the code in critical regions to enhance understanding.
4. Fourth, we completed all pattern matching expressions. This would be identified by the OCaml compiler if we left out any expressions.
5. Fifth, we tried hard to compress our code as to not to break expressions over multiple lines.
6. Sixth, to our knowledge, we did not rewrite any standard OCaml library functions and were careful to (re)use applicable standard OCaml library functions.
7. Seventh, to our knowledge, we did not compute any values twice and worked hard to ensure that the our implementation was devoid of repetition.

We followed the naming convention style used by the standard OCaml libraries when writing tokens to indicate certain meanings. The table below summarizes our token style used throughout the compiler.

Token	**OCaml Convention**	**Example**
Variables	*First letter lower case. *CamelCase or underscore multiwords.	a_var
Constructors	*First letter upper case. *Underscores between multiwords	Abs
Types	*Entirely lower case. *Underscores between multiwords.	member_list
Signatures	*Entirely upper case. *Underscores between multiwords.	ABC_DEF
Structures	*First letter upper case. *CamelCase for multiwords.	Abstract
Functors	*First letter upper case *CamelCase for multiwords. *Fn completes name.	Function

4.3. Project timeline

We started writing the compiler July 19 and completed the compiler on August 11. The table below indicates the milestones of our project.

In the table below, the days of the week are indicated as Monday (M), Tuesday (T), Wednesday (W), Thursday (R), Friday (F), Saturday (S) and Sunday (Z).

Date	Milestone
July	
11 (M)	Proposal Due due to TA
17 (Z)	Environment set-up (e.g. Amazon, Docker, Asana, and GIT)
19 (T)	Begin work on scanner, parser and abstract syntax tree
20 (W)	Language Reference Manual (LRM) due to TA
29 (W)	In addition to the basic C statement functionality. Adam and Suzanna had 1-dimensional arrays working; Zac and Rui implemented getchar.
28 (R)	“Hello World” demo to TA
31 (S)	Suzanna implemented break and continue statements
August	
1 (M)	“Hello World” program compiled
3 (W)	Rui added putchar, Suzanna added char, Zach improved char, Adam working on Arrays and Suzanna implemented the regression test suite and scripts.
5 (R)	Rui implemented extern function references
6 (F)	Suzanna added float variables and respective binary operations
8 (Z)	Suzanna implemented casting
11 (R)	Adam added multi-dimensional arrays and Rui, Suzanna and Zac added struct declarations
11 (R)	Demo to Dr. Stephen A. Edwards

4.4. Team Member Roles and Responsibilities

Because of the compressed class, everyone had to wear multiple-hats. Our Git had over ____ committs in total over 4 weeks. We estimated that we spent ___ hours in total on the project. That said we each owned certain tasks which may have been integrated or extended by another team member.

4.4.1. Adam Hadar, anh2130

I was the lead language guru, and did much of the backend code generation. I managed the group in defining what we wanted the language to do/look like. I took the lead in writing up the Language Reference Manual. My other major roles were in implementing the modulo operator, the crement operators, and the mathematical assignment operators (+=, *=, etc). Finally, I did all the back-end code generation for arrays. This included: allocating memory for an array and storing its pointer in the symbol table, accessing elements within the array, creating the literal array (marking values with {| and |} generates an array literal). I also handled passing arrays to functions, where you could specify a static size and pass by value, or not specify the size and pass by pointer. I attempted to implement multidimensional arrays, but was stressed for time and only implemented up to two dimensional arrays. Time constraints also kept me from passing back arrays from functions. I also wrote the standard library functions.

4.4.2. Zachary Moffitt, znm2104

I was the Systems Architect for our project and assisted all of my teammates when possible. I took the lead at preparing a uniform test environment that had the potential for stability under heavy utilization. With this we also used Dockers to keep our environment together. My original scope for the project changed overtime and we moved many of our processes to new work flows (Asana, Git, etc.) As the System Architect for the project team it was also my responsibility to pick up on code cleanup; if possible, and ensure that we were keeping our function calls clean and concise. We would hold weekly meetings and I would conduct a merge code review where we would evaluate the changes that were being made to our system and ensure that they didn’t conflict with other planned changes. During the tail end of our project we began to become a bit adventurous with our features and misjudged our time. We had many pieces of code which are available (as partially prepared); however, the results are not quite as clean as we’d like.

4.4.3. Suzanna Schmeelk, ss4648

Suzanna was the project manager and test implementer. Suzanna broke apart the compiler and the write-up in Asana and laid out the first 40 pages of the final write-up outline. The asana lists over 70 tasks to complete. The other team members added tasking in asana as they found additional cases. Suzanna tasked each team member with deadlines that were rarely attended too. Suzanna implemented the the basic flow of the compiler with adding the break and continue statements. Suzanna added char variable times and reference throughout the program (which Zac improved). Suzanna worked with Adam to get 1-dimensional array functionality working. Adam continued work on arrays through the remainder of the project. Suzanna added float variables throughout the compiler and the respective llvm operations. Suzanna implemented the casting of primitive values for math operations (e.g. int to float). Suzanna jumped in to help out with struct declarations which were successful. Suzanna created the regression tests and a repository of over 150 test cases. The regression test script prints out the number of tests run, the number of tests that pass and the number of tests that fail.

4.4.4. Rui Gu, rg2970

Rui takes part in architecture design, function implementation and testing. Rui implements putchar() and getchar() as a compiler built-in. This two functions serve as the basic primitive to implement our IO function. He also implemented “extern” keywords to allow external calls and let the linker resolve external functions. Thanks to the strong help from the Suzanna & Zach, Rui is able to finish struct declaration, which allows user to declare their own synthetic types. All the above mentioned implemented functionalities are complete and works 100% in different situations.

4.5. Software Development Environment

4.5.1. Software Development Tools

4.5.1.1.Version Control System: GitHub

Our team used Git (specifically GIthub) to oversee changes to code done by the various team members. We each used our own separate branches from the master and would merge code as scheduled intervals.

4.5.1.2. Docker

Our team used Docker to keep a consistent building environment. Each of us maintains one to several docker containers other people can attach. Docker helps us collaborate efficiently and also ensured that we had access to the same environment; on and off campus.

4.5.2. Software Development Languages

4.5.2.1. OCaml

OCaml, created in 1996, was originally known as Objective Caml. OCaml is a member of the ML language family. It extends Caml language with object-oriented constructs. OCaml is a free open-source project. It is primarily managed and maintained by INRIA in France.

4.5.2.2. LLVM: Low Level Virtual Machine

The LLVM compiler infrastructure project is formerly known as the *Low Level Virtual Machine*. The project started in 2000 at the University of Illinois at Urbana–Champaign, under the direction of Vikram Adve and Chris Lattner. According to the LLVM website, it is a collection of modular and reusable compiler and toolchain technologies used to develop compiler front ends and back ends .

LLVM is written in C++ and is designed for compile-time, link-time, run-time, and "idle-time" optimization of programs written in arbitrary programming languages. OCaml has a library where it can write LLVM intermediate representation (IR) code.

4.5.2.3. Bash Scripts for Regression Testing

The Bash command language, first released in 1989, was written by Brian Fox for the GNU Project as replacement for the Bourne shell. It is widely released with Linux distributions.

4.6. Project Log (Asana)

Please see Appendix B

5. Language Evolution (by Guru)

5.1. Phase 1: Proposal Commitments

We initially committed to a language with strict typing (and no casting), and a spartan/simple type of C (that is where we made up the name SimpliCty).

5.2. Phase 2: Language Reference Enhancements

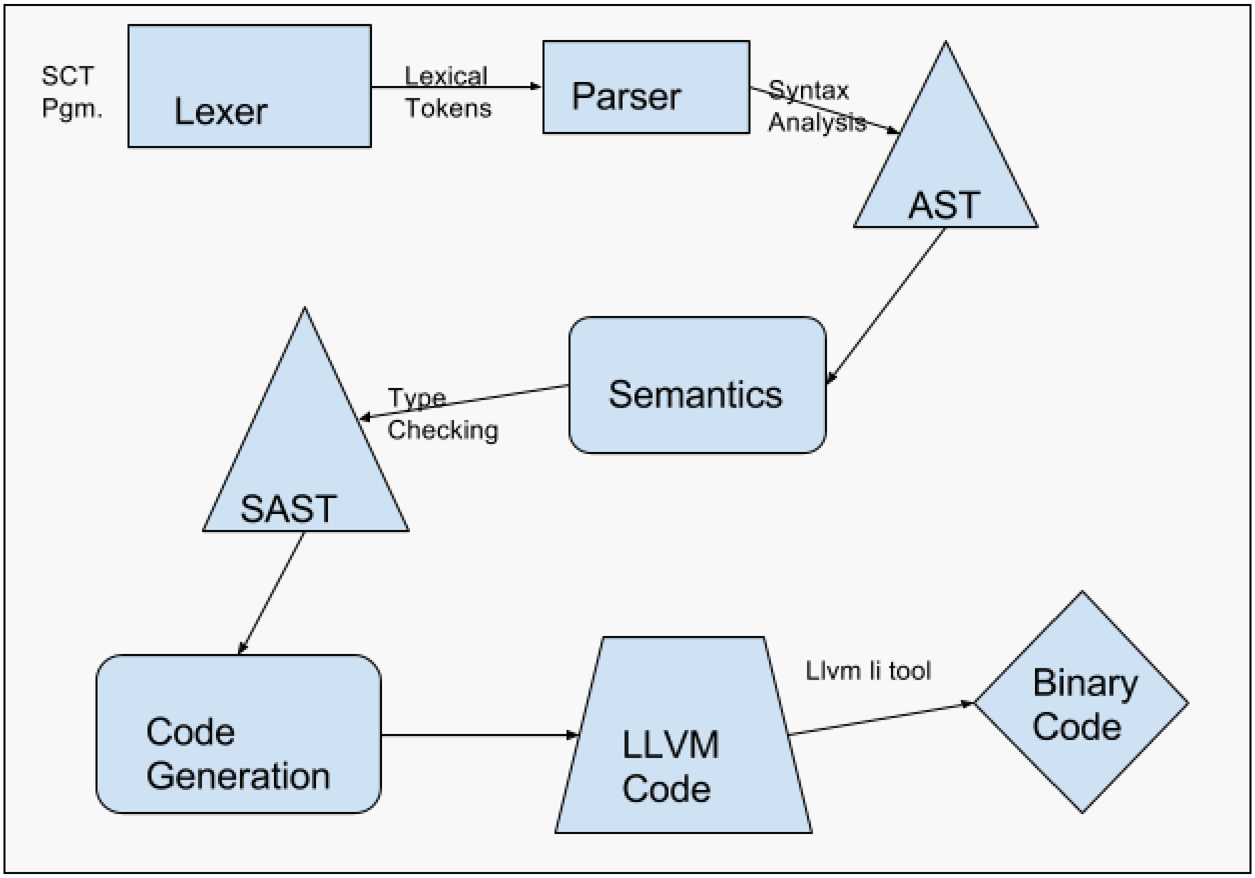
After discussing the proposal with both the professor and the TA, we realized that such a language would be too simplistic and easy for us to do. We then saw the baseline of Micro C and that confirmed it to us. We therefore transitioned to try to implement as much as possible that C has. Pointers were too complicated, but we settled on implementing arrays and structures, and we had agreed with our TA that it would be appealing if we could implement syscalls (machine specific, and we were planning on doing it in x86).

5.3. Phase 3: Final Enhancements

We were able to implement declaration and assignment in the same line, and global variables (which we did not specify in our original documentation). Single dimensional arrays were correctly implemented, but the transition to multi-dimensional arrays broke much of that code. We were unable to implement syscalls, but we did build much of our standard library from the two basic C functions getchar() and putchar().

6. Translator Architectural Design (by Sys. Architect)

6.1. Block Diagram of the Major Components of our Compiler



6.2. Interfaces between the components

There are multiple interfaces between components. First, the source program is passed into the compiler. A scanner scans the program and attempts to parse it. Parsing creates an abstract syntax tree. The abstract syntax tree is passed to semantic checking ensuring a syntactically correct AST. The SAST is passed to the LLVM code generator. The code generator performs a poster-order AST transversal to create the respective LLVM. The LLVM JIT compiler transforms the llvm into machine code which can be executed.

6.3. Who implemented each component

We all worked together to get the basic control-flow, local variables, global variables defined and semantic checking working.

Rui	Adam	Zac	Suzanna
*Docker *Environment Setting *Externs *Putchar *Getchar *Struct decls *Tests *Regression Testing script robustness	*Array declaration *Array index assignment *Array bulk assignment *Array literal *Multidimensional arrays *modulo *crement operators *mathematical assignment *Tests	*Docker *Environment Setting *Managed GIT *Finished char codegen *Array -> Char combine *Strings *Tests *Chars *Scanner Syntax *AST Matching *2nd Proj Mgmt	*Asana tasking *Proj. mgnt. *Array declaration *Char scanning, AST, *Float scanning, parsing *Float operations *Casting *Break Statements *Cont. Statements *Struct decls in llvm *Regression Testing script robustness *Tests

7. Test Plan and Scripts (by Tester)

```
#!/bin/sh
# Project:  COMS S4115, SimpliCty Compiler
# Filename: src/testall.sh
# Authors:  - Rui Gu,            rg2970
#            - Adam Hadar,       anh2130
#            - Zachary Moffitt,  znm2104
#            - Suzanna Schmeelk, ss4648
# Purpose:  * Regression testing script for SimpliCty
#            * Steps through list of files:
#            * Expected to work: compile, run, check output
#            * Expected to fail: compile, check error
# Modified: 2016-07-24

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

# Path to the simpliCty compiler.  Usually "./simplicty.native"
# Try "_build/simplicty.native" if ocamlbuild was unable to create a symbolic link.
SIMPLICTY="../simplicty"

# Set time limit for all operations
ulimit -t 30

globallog=test.log
```

```
rm -f $globallog
error=0
testcountall=0
testcounttest=0
testcounttestpass=0
testcounttestfail=0
testcountfail=0
testcountfailpass=0
testcountfailfail=0
globalerror=0
dirout=./test-output/
keep=0

Usage() {
       echo "Usage: testall.sh [options] [.mc files]"
       echo "-h      Print this help"
       echo "-k      Keep intermediate files"
       echo "-l      Test loop statements"
       echo "-d      Test declaration statements"
       echo "-p      Test print statements"
       echo "-s      Test scan statements"
       echo "-r      Test array statements"
       echo "-t      Test struct statements"
       echo "-f      Test function statements"
       echo "-i      Test if statements"
       echo "-o      Test operator statements"
       echo "-g      Test assignment statements"
       echo "-a      Test all statements"
       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 ${dirout}$1 $2 ">" $3 1>&2
       diff -b "${dirout}$1" "$2" > "${dirout}$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/.sct//'`
       reffile=`echo $1 | sed 's/.sct$//'`
       basedir="`echo $1 | sed 's/\/[^\/]*$//'`/."

       echo -n "$basename..."

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

       generatedfiles=""
       testcountall=$((testcountall+1))
       testcounttest=$((testcounttest+1))
       generatedfiles="$generatedfiles ${basename}.ll ${basename}.out" &&
       Run "$SIMPLICTY" $1 ">" "${dirout}${basename}.ll" &&
       Run "$LLI" "${dirout}${basename}.ll" ">" "${dirout}${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
       rm -f ${dirout}/*
    fi
    echo "OK"
    echo "###### SUCCESS" 1>&2
       testcounttestpass=$((testcounttestpass+1))
       else
    echo "###### FAILED" 1>&2
       testcounttestfail=$((testcounttestfail+1))
    globalerror=$error
       fi
}

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

       echo -n "$basename..."

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

```
        testcountall=$((testcountall+1))
        testcountfail=$((testcountfail+1))
        generatedfiles=""

        generatedfiles="$generatedfiles ${basename}.err ${basename}.diff" &&
        RunFail "$SIMPLICTY" $1 "2>" "${dirout}${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 $generatedfiles
    fi
    echo "OK"
    echo "###### SUCCESS" 1>&2
        testcountfailpass=$((testcountfailpass+1))
        else
    echo "###### FAILED" 1>&2
        testcountfailfail=$((testcountfailfail+1))
    globalerror=$error
        fi
}

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
}

Test() {
        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

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

}

TestLoop(){
        echo "Testing Loops"
        files="test/loops/test-*.sct fail/loops/fail-*.sct"
        Test
}

TestDec(){
        echo "Testing Declarations"
        files="test/declaration/test-*.sct fail/declaration/fail-*.sct"
        Test
}

TestPrint(){
        echo "Testing Print"
        files="test/print/test-*.sct fail/print/fail-*.sct"
        Test
}

TestScan(){
        echo "Testing Scan"
        files="test/scan/test-*.sct fail/scan/fail-*.sct"
        Test
}

TestArray(){
        echo "Testing Array"
        files="test/array/test-*.sct fail/array/fail-*.sct"
        Test
}

TestStruct(){
        echo "Testing Struct"
        files="test/struct/test-*.sct fail/struct/fail-*.sct"
        Test
}

TestFun(){
        echo "Testing Functions"
        files="test/functions/test-*.sct fail/functions/fail-*.sct"
        Test
}

TestIf(){
        echo "Testing If"
        files="test/if/test-*.sct fail/if/fail-*.sct"
        Test
}

TestOp(){
        echo "Testing Operators"
        files="test/operators/test-*.sct fail/operators/fail-*.sct"
        Test
        files="test/operators/binary/test-*.sct fail/operators/binary/fail-*.sct"
        Test
        files="test/operators/unary/test-*.sct fail/operators/unary/fail-*.sct"
        Test
}

TestAssign(){
        echo "Testing Assignments"
        files="test/assignment/test-*.sct fail/assignment/fail-*.sct"
        Test
        files="test/assignment/lvalue/test-*.sct fail/assignment/lvalue/fail-*.sct"
        Test
        files="test/assignment/literal/test-*.sct fail/assignment/literal/fail-*.sct"
        Test
        files="test/assignment/char/test-*.sct fail/assignment/char/fail-*.sct"
        Test
}
```

```
TestAll(){
  TestLoop
  TestDec
  TestPrint
  TestScan
  TestArray
  TestStruct
  TestFun
  TestIf
  TestOp
  TestAssign
}

options='khldpsrtfioga*'
while getopts $options optchar; do
       case $optchar in
    k) # Keep intermediate files
       keep=1
       ;;
    h) # Help
       Usage; exit 0;
       ;;
       l) # Test loop statements
       TestLoop; exit 0;
             ;;
       d) # Test declaration statements
       TestDec; exit 0;
             ;;
       p) # Test print statements
       TestPrint; exit 0;
             ;;
       s) # Test scan statements
       TestScan; exit 0;
             ;;
       r) # Test array statements
             TestArray; exit 0;
             ;;
       t) # Test struct statements
             TestStruct; exit 0;
             ;;
       f) # Test fun statements
             TestFun; exit 0;
             ;;
       i) # Test if statements
             TestIf; exit 0;
             ;;
       o) # Test operator statements
             TestOp; exit 0;
             ;;
       g) # Test assign statements
             TestAssign; exit 0;
             ;;
       a) # Test all statements
             TestAll; exit 0;
             ;;
       *) # Default
             TestAll; exit 0;
             ;;
       esac
```

```
done
TestAll
echo "Number of all tests executed " $testcountall"!"
echo "Number of test tests executed " $testcounttest " of which " $testcounttestpass
" passed and " $testcounttestfail " failed"
echo "Number of fail tests executed " $testcountfail " of which " $testcountfailpass
" passed and " $testcountfailfail " failed"
shift `expr $OPTIND - 1`

exit $globalerror
```

7.1. Source to Target Examples

7.1.1. Example 1: Tic Tac Toe

```
void prints(char[] val)
{
        int ptr;
        while(val[ptr] != 0)
                putchar(val[ptr++]);
        putchar(10);
        return;
}

void printc(char val)
{
        putchar(val);
        putchar(10);
        return;
}

char[3][3] board = {| {|' ',' ',' '|}, {|' ',' ',' '|}, {|' ',' ',' '|} |};
char[5] turn = {|'t','u','r','n',0|};
char[10] row = {|'i','n','p','u','t',' ','r','o','w',0|};
char[10] col = {|'i','n','p','u','t',' ','c','o','l',0|};
char[10] end = {|'g','a','m','e',' ','o','v','e','r',0|};
char[14] winner = {|'t','h','e',' ','w','i','n','n','e','r',' ','i','s',0|};


void printBoard()
{
        int i;
        int j;
        for(i = 0; i < 3; i++)
        {
                for(j = 0; j < 3; j++)
                {
                                putchar(board[i][j]);
                                if(j != 2)
                                        putchar(124); /* the | character */
                }
                putchar(10);
                if(i != 2)
                {
                                putchar(95);            /* the _ character */
                                putchar(95);            /* for pretty printing */
                                putchar(95);
                                putchar(95);
                                putchar(95);
                                putchar(95);
                                putchar(10);
                }
        }
        return;
}
bool is_gameOver()
{
        int i;
        for(i = 0; i < 3; i++)
        {
                if((board[i][0] == board[i][1] && board[i][0] == board[i][2])
                || (board[0][i] == board[1][i] && board[0][i] == board[2][i]))
                                return true;
        }
        if((board[0][0] == board[1][1] && board[0][0] == board[2][2])
        || (board[0][2] == board[1][1] && board[0][2] == board[2][0]))
                return true;

        return false;
}

int main()
{
        int i;
        int j;
        int x;
        int y;
        bool cont = true;
        bool is_x = true;

        for(i = 0; i < 9 && cont; i++)
        {
                prints(turn);
                if(is_x) printc('X');
                else     printc('O');

                prints(row);
                x = getchar() - '0';
                prints(col);
                y = getchar() - '0';

                if(is_x) board[x][y] = 'X';
                else     board[x][y] = 'O';

                printBoard();

                if(is_gameOver())
                {
                                prints(end);
                                prints(winner);
                                if(is_x)
                                        printc('X');
                                else
                                        printc('O');
                                cont = false;
                }
                is_x = !is_x;
        }
        return 0;
}
```

7.1.2. Example 2: Extern Functions

```
extern int add (int a, int b);
extern int minus (int a, int b);
extern int times (int a, int b);
int main() {
    int a = 65;
    int b = 1;
    int c = 0;
    c = add(a, b);
    putchar(c);
    return 0;
}
```

```
Output:
> ./regression-test.sh

    ; ModuleID = 'SimpliCty'
declare i32 @putchar(i32)
declare i32 @getchar(...)
declare i32 @add(i32, i32)
declare i32 @minus(i32, i32)
declare i32 @times(i32, i32)
define i32 @main() {
entry:
  %a = alloca i32
  store i32 65, i32* %a
  %b = alloca i32
  store i32 1, i32* %b
  %c = alloca i32
  store i32 0, i32* %c
  %lv = load i32, i32* %b
  %lv1 = load i32, i32* %a
  %add_result = call i32 @add(i32 %lv1, i32 %lv)
  store i32 %add_result, i32* %c
  %lv2 = load i32, i32* %c
  %putchar = call i32 @putchar(i32 %lv2)
  ret i32 0
}
```

7.1.3. Example 3: test-float-mixed.sct

```
float f;
int i;
int main( ) {
   int x;
   float y;
   f = 123.456;
   i = 999;
   x = i;
   y = 123.45 + 999 + 12.4567;
   f = 198.2 + 6 + 97.1;
   return 0;
}
```

```
Output:
> ./regression-test.sh
root@f63df2ece601:# ./simplicty test-float-mixed.sct
; ModuleID = 'SimpliCty'

@i = global i32 0
@f = global float 0.000000e+00

declare i32 @putchar(i32)

define i32 @main() {
entry:
  %x = alloca i32
  store i32 0, i32* %x
  %y = alloca float
  store float 0.000000e+00, float* %y
  store float 0x405EDD2F20000000, float* @f
  store i32 999, i32* @i
  %lv = load i32, i32* @i
  store i32 %lv, i32* %x
  store float 0x4091BBA060000000, float* %y
  store float 0x4072D4CCC0000000, float* @f
  ret i32 0
}
```

7.1.4. Example 4: Test-struct4.sct

Our compiler parses and declares global structs based on the embedded fields.

```
root@f63df2ece601:/soft-link/archive/8_11_2016_350pm# cat test-struct4.sct

struct Books {
  int id;
  int id1;
  int id2;
  int id3;
  float f1;
  float f2;
};
```

```
int main( ) {
   return 0;
}
```

```
root@f63df2ece601:# ./simplicty test-struct4.sct
; ModuleID = 'SimpliCty'

%Books = type { i32, i32, i32, i32, float, float }

@Books = external global %Books

declare i32 @putchar(i32)

define i32 @main() {
entry:
  ret i32 0
}
```

7.1.5. Example 5: Break Statements

```
root@f63df2ece601:# cat tests/test/loops/test-for-for-break0.sct

int main()
{
  int i;
  int x;
  int j;
  for (i = 0 ; i < 5 ; i++ ) {
       i++;
       if(i == 9)
       {
          for(j = 0; j < 9; j++)
       {
          if(i ==9)
          {
             break;
           }
        }
       }
       else{
       x = 9999;
       }
  }
  return 0;
}
```

```
root@f63df2ece601:# ./simplicty tests/test/loops/test-for-for-break0.sct
; ModuleID = 'SimpliCty'
```

```
%aname = type { i32, float }
%aname.0 = type { i32, float }
%aname.1 = type { i32, float }

@aname = external global %aname

declare i32 @putchar(i32)

define i32 @main() {
entry:
  %aname = alloca %aname
  %i = alloca i32
  store i32 0, i32* %i
  %aname1 = alloca %aname.0
  %x = alloca i32
  store i32 0, i32* %x
  %aname2 = alloca %aname.1
  %j = alloca i32
  store i32 0, i32* %j
  store i32 0, i32* %i
  br label %while.cmp.block

while.cmp.block:                               ; preds = %if.else.merge,
%entry
  %lv19 = load i32, i32* %i
  %tmp20 = icmp slt i32 %lv19, 5
  br i1 %tmp20, label %while.body, label %while.merge.block

while.body:                                    ; preds = %while.cmp.block
  %lv = load i32, i32* %i
  %tmp = add i32 %lv, 1
  store i32 %tmp, i32* %i
  %lv3 = load i32, i32* %i
  %tmp4 = icmp eq i32 %lv3, 9
  br i1 %tmp4, label %if.then, label %if.else16

while.merge.block:                             ; preds = %while.cmp.block
  ret i32 0

declare i32 @putchar(i32)

define i32 @main() {
entry:
  ret i32 0
}
if.else.merge:                                 ; preds = %if.else16,
%while.merge.block7
  %lv17 = load i32, i32* %i
  %tmp18 = add i32 %lv17, 1
  store i32 %tmp18, i32* %i
  br label %while.cmp.block

if.then:                                       ; preds = %while.body
  store i32 0, i32* %j
  br label %while.cmp.block5

while.cmp.block5:                              ; preds =
%if.else.merge10, %if.then
  %lv14 = load i32, i32* %j
  %tmp15 = icmp slt i32 %lv14, 9
  br i1 %tmp15, label %while.body6, label %while.merge.block7

while.body6:                                      ; preds =
%while.cmp.block5
  %lv8 = load i32, i32* %i
  %tmp9 = icmp eq i32 %lv8, 9
  br i1 %tmp9, label %if.then11, label %if.else

while.merge.block7:                               ; preds =
%while.cmp.block5, %if.then11
  br label %if.else.merge

if.else.merge10:                                  ; preds = %if.else,
%after.break
  %lv12 = load i32, i32* %j
  %tmp13 = add i32 %lv12, 1
  store i32 %tmp13, i32* %j
  br label %while.cmp.block5

if.then11:                                        ; preds = %while.body6
  br label %while.merge.block7

after.break:                                      ; No predecessors!
  br label %if.else.merge10

if.else:                                          ; preds = %while.body6
  br label %if.else.merge10

if.else16:                                        ; preds = %while.body
  store i32 9999, i32* %x
  br label %if.else.merge
}
```

7.1.5. Example 5: Continue Statements

We implemented continue statements.

```
root@f63df2ece601# cat tests/test/loops/test-while-while-continue0.sct

int main()
{
  int i;
  int j;
  int x;
  i = 5;
  j = 6;
  x = 9;
  while (i > 0) {
       i = i - 1;
       if ( i == 9 ) {
       while( j > 0)
```

```
        {
        if (x == 9)
        {
                continue;
        }
        }
        }
  }
  return 0;
}
```

```
root@f63df2ece601:/soft-link/archive/8_11_2016_350pm# ./simplicty
../../tests/test/loops/test-while-while-continue0.sct
; ModuleID = 'SimpliCty'

%aname = type { i32, float }
%aname.0 = type { i32, float }
%aname.1 = type { i32, float }

@aname = external global %aname

declare i32 @putchar(i32)

define i32 @main() {
entry:
  %aname = alloca %aname
  %i = alloca i32
  store i32 0, i32* %i
  %aname1 = alloca %aname.0
  %j = alloca i32
  store i32 0, i32* %j
  %aname2 = alloca %aname.1
  %x = alloca i32
  store i32 0, i32* %x
  store i32 5, i32* %i
  store i32 6, i32* %j
  store i32 9, i32* %x
  br label %while.cmp.block

while.cmp.block:                                  ; preds = %if.else.merge,
%entry
  %lv15 = load i32, i32* %i
  %tmp16 = icmp sgt i32 %lv15, 0
  br i1 %tmp16, label %while.body, label %while.merge.block

while.body:                                       ; preds = %while.cmp.block
  %lv = load i32, i32* %i

  %tmp = sub i32 %lv, 1
  store i32 %tmp, i32* %i
  %lv3 = load i32, i32* %i
  %tmp4 = icmp eq i32 %lv3, 9
  br i1 %tmp4, label %if.then, label %if.else14

while.merge.block:                                ; preds = %while.cmp.block
  ret i32 0

if.else.merge:                                    ; preds = %if.else14,
%while.merge.block7
  br label %while.cmp.block
if.then:                                       ; preds = %while.body
  br label %while.cmp.block5

while.cmp.block5:                              ; preds =
%if.else.merge10, %if.then11, %if.then
  %lv12 = load i32, i32* %j
  %tmp13 = icmp sgt i32 %lv12, 0
  br i1 %tmp13, label %while.body6, label %while.merge.block7

while.body6:                                   ; preds =
%while.cmp.block5
  %lv8 = load i32, i32* %x
  %tmp9 = icmp eq i32 %lv8, 9
  br i1 %tmp9, label %if.then11, label %if.else

while.merge.block7:                            ; preds =
%while.cmp.block5
  br label %if.else.merge

if.else.merge10:                               ; preds = %if.else,
%after.cont
  br label %while.cmp.block5

if.then11:                                     ; preds = %while.body6
  br label %while.cmp.block5

after.cont:                                    ; No predecessors!
  br label %if.else.merge10

if.else:                                       ; preds = %while.body6
  br label %if.else.merge10

if.else14:                                     ; preds = %while.body
  br label %if.else.merge
}
```

7.2. Test Suites

7.2.1. Why and How these Test Cases were Chosen

Our test cases fall into two categories. The first category is basic tests. Basic tests server as unit tests that targets on particular basic functionalities we implement. For example, string declaration, assignment, etc..

The second category is synthetic test cases. These test cases involves multiple components and implemented functionalities, like our tic-tac-toe and hello world.

7.2.2. Type of Automation Used In Testing

We used a bash script to test over 150 test cases with both tests and failure scenarios. As we started to get a huge number of test cases, Suzanna wrote a script to output errors into log files and read in test files based directory structure. The script has command-line features to run certain regression tests. Typing ./regression-test.sh -h lists the different test cases that can be run separately. If no command-line arguments are passed to the test script then all the 150+ test cases are run.

7.2.3. Who Did What in Testing

The table below summarizes each person's contributions to testing.

Team Member	**Testing Role**
Rui Gu	Wrote the main regression testing script. Wrote test cases for sections implemented.
Adam Hadar	Wrote test cases for the sections I implemented: modulo, array assignment, declaration, passing.
Zachary Moffitt	Wrote test cases for sections implemented.
Suzanna Schmeelk	Suzanna wrote the main regression testing script. The script counts the tests that pass and fail. Suzann created testcases and created directories based on the regression test cases.

8. Lessons Learned (by Team)

In this section we describe the lessons learned by the team for an accelerated summer class.

8.1. Rui Gu, rg2970

One thing I learnt is a good time management skill is an absolute requirement for a group project. Other things would be always run regression test before commit. Also, docker is really convenient.

8.2. Adam Hadar, anh2130

Start early, and test often. Communicate with your TA regularly (and more often than the regular once a week meetings) regarding issues and overall direction. Never merge to master if there are bugs. Git is incredibly annoying to resolve merge conflicts, so make sure to allocate a large amount of time to merging code together. Or get a better VC system. Merging code from various branches will also be a huge time suck, and literally everything you've done will break, so take that amount of time you devoted to merging code and double it. Stay in daily contact with your teammates, to make sure that if they are struggling you can help them, (or to make sure they are working at all). You will all be working on the same pieces of code, so there will be a lot of overlap between yourself and your teammates in terms of the things you are writing (Zach and I wrote the same Array Literal code independently). This will lead to many, many merge conflicts. And deleting redundancies. And making your life very difficult in the hours before you submit your final project.

8.3. Zachary Moffitt, znm2104

I learned that you should keep strict deadlines with your team and always pad at least 50% of your time, if not more. Both Suzanna and I work full time and had difficulty securing a time that would work well for everyone. I believe that we spent so much of our time trying to organize how we were going to complete the project and meet our goals and at the same time underestimating the varying difficulty between different tasks. With this we also found that working on the same pieces of code and trying to access memory elements via LLVM ends up being extremely difficult and we do not realize what the necessary variables are until you've completely worked out a function.

8.4. Suzanna Schmeelk, ss4648

I learned a lot during this awesome five week intensive course--especially time management. For prospective students, you will need to manage homeworks, extended classes, deadlines, final exam, final write-up, teammate schedules, as well as writing a compiler in OCaml and potentially learn LLVM in 5 weeks total.). As the project manager, I spent a great deal of time tasking out our compiler features on Asana. Asana turned out to be a huge help for the team but the team left some of their features unimplemented. Also, our team was awesome with version control, so we never had a serious merge issue until the very end. I was pleasantly surprised by how well Git ended up working out, albeit Git can be quite frustrating at times. All teammates tested their code before merging with the master branch which I seriously advise to prospective students.

9. Appendix - Full Code Listing (by Team)

9.1 Main file (simplicty.ml)

```
(*
Project:  COMS S4115, SimpliCty Compiler
Filename: src/simplicty.ml
Authors:  - Rui Gu,           rg2970
          - Adam Hadar,       anh2130
          - Zachary Moffitt,  znm2104
          - Suzanna Schmeelk, ss4648
Purpose:  * Top level for SimpliCty compiler
            * Scan & parse input, global variables
            * Check each function in the resulting AST, generate LLVM IR, dump module
Modified: 2016-08-11
*)

type action = Ast | LLVM_IR | Compile

let _ =
  if Array.length Sys.argv <= 1 then (
    print_string "Usage: ./simplicty [-a|-l|-c] source_code\n";
    exit 1);;
  let action = if Array.length Sys.argv > 2 then
    List.assoc Sys.argv.(1) [ ("-a", Ast);   (* Print the AST only *)
                             ("-l", LLVM_IR);  (* Generate LLVM, don't check *)
                             ("-c", Compile) ] (* Generate, check LLVM IR *)
  else Compile in
  let inputfile = Sys.argv.(Array.length Sys.argv - 1) in
  let lexbuf = Lexing.from_channel (open_in inputfile) in
  let ast = Parser.program Scanner.token lexbuf in
  Semant.check ast;
  match action with
    Ast -> print_string (Ast.string_of_program ast)
  | LLVM_IR -> print_string (Llvm.string_of_llmodule (Codegen.translate ast))
  | Compile -> let m = Codegen.translate ast in
    Llvm_analysis.assert_valid_module m;
    print_string (Llvm.string_of_llmodule m)
```

9.2 Scanner

```
(*
Project:  COMS S4115, SimpliCty Compiler
Filename: src/scanner.mll
Authors:  - Rui Gu,           rg2970
       - Adam Hadar,          anh2130
       - Zachary Moffitt,  znm2104
       - Suzanna Schmeelk, ss4648
Purpose:  * Scan an inputted SimpliCty file
Modified: 2016-08-11
*)

{ open Parser }
```

```
rule token = parse
  [' ' '\t' '\r' '\n'] { token lexbuf } (* Whitespace *)
| "/*"  { comment lexbuf }            (* Comments *)
| '('   { LPAREN }
| ')'   { RPAREN }
| '{'   { LBRACE }
| '}'   { RBRACE }
| '['   { LBRACKET }
| '''   { SINGLEQT }
| '"'   { DOUBLEQT }
| ']'   { RBRACKET }
| "{|"        { OPENARR }
| "|}"  { CLOSEARR }
| ';'   { SEMI }
| ','   { COMMA }
| '+'   { PLUS }
| '-'   { MINUS }
| '*'   { TIMES }
| '/'   { DIVIDE }
| '%'   { MODULO }
| '='   { ASSIGNREG }
| "+="  { ASSIGNADD }
| "-="  { ASSIGNSUB }
| "*="  { ASSIGNMULT }
| "/="  { ASSIGNDIV }
| "%="  { ASSIGNMOD }
| "=="  { EQ }
| "!="  { NEQ }
| '<'   { LT }
| "<="  { LEQ }
| ">"   { GT }
| ">="  { GEQ }
| "&&"  { AND }
| "||"  { OR }
| "!"   { NOT }
| "++"  { PLUSPLUS }
| "--"  { MINUSMINUS }
| "if"  { IF }
| "else"   { ELSE }
| "for"{ FOR }
| "while"  { WHILE }
| "break"  { BREAK }
| "continue" { CONTINUE }
| "return" { RETURN }
| "int"{ INT }
| "float"  { FLOAT }
| "char"   { INT }
| "bool"   { BOOL }
| "void"   { VOID }
| "true"   { TRUE }
| "string" { STRING }
| "false"  { FALSE }
| "extern" { EXTERN }
| ['+' '-']?['0'-'9']+ as lxm { INTLIT(int_of_string lxm) }
| ['a'-'z' 'A'-'Z']['a'-'z' 'A'-'Z' '0'-'9' '_']* as lxm { ID(lxm) }
| '\''['a'-'z' 'A'-'Z' ' ' '!' '0'-'9']*'\'' as lxm { INTLIT(int_of_char lxm.[1]) }
| '"'['a'-'z' 'A'-'Z' ' ' '!' '0'-'9']+'"' as lxm { STRINGS(lxm) }
| ['+' '-']?['0'-'9']*'.'['0'-'9']* as lxm { FLOATLIT(float_of_string lxm) }
| ['+' '-']?['0'-'9']['.']?['0'-'9']*'e'['-' '+']?['0'-'9']* as lxm {
FLOATLIT(float_of_string lxm) }
| ['+' '-']?['0'-'9']*'e'['-' '+']?['0'-'9']* as lxm { FLOATLIT(float_of_string lxm) }
| eof { EOF }
| _ as char { raise (Failure("illegal character " ^ Char.escaped char)) }

and comment = parse
```

```
  "*/" { token lexbuf }
| _ { comment lexbuf }
```

9.2. Parser

```
/*
Project:  COMS S4115, SimpliCty Compiler
Filename: src/parser.mly
Authors:  - Rui Gu,           rg2970
       - Adam Hadar,          anh2130
       - Zachary Moffitt,  znm2104
       - Suzanna Schmeelk, ss4648
Purpose:  * Ocamlyacc parser for SimpliCty
Modified: 2016-08-11
*/

%{
open Ast

let explode s = let rec f t = function
       | -1 -> t
       | h -> f (s.[h] :: t) (h - 1)
  in f [] (String.length s - 1)
%}

%token SEMI LPAREN RPAREN LBRACE RBRACE LBRACKET RBRACKET COMMA SINGLEQT DOUBLEQT OPENARR
CLOSEARR
%token PLUS MINUS TIMES DIVIDE MODULO
%token NOT PLUSPLUS MINUSMINUS
%token ASSIGNREG ASSIGNADD ASSIGNSUB ASSIGNMULT ASSIGNDIV ASSIGNMOD
%token EQ NEQ LT LEQ GT GEQ TRUE FALSE AND OR
%token RETURN BREAK CONTINUE IF ELSE FOR WHILE INT FLOAT BOOL VOID CHAR STRING
%token PRINT EXTERN
%token <int> INTLIT
%token <string> ID
%token <float> FLOATLIT
%token <char> CHARLIT
%token <string> STRINGS
%token EOF

%nonassoc NOELSE
%nonassoc ELSE
%nonassoc PRINT EXTERN
%right ASSIGNADD ASSIGNSUB ASSIGNMULT ASSIGNDIV ASSIGNMOD
%right ASSIGNREG
%left OR
%left AND
%left EQ NEQ
%left LT GT LEQ GEQ
%left PLUS MINUS
%left TIMES DIVIDE MODULO
%nonassoc PLUSPLUS MINUSMINUS
%right NOT NEG

%start program
%type <Ast.program> program

%%

program:
  modul_definitions EOF { $1 }

modul_definitions:
        /* nothing */ { [], [], [] }
 | modul_definitions declaration      { let (a, b, c) = $1 in ($2 :: a), b, c }
 | modul_definitions extern_func_decl { let (a, b, c) = $1 in a, ($2 :: b), c }
 | modul_definitions func_definition { let (a, b, c) = $1 in a, b, ($2 :: c) }

func_definition:
   typ_specifier ID LPAREN parameter_list_opt RPAREN LBRACE declaration_list statement_list 
RBRACE
        { { typ = $1;
     fname = $2;
     formals = $4;
     locals = List.rev $7;
     body = List.rev $8 } }

extern_func_decl:
   EXTERN typ_specifier ID LPAREN parameter_list_opt RPAREN SEMI
        { { e_typ = $2;
     e_fname = $3;
     e_formals = $5 } }

parameter_list_opt:
        /* nothing */ { [] }
  | parameter_list   { List.rev $1 }

parameter_list:
  | parameter { [$1] }
  | parameter_list COMMA  parameter { $3 :: $1 }

parameter:
        typ_specifier ID                      { ($1,$2, Primitive, [1]) }
  | typ_specifier LBRACKET RBRACKET ID { ($1,$4, Array,     [0]) }
  | typ_specifier size_decl ID       { ($1,$3, Array,      List.rev $2) }

typ_specifier:
        INT   { Int }
  | FLOAT { Float }
  | CHAR  { Char }
  | BOOL  { Bool }
  | STRING { String }
  | VOID  { Void }

declaration_list:
        /* nothing */  { [] }
  | declaration_list declaration { $2 :: $1 }

declaration:
        typ_specifier ID SEMI                                       { ($1, $2, Primitive, [0], 
[] ) }
  | typ_specifier ID ASSIGNREG primary SEMI                 { ($1, $2, Primitive, [0], [$4]) }
  | typ_specifier size_decl ID SEMI                         { ($1, $3, Array,     List.rev $2, 
[]) }
  | typ_specifier size_decl ID ASSIGNREG decl_assign_arr SEMI { ($1, $3, Array,   List.rev $2, 
$5) }

decl_assign_arr:
        OPENARR arr_assign CLOSEARR {List.rev $2}

arr_assign:
        primary {[$1]}
  | arr_assign STRINGS                       { (List.map (fun x -> (CharLit(x))) (explode 
$2)) }
  | arr_assign COMMA primary {$3::$1}
```

```
size_decl:
        LBRACKET INTLIT RBRACKET             { [$2] }
  | size_decl LBRACKET INTLIT RBRACKET { $3::$1 }

statement_list:
        /* nothing */  { [] }
  | statement_list statement { $2 :: $1 }

statement:
        expression SEMI               { Expr $1 }
  | LBRACE statement_list RBRACE { Block(List.rev $2) }
  | IF LPAREN expression RPAREN statement %prec NOELSE   { If($3, $5, Block([])) }
  | IF LPAREN expression RPAREN statement ELSE statement { If($3, $5, $7) }
  | WHILE LPAREN expression RPAREN statement { While($3, $5) }
  | FOR LPAREN expression_opt SEMI expression SEMI expression_opt RPAREN statement
        { For($3, $5, $7, $9) }
  | BREAK SEMI                { Break }
  | CONTINUE SEMI             { Continue }
  | RETURN SEMI               { Return Noexpr }
  | RETURN expression SEMI { Return $2 }

expression_opt:
        /* nothing */ { Noexpr }
  | expression { $1 }

expression:
        primary                       { Primary($1) }
  | LPAREN expression RPAREN  { $2 }
  | OPENARR expression_list CLOSEARR {ArrLit($2)}
  | lvalue arr_pos { Lvarr($1,List.rev $2) }
  | expression PLUS   expression { Binop($1, Add,    $3) }
  | expression MINUS  expression { Binop($1, Sub,    $3) }
  | expression TIMES  expression { Binop($1, Mult,   $3) }
  | expression DIVIDE expression { Binop($1, Div,    $3) }
  | expression MODULO expression { Binop($1, Mod,    $3) }
  | expression EQ     expression { Binop($1, Equal,   $3) }
  | expression NEQ    expression { Binop($1, Neq,   $3) }
  | expression LT     expression { Binop($1, Less,  $3) }
  | expression LEQ    expression { Binop($1, Leq,   $3) }
  | expression GT     expression { Binop($1, Greater, $3) }
  | expression GEQ    expression { Binop($1, Geq,   $3) }
  | expression AND    expression { Binop($1, And,   $3) }
  | expression OR     expression { Binop($1, Or,    $3) }
  | MINUS expression %prec NEG   { Unop(Neg, $2) }
  | NOT expression            { Unop(Not, $2) }
  | PLUSPLUS expression       { Crement(Pre,  PlusPlus,   $2) }
  | MINUSMINUS expression     { Crement(Pre,  MinusMinus, $2) }
  | expression PLUSPLUS       { Crement(Post, PlusPlus,   $1) }
  | expression MINUSMINUS     { Crement(Post, MinusMinus, $1) }
  | expression ASSIGNREG expression  { Assign($1, AssnReg,  $3) }
  | expression ASSIGNADD expression  { Assign($1, AssnAdd,  $3) }
  | expression ASSIGNSUB expression  { Assign($1, AssnSub,  $3) }
  | expression ASSIGNMULT expression { Assign($1, AssnMult, $3) }
  | expression ASSIGNDIV expression  { Assign($1, AssnDiv,  $3) }
  | expression ASSIGNMOD expression  { Assign($1, AssnMod,  $3) }
  | ID LPAREN expression_list_opt RPAREN { Call($1, $3) }

primary:
        INTLIT   { IntLit($1) }
  | FLOATLIT { FloatLit($1) }
  | CHARLIT  { CharLit($1) }
  | TRUE       { BoolLit(true) }
  | FALSE      { BoolLit(false) }
  | lvalue   { Lvalue($1) }

lvalue:
       ID                            { Id($1) }
  /*| ID LBRACKET expression RBRACKET { Arr($1,$3) }*/

arr_pos:
       LBRACKET expression RBRACKET          {[$2]}
  | arr_pos LBRACKET expression RBRACKET {$3::$1}

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

expression_list:
       expression                    { [$1] }
  | STRINGS                  { ((List.map (fun x -> (StringConv(x))) (explode $1))) }
| expression_list COMMA expression { $3 :: $1 }
```

9.4. AST

```
(*
Project:  COMS S4115, SimpliCty Compiler
Filename: src/ast.ml
Authors:  - Rui Gu,           rg2970
       - Adam Hadar,          anh2130
       - Zachary Moffitt,  znm2104
       - Suzanna Schmeelk, ss4648
Purpose:  * Generate abstract syntax tree
       * Functions for printing the AST
Modified: 2016-08-11
*)

type decl = Primitive | Array (* | Struct *)

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

type uop = Neg | Not

type crement = PlusPlus | MinusMinus

type crementDir = Pre | Post

type typ = Int | Float | Bool | Void | Char | String

type assn = AssnReg | AssnAdd | AssnSub | AssnMult | AssnDiv | AssnMod

type lvalue =
       Id of string
  (*| Arr of string * int *)

type primary =
       IntLit of int
  | FloatLit of float
  | CharLit of char
  | BoolLit of bool
  | Lvalue of lvalue

type expr =
        Primary of primary
  | ArrLit of expr list
  | Lvarr of lvalue * (expr list)
  | Binop of expr * op * expr
  | Unop of uop * expr
  | Crement of crementDir * crement * expr
  | Assign of expr * assn * expr
  | Call of string * expr list
  | StringConv of char
  | Noexpr

type parameter = typ * string * decl * (int list)

type declaration = typ * string * decl * (int list) * (primary list)

type function_declaration = typ * string * decl * expr

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

type extern_func_decl = {
        e_typ : typ;
        e_fname : string;
        e_formals : parameter list;
  }

type func_decl = {
        typ : typ;
        fname : string;
        formals : parameter list;
        locals : declaration list;
        body : stmt list;
  }

type program = declaration list * extern_func_decl list * func_decl list

(* Pretty-printing functions *)

let string_of_decl = function
        Primitive -> "prime"
  | Array -> "array"
  (*| Struct -> "struct"*)

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 string_of_crement = function
        PlusPlus   -> "++"
  | MinusMinus -> "--"

let string_of_crementDir = function
        Pre -> "pre"
  | Post-> "post"

let string_of_assn = function
        AssnReg  -> "="
  | AssnAdd  -> "+="
  | AssnSub  -> "-="
  | AssnMult -> "*="
  | AssnDiv  -> "/="
  | AssnMod  -> "%="

let string_of_lvalue = function
        Id(s)   -> s

let string_of_primary = function
        IntLit(i)      -> string_of_int i
  | FloatLit(f)   -> string_of_float f
  | CharLit(c) -> string_of_int (int_of_char c)
  | BoolLit(l) -> if l = true then "true" else "false"
  | Lvalue(l)  -> string_of_lvalue l

let rec string_of_expr = function
        Primary(l)            ->
        string_of_primary l
  | StringConv(s)     -> string_of_int(Char.code s)
  | ArrLit(lp) ->
    "{|"^ String.concat ", " (List.map string_of_expr lp) ^ "|}"
  | Lvarr(lv, le)      ->
        string_of_lvalue lv ^"["^ String.concat "][" (List.map string_of_expr le) ^"]"
  | 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
  | Crement(oD, o, e_lv)-> (match oD with
        Pre  -> string_of_crement o ^" "^ string_of_expr e_lv
        | Post -> string_of_expr e_lv ^" "^ string_of_crement o)
  | Assign(e_lv, o, e)  ->
        string_of_expr e_lv ^" "^ string_of_assn o ^" "^ 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";
  | Break              -> "break;\n";
  | Continue           -> "continue;\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"
  | Float -> "float"
  | Char  -> "char"
  | Bool  -> "bool"
  | String -> "string"
  | Void  -> "void"

let string_of_vdecl (t, id, decl, size_list, prim_list) =
   let size' =
       if decl = Primitive then ""
       else "["^ string_of_int (List.hd size_list) ^"]"
   and assn =
       if List.length prim_list = 0 then ""
       else
       let value =
       if decl = Primitive then string_of_primary (List.hd prim_list)
       else "{|"^ String.concat ", " (List.map string_of_primary prim_list) ^ "|}"
       in
       " = " ^ value
   in
   string_of_typ t ^ size' ^" "^ id ^ assn ^";\n"

let snd_of_four (_,id,_,_) = id

let string_of_fdecl fdecl =
  string_of_typ fdecl.typ ^ " " ^
  fdecl.fname ^ "(" ^ String.concat ", " (List.map snd_of_four 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_extern_fdecl efdecl =
  string_of_typ efdecl.e_typ ^ " " ^
  efdecl.e_fname ^ "(" ^ String.concat ", " (List.map snd_of_four efdecl.e_formals) ^
  ");\n"

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

9.5. Semantic Checking

```
(*
Project:  COMS S4115, SimpliCty Compiler
Filename: src/semant.ml
Authors:  - Rui Gu,           rg2970
       - Adam Hadar,          anh2130
       - Zachary Moffitt,  znm2104
       - Suzanna Schmeelk, ss4648
Purpose:  * Semantic checking for the SimpliCty compiler
       * Returns void if successful. Otherwise throws exception.
Modified: 2016-08-11
*)
```

```
open Ast

module StringMap = Map.Make(String)

let check (globals, externs, 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 snd_of_four (_,id,_,_) = id in
  let snd_of_five (_, id, _, _, _) = id in
  let check_not_void_four exceptf = function
        (Void, n, _, _) -> raise (Failure (exceptf n))
        | _ -> ()
  in
  let check_not_void_five 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 Global Variables ****)
  List.iter (check_not_void_five (fun n -> "illegal void global " ^ n)) globals;
  report_duplicate (fun n -> "duplicate global " ^ n) (List.map snd_of_five globals);

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

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

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

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

  (* Function declaration for a named function *)
  let built_in_decls =  StringMap.add "print"
        { typ = Void; fname = "print"; formals = [(Int, "x", Primitive, [])];
        locals = []; body = [] } (StringMap.singleton "printb"
        { typ = Void; fname = "printb"; formals = [(Bool, "x", Primitive, [])];
        locals = []; body = [] })
  in
  let built_in_decls =  StringMap.add "putchar"
        { typ = Void; fname = "putchar"; formals = [(Int, "x", Primitive, [])];
        locals = []; body = [] } built_in_decls
  in
  let built_in_decls =  StringMap.add "getchar"
        { typ = Int; fname = "getchar"; formals = [];
        locals = []; body = [] } built_in_decls
```

```
in
let function_decls = List.fold_left (fun m fd -> StringMap.add fd.fname fd m)
                     built_in_decls functions
in
let function_decls = List.fold_left (fun m ed -> StringMap.add ed.e_fname
      { typ = ed.e_typ; fname = ed.e_fname; formals = ed.e_formals;
      locals = []; body = [] } m)
                     function_decls externs
in
let function_decl s = try StringMap.find s function_decls
      with Not_found -> raise (Failure ("unrecognized function " ^ s))
in

(*let _ = function_decl "main" in*) (* Ensure "main" is defined *)

let check_function func =

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

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

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

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

      (* Type of each variable (global, formal, or local *)
      let symbols = List.fold_left (fun m (t, n, _, _, _) -> StringMap.add n t m)
      StringMap.empty globals
      in
      let symbols = List.fold_left (fun m (t, n, _, _) -> StringMap.add n t m)
  symbols func.formals
      in
      let symbols = List.fold_left (fun m (t, n, _, _, _) -> StringMap.add n t m)
      symbols func.locals
      in

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

      let primary = function
      IntLit _  -> Int
      | FloatLit _  -> Float
      | BoolLit _  -> Bool
      | CharLit _  -> Char
      | Lvalue Id(s) -> type_of_identifier s
      in
      (* Return the type of an expression or throw an exception *)
      let rec expr = function
      Primary p -> primary p
      | ArrLit _ -> Int (*TODO-ADAM: TrASH*)
      | Lvarr(Id(s),_) -> type_of_identifier s (*TODO-ADAM: semantic checking*)
      | 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
      | Add | Sub | Mult | Div | Mod when t1 = Float && t2 = Float -> Float
      | Add | Sub | Mult | Div | Mod when t1 = Int && t2 = Float -> Float
      | Add | Sub | Mult | Div | Mod when t1 = Float && t2 = Int -> Float
```

```
        | Add | Sub | Mult | Div | Mod when t1 = Bool && (t2 = Int || t2 == Float) -> raise
(Failure (
                "illegal cast with operator "^ string_of_typ t1 ^" "^ string_of_op op ^" "^
                string_of_typ t2 ^" in "^ string_of_expr e
                ))
        | Add | Sub | Mult | Div | Mod when (t1 = Int || t1 = Float) && t2 = Bool -> raise
(Failure (
                "illegal cast with operator "^ string_of_typ t1 ^" "^ string_of_op op ^" "^
                string_of_typ t2 ^" in "^ string_of_expr e
                ))
        | Equal | Neq when t1 = t2                                  -> Bool
        | Less | Leq | Greater | Geq when t1 = Int && t2 = Int   -> Bool
        | Less | Leq | Greater | Geq when t1 = Float && t2 = Float   -> 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
                ))
        )
        | Unop(op, e_lv) as ex ->
        (*TODO-ADAM: failure if thing is an array*)
        let t = expr e_lv 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
                ))
        )
        | Crement(opDir, op, e_lv) as ex ->
        (*TODO-ADAM: failure if thing is an array*)
        let t = expr e_lv in
        (match op with
                _ when t = Int -> Int
        | _            -> raise (Failure (
                "illegal "^ string_of_crementDir opDir ^ string_of_crement op ^
                " "^ string_of_typ t ^" in "^ string_of_expr ex
                ))
        )
        | Noexpr -> Void
        | Assign(e_lv, op, e) as ex ->
        (*TODO-ADAM: check that arrays are assigned to arrays/arrays of same size/no math*)
        let lt = expr e_lv
        and rt = expr e in
      (match op with
                _ -> 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 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
        in

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

        (* Verify a statement or throw an exception *)
        let rec stmt = function
    Block sl -> let rec check_block = function
        [Return _ as s] -> stmt s
        | Return _ :: _ -> raise (Failure "nothing may follow a return")
        | Block sl :: ss -> check_block (sl @ ss)
        | s :: ss -> stmt s ; check_block ss
        | [] -> ()
        in check_block sl
        | Expr e -> ignore (expr e)
        | Break -> ignore ()   (*TODO: Include outside loop check *)
        | Continue -> ignore ()  (*TODO: Include outside loop check *)
        | Return e -> let t = expr e in if t = func.typ then () else
        raise (Failure ("return gives " ^ string_of_typ t ^ " expected " ^
                       string_of_typ func.typ ^ " in " ^ string_of_expr e))

        | If(p, b1, b2) -> check_bool_expr p; stmt b1; stmt b2
        | For(e1, e2, e3, st) -> ignore (expr e1); check_bool_expr e2;
                               ignore (expr e3); stmt st
        | While(p, s) -> check_bool_expr p; stmt s
        in

        stmt (Block func.body)

  in
List.iter check_function functions
```

9.6. CodeGen

```
(*
Project:  COMS S4115, SimpliCty Compiler
Filename: src/codegen.ml
Authors:  - Rui Gu,           rg2970
        - Adam Hadar,         anh2130
        - Zachary Moffitt,  znm2104
        - Suzanna Schmeelk, ss4648
Purpose:  * Translates semantically checked SimpliCty AST to LLVM IR
        * Functions for printing the AST
Modified: 2016-08-11
*)
(*: Make sure to read the OCaml version of the tutorial
http://llvm.org/docs/tutorial/index.html
Detailed documentation on the OCaml LLVM library:
http://llvm.moe/
http://llvm.moe/ocaml/
*)

module L = Llvm
module A = Ast
module StringMap = Map.Make(String)

let translate (globals, externs, functions) =
  let context = L.global_context () in
  let the_module = L.create_module context "SimpliCty"
  and i32_t  = L.i32_type   context
  and f32_t  = L.float_type context
  and i1_t   = L.i1_type      context
  and void_t = L.void_type  context in

  let ltype_of_typ = function
        A.Int    -> i32_t
        | A.Float -> f32_t
        | A.Char  -> i32_t
        | A.String -> i32_t
        | A.Bool  -> i1_t
        | A.Void  -> void_t
  in
  let primary_decompose = function
        A.IntLit(i)   -> i
        | A.BoolLit(b)  -> if b then 1 else 0
        | A.FloatLit(f) -> int_of_float f
        | _            -> 0

  and primary_float_decompose = function
        A.IntLit(i)   -> float_of_int i
        | A.BoolLit(b)  -> if b then 1.0 else 0.0
        | A.FloatLit(f) -> f
        | _            -> 0.0

  in

  (* Store memory *)
  let store_primitive addr typ' value builder =
        L.build_store (L.const_int typ' (if List.length value <> 0 then primary_decompose
(List.hd value)
        else 0)
        ) addr builder
  and store_array_idx addr index typ' value builder =
        let i  = [|L.const_int i32_t index|]
        and v' = L.const_int typ' (if List.length value <> 0 then primary_decompose (List.hd
value)
        else 0)
        in
        let addr' = L.build_in_bounds_gep addr i "storeArrIdx" builder in
        L.build_store v' addr' builder
  and store_float_primitive addr typ' value builder =
        L.build_store (L.const_float typ' (if List.length value <> 0
        then primary_float_decompose (List.hd value)
        else 0.0)
        ) addr builder
  and copy_array size old_addr new_addr builder =
        let rec copy_idx idx =(match idx with
        -1 -> 0
        | _  ->
        let idx' = [|L.const_int i32_t idx|] in
        let idx_ptr_n = L.build_in_bounds_gep new_addr idx' "newArr" builder
        and idx_ptr_o = L.build_in_bounds_gep old_addr idx' "oldArr" builder
        in
        let val_old = L.build_load idx_ptr_o "oldArrIdx" builder in
        ignore(L.build_store val_old idx_ptr_n builder); copy_idx (idx-1)
        ) in copy_idx (size-1)
  in

  (* Declare each global variable; remember its value in a map *)
  (*TODO-ADAM: global scoped arrays*)
  let global_vars =
        let global_var m (typ, name, decl, size_list, values) =
        let typ' = ltype_of_typ typ in
        let init_val v =
        (match typ with
        A.Float -> L.const_float typ' (if List.length values <> 0 then
primary_float_decompose v else 0.0)
```

```
      | _     -> L.const_int   typ' (if List.length values <> 0 then primary_decompose v
      else 0)
      )
      in
      let init = (match decl with
      A.Primitive -> init_val (List.hd values)
      | A.Array      -> L.const_array typ' (Array.of_list (List.map init_val values))
      ) in
      let addr = L.define_global name init the_module in
      StringMap.add name (addr, decl, size_list) m
      in
      List.fold_left global_var StringMap.empty globals in

(* Declare putchar(), which the putchar built-in function will call *)
let putchar_t = L.function_type i32_t [| i32_t |] in
let putchar_func = L.declare_function "putchar" putchar_t the_module in

(* Define each function (arguments and return type) so we can call it *)
(*L.pointer_type (ltype_of_typ t)*)
let param_type (typ,_,decl,_) =
      (match decl with
      A.Primitive -> ltype_of_typ typ
      | A.Array      -> L.pointer_type (ltype_of_typ typ)
      )
in
let function_decls =
      let function_decl m fdecl =
      let name = fdecl.A.fname
      and formal_types = Array.of_list (List.map param_type 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) m in
      List.fold_left function_decl StringMap.empty functions in

let extern_decls = List.fold_left (fun ed e ->
      { A.typ = e.A.e_typ; A.fname = e.A.e_fname; A.formals = e.A.e_formals;
      A.locals = []; A.body = [] } :: ed)
                     [] externs
in

let function_decls =
      let function_decl m fdecl =
      let name = fdecl.A.fname
      and formal_types =
  Array.of_list (List.map param_type fdecl.A.formals)
      in let ftype = L.function_type (ltype_of_typ fdecl.A.typ) formal_types in
      StringMap.add name (L.declare_function name ftype the_module, fdecl) m in
      List.fold_left function_decl function_decls extern_decls 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

      (* 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 *)
      let local_vars =
      let add_formal m (typ, name, decl, size_list) p =
      L.set_value_name name p;
      let typ' = ltype_of_typ typ in
      (match decl with
      A.Primitive ->
              let addr = L.build_alloca typ' name builder in
      ignore(L.build_store p addr builder); StringMap.add name (addr,decl,size_list) m
```

```
        | A.Array ->
                let full_size = List.fold_left (fun i s -> i*s) 1 size_list in
                if full_size <> 0 then
                let size' = L.const_int i32_t full_size in
                let addr = L.build_array_alloca typ' size' name builder in
                ignore(copy_array full_size p addr builder); StringMap.add name
(addr,decl,size_list) m
                else
                StringMap.add name (p,decl,size_list) m
        )
        in
        let add_local m (typ, name, decl, size_list, values) =
        let typ' = ltype_of_typ typ in
        let addr = (match decl with
        A.Primitive -> L.build_alloca typ'
        | A.Array      ->
                let full_size = List.fold_left (fun i s -> i*s) 1 size_list in
                let size' = L.const_int i32_t full_size in
                L.build_array_alloca typ' size') name builder in
        (match decl with
        A.Primitive -> (match typ with
                A.Float -> ignore(store_float_primitive addr typ' values builder)
                | _ ->  ignore(store_primitive addr typ' values builder))
        | A.Array      ->
                ignore(List.fold_left (fun index _vals ->
                ignore(store_array_idx addr index typ' [_vals] builder);index+1) 0 values)
        ); StringMap.add name (addr,decl,size_list) 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_addr n =
        (fun (a,_,_) -> a)
        (try StringMap.find n local_vars
        with Not_found -> StringMap.find n global_vars)
        and lookup_decl n =
        (fun (_,b,_) -> b)
        (try StringMap.find n local_vars
        with Not_found -> StringMap.find n global_vars)
        and lookup_size n =
        let (_,_,c) =
        (try StringMap.find n local_vars
        with Not_found -> StringMap.find n global_vars)
        in c
        in
        (*Construct code for lvalues; return value pointed to*)

        let primary builder = function
        A.IntLit i   -> ([L.const_int i32_t i]                       , A.Primitive, [0], i32_t)
        | A.FloatLit f -> ([L.const_float f32_t f]                           , A.Primitive, [0],
f32_t)
        | A.CharLit c  -> ([L.const_int i32_t (int_of_char c)]               , A.Primitive, [0],
i32_t)
        | A.BoolLit b  -> ([L.const_int i1_t (if b then 1 else 0)]           , A.Primitive, [0],
i1_t)
        | A.Lvalue (A.Id(s))  ->
        let addr = lookup_addr s and decl = lookup_decl s and size_list = lookup_size s
        in
        (match decl with
        A.Primitive -> ([L.build_load addr "lv" builder], decl, size_list, i32_t)
        | A.Array      -> ([addr], decl, size_list, i32_t))
        in
        (* Construct code for an expression; return its value *)
        let rec expr builder = function
        A.Primary p            -> primary builder p
        | A.ArrLit lp ->
        let list_primary = List.fold_left (fun li p ->
                let (p',_,_,_) = expr builder p in
                (List.hd p')::li
                ) [] lp in
        (list_primary, A.Array, [List.length list_primary], i32_t)
        | A.Lvarr (A.Id(lv), e_list)->
        let lv' = lookup_addr lv
        and decl = lookup_decl lv
        and size = lookup_size lv
        (*TODO-ADAM: throwing away values*)
        and pos_list = List.map (fun e ->
                let (e',_,_,_) = expr builder e in
                List.hd e'
        ) e_list
        in
        let e'' = (
                if List.length size = 2 then
                let mul = L.build_mul (List.hd pos_list) (L.const_int i32_t (List.nth size
1)) "mult" builder in
                L.build_add mul (List.nth pos_list 1) "add" builder
                else if List.length size = 1 then List.hd pos_list
                else L.const_int i32_t 0)
        in
        (*let addr = L.build_in_bounds_gep lv' [|L.const_int i32_t 0|] "arrPtr" builder in
        let addr' = L.build_in_bounds_gep addr [|e'|] "arrIdx" builder in*)
        let addr' = L.build_gep lv' [|e''|] "arrIdx" builder in
        ([L.build_load addr' "idxIn" builder],decl,[0], i32_t)
        | A.Noexpr             -> ([L.const_int i32_t 0], A.Primitive, [0], i32_t)
        | A.Binop (e1, op, e2) ->
        let e1' =
         let (t1,_,_,t2) = expr builder e1 in
         ((List.hd t1),t2)
        (*match (expr builder e1) with
                (c ,A.Primitive,_,e1_type) -> (c, e1_type)
        | (p ,_,_,e1_type) ->   (L.const_ptrtoint (List.hd p) (i32_t), e1_type)
      *)and e2' =
         let(t1,_,_,t2) = expr builder e2 in
         ((List.hd t1),t2)  (*match (expr builder e2) with
                (c,A.Primitive,_,e2_type) -> (c, e2_type)
        | (p,_,_,e2_type)             ->  (L.const_inttoptr (List.hd p) (i32_t), e2_type)
        *)in
    let flot = f32_t and iont = i32_t and ff = f32_t in
      let (e1', e2') = (match (snd e1', snd e2') with
                ff, f32_t -> ((fst e1' , ff) , (fst e2' ,  f32_t))
                | i32_t, f32_t -> ((L.const_sitofp (fst e1') f32_t,  f32_t), (fst e2' ,
f32_t))
                | f32_t, i32_t -> ((fst e1', flot), (L.const_sitofp (fst e2') f32_t, f32_t))
                | _ , _ -> ((fst e1', iont), (fst e2', iont))
        ) in
        (match snd e1', snd e2' with
        ff,  f32_t ->
                ([(match op with
                A.Add   -> L.build_fadd
                | A.Sub        -> L.build_fsub
                | A.Mult       -> L.build_fmul
                | A.Div        -> L.build_fdiv
                | A.Mod        -> L.build_frem
                | A.And        -> L.build_and
                | A.Or         -> L.build_or
                | A.Equal   -> L.build_fcmp L.Fcmp.Ueq
```

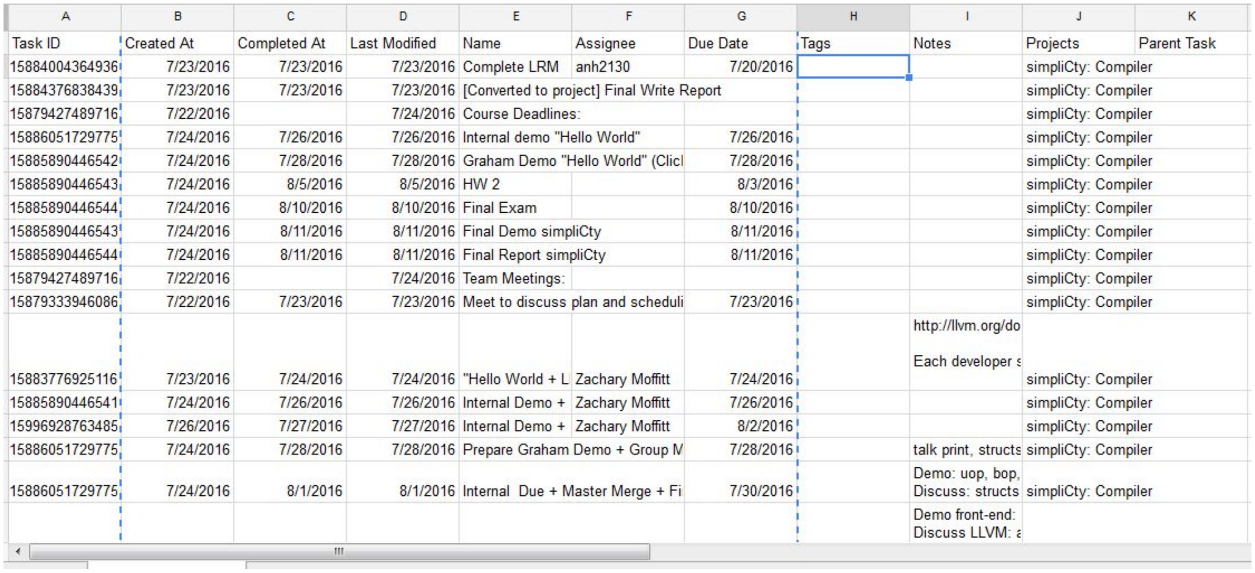
```
               | A.Neq        -> L.build_fcmp L.Fcmp.Une
               | A.Less       -> L.build_fcmp L.Fcmp.Ult
               | A.Leq        -> L.build_fcmp L.Fcmp.Ule
               | A.Greater -> L.build_fcmp L.Fcmp.Ugt
               | A.Geq        -> L.build_fcmp L.Fcmp.Uge)
               (fst e1') (fst e2') "tmp" builder], A.Primitive, [0], f32_t)
        |  _,  _ ->
               ([(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)
               (fst e1') (fst e2') "tmp" builder], A.Primitive, [0], f32_t)
        )
        | A.Unop(op, e_lv) ->
        (*TODO-ADAM: Semantic checking should make sure e_lv is an lv*)
        let (e',_,_,typ) = expr builder e_lv in
    let e'' = (List.hd e') in
      ([(match op with
        A.Neg  -> L.build_neg
        | A.Not        -> L.build_not) e'' "unop" builder], A.Primitive, [0], typ)
        | A.Crement(opDir, op, e_lv) ->
        (*TODO-ADAM: Semantic checking should make sure e_lv is an lv*)
        (match opDir with
               A.Pre  -> expr builder (A.Assign(e_lv, (match op with
               A.PlusPlus   -> A.AssnAdd
               | A.MinusMinus -> A.AssnSub), (A.Primary (A.IntLit 1))))
        | A.Post ->
               let (value,decl,_,typ) = expr builder e_lv in
               ignore(expr builder (A.Crement(A.Pre, op, e_lv))); (value, decl, [0], typ)
        )
        | A.Assign (e_lv, op, e) ->
        (*TODO-ADAM: Allow array assignment*)
        (*TODO-ADAM: Semantic checking should make sure e_lv is an lv*)
        let (addr,decl,size) = (match e_lv with
               A.Lvarr(A.Id(lvInner), eInner_list) ->
               let lvI' = lookup_addr lvInner
               and lvdecl = lookup_decl lvInner
               and lvsize = lookup_size lvInner
               (*TODO-ADAM: throwing away values*)
               and pos_list = List.map (fun e->
               let (e',_,_,_) = expr builder e in
               (List.hd e')
               ) eInner_list in
               let eI'' = (
               if List.length lvsize = 2 then
               let mul = L.build_mul (List.hd pos_list) (L.const_int i32_t (List.nth lvsize
1)) "mult" builder in
               L.build_add mul (List.nth pos_list 1) "add" builder
               else if List.length lvsize = 1 then List.hd pos_list
               else L.const_int i32_t 0
               )in
               (*let addrIn = L.build_in_bounds_gep lvI' [|L.const_int i32_t 0|] "arrPtr"
builder in
               *)(L.build_in_bounds_gep lvI' [|L.const_int i32_t 0; eI''|] "arrIdx" builder,
A.Primitive, [0])
```

```
        | A.Primary(A.Lvalue(A.Id(s))) ->
               (lookup_addr s, lookup_decl s, lookup_size s)
        | _ ->
               (*TODO-ADAM: Semantic checking should catch this trash*)
               let trash = L.const_inttoptr (L.const_int i32_t 0) (L.pointer_type i32_t) in
               (L.build_in_bounds_gep trash [|L.const_int i32_t 0|] "trash" builder,
A.Primitive, [0])
        )
        in
        let full_size = List.hd size in
        let eval = (match op with
               A.AssnReg      -> expr builder e
        | A.AssnAdd    -> expr builder (A.Binop(e_lv, A.Add,  e))
        | A.AssnSub    -> expr builder (A.Binop(e_lv, A.Sub,  e))
        | A.AssnMult   -> expr builder (A.Binop(e_lv, A.Mult, e))
        | A.AssnDiv    -> expr builder (A.Binop(e_lv, A.Div,  e))
        | A.AssnMod    -> expr builder (A.Binop(e_lv, A.Mod,  e))
        ) in
        (*TODO-ADAM: throwing away values*)
        (*let (eval',_,_) = eval in*)
        (match decl with
               A.Primitive ->
               let eval' = match eval with (e,_,_,_)-> List.hd e in
               ignore(L.build_store eval' addr builder)
        | A.Array ->
               let (eval',_,siz,typT) = eval in
               let full_siz = List.hd siz in
               if List.length eval' = 1 then
               ignore(copy_array full_size (List.hd eval') addr builder)
               else
               let arrLitAddr = L.build_array_alloca typT (L.const_int i32_t full_siz)
"arrLit" builder in
               ignore(List.fold_left (fun index _vals ->
                       let i = [|L.const_int i32_t index|] in
                       let arrLitIdx = L.build_in_bounds_gep arrLitAddr i "ArrLitIdx" builder
in
                       ignore(L.build_store _vals arrLitIdx builder); index+1)
               0 eval'); ignore(copy_array full_size arrLitAddr addr builder)
    ); eval
        | A.Call ("putchar", [e]) ->
        (*TODO-ADAM: throwing away values*)
        let (actual,_,_,_) = expr builder e in
    let actual' = List.hd actual in
        ([L.build_call putchar_func [|actual'|] "putchar" builder], A.Primitive, [0], i32_t)
        | A.Call (f, act) ->
        let (fdef, fdecl) = StringMap.find f function_decls in
     let actuals = List.rev (List.map (fun a ->
        match expr builder a with (p,_,_,_)->List.hd p) (List.rev act)) in
     let result = (match fdecl.A.typ with
        A.Void -> ""
        | _ -> f ^ "_result") in
        (* TODO-ADAM: convert fdecl.A.typ to A.decl *)
        ([L.build_call fdef (Array.of_list actuals) result builder], A.Primitive, [0],
ltype_of_typ fdecl.A.typ)
        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 llvm code for function statements; return the builder for the statement's
successor *)
```

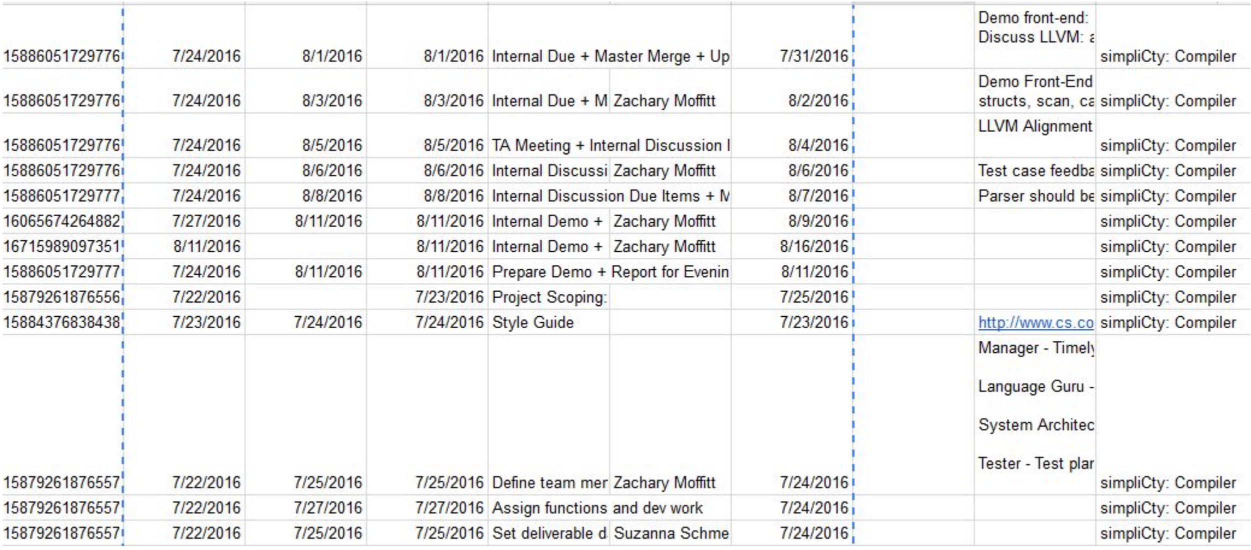
```
        (*let dummy_bb = L.append_block context "dummy.toremove.block" the_function in
        let break_builder = dummy_bb and continue_builder = dummy_bb in*)
        let rec stmt (builder, break_bb, cont_bb) = function
        A.Block sl ->
        List.fold_left stmt (builder, break_bb, cont_bb) sl
        | A.Expr e ->
        ignore (expr builder e); (builder, break_bb, cont_bb)
        | A.Break ->
        ignore(add_terminal builder (L.build_br break_bb));
        let new_block = L.append_block context "after.break" the_function in
        let builder = L.builder_at_end context new_block in (builder, break_bb, cont_bb)
        | A.Continue ->
        ignore(add_terminal builder (L.build_br cont_bb));
        let new_block = L.append_block context "after.cont" the_function in
        let builder = L.builder_at_end context new_block in (builder, break_bb, cont_bb)
        | A.Return e ->
        ignore (match fdecl.A.typ with
        A.Void -> L.build_ret_void builder
        (*TODO-ADAM: return array*)
        (*TODO-ADAM: throwing away value*)
        | _    -> L.build_ret (match expr builder e with (p,_,_,_)->List.hd p) builder);
(builder, break_bb, cont_bb)
        | A.If (predicate, then_stmt, else_stmt) ->
        (*TODO-ADAM: throwing away value*)
        let (bool_val,_,_,_) = expr builder predicate in
    let bool_val' = List.hd bool_val in
        let if_merge_bb = L.append_block context "if.else.merge" the_function in

        let if_then_bb = L.append_block context "if.then" the_function in
        let b = L.builder_at_end context if_then_bb in
        let (temp1, _, _) = stmt (b, break_bb, cont_bb) then_stmt in
        ignore(add_terminal temp1 (L.build_br if_merge_bb));

        let if_else_bb = L.append_block context "if.else" the_function in
        let b = L.builder_at_end context if_else_bb in
        let (temp1, _, _) = stmt (b, break_bb, cont_bb) else_stmt in

        ignore(add_terminal temp1 (L.build_br if_merge_bb));
        ignore (L.build_cond_br bool_val' if_then_bb if_else_bb builder);
        ((L.builder_at_end context if_merge_bb), break_bb, cont_bb)
        | A.While (predicate, body) ->
        let while_pred_bb = L.append_block context "while.cmp.block" the_function in
        ignore (L.build_br while_pred_bb builder);
        let while_body_bb = L.append_block context "while.body" the_function in
        let while_merge_bb = L.append_block context "while.merge.block" the_function in
        let break_builder = while_merge_bb and continue_builder = while_pred_bb in
        let b = L.builder_at_end context while_body_bb in
        let (temp1, _, _) = stmt (b, break_builder, continue_builder) body in
        ignore(add_terminal temp1 (L.build_br while_pred_bb));
        (*if(L.fold_left_instrs ~f:(s->is_terminator s) ~init:() temp1)  (*instr_opcode*)
        then{
        ignore(add_terminal temp1 (L.build_br while_pred_bb));
        }
        else{
        ignore(add_terminal temp1 (L.build_br while_pred_bb));
        }*)
        let pred_builder = L.builder_at_end context while_pred_bb in
        (*TODO-ADAM: throwing away value*)
        let bool_val = match expr pred_builder predicate with (p,_,_,_)->List.hd p in
        ignore (L.build_cond_br bool_val while_body_bb while_merge_bb pred_builder);
        (*ignore(L.replace_all_uses_with (L.build_br dummy_bb) (L.build_br
while_merge_bb));*)
        ((L.builder_at_end context while_merge_bb), break_builder, continue_builder)
        | A.For (e1, e2, e3, body) ->
        stmt (builder, break_bb, cont_bb)
        ( A.Block [A.Expr e1 ; A.While (e2, A.Block [body ; A.Expr e3]) ] )
        in
        (* Build llvm code for each statement in a function *)
        let dummy_bb = L.append_block context "dummy.toremove.block" the_function in
        let break_builder = dummy_bb and continue_builder = dummy_bb in
        let (builder, _, _) = (stmt (builder, break_builder, continue_builder) (A.Block
fdecl.A.body))
        in
        (*let builder = L.builder_at_end context dummy_bb in
        let rec vist_bb_add_terminals = fold_left_blocks (L.block_terminator x) in
        visit_bb_add_terminals the_function*)
        (* Add a return if the last basic block is at the end *)
        add_terminal builder (match fdecl.A.typ with
        A.Void -> L.build_ret_void
        (*TODO-ADAM: return array*)
        | t -> L.build_ret (L.const_int (ltype_of_typ t) 0));
        ignore(L.builder_at_end context dummy_bb);
        ignore(L.block_terminator dummy_bb);
        ignore(L.delete_block dummy_bb);
  in

  List.iter build_function_body functions;
the_module
```

9.7 GitLog

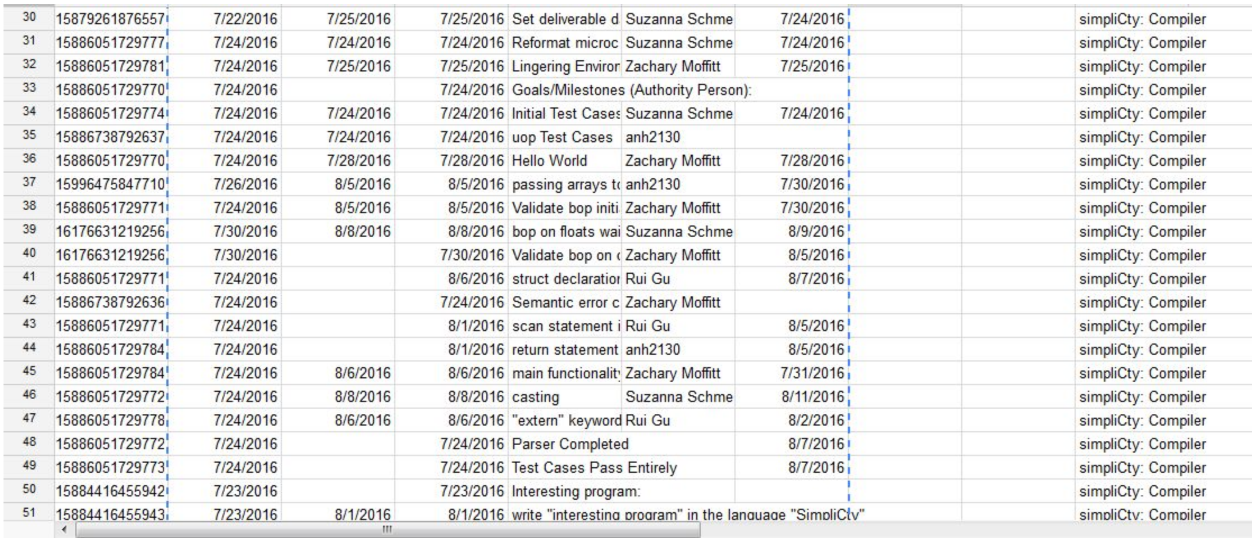
10. Appendix - Asana Project Log



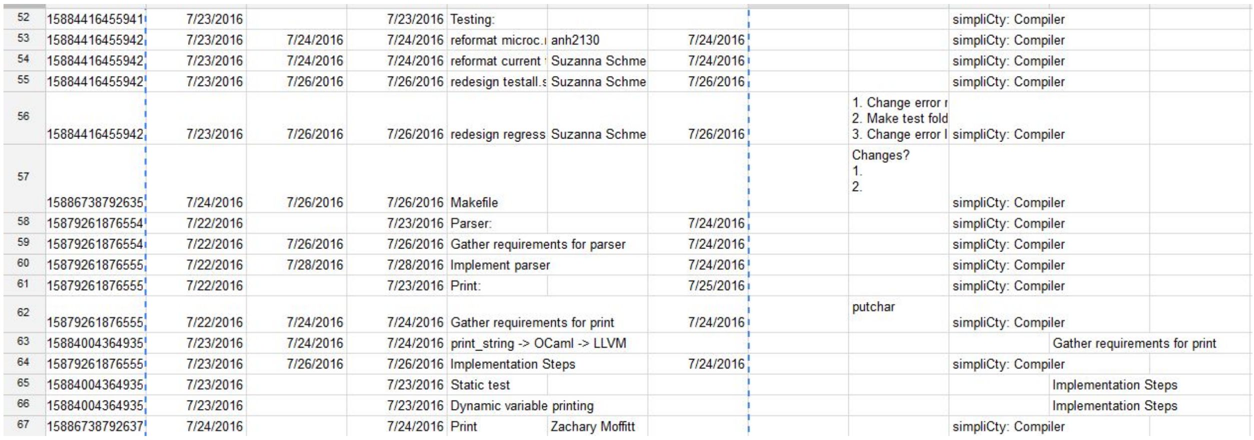
A	B	C	D	E	F	G	H	I	J	K
Task ID	Created At	Completed At	Last Modified	Name	Assignee	Due Date	Tags	Notes	Projects	Parent Task
15884004364936	7/23/2016	7/23/2016	7/23/2016	Complete LRM	anh2130	7/20/2016			simpliCty: Compiler	
15884376838439	7/23/2016	7/23/2016	7/23/2016	[Converted to project] Final Write Report					simpliCty: Compiler	
15879427489716	7/22/2016		7/24/2016	Course Deadlines:					simpliCty: Compiler	
15886051729775	7/24/2016	7/26/2016	7/26/2016	Internal demo "Hello World"		7/26/2016			simpliCty: Compiler	
15885890446542	7/24/2016	7/28/2016	7/28/2016	Graham Demo "Hello World" (Clicl		7/28/2016			simpliCty: Compiler	
15885890446543	7/24/2016	8/5/2016	8/5/2016	HW 2		8/3/2016			simpliCty: Compiler	
15885890446544	7/24/2016	8/10/2016	8/10/2016	Final Exam		8/10/2016			simpliCty: Compiler	
15885890446543	7/24/2016	8/11/2016	8/11/2016	Final Demo simpliCty		8/11/2016			simpliCty: Compiler	
15885890446544	7/24/2016	8/11/2016	8/11/2016	Final Report simpliCty		8/11/2016			simpliCty: Compiler	
15879427489716	7/22/2016		7/24/2016	Team Meetings:					simpliCty: Compiler	
15879333946086	7/22/2016	7/23/2016	7/23/2016	Meet to discuss plan and scheduli		7/23/2016			simpliCty: Compiler	
15883776925116	7/23/2016	7/24/2016	7/24/2016	"Hello World + L	Zachary Moffitt	7/24/2016		http://llvm.org/do Each developer s	simpliCty: Compiler	
15885890446541	7/24/2016	7/26/2016	7/26/2016	Internal Demo +	Zachary Moffitt	7/26/2016			simpliCty: Compiler	
15996928763485	7/26/2016	7/27/2016	7/27/2016	Internal Demo +	Zachary Moffitt	8/2/2016			simpliCty: Compiler	
15886051729775	7/24/2016	7/28/2016	7/28/2016	Prepare Graham Demo + Group M		7/28/2016		talk print, structs	simpliCty: Compiler	
15886051729775	7/24/2016	8/1/2016	8/1/2016	Internal Due + Master Merge + Fi		7/30/2016		Demo: uop, bop, Discuss: structs	simpliCty: Compiler	
								Demo front-end: Discuss LLVM: a		



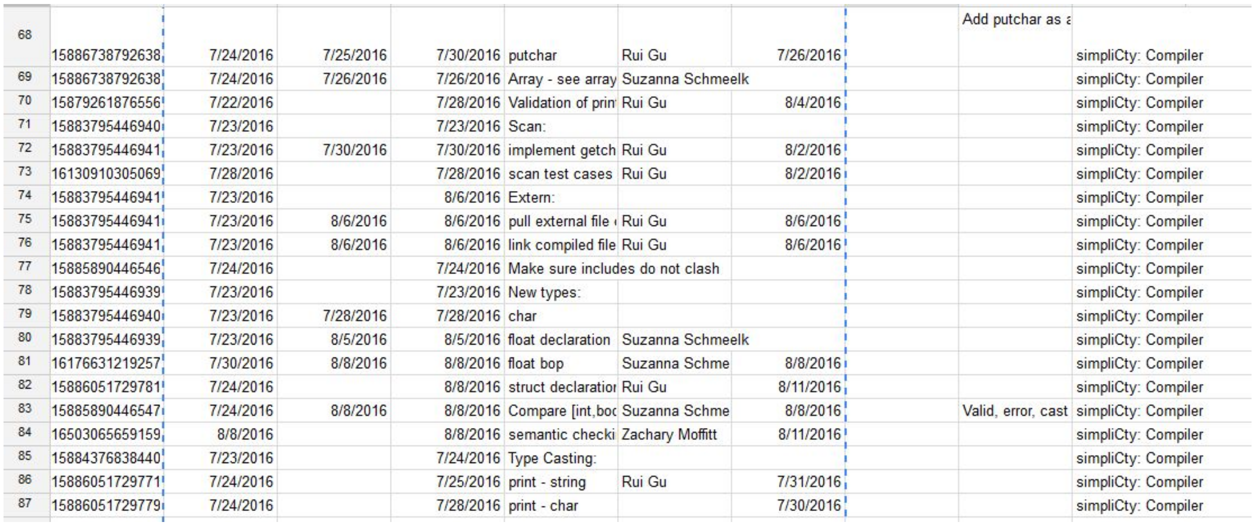
15886051729776	7/24/2016	8/1/2016	8/1/2016	Internal Due + Master Merge + Up		7/31/2016		Demo front-end: Discuss LLVM: a	simpliCty: Compiler
15886051729776	7/24/2016	8/3/2016	8/3/2016	Internal Due + M	Zachary Moffitt	8/2/2016		Demo Front-End structs, scan, ca	simpliCty: Compiler
15886051729776	7/24/2016	8/5/2016	8/5/2016	TA Meeting + Internal Discussion I		8/4/2016		LLVM Alignment	simpliCty: Compiler
15886051729776	7/24/2016	8/6/2016	8/6/2016	Internal Discussi	Zachary Moffitt	8/6/2016		Test case feedba	simpliCty: Compiler
15886051729777	7/24/2016	8/8/2016	8/8/2016	Internal Discussion Due Items + M		8/7/2016		Parser should be	simpliCty: Compiler
16065674264882	7/27/2016	8/11/2016	8/11/2016	Internal Demo +	Zachary Moffitt	8/9/2016			simpliCty: Compiler
16715989097351	8/11/2016		8/11/2016	Internal Demo +	Zachary Moffitt	8/16/2016			simpliCty: Compiler
15886051729777	7/24/2016	8/11/2016	8/11/2016	Prepare Demo + Report for Evenin		8/11/2016			simpliCty: Compiler
15879261876556	7/22/2016		7/23/2016	Project Scoping:		7/25/2016			simpliCty: Compiler
15884376838438	7/23/2016	7/24/2016	7/24/2016	Style Guide		7/23/2016		http://www.cs.co	simpliCty: Compiler
15879261876557	7/22/2016	7/25/2016	7/25/2016	Define team mer	Zachary Moffitt	7/24/2016		Manager - Timely Language Guru - System Architec Tester - Test plar	simpliCty: Compiler
15879261876557	7/22/2016	7/27/2016	7/27/2016	Assign functions and dev work		7/24/2016			simpliCty: Compiler
15879261876557	7/22/2016	7/25/2016	7/25/2016	Set deliverable d	Suzanna Schme	7/24/2016			simpliCty: Compiler



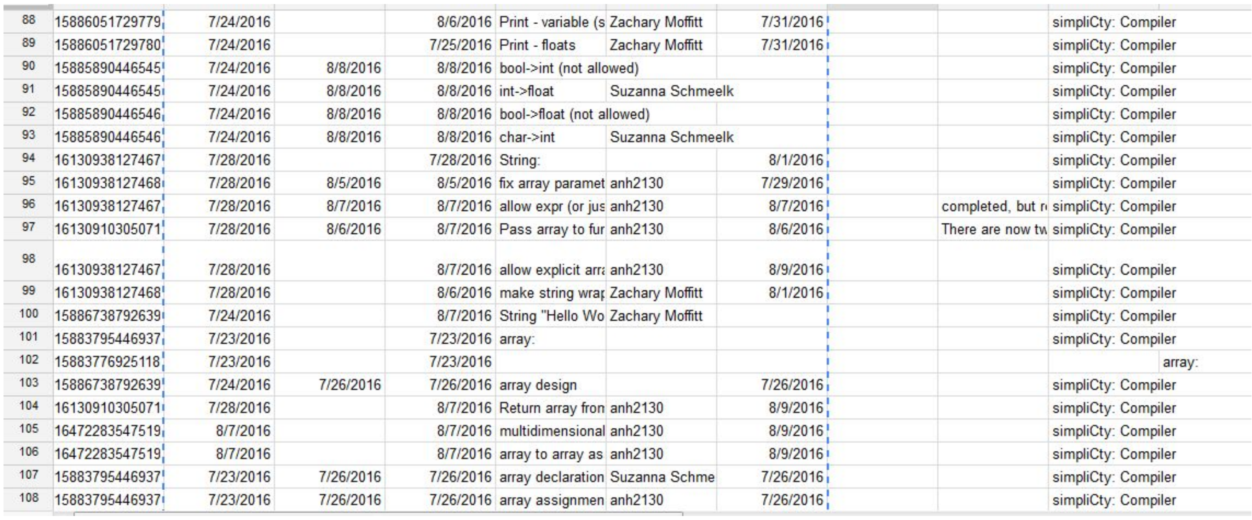
30	15879261876557	7/22/2016	7/25/2016	7/25/2016	Set deliverable d	Suzanna Schme	7/24/2016			simpliCty: Compiler
31	15886051729777	7/24/2016	7/24/2016	7/24/2016	Reformat microc	Suzanna Schme	7/24/2016			simpliCty: Compiler
32	15886051729781	7/24/2016	7/25/2016	7/25/2016	Lingering Enviror	Zachary Moffitt	7/25/2016			simpliCty: Compiler
33	15886051729770	7/24/2016		7/24/2016	Goals/Milestones (Authority Person):					simpliCty: Compiler
34	15886051729774	7/24/2016	7/24/2016	7/24/2016	Initial Test Cases	Suzanna Schme	7/24/2016			simpliCty: Compiler
35	15886738792637	7/24/2016	7/24/2016	7/24/2016	uop Test Cases	anh2130				simpliCty: Compiler
36	15886051729770	7/24/2016	7/28/2016	7/28/2016	Hello World	Zachary Moffitt	7/28/2016			simpliCty: Compiler
37	15996475847710	7/26/2016	8/5/2016	8/5/2016	passing arrays t	anh2130	7/30/2016			simpliCty: Compiler
38	15886051729771	7/24/2016	8/5/2016	8/5/2016	Validate bop initi	Zachary Moffitt	7/30/2016			simpliCty: Compiler
39	16176631219256	7/30/2016	8/8/2016	8/8/2016	bop on floats wai	Suzanna Schme	8/9/2016			simpliCty: Compiler
40	16176631219256	7/30/2016		7/30/2016	Validate bop on	Zachary Moffitt	8/5/2016			simpliCty: Compiler
41	15886051729771	7/24/2016		8/6/2016	struct declaratior	Rui Gu	8/7/2016			simpliCty: Compiler
42	15886738792636	7/24/2016		7/24/2016	Semantic error c	Zachary Moffitt				simpliCty: Compiler
43	15886051729771	7/24/2016		8/1/2016	scan statement i	Rui Gu	8/5/2016			simpliCty: Compiler
44	15886051729784	7/24/2016		8/1/2016	return statement	anh2130	8/5/2016			simpliCty: Compiler
45	15886051729784	7/24/2016	8/6/2016	8/6/2016	main functionality	Zachary Moffitt	7/31/2016			simpliCty: Compiler
46	15886051729772	7/24/2016	8/8/2016	8/8/2016	casting	Suzanna Schme	8/11/2016			simpliCty: Compiler
47	15886051729778	7/24/2016	8/6/2016	8/6/2016	"extern" keyword	Rui Gu	8/2/2016			simpliCty: Compiler
48	15886051729772	7/24/2016		7/24/2016	Parser Completed		8/7/2016			simpliCty: Compiler
49	15886051729773	7/24/2016		7/24/2016	Test Cases Pass Entirely		8/7/2016			simpliCty: Compiler
50	15884416455942	7/23/2016		7/23/2016	Interesting program:					simpliCty: Compiler
51	15884416455943	7/23/2016	8/1/2016	8/1/2016	write "interesting program" in the language "SimpliCty"					simpliCty: Compiler



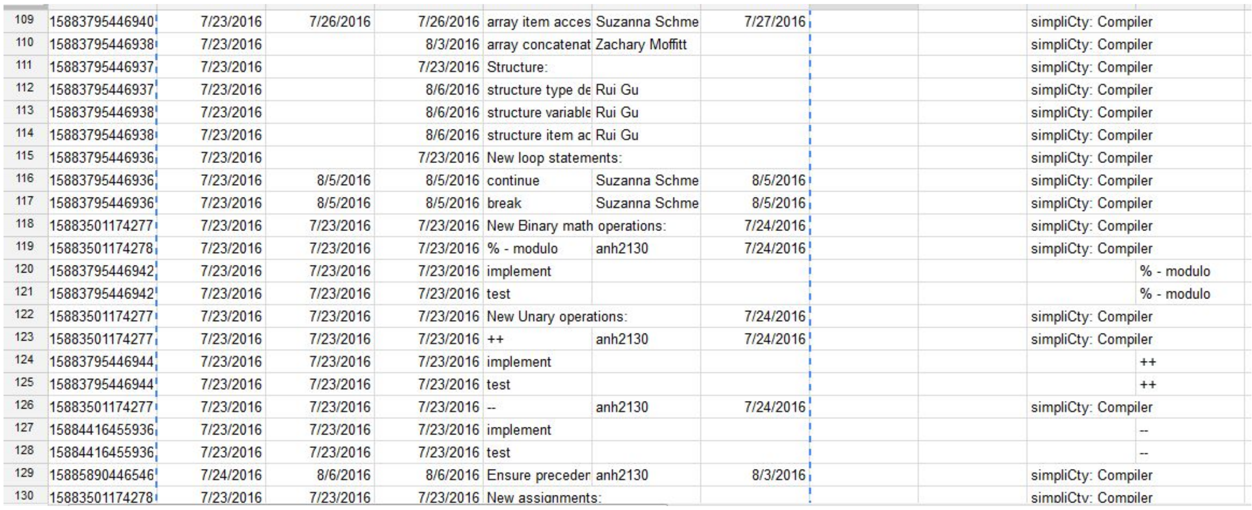
52	15884416455941	7/23/2016		7/23/2016	Testing:					simpliCty: Compiler	
53	15884416455942	7/23/2016	7/24/2016	7/24/2016	reformat microc.	anh2130	7/24/2016			simpliCty: Compiler	
54	15884416455942	7/23/2016	7/24/2016	7/24/2016	reformat current	Suzanna Schme	7/24/2016			simpliCty: Compiler	
55	15884416455942	7/23/2016	7/26/2016	7/26/2016	redesign testall.s	Suzanna Schme	7/26/2016			simpliCty: Compiler	
56	15884416455942	7/23/2016	7/26/2016	7/26/2016	redesign regress	Suzanna Schme	7/26/2016		1. Change error r 2. Make test fold 3. Change error l	simpliCty: Compiler	
57	15886738792635	7/24/2016	7/26/2016	7/26/2016	Makefile				Changes? 1. 2.	simpliCty: Compiler	
58	15879261876554	7/22/2016		7/23/2016	Parser:		7/24/2016			simpliCty: Compiler	
59	15879261876554	7/22/2016	7/26/2016	7/26/2016	Gather requirements for parser		7/24/2016			simpliCty: Compiler	
60	15879261876555	7/22/2016	7/28/2016	7/28/2016	Implement parser		7/24/2016			simpliCty: Compiler	
61	15879261876555	7/22/2016		7/23/2016	Print:		7/25/2016			simpliCty: Compiler	
62	15879261876555	7/22/2016	7/24/2016	7/24/2016	Gather requirements for print		7/24/2016		putchar	simpliCty: Compiler	
63	15884004364935	7/23/2016	7/24/2016	7/24/2016	print_string -> OCaml -> LLVM						Gather requirements for print
64	15879261876555	7/23/2016	7/26/2016	7/26/2016	Implementation Steps		7/24/2016			simpliCty: Compiler	
65	15884004364935	7/23/2016		7/23/2016	Static test						Implementation Steps
66	15884004364935	7/23/2016		7/23/2016	Dynamic variable printing						Implementation Steps
67	15886738792637	7/24/2016		7/24/2016	Print	Zachary Moffitt				simpliCty: Compiler	



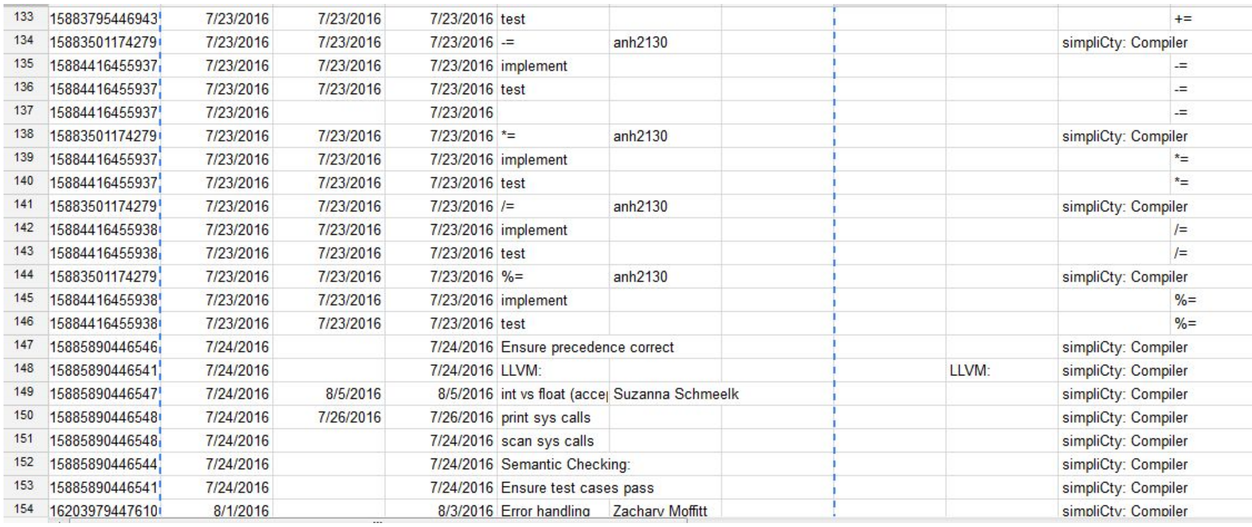
68	15886738792638	7/24/2016	7/25/2016	7/30/2016	putchar	Rui Gu	7/26/2016		Add putchar as ā	simpliCty: Compiler	
69	15886738792638	7/24/2016	7/26/2016	7/26/2016	Array - see array	Suzanna Schmeelk				simpliCty: Compiler	
70	15879261876556	7/22/2016		7/28/2016	Validation of prin	Rui Gu	8/4/2016			simpliCty: Compiler	
71	15883795446940	7/23/2016		7/23/2016	Scan:					simpliCty: Compiler	
72	15883795446941	7/23/2016	7/30/2016	7/30/2016	implement getch	Rui Gu	8/2/2016			simpliCty: Compiler	
73	16130910305069	7/28/2016		7/28/2016	scan test cases	Rui Gu	8/2/2016			simpliCty: Compiler	
74	15883795446941	7/23/2016		8/6/2016	Extern:					simpliCty: Compiler	
75	15883795446941	7/23/2016	8/6/2016	8/6/2016	pull external file (	Rui Gu	8/6/2016			simpliCty: Compiler	
76	15883795446941	7/23/2016	8/6/2016	8/6/2016	link compiled file	Rui Gu	8/6/2016			simpliCty: Compiler	
77	15885890446546	7/24/2016		7/24/2016	Make sure includes do not clash					simpliCty: Compiler	
78	15883795446939	7/23/2016		7/23/2016	New types:					simpliCty: Compiler	
79	15883795446940	7/23/2016	7/28/2016	7/28/2016	char					simpliCty: Compiler	
80	15883795446939	7/23/2016	8/5/2016	8/5/2016	float declaration	Suzanna Schmeelk				simpliCty: Compiler	
81	16176631219257	7/30/2016	8/8/2016	8/8/2016	float bop	Suzanna Schme	8/8/2016			simpliCty: Compiler	
82	15886051729781	7/24/2016		8/8/2016	struct declaratio	Rui Gu	8/11/2016			simpliCty: Compiler	
83	15885890446547	7/24/2016	8/8/2016	8/8/2016	Compare [int,boc	Suzanna Schme	8/8/2016		Valid, error, cast	simpliCty: Compiler	
84	16503065659159	8/8/2016		8/8/2016	semantic checki	Zachary Moffitt	8/11/2016			simpliCty: Compiler	
85	15884376838440	7/23/2016		7/24/2016	Type Casting:					simpliCty: Compiler	
86	15886051729771	7/24/2016		7/25/2016	print - string	Rui Gu	7/31/2016			simpliCty: Compiler	
87	15886051729779	7/24/2016		7/28/2016	print - char		7/30/2016			simpliCty: Compiler	



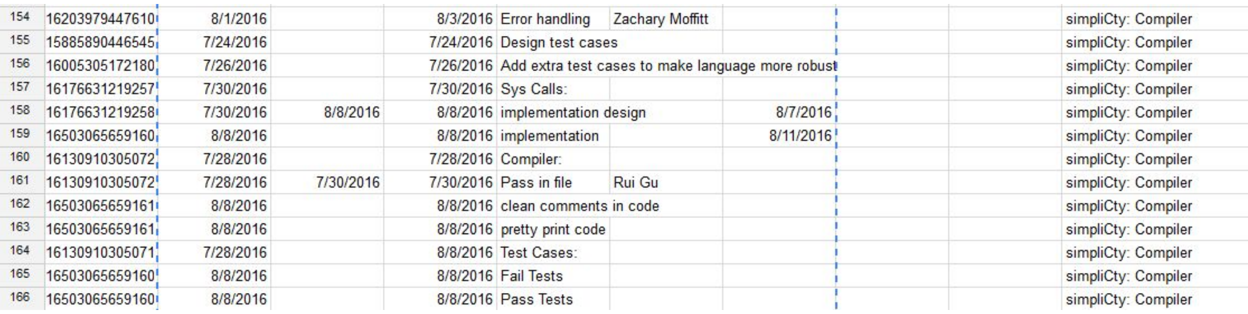
88	15886051729779	7/24/2016		8/6/2016	Print - variable (s	Zachary Moffitt	7/31/2016			simpliCty: Compiler	
89	15886051729780	7/24/2016		7/25/2016	Print - floats	Zachary Moffitt	7/31/2016			simpliCty: Compiler	
90	15885890446545	7/24/2016	8/8/2016	8/8/2016	bool->int (not allowed)					simpliCty: Compiler	
91	15885890446545	7/24/2016	8/8/2016	8/8/2016	int->float	Suzanna Schmeelk				simpliCty: Compiler	
92	15885890446546	7/24/2016	8/8/2016	8/8/2016	bool->float (not allowed)					simpliCty: Compiler	
93	15885890446546	7/24/2016	8/8/2016	8/8/2016	char->int	Suzanna Schmeelk				simpliCty: Compiler	
94	16130938127467	7/28/2016		7/28/2016	String:		8/1/2016			simpliCty: Compiler	
95	16130938127468	7/28/2016	8/5/2016	8/5/2016	fix array paramet	anh2130	7/29/2016			simpliCty: Compiler	
96	16130938127467	7/28/2016	8/7/2016	8/7/2016	allow expr (or jus	anh2130	8/7/2016		completed, but r	simpliCty: Compiler	
97	16130910305071	7/28/2016	8/6/2016	8/7/2016	Pass array to fur	anh2130	8/6/2016		There are now tw	simpliCty: Compiler	
98	16130938127467	7/28/2016		8/7/2016	allow explicit arra	anh2130	8/9/2016			simpliCty: Compiler	
99	16130938127468	7/28/2016		8/6/2016	make string wrap	Zachary Moffitt	8/1/2016			simpliCty: Compiler	
100	15886738792639	7/24/2016		8/7/2016	String "Hello Wo	Zachary Moffitt				simpliCty: Compiler	
101	15883795446937	7/23/2016		7/23/2016	array:					simpliCty: Compiler	
102	15883776925118	7/23/2016		7/23/2016							array:
103	15886738792639	7/24/2016	7/26/2016	7/26/2016	array design		7/26/2016			simpliCty: Compiler	
104	16130910305071	7/28/2016		8/7/2016	Return array fron	anh2130	8/9/2016			simpliCty: Compiler	
105	16472283547519	8/7/2016		8/7/2016	multidimensional	anh2130	8/9/2016			simpliCty: Compiler	
106	16472283547519	8/7/2016		8/7/2016	array to array as	anh2130	8/9/2016			simpliCty: Compiler	
107	15883795446937	7/23/2016	7/26/2016	7/26/2016	array declaration	Suzanna Schme	7/26/2016			simpliCty: Compiler	
108	15883795446937	7/23/2016	7/26/2016	7/26/2016	array assignmen	anh2130	7/26/2016			simpliCty: Compiler	



109	15883795446940	7/23/2016	7/26/2016	7/26/2016	array item acces	Suzanna Schme	7/27/2016			simpliCty: Compiler	
110	15883795446938	7/23/2016		8/3/2016	array concatenat	Zachary Moffitt				simpliCty: Compiler	
111	15883795446937	7/23/2016		7/23/2016	Structure:					simpliCty: Compiler	
112	15883795446937	7/23/2016		8/6/2016	structure type de	Rui Gu				simpliCty: Compiler	
113	15883795446938	7/23/2016		8/6/2016	structure variable	Rui Gu				simpliCty: Compiler	
114	15883795446938	7/23/2016		8/6/2016	structure item ac	Rui Gu				simpliCty: Compiler	
115	15883795446936	7/23/2016		7/23/2016	New loop statements:					simpliCty: Compiler	
116	15883795446936	7/23/2016	8/5/2016	8/5/2016	continue	Suzanna Schme	8/5/2016			simpliCty: Compiler	
117	15883795446936	7/23/2016	8/5/2016	8/5/2016	break	Suzanna Schme	8/5/2016			simpliCty: Compiler	
118	15883501174277	7/23/2016	7/23/2016	7/23/2016	New Binary math operations:		7/24/2016			simpliCty: Compiler	
119	15883501174278	7/23/2016	7/23/2016	7/23/2016	% - modulo	anh2130	7/24/2016			simpliCty: Compiler	
120	15883795446942	7/23/2016	7/23/2016	7/23/2016	implement						% - modulo
121	15883795446942	7/23/2016	7/23/2016	7/23/2016	test						% - modulo
122	15883501174277	7/23/2016	7/23/2016	7/23/2016	New Unary operations:		7/24/2016			simpliCty: Compiler	
123	15883501174277	7/23/2016	7/23/2016	7/23/2016	++	anh2130	7/24/2016			simpliCty: Compiler	
124	15883795446944	7/23/2016	7/23/2016	7/23/2016	implement						++
125	15883795446944	7/23/2016	7/23/2016	7/23/2016	test						++
126	15883501174277	7/23/2016	7/23/2016	7/23/2016	--	anh2130	7/24/2016			simpliCty: Compiler	
127	15884416455936	7/23/2016	7/23/2016	7/23/2016	implement						--
128	15884416455936	7/23/2016	7/23/2016	7/23/2016	test						--
129	15885890446546	7/24/2016	8/6/2016	8/6/2016	Ensure preceder	anh2130	8/3/2016			simpliCty: Compiler	
130	15883501174278	7/23/2016	7/23/2016	7/23/2016	New assignments:					simpliCty: Compiler	



133	15883795446943	7/23/2016	7/23/2016	7/23/2016	test						+=
134	15883501174279	7/23/2016	7/23/2016	7/23/2016	-=	anh2130				simpliCty: Compiler	
135	15884416455937	7/23/2016	7/23/2016	7/23/2016	implement						-=
136	15884416455937	7/23/2016	7/23/2016	7/23/2016	test						-=
137	15884416455937	7/23/2016		7/23/2016							-=
138	15883501174279	7/23/2016	7/23/2016	7/23/2016	*=	anh2130				simpliCty: Compiler	
139	15884416455937	7/23/2016	7/23/2016	7/23/2016	implement						*=
140	15884416455937	7/23/2016	7/23/2016	7/23/2016	test						*=
141	15883501174279	7/23/2016	7/23/2016	7/23/2016	/=	anh2130				simpliCty: Compiler	
142	15884416455938	7/23/2016	7/23/2016	7/23/2016	implement						/=
143	15884416455938	7/23/2016	7/23/2016	7/23/2016	test						/=
144	15883501174279	7/23/2016	7/23/2016	7/23/2016	%=	anh2130				simpliCty: Compiler	
145	15884416455938	7/23/2016	7/23/2016	7/23/2016	implement						%=
146	15884416455938	7/23/2016	7/23/2016	7/23/2016	test						%=
147	15885890446546	7/24/2016		7/24/2016	Ensure precedence correct					simpliCty: Compiler	
148	15885890446541	7/24/2016		7/24/2016	LLVM:				LLVM:	simpliCty: Compiler	
149	15885890446547	7/24/2016	8/5/2016	8/5/2016	int vs float (acce	Suzanna Schmeelk				simpliCty: Compiler	
150	15885890446548	7/24/2016	7/26/2016	7/26/2016	print sys calls					simpliCty: Compiler	
151	15885890446548	7/24/2016		7/24/2016	scan sys calls					simpliCty: Compiler	
152	15885890446544	7/24/2016		7/24/2016	Semantic Checking:					simpliCty: Compiler	
153	15885890446541	7/24/2016		7/24/2016	Ensure test cases pass					simpliCty: Compiler	
154	16203979447610	8/1/2016		8/3/2016	Error handling	Zachary Moffitt				simpliCty: Compiler	



154	16203979447610	8/1/2016		8/3/2016	Error handling	Zachary Moffitt				simpliCty: Compiler
155	15885890446545	7/24/2016		7/24/2016	Design test cases					simpliCty: Compiler
156	16005305172180	7/26/2016		7/26/2016	Add extra test cases to make language more robust					simpliCty: Compiler
157	16176631219257	7/30/2016		7/30/2016	Sys Calls:					simpliCty: Compiler
158	16176631219258	7/30/2016	8/8/2016	8/8/2016	implementation design		8/7/2016			simpliCty: Compiler
159	16503065659160	8/8/2016		8/8/2016	implementation		8/11/2016			simpliCty: Compiler
160	16130910305072	7/28/2016		7/28/2016	Compiler:					simpliCty: Compiler
161	16130910305072	7/28/2016	7/30/2016	7/30/2016	Pass in file	Rui Gu				simpliCty: Compiler
162	16503065659161	8/8/2016		8/8/2016	clean comments in code					simpliCty: Compiler
163	16503065659161	8/8/2016		8/8/2016	pretty print code					simpliCty: Compiler
164	16130910305071	7/28/2016		8/8/2016	Test Cases:					simpliCty: Compiler
165	16503065659160	8/8/2016		8/8/2016	Fail Tests					simpliCty: Compiler
166	16503065659160	8/8/2016		8/8/2016	Pass Tests					simpliCty: Compiler