PartialC

Final Report

Mingjie Lao (ml4545): System Architect, Tester

Jiaying Song (js5799): Manager, Language Guru

COMS W4115 Programming Languages and Translators
Spring 2021

Table of Contents

1 Introduction

PartialC is a general-purpose imperative language that implements part of the functions in C language. It is an extended version of MicroC and provides support for compound data types including string, array and struct.

It also implements the primitive data types such as integer, float number, boolean, as well as string. All common arithmetic operations (addition, subtraction, multiplication, division and modulo), logical operations, and control flows are supported. Some compound types like arrays are also implemented in PartialC.

With the functionalities provided by PartialC, many common algorithms like fibonacci and dynamic programming can be implemented with simplicity.

User-defined data type struct also provides a certain degree of flexibility in programming. It leans towards OOP and is scalable to become an OOP like c++.=

2 Language Reference Manual

2. 1 Data Types

The data types are defined as below:

Data Types	Description	Syntax
int	integer	int x = 1
float	real number	float x=2.3
bool	boolean	bool x = true
string	string	string x = 'abcde'
int/float/bool/string array	arr int, arr float, arr bool, array string	arr int x = [1,2,3]
void	return type when no value is returned in function	void functionName()

struct	User defined compound data type	struct Student {
		int sid;
		float grade;
		bool graduated;
		};

2.2 Lexical Conventions

2.2.1 Key word

Here is a list of tokens been reserved for PartialC:

Data type: int / float / bool / string / arr int / arr float / arr bool / arr string / void / struct
Data value: true / false / null
Control flow: if / else / for / while / return
Function: main(), sizeof(), printi(), printf(), prints()

2.2.2 Identifier

An identifier for data types (except struct name) is in lower snake case. In other words, a sequence of lower case letters or underscore is used to represent variable names. It should be defined before using and the data type should be defined explicitly as well.

Example: int **var_a** = 13

An identifier for data type struct should consist of lower case letters started with a single upper case alphabet.

Example: struct Student{
};

2.2.3 Literal

2.2.3.1 Integer

A sequence of digits 0-9 with optional - character to indicate the minus sign.

Example: 98, -4

2.2.3.2 Real number

A sequence of digits 0-9 with one period '.' in between to represent float number. It also has the optional - character to indicate the minus sign. The part before the period represents the integer part while the part after represents the fraction part.

Example: 2.3, -988.20

2.2.3.3 Logical literal

Both true and false are reserved keywords to represent logical value.

Example: true, false

2.2.3.4 String

An immutable sequence of characters surrounded by single quotes.

Example: 'abc'

2.2.3.5 Array

Array is a list of elements of the same data type. Elements can be accessed by its position index ranging from 0, **enclosed** in a square bracket. This immutable single dimension data structure supports int, float, bool, or string as elements. The elements are separated by comma.

Example: arr int a = **[1,2,3]**

2.2.3.6 Array Element

The specific element in the array is accessed by specifying array identifier and the index (start from 0).

Example: we first declare arr int a = [1, 2, 3], then we have **a[1]** = 2

2.2.4 Comment

Comment is any single-line sequence of characters that would be ignored by the language compiler. It is started with two consecutive front slashes //.

Example: // This is a comment.

2.2.5 White space

White space is ignored by the compiler automatically. Newline \n, Return \r, Tab \t, and space are all considered whitespace.

2.2.6 Separator

Separator type	Description
(	Left parenthesis used in expression and control flow conditions
)	Right parenthesis used in expression and control flow conditions
[	Left bracket for array index opening
]	Right bracket for array index closing
{	Left curly brace for control flow and function statement body
}	Right curly brace for control flow and function statement body
;	Semicolon to separate two expressions in control flow condition
,	Comma to separate array elements or arguments in function declaration

2.2.7 Operator

Operator	Description	Example
+	binary arithmetic addition (applied for int and float type)	1 + 5 1.0 + 5.2
-	binary arithmetic subtraction (applied for int and float type)	1 – 5 1.0 - 5.2
*	binary arithmetic multiplication (applied for int and float type)	1 * 5 1.0 * 5.2
/	binary arithmetic division (applied for int and float type)	1 / 5 1.0 / 5.2
%	binary arithmetic modulus (applied for int type only)	1 % 5
>	binary relational greater than (applied for int and float type)	a > 5
>=	binary relational greater than or equal to (applied for int and float type)	a >= 5
<	binary relational less than (applied for int and float type)	a < 5
<=	binary relational less than or equal (applied for int and float type)	a <= 5

<table>
<tr><td>==</td><td>binary relational equal (applied for int and float type)</td><td>a == 5</td></tr>
<tr><td>!=</td><td>binary relational not equal (applied for int and float type)</td><td>a != 5</td></tr>
<tr><td>&&</td><td>binary logical AND for boolean expression</td><td>m && n</td></tr>
<tr><td>||</td><td>binary logical OR for boolean expression</td><td>m || n</td></tr>
<tr><td>!</td><td>unary logical NOT for boolean expression</td><td>!m</td></tr>
</table>

2.2.8 Assignment

The equal sign = is used to perform the assignment of right-hand expression to left-hand side expression.

Example: a = 3 or a[1] = 4

2.3 Expression

Expression consists of the literals defined in Section 2.3. An expression also has an evaluated value corresponding to one of the data types defined in Section 2.1.

2.3.1 Basic expression

The expression can be a single literal defined in Section 2.2.3.
For example, any integer, float, bool, array, array element, string, or identifier.

2.3.2 Compound expression

The expression can also be composed of one of the following formats: expression BINARY_OPERATOR expression, UNARY_OPERATOR expression, (expression) or expression = expression. The type of expression value must be compatible with the operator (Compatibility is indicated in section 2.2.7.

2.3.3 Operator Precedence

Expressions are evaluated based on the following precedence (from high to low):

Operator	**Associativity**
()	left
[]	left
!	right
*, /, %	left
+, -	left

>, >=, <, <=	left
==, !=	left
&&, ||	left
=	right
, ;	left

2.4 Statement

There are five types of statements in PartialC: declaration, if/else, for, while and return. The statements are terminated by a semicolon.

2.4.1 Declaration with/without assignment

The declaration statement is used to define an identifier with data type. This statement can be used with or without assignment. The declaration statement can be implemented in any order within the statement body given the identifier is defined before being accessed in expression.

Example with assignment:

```
int a = 3;
arr int a = [1,1,1];
```

Example without assignment:

```
int a;
arr int a[3];
```

2.4.2 If/else

Similar to C language, this if/else control flow executes a block of code if a specified condition (logical expression) is true. If the condition is false, another block of code can be executed.
The two blocks of statement are compound statement lists that are enclosed by {} with no semicolon at the end. The structure is defined below:

```
if (logical expression) {
   statement1;
   statement2;
   ...
}else{
```

```
    statement3;
    statement4;
    ...
}
```

2.4.3 For

The for statement creates a loop that executes a block of statements as long as a termination condition is true. Before the iteration starts, the initialization statement is executed first and at the end of each iteration, the increment expression is executed. This block of statement is enclosed by {} with no semicolon at the end. The structure is defined below:

```
for (<initialization statement>; <logical termination expression>; <increment
expression>){
    statement1;
    statement2;
    ...
}
```

2.4.4 While

The while control flow creates a loop that executes a block of statements as long as the logical expression is evaluated as true.
This block of statement list is enclosed by {} with no semicolon at the end. The structure is defined below:

```
while ( <logical expression> ) {
    statement1;
    statement2;
    ...
}
```

2.4.5 Return

The return statement is used to define the value to be passed back from the current function. The data type returned must be consistent with the data type specified in the function declaration.

Example: return a;

2.5 Function

There are two types of functions in PartialC: built-in function and user-defined function.

2.5.1 Built-in function

2.5.1.1 sizeof()

The sizeof() function is used to return the length of the array identifier passed in. It is exclusively designed for arrays.

Example:

```
arr int a = [1,2,3];
int b = sizeof(a); // b has value of 3
```

2.5.1.2 prints()

The prints() function takes an expression of string value as input and prints it out. The input value can be an identifier or a raw string with single quotes.

Example:

```
prints(‘hello world’);
```

2.5.1.3 printi()

The printi() function takes an expression of integer value as input and prints it out.

Example:

```
printi(5);
printi(a+3);
```

2.5.1.4 printf()

The printi() function takes an expression of float value as input and prints it out.

Example:

```
printi(5.2);
printi(a+3.0);
```

2.5.2 User defined function

Users can define functions in PartialC.
The function takes a list of formals defined in the enclosed parenthesis (separated by comma) followed by a block of statements enclosed in {}. At the end of the statement body, the function may return an expression of the type as specified in the function prototype.

Similar to C, there must be a function with name main() which will be executed first.

An example of function declaration is given below:

```
ReturnDataType FunctionName (formal a, formal b, ...){
    statement1;
    statement2;
    …
    return expression;
}
```

3 Project Plan

3.1 Project process

3.1.1 Planning

As a two-member team, a regular meeting is scheduled every two weeks. The incremental development approach is adopted to ensure multiple milestones could be delivered in time.

3.1.2 Specification

The PartialC is inspired by the C language. The initial objective is to solve array related algorithms efficiently. During the proposal and LRM written stage, some basic features are finalized to support both primitive type and array operations. Along the development stage, some new features are introduced to make PartialC more complete and powerful.

3.1.3 Development

Development can be divided into two stages. During the first stage, the primitive version of scanner and parser are developed along with a . After the hello world demo was done, new features are added one by one

3.1.4 Test

Testing process is in parallel with the development process to ensure both backward compatibility when adding new functions. Over ?????? test cases are developed to improve the test coverage.

3.2 Software development tools

Hardware	MacBook Pro (13-inch, 2020, Four Thunderbolt 3 ports)
Operating system	macOS Catalina version 10.15.7 Ubuntu 18.04.1 LTS
Language and tool	OCaml version 4.05.0 GCC version (Ubuntu 7.3.0-27ubuntu1~18.04) 7.3.0 LLVM version 6.0.0 Docker version 19.03.13
Version control	Git (repo at https://github.com/songjytx/partialC)
Text editor	Sublime

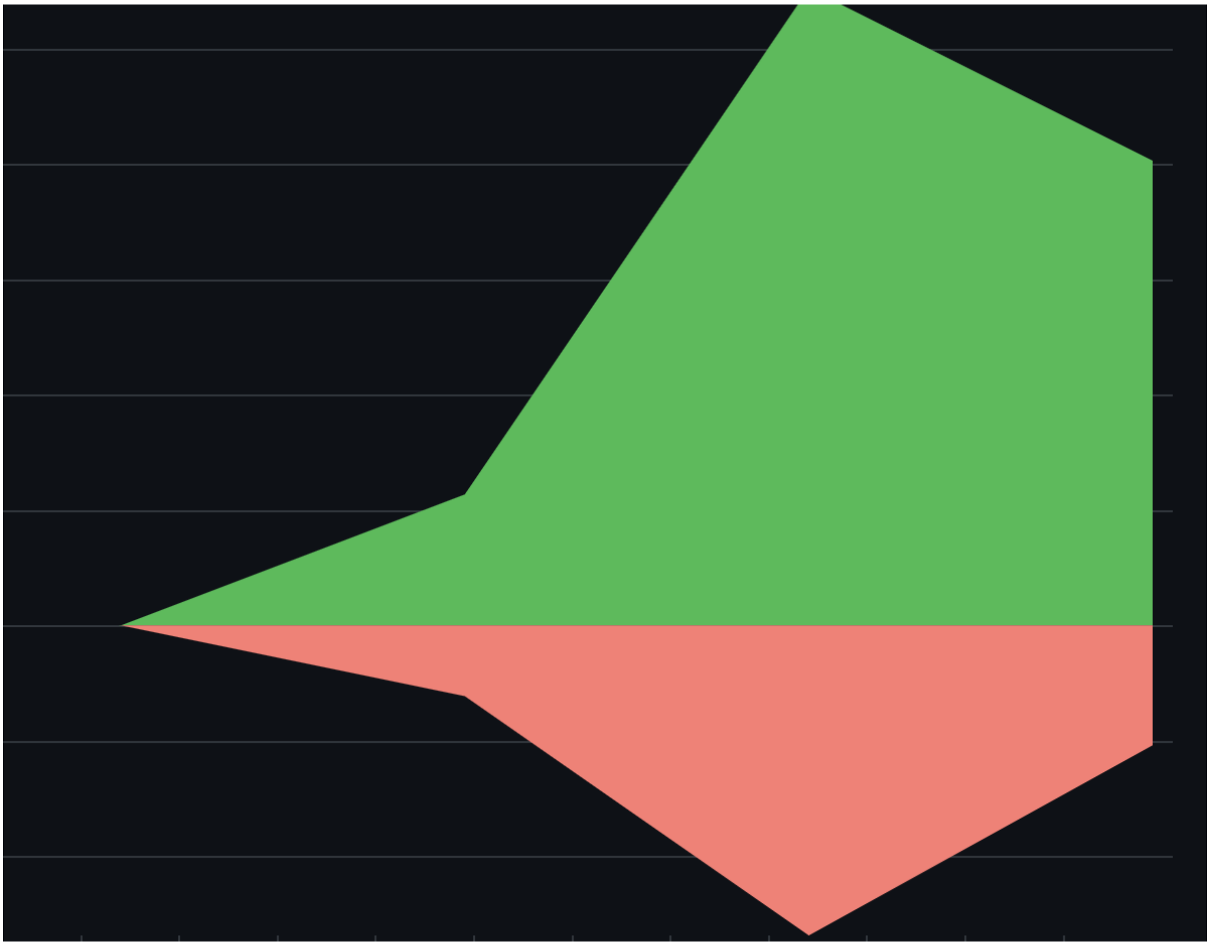
3.3 Project timeline

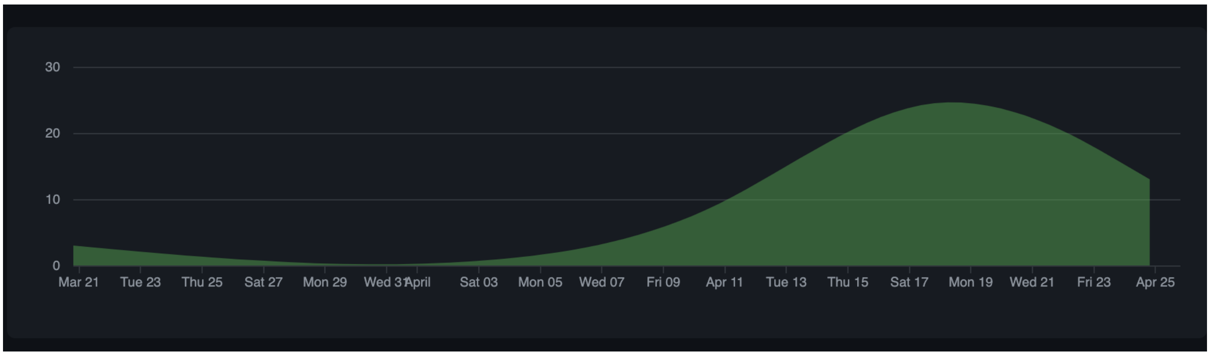
Feb 1st	Proposal discussion
Feb 3rd	Proposal submission
Feb 15th	Development environment setup
Feb 20th	LRM and parser discussion
Feb 24th	LRM and parser submission
Mar 18th	Scanner, parser, ast, sast, semant and codegen preliminary version
Mar 24th	Hello world demo
Apr 1st - Apr 25th	Iterative development and test
Apr 26th	Final presentation and report submission

3.4 Roles and responsibilities

Jiaying assumed the roles of Manager and Language Guru.
Mingjie assumed the roles of System Architect and Tester.
The final report is written in a joint effort.

3.5 Project log

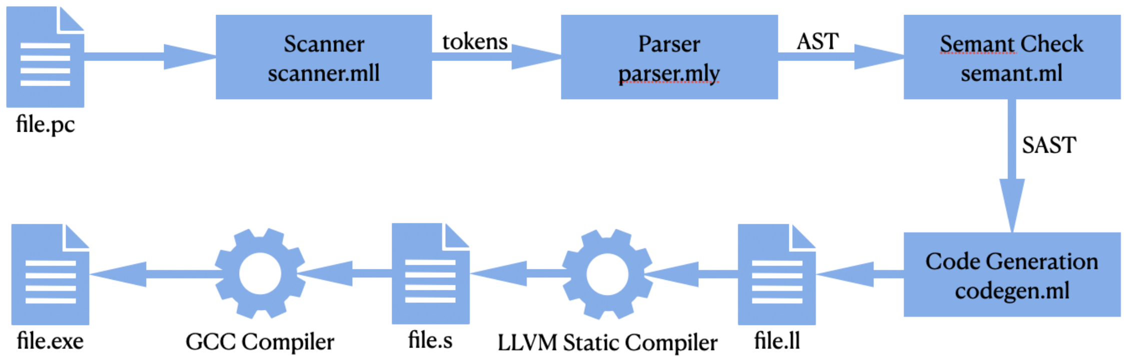




4 Architectural Design

4.1 Overview

The general architecture of compiler shows how the LLVM intermediate code if generated from a PartialC source file and finally compiled into the executable machine code using LLVM static compiler and GCC compiler:



The partialC source code file with extension .pc is firstly fed into the scanner and generates a sequence of tokens. Then the token sequence is consumed by the parser to generate the abstract syntax tree. AST is then evaluated by semant check file to append type on expression and execute semant check. Finally the generated SAST is passed to code generation to compose LLVM intermediate code. Executable machine code is generated based on this ll file.

4.2 Scanner

The Scanner takes as input a .pc source file and then creates tokens for identifiers, operators, keywords and different values. The most interesting part is to capture the keyword so that that particular keyword has special meaning for your compiler. You have great authority of defining the scanner in terms of whether an uppercase letter is allowed during scanning, etc,. The scanned tokens will then be fed into the parser to create a meaning.

4.3 Parser

The tokers generated from the scanner are then fed into the parder. An abstract syntax tree (AST), based on the grammar is generated. A token plays a crucial role in the AST, it directly defines the meaning of the code you write. Parser is especially important for the scalability of your compiler. The reason is simple, if you define your parser nicely, you can have a much shorter code that can capture much more grammar than the poor ones. Design parser wisely and it will save you plenty of time. The parsed AST must go through the semantic checking to resolve various semantic errors.

4.4 The Semantics Checker

The semantic checker converts the AST from the last step into a SAST which basically means an AST that has been semantically checked. The checking includes various type checking, assignment type checking, build-in function checking, logical operation type checking, duplicate variable or function name checking and much more. For our project the semantic iteratively traverses the Struct and function statements to check if there are any errors and report it before the potential error propagates to the next level. Semantic checker is fuzzy when you have a compound type because you need more work to store variable names in the map and do a thorough check. Semantic checked SAST is now safe to enter the code generator and generate llvm code.

4.5 The Code Generator

The Code Generator takes a SAST as an input and generates llvm code. Each SAST element has a corresponding llvm type and with careful implementation, the llvm code will define everything that is needed to generate machine code.

5 Test Plan

5.1 Test Automation

In the /test folder under root directory, two sets of test files are created for unit tests. The file started with test*.pc are for success cases and the file started with fail*.pc are for failure cases.

The automation test script testall.sh is borrowed from MicroC. This script compiles and runs all of the test*.pc files in the /test folder and compares the output to the corresponding *.out file.

5.2 Single file test

Following commands can be executed for generating executable machine code:

```
./partialc.native program_file_path.pc > program_file.ll
llc -relocation-model=pic program_file.ll
cc -o program_file.exe program_file.s
./program_file.exe
```

5.3 Sample test suite

5.3.1 Fibonacci Sequence

Here is the sample code for calculating fibonacci sequence.

```
void main(){
    int target = 9;
    arr int b[10];
    fib(b, target);
    printi(b[9]);
}

int fib(arr int f, int t){
    f[0] = 1;
    f[1] = 1;
    for(int i = 2; i <= t; i=i+1)
    {
       f[i] = f[i - 1] + f[i - 2];
    }
}
```

Generated LLVM code:

```
; ModuleID = 'PartialC'
source_filename = "PartialC"

@0 = private unnamed_addr constant [4 x i8] c"%s\0A\00"
@1 = private unnamed_addr constant [4 x i8] c"%d\0A\00"
@2 = private unnamed_addr constant [4 x i8] c"%f\0A\00"
@3 = private unnamed_addr constant [4 x i8] c"%s\0A\00"
@4 = private unnamed_addr constant [4 x i8] c"%d\0A\00"
@5 = private unnamed_addr constant [4 x i8] c"%f\0A\00"

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

declare { i8* } @strcat({ i8* }, { i8* })

define i32 @fib(i32 %t, { i32*, i32, i32 } %f) {
entry:
  %t1 = alloca i32
  store i32 %t, i32* %t1
  %f2 = alloca { i32*, i32, i32 }
  store { i32*, i32, i32 } %f, { i32*, i32, i32 }* %f2
  %0 = getelementptr inbounds { i32*, i32, i32 }, { i32*, i32, i32 }*
%f2, i32 0, i32 0
  %1 = load i32*, i32** %0
  %2 = getelementptr i32, i32* %1, i32 0
  store i32 1, i32* %2
  %3 = getelementptr inbounds { i32*, i32, i32 }, { i32*, i32, i32 }*
%f2, i32 0, i32 0
  %4 = load i32*, i32** %3
  %5 = getelementptr i32, i32* %4, i32 1
  store i32 1, i32* %5
  %i = alloca i32
  store i32 2, i32* %i
  br label %while

while:                                            ; preds = %while_body,
%entry
  %i10 = load i32, i32* %i
  %t11 = load i32, i32* %t1
  %"general op12" = icmp sle i32 %i10, %t11
  br i1 %"general op12", label %while_body, label %merge

while_body:                                       ; preds = %while
  %6 = getelementptr inbounds { i32*, i32, i32 }, { i32*, i32, i32 }*
%f2, i32 0, i32 0
  %7 = load i32*, i32** %6
```

```
  %i3 = load i32, i32* %i
  %"general op" = sub i32 %i3, 1
  %8 = getelementptr i32, i32* %7, i32 %"general op"
  %9 = load i32, i32* %8
  %10 = getelementptr inbounds { i32*, i32, i32 }, { i32*, i32, i32 }*
%f2, i32 0, i32 0
  %11 = load i32*, i32** %10
  %i4 = load i32, i32* %i
  %"general op5" = sub i32 %i4, 2
  %12 = getelementptr i32, i32* %11, i32 %"general op5"
  %13 = load i32, i32* %12
  %"general op6" = add i32 %9, %13
  %14 = getelementptr inbounds { i32*, i32, i32 }, { i32*, i32, i32 }*
%f2, i32 0, i32 0
  %15 = load i32*, i32** %14
  %i7 = load i32, i32* %i
  %16 = getelementptr i32, i32* %15, i32 %i7
  store i32 %"general op6", i32* %16
  %i8 = load i32, i32* %i
  %"general op9" = add i32 %i8, 1
  store i32 %"general op9", i32* %i
  br label %while

merge:                                            ; preds = %while
  ret i32 0
}

define void @main() {
entry:
  %target = alloca i32
  store i32 9, i32* %target
  %b = alloca { i32*, i32, i32 }
  %alloc = alloca { i32*, i32, i32 }
  %data_field_loc = getelementptr inbounds { i32*, i32, i32 }, { i32*,
i32, i32 }* %alloc, i32 0, i32 0
  %0 = getelementptr inbounds { i32*, i32, i32 }, { i32*, i32, i32 }*
%alloc, i32 0, i32 1
  %data_loc = alloca i32, i32 20
  %item_loc = getelementptr i32, i32* %data_loc, i32 0
  store i32 0, i32* %item_loc
  %item_loc1 = getelementptr i32, i32* %data_loc, i32 1
  store i32 0, i32* %item_loc1
  %item_loc2 = getelementptr i32, i32* %data_loc, i32 2
  store i32 0, i32* %item_loc2
  %item_loc3 = getelementptr i32, i32* %data_loc, i32 3
  store i32 0, i32* %item_loc3
```

```
  %item_loc4 = getelementptr i32, i32* %data_loc, i32 4
  store i32 0, i32* %item_loc4
  %item_loc5 = getelementptr i32, i32* %data_loc, i32 5
  store i32 0, i32* %item_loc5
  %item_loc6 = getelementptr i32, i32* %data_loc, i32 6
  store i32 0, i32* %item_loc6
  %item_loc7 = getelementptr i32, i32* %data_loc, i32 7
  store i32 0, i32* %item_loc7
  %item_loc8 = getelementptr i32, i32* %data_loc, i32 8
  store i32 0, i32* %item_loc8
  %item_loc9 = getelementptr i32, i32* %data_loc, i32 9
  store i32 0, i32* %item_loc9
  %item_loc10 = getelementptr i32, i32* %data_loc, i32 10
  store i32 0, i32* %item_loc10
  store i32* %data_loc, i32** %data_field_loc
  store i32 10, i32* %0
  %value = load { i32*, i32, i32 }, { i32*, i32, i32 }* %alloc
  store { i32*, i32, i32 } %value, { i32*, i32, i32 }* %b
  %b11 = load { i32*, i32, i32 }, { i32*, i32, i32 }* %b
  %target12 = load i32, i32* %target
  %fib_result = call i32 @fib(i32 %target12, { i32*, i32, i32 } %b11)
  %1 = getelementptr inbounds { i32*, i32, i32 }, { i32*, i32, i32 }*
%b, i32 0, i32 0
  %2 = load i32*, i32** %1
  %3 = getelementptr i32, i32* %2, i32 9
  %4 = load i32, i32* %3
  %printf = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([4 x
i8], [4 x i8]* @4, i32 0, i32 0), i32 %4)
  ret void
}
```

5.3.2 Coin Change based on DP

Here is the sample code for cracking the coin change problem based on dynamic programming. sizeof() function is utilized to return the size of the array.

```
// Coin Change
// Suppose we have coins of value 1, 2 and 5. Given a target of N,
return the fewest number of coins to make up N

// result[N] = min ({result[N - vi] + 1}) for N > 0 and vi = 1, 2, 5
// To make change for n cents, we are going to figure out how to make
change for every value x < n first.

void main(){
```

```
    arr int change = [1, 2, 5];

    arr int result[28]; //all values initialized to

    // mark result[1] result[2] and result[5] as 1.
    for(int i=0; i<=sizeof(change); i=i+1){
        result[change[i]] = 1;
    }

    for(int j=1; j<sizeof(result); j=j+1){
        for(int k=0; k<=sizeof(change); k=k+1){
            if(change[k] < j && (result[j] > (result[j- change[k]] +
result[change[k]]) || result[j] == 0)){
                result[j] = result[j-change[k]] + 1;
            }
        }
    }

    for(int y=1; y<sizeof(result); y=y+1){
        printi(result[y]);
    }
}
```

Generated LLVM code:

```
; ModuleID = 'PartialC'
source_filename = "PartialC"

@0 = private unnamed_addr constant [4 x i8] c"%s\0A\00"
@1 = private unnamed_addr constant [4 x i8] c"%d\0A\00"
@2 = private unnamed_addr constant [4 x i8] c"%f\0A\00"

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

declare { i8* } @strcat({ i8* }, { i8* })

define void @main() {
entry:
  %change = alloca { i32*, i32, i32 }
  %alloc = alloca { i32*, i32, i32 }
  %data_field_loc = getelementptr inbounds { i32*, i32, i32 }, { i32*,
i32, i32 }* %alloc, i32 0, i32 0
  %0 = getelementptr inbounds { i32*, i32, i32 }, { i32*, i32, i32 }*
%alloc, i32 0, i32 1
  %data_loc = alloca i32, i32 6
  %item_loc = getelementptr i32, i32* %data_loc, i32 0
```

```
  store i32 1, i32* %item_loc
  %item_loc1 = getelementptr i32, i32* %data_loc, i32 1
  store i32 2, i32* %item_loc1
  %item_loc2 = getelementptr i32, i32* %data_loc, i32 2
  store i32 5, i32* %item_loc2
  store i32* %data_loc, i32** %data_field_loc
  store i32 3, i32* %0
  %value = load { i32*, i32, i32 }, { i32*, i32, i32 }* %alloc
  store { i32*, i32, i32 } %value, { i32*, i32, i32 }* %change
  %result = alloca { i32*, i32, i32 }
  %alloc3 = alloca { i32*, i32, i32 }
  %data_field_loc4 = getelementptr inbounds { i32*, i32, i32 }, { i32*,
i32, i32 }* %alloc3, i32 0, i32 0
  %1 = getelementptr inbounds { i32*, i32, i32 }, { i32*, i32, i32 }*
%alloc3, i32 0, i32 1
  %data_loc5 = alloca i32, i32 56
  %item_loc6 = getelementptr i32, i32* %data_loc5, i32 0
  store i32 0, i32* %item_loc6
  %item_loc7 = getelementptr i32, i32* %data_loc5, i32 1
  store i32 0, i32* %item_loc7
  %item_loc8 = getelementptr i32, i32* %data_loc5, i32 2
  store i32 0, i32* %item_loc8
  %item_loc9 = getelementptr i32, i32* %data_loc5, i32 3
  store i32 0, i32* %item_loc9
  %item_loc10 = getelementptr i32, i32* %data_loc5, i32 4
  store i32 0, i32* %item_loc10
  %item_loc11 = getelementptr i32, i32* %data_loc5, i32 5
  store i32 0, i32* %item_loc11
  %item_loc12 = getelementptr i32, i32* %data_loc5, i32 6
  store i32 0, i32* %item_loc12
  %item_loc13 = getelementptr i32, i32* %data_loc5, i32 7
  store i32 0, i32* %item_loc13
  %item_loc14 = getelementptr i32, i32* %data_loc5, i32 8
  store i32 0, i32* %item_loc14
  %item_loc15 = getelementptr i32, i32* %data_loc5, i32 9
  store i32 0, i32* %item_loc15
  %item_loc16 = getelementptr i32, i32* %data_loc5, i32 10
  store i32 0, i32* %item_loc16
  %item_loc17 = getelementptr i32, i32* %data_loc5, i32 11
  store i32 0, i32* %item_loc17
  %item_loc18 = getelementptr i32, i32* %data_loc5, i32 12
  store i32 0, i32* %item_loc18
  %item_loc19 = getelementptr i32, i32* %data_loc5, i32 13
  store i32 0, i32* %item_loc19
  %item_loc20 = getelementptr i32, i32* %data_loc5, i32 14
  store i32 0, i32* %item_loc20
```

```
  %item_loc21 = getelementptr i32, i32* %data_loc5, i32 15
  store i32 0, i32* %item_loc21
  %item_loc22 = getelementptr i32, i32* %data_loc5, i32 16
  store i32 0, i32* %item_loc22
  %item_loc23 = getelementptr i32, i32* %data_loc5, i32 17
  store i32 0, i32* %item_loc23
  %item_loc24 = getelementptr i32, i32* %data_loc5, i32 18
  store i32 0, i32* %item_loc24
  %item_loc25 = getelementptr i32, i32* %data_loc5, i32 19
  store i32 0, i32* %item_loc25
  %item_loc26 = getelementptr i32, i32* %data_loc5, i32 20
  store i32 0, i32* %item_loc26
  %item_loc27 = getelementptr i32, i32* %data_loc5, i32 21
  store i32 0, i32* %item_loc27
  %item_loc28 = getelementptr i32, i32* %data_loc5, i32 22
  store i32 0, i32* %item_loc28
  %item_loc29 = getelementptr i32, i32* %data_loc5, i32 23
  store i32 0, i32* %item_loc29
  %item_loc30 = getelementptr i32, i32* %data_loc5, i32 24
  store i32 0, i32* %item_loc30
  %item_loc31 = getelementptr i32, i32* %data_loc5, i32 25
  store i32 0, i32* %item_loc31
  %item_loc32 = getelementptr i32, i32* %data_loc5, i32 26
  store i32 0, i32* %item_loc32
  %item_loc33 = getelementptr i32, i32* %data_loc5, i32 27
  store i32 0, i32* %item_loc33
  %item_loc34 = getelementptr i32, i32* %data_loc5, i32 28
  store i32 0, i32* %item_loc34
  store i32* %data_loc5, i32** %data_field_loc4
  store i32 28, i32* %1
  %value35 = load { i32*, i32, i32 }, { i32*, i32, i32 }* %alloc3
  store { i32*, i32, i32 } %value35, { i32*, i32, i32 }* %result
  %i = alloca i32
  store i32 0, i32* %i
  br label %while

while:                                            ; preds = %while_body,
%entry
  %i38 = load i32, i32* %i
  %2 = getelementptr inbounds { i32*, i32, i32 }, { i32*, i32, i32 }*
%change, i32 0, i32 1
  %3 = load i32, i32* %2
  %"general op39" = icmp sle i32 %i38, %3
  br i1 %"general op39", label %while_body, label %merge

while_body:                                       ; preds = %while
```

```
  %4 = getelementptr inbounds { i32*, i32, i32 }, { i32*, i32, i32 }*
%result, i32 0, i32 0
  %5 = load i32*, i32** %4
  %6 = getelementptr inbounds { i32*, i32, i32 }, { i32*, i32, i32 }*
%change, i32 0, i32 0
  %7 = load i32*, i32** %6
  %i36 = load i32, i32* %i
  %8 = getelementptr i32, i32* %7, i32 %i36
  %9 = load i32, i32* %8
  %10 = getelementptr i32, i32* %5, i32 %9
  store i32 1, i32* %10
  %i37 = load i32, i32* %i
  %"general op" = add i32 %i37, 1
  store i32 %"general op", i32* %i
  br label %while

merge:                                            ; preds = %while
  %j = alloca i32
  store i32 1, i32* %j
  br label %while40

while40:                                          ; preds = %merge68,
%merge
  %j71 = load i32, i32* %j
  %11 = getelementptr inbounds { i32*, i32, i32 }, { i32*, i32, i32 }*
%result, i32 0, i32 1
  %12 = load i32, i32* %11
  %"general op72" = icmp slt i32 %j71, %12
  br i1 %"general op72", label %while_body41, label %merge73

while_body41:                                     ; preds = %while40
  %k = alloca i32
  store i32 0, i32* %k
  br label %while42

while42:                                          ; preds = %merge58,
%while_body41
  %k66 = load i32, i32* %k
  %13 = getelementptr inbounds { i32*, i32, i32 }, { i32*, i32, i32 }*
%change, i32 0, i32 1
  %14 = load i32, i32* %13
  %"general op67" = icmp sle i32 %k66, %14
  br i1 %"general op67", label %while_body43, label %merge68

while_body43:                                     ; preds = %while42
  %15 = getelementptr inbounds { i32*, i32, i32 }, { i32*, i32, i32 }*
```

```
%change, i32 0, i32 0
  %16 = load i32*, i32** %15
  %k44 = load i32, i32* %k
  %17 = getelementptr i32, i32* %16, i32 %k44
  %18 = load i32, i32* %17
  %j45 = load i32, i32* %j
  %"general op46" = icmp slt i32 %18, %j45
  %19 = getelementptr inbounds { i32*, i32, i32 }, { i32*, i32, i32 }*
%result, i32 0, i32 0
  %20 = load i32*, i32** %19
  %j47 = load i32, i32* %j
  %21 = getelementptr i32, i32* %20, i32 %j47
  %22 = load i32, i32* %21
  %23 = getelementptr inbounds { i32*, i32, i32 }, { i32*, i32, i32 }*
%result, i32 0, i32 0
  %24 = load i32*, i32** %23
  %j48 = load i32, i32* %j
  %25 = getelementptr inbounds { i32*, i32, i32 }, { i32*, i32, i32 }*
%change, i32 0, i32 0
  %26 = load i32*, i32** %25
  %k49 = load i32, i32* %k
  %27 = getelementptr i32, i32* %26, i32 %k49
  %28 = load i32, i32* %27
  %"general op50" = sub i32 %j48, %28
  %29 = getelementptr i32, i32* %24, i32 %"general op50"
  %30 = load i32, i32* %29
  %31 = getelementptr inbounds { i32*, i32, i32 }, { i32*, i32, i32 }*
%result, i32 0, i32 0
  %32 = load i32*, i32** %31
  %33 = getelementptr inbounds { i32*, i32, i32 }, { i32*, i32, i32 }*
%change, i32 0, i32 0
  %34 = load i32*, i32** %33
  %k51 = load i32, i32* %k
  %35 = getelementptr i32, i32* %34, i32 %k51
  %36 = load i32, i32* %35
  %37 = getelementptr i32, i32* %32, i32 %36
  %38 = load i32, i32* %37
  %"general op52" = add i32 %30, %38
  %"general op53" = icmp sgt i32 %22, %"general op52"
  %39 = getelementptr inbounds { i32*, i32, i32 }, { i32*, i32, i32 }*
%result, i32 0, i32 0
  %40 = load i32*, i32** %39
  %j54 = load i32, i32* %j
  %41 = getelementptr i32, i32* %40, i32 %j54
  %42 = load i32, i32* %41
  %"general op55" = icmp eq i32 %42, 0
```

```
  %"general op56" = or i1 %"general op53", %"general op55"
  %"general op57" = and i1 %"general op46", %"general op56"
  br i1 %"general op57", label %then, label %else

merge58:                                          ; preds = %else, %then
  %k64 = load i32, i32* %k
  %"general op65" = add i32 %k64, 1
  store i32 %"general op65", i32* %k
  br label %while42

then:                                             ; preds =
%while_body43
  %43 = getelementptr inbounds { i32*, i32, i32 }, { i32*, i32, i32 }*
%result, i32 0, i32 0
  %44 = load i32*, i32** %43
  %j59 = load i32, i32* %j
  %45 = getelementptr inbounds { i32*, i32, i32 }, { i32*, i32, i32 }*
%change, i32 0, i32 0
  %46 = load i32*, i32** %45
  %k60 = load i32, i32* %k
  %47 = getelementptr i32, i32* %46, i32 %k60
  %48 = load i32, i32* %47
  %"general op61" = sub i32 %j59, %48
  %49 = getelementptr i32, i32* %44, i32 %"general op61"
  %50 = load i32, i32* %49
  %"general op62" = add i32 %50, 1
  %51 = getelementptr inbounds { i32*, i32, i32 }, { i32*, i32, i32 }*
%result, i32 0, i32 0
  %52 = load i32*, i32** %51
  %j63 = load i32, i32* %j
  %53 = getelementptr i32, i32* %52, i32 %j63
  store i32 %"general op62", i32* %53
  br label %merge58

else:                                             ; preds =
%while_body43
  br label %merge58

merge68:                                          ; preds = %while42
  %j69 = load i32, i32* %j
  %"general op70" = add i32 %j69, 1
  store i32 %"general op70", i32* %j
  br label %while40

merge73:                                          ; preds = %while40
  %y = alloca i32
```

```
  store i32 1, i32* %y
  br label %while74

while74:                                          ; preds =
%while_body75, %merge73
  %y79 = load i32, i32* %y
  %54 = getelementptr inbounds { i32*, i32, i32 }, { i32*, i32, i32 }*
%result, i32 0, i32 1
  %55 = load i32, i32* %54
  %"general op80" = icmp slt i32 %y79, %55
  br i1 %"general op80", label %while_body75, label %merge81

while_body75:                                     ; preds = %while74
  %56 = getelementptr inbounds { i32*, i32, i32 }, { i32*, i32, i32 }*
%result, i32 0, i32 0
  %57 = load i32*, i32** %56
  %y76 = load i32, i32* %y
  %58 = getelementptr i32, i32* %57, i32 %y76
  %59 = load i32, i32* %58
  %printf = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([4 x
i8], [4 x i8]* @1, i32 0, i32 0), i32 %59)
  %y77 = load i32, i32* %y
  %"general op78" = add i32 %y77, 1
  store i32 %"general op78", i32* %y
  br label %while74

merge81:                                          ; preds = %while74
  ret void
}
```

5.3.2 Struct

Here is the sample code for defining struct and assigning values to its members.

```
struct Test{
      int a;
      float b;
      bool c;
};

void main(){
      Test x;
      x.a = 1;
      x.b = 4.5;
      x.c = true;

      printi(x.a);
      printf(x.b);
      if (x.c){
          prints('Success!');
      }
}
```

Generated LLVM code:

```
; ModuleID = 'PartialC'
source_filename = "PartialC"

@0 = private unnamed_addr constant [4 x i8] c"%s\0A\00"
@1 = private unnamed_addr constant [4 x i8] c"%d\0A\00"
@2 = private unnamed_addr constant [4 x i8] c"%f\0A\00"

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

define void @main() {
entry:
  %x = alloca { i32, double, i1 }
  %struct_p = getelementptr inbounds { i32, double, i1 }, { i32, double,
i1 }* %x, i32 0, i32 0
  store i32 1, i32* %struct_p
  %struct_p1 = getelementptr inbounds { i32, double, i1 }, { i32,
double, i1 }* %x, i32 0, i32 1
  store double 4.500000e+00, double* %struct_p1
  %struct_p2 = getelementptr inbounds { i32, double, i1 }, { i32,
```

```
double, i1 }* %x, i32 0, i32 2
  store i1 true, i1* %struct_p2
  %struct_p3 = getelementptr inbounds { i32, double, i1 }, { i32,
double, i1 }* %x, i32 0, i32 0
  %member_v = load i32, i32* %struct_p3
  %printf = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([4 x
i8], [4 x i8]* @1, i32 0, i32 0), i32 %member_v)
  %struct_p4 = getelementptr inbounds { i32, double, i1 }, { i32,
double, i1 }* %x, i32 0, i32 1
  %member_v5 = load double, double* %struct_p4
  %printf6 = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([4
x i8], [4 x i8]* @2, i32 0, i32 0), double %member_v5)
  ret void
}
```

6 Lessons Learned

Mingjie: Start the project early because you will need a lot of time to design and debug. Some small features may take you hours to debug. Some features like compound types are tricky, you need to be very familiar with LLVM code in order to generate proper executables.

Ocaml is a fun language to use and learn. It is the first functional programming language that I have ever learned. It takes time to think in the way ocaml usually works. Once you are familiar with ocaml, you would think this is the perfect language to create compilers and you will be amazed that it works perfectly.

Semantic check is always my headache because every new feature you add must go through semantic check in order to generate the proper abstract semantic tree. It involves a lot of fuzzy checking when I create compound types like struct. But it is worth the time because you can implement some interesting features at that layer, for instance, out of bound check.

Last but not least, collaborate with your teammate, no matter how good you are. You might have a great idea but it also means a hell amount of work. It is always good to work with your teammates and you will learn a lot.

Jiaying: Parsing rules will determine the scalability of your compiler. There are many times that we found the parser needs to be refined in order to add some new features. Design it wisely! Semant check is really fuzzy and time consuming, however, it helps a lot to improve the compiler quality.

7 Future Improvement

a. Support array as struct member type.
b. Add min() and max() function to improve efficiency in dynamic programming coding
c. Support pointer to struct
d. Add dictionaries

Appendix A -- Compiler code

Scanner.mll

```
{ open Parser }
let digit = ['0'-'9']

rule tokenize = parse
  [' ' '\t' '\r' '\n'] { tokenize lexbuf }
| "//"              { comment lexbuf }
| '.'      { DOT }
| '+'               { PLUS }
| '-'               { MINUS }
| '*'               { TIMES }
| '/'               { DIVIDE }
| '%'               { MOD }
| '>'               { GT }
| ">="              { GEQ }
| '<'               { LT }
| "<="              { LEQ }
| "=="              { EQ }
| "!="              { NEQ }
| "&&"              { AND }
| "||"              { OR }
| '!'               { NOT }
| '='               { ASSIGN }
| ';'               { SEMI }
| ','               { COMMA }
| '{'               { LBRACE }
| '}'               { RBRACE }
| '('               { LPAREN }
| ')'               { RPAREN }
| '['               { LBRACKET }
| ']'               { RBRACKET }
| "int"             { INT }
| "float"   { FLOAT }
| "bool"    { BOOL }
| "string"  { STRING }
| "void"    { VOID }
| "arr"     { ARRAY}
| "true"    { BOOL_L(true) }
| "false"   { BOOL_L(false) }
| "null"    { NULL }
| "struct"  { STRUCT }
| "if"      { IF }
```

```
| "else"    { ELSE }
| "for"     { FOR }
| "while"   { WHILE }
| "return"  { RETURN }
| ['-']?digit+ as lxm { INT_L(int_of_string lxm) }
| ['-']?digit+['.']digit+ as lxm {FLOAT_L(float_of_string lxm)}
| "\'" [^''']+ "\'" as lxm { STRING_L(lxm) }
| ['a'-'z''_' ]+ as lxm { ID(lxm) }
| ['A'-'Z']['a'-'z' 'A'-'Z']* as structLit { STRUCT_ID(structLit) }
| eof { EOF }
| _ as ch { raise (Failure("illegal character detected " ^ Char.escaped
ch)) }

and comment = parse
  '\n' { tokenize lexbuf }
| _    { comment lexbuf }
```

Parser.mly

```
%{ open Ast %}

%token LBRACE RBRACE
%token LPAREN RPAREN
%token LBRACKET RBRACKET
%token NOT NEGATE
%token DOT
%token TIMES DIVIDE MOD
%token PLUS MINUS
%token GEQ GT LEQ LT
%token EQ NEQ
%token AND OR
%token IF ELSE
%token FOR WHILE RETURN PRINT

%token ASSIGN
%token SEMI
%token COMMA
%token MAIN
%token TRUE
%token FALSE
%token NULL
%token INT BOOL STRING FLOAT VOID
%token ARRAY
%token STRUCT
%token <int> INT_L
```

```
%token <float> FLOAT_L
%token <bool> BOOL_L
%token <string> ID STRING_L
%token <string> STRUCT_ID
%token RETURN
%token EOF

%left SEMI
%right ASSIGN
%left AND OR
%left EQ NEQ
%left GEQ GT LEQ LT
%left PLUS MINUS
%left TIMES DIVIDE MOD
%right NOT
%left LBRACKET RBRACKET
%left LPAREN RPAREN

%start program
%type <Ast.program> program

%%

program:
  { [], [] }
| program struct_decl { ($2 :: fst $1), snd $1 }
| program func_decl   { fst $1, ($2 :: snd $1) }


struct_decl:
  STRUCT STRUCT_ID LBRACE member_list RBRACE SEMI
  { { sname = $2;
      members = List.rev $4;} }

func_decl:
  dtype ID LPAREN formals_opt RPAREN LBRACE stmt_list RBRACE
  { { typ = $1;
    fname = $2;
    formals = $4;
    fstmts = List.rev $7 } }

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

formal_list:
```

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

member_list:
    /* nothing */ { [] }
  | member_list member   { $2 :: $1 }

member:
  dtype ID SEMI  {($1,$2) }

vname:
    ID {Id($1)}

/*array:
  ID LBRACKET INT_L RBRACKET {ArrayIndex(Id($1), IntLit($3))}*/

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

stmt:
  expr SEMI  {Expr $1}
| dtype ID ASSIGN expr SEMI { VarDecl($1, $2, $4) }
| dtype ID SEMI { VarDecl($1, $2, Noexpr($1)) }
| IF LPAREN expr RPAREN stmt ELSE stmt  { If($3, $5, $7) }
| IF LPAREN expr RPAREN stmt { If($3, $5, Block([])) }
| FOR LPAREN stmt expr SEMI expr RPAREN stmt { For($3, $4, $6, $8) }
| WHILE LPAREN expr RPAREN stmt { While($3, $5) }
| RETURN expr SEMI { Return($2) }
| LBRACE stmt_list RBRACE { Block(List.rev $2)}
| dtype ID LBRACKET expr RBRACKET SEMI{ ArrayDecl($1, $2, $4,
Noexpr($1)) }

expr:
    LPAREN expr RPAREN { $2 }
  | expr PLUS  expr { Binop($1, Add, $3) }
  | expr MINUS  expr { Binop($1, Sub, $3) }
  | expr TIMES  expr { Binop($1, Mul, $3) }
  | expr DIVIDE expr { Binop($1, Div, $3) }
  | expr MOD expr { Binop($1, Mod, $3) }
  | expr EQ  expr { Binop($1, Eq, $3) }
  | expr NEQ  expr { Binop($1, Neq, $3) }
  | expr GEQ  expr { Binop($1, Geq, $3) }
  | expr GT  expr { Binop($1, Gt, $3) }
  | expr LEQ  expr { Binop($1, Leq, $3) }
  | expr LT  expr { Binop($1, Lt, $3) }
```

```
  | expr AND  expr { Binop($1, And, $3) }
  | expr OR  expr { Binop($1, Or, $3) }
  | NOT expr           { Not($2) }
  | STRING_L           { StringLit($1) }
  | FLOAT_L           { FloatLit($1) }
  | INT_L             { IntLit($1)  }
  | BOOL_L            { BoolLit($1)  }
  | ID               { Id($1) }
  | LBRACKET array_opt RBRACKET          { ArrayLit(List.rev $2) }
  | ID LPAREN args_opt RPAREN { Call($1, $3)  }
  | vname ASSIGN expr {AssignOp($1, $3)}
  | ID LBRACKET expr RBRACKET ASSIGN expr {ArrayAssignOp(Id($1), $3,
$6)}
  | ID LBRACKET expr RBRACKET {ArrayIndex(Id($1), $3)}
  | ID DOT ID {StructAccess(Id($1), Id($3))}
  | ID DOT ID ASSIGN expr {StructAssignOp(Id($1), Id($3), $5)}

args_opt:
    /* nothing */ { [] }
  | args_list  { $1 }

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

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

dtype:
  | INT { Int }
  | FLOAT { Float }
  | BOOL { Bool }
  | STRING { String }
  | VOID { Void }
  | ARRAY dtype { Array($2) }
  | STRUCT_ID {Struct($1)}
```

ast.ml

```
type typ = Int | Float | Bool | Void | String | Array of typ |  Struct
of string
type operator = Add | Sub | Mul | Div | Mod | Sep | Eq | Neq | Lt | Leq
```

```
| Gt | Geq | And | Or
type assignment = Assign

type expr =

    Binop of expr * operator * expr
  | Not of expr
  | AssignOp of expr * expr
  | ArrayAssignOp of expr * expr * expr
  | Var of string
  | StringLit of string
  | FloatLit of float
  | IntLit of int
  | BoolLit of bool
  | Id of string
  | Call of string * expr list
  | ArrayLit of expr list
  | ArrayIndex of expr * expr
  | StructAccess of expr * expr
  | StructAssignOp of expr * expr * expr
  | Noexpr of typ

type bind = typ * string

type stmt =
        Block of stmt list
      | VarDecl of typ * string * expr
  | ArrayDecl of typ * string * expr * expr
      | If of expr * stmt * stmt
      | For of stmt * expr * expr * stmt
      | While of expr * stmt
      | Print of expr
      | Return of expr
  | Expr of expr

type struct_decl = {
    sname: string;
    members: bind list;
  }

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

```
type program = struct_decl list * func_decl list

let rec string_of_typ = function
    Int -> "int"
  | Bool -> "bool"
  | Float -> "float"
  | Void -> "void"
  | String -> "string"
  | Array(t) -> string_of_typ(t) ^ " array"
  | Struct(t) -> t


let string_of_op = function
    Add -> "+"
  | Sub -> "-"
  | Mul -> "*"
  | Div -> "/"
  | Eq -> "=="
  | Neq -> "!="
  | Lt -> "<"
  | Leq -> "<="
  | Gt -> ">"
  | Geq -> ">="
  | And -> "&&"
  | Or -> "||"

let rec string_of_expr = function
      Noexpr(t) -> ""
      | IntLit(i) -> string_of_int i
      | StringLit(s) -> s
  | Call(f, el) ->
      f ^ "(" ^ String.concat ", " (List.map string_of_expr el) ^ ")"
  | FloatLit(f) -> string_of_float f
      | BoolLit(true) -> "true"
      | BoolLit(false) -> "false"
      | Id(s) -> s
  | AssignOp(v, e) -> string_of_expr v ^ " = " ^ string_of_expr e
  | ArrayAssignOp(v, i, e) -> string_of_expr v ^  "[" ^ string_of_expr i
^ "]"^" = " ^ string_of_expr e
  | ArrayLit(l) -> "[" ^ (String.concat ", " (List.map string_of_expr
l)) ^ "]"
  | ArrayIndex(v, i) -> string_of_expr v ^ "[" ^ string_of_expr i ^ "]"
  | StructAssignOp(v, m, e) -> string_of_expr v ^ "." ^ string_of_expr m
^ " = "^ string_of_expr e ^";"
  | StructAccess(v, m) -> string_of_expr v ^ "." ^ string_of_expr m
```

```
  | _ -> "no expression matched*******"

let string_of_vdecl = function
        VarDecl(t, id, Noexpr(ty)) -> string_of_typ t ^ " " ^ id
  | VarDecl(t, id, e) -> string_of_typ t ^ " " ^ id ^ " = " ^
string_of_expr e

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";
  | VarDecl(t, s1, Noexpr(ty)) ->  string_of_typ t ^" " ^s1 ^ ";\n"
  | VarDecl(t, s1, e1) -> string_of_typ t ^" " ^s1 ^ " = " ^
string_of_expr e1 ^ ";\n"
  | ArrayDecl(t, v, e, Noexpr(ty)) -> string_of_typ t ^ " " ^ v ^  "[" ^
string_of_expr e ^ "];\n"
  |   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_stmt 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
  | Return(e) -> "return " ^ string_of_expr e
  | _ -> "Statement Not Matched??"


let string_of_fdecl fdecl =
  string_of_typ fdecl.typ ^ " " ^
  fdecl.fname ^ "(" ^ String.concat ", " (List.map snd fdecl.formals) ^
  ")\n{\n" ^
  String.concat "" (List.map string_of_stmt fdecl.fstmts) ^
  "}\n"

let string_of_structs sdecl =
  "struct " ^ sdecl.sname ^ "{\n" ^
  String.concat "" (List.map snd sdecl.members) ^
  "};\n"

let string_of_program (structs, funcs) =
  String.concat "" (List.map string_of_structs structs) ^ "\n" ^
  String.concat "\n" (List.map string_of_fdecl funcs)
```

sast.ml

```ocaml
open Ast

type sexpr = typ * sx
and sx =
    SBinop of sexpr * operator * sexpr
  | SNot of sexpr
  | SAssignOp of sexpr * sexpr
  | SArrayAssignOp of sexpr * sexpr * sexpr
  | SStructAssignOp of sexpr * expr * sexpr
  | SVar of string
  | SStringLit of string
  | SFloatLit of float
  | SIntLit of int
  | SBoolLit of bool
  | SArrayLit of sexpr list
  | SArrayIndex of sexpr * sexpr
  | SStructAccess of sexpr * expr
  | SId of string
  | SCall of string * sexpr list
  | SNoexpr of typ

type sstmt =
    SBlock of sstmt list
  | SExpr of sexpr
  | SVarDecl of typ * string * sexpr
  | SArrayDecl of typ * string * sexpr * sexpr
  | SIf of sexpr * sstmt * sstmt
  | SFor of sstmt * sexpr * sexpr * sstmt
  | SWhile of sexpr * sstmt
  | SPrint of sexpr
  | SReturn of sexpr


type sstruct_decl = {
    ssname: string;
    smembers: bind list;
  }

type sfunc_decl = {
    styp : typ;
    sfname : string;
    sformals : bind list;
    sfstmts : sstmt list;
  }
```

```
type sprogram = sstruct_decl list * sfunc_decl list

(* pretty printing function*)

let rec string_of_sexpr (sex:sexpr) = match snd sex with
    SNoexpr(t) -> ""
  | SIntLit(i) -> string_of_int i
  | SStringLit(s) -> s
  | SArrayLit(l) -> "[" ^ (String.concat ", " (List.map string_of_sexpr
l)) ^ "]"
  | SArrayIndex(v, i) -> string_of_sexpr v ^ "[" ^ string_of_sexpr i ^
"]"
  | SCall(f, el) -> f ^ "(" ^ String.concat ", " (List.map
string_of_sexpr el) ^ ")"
  | SId(s) -> s
  | SAssignOp(v, e) -> string_of_sexpr v ^ " = " ^ string_of_sexpr e
  | SArrayAssignOp(v, i, e) -> string_of_sexpr v ^  "[" ^
string_of_sexpr i ^ "]"^" = " ^ string_of_sexpr e
  | _ -> "NOT FOUND"

let string_of_svdecl = function
    VarDecl(t, id, Noexpr(ty)) -> string_of_typ t ^ " " ^ id
  | VarDecl(t, id, e) -> string_of_typ t ^ " " ^ id ^ " = "

let rec string_of_sstmt = function
    SBlock(stmts) ->
      "{\n" ^ String.concat "" (List.map string_of_sstmt stmts) ^ "}\n"
  | SExpr(expr) -> string_of_sexpr expr ^ ";\n";
  (* | SVarDecl(t, s1, SNoexpr) -> string_of_typ t ^" " ^s1 ^ ";\n" *)
  | SVarDecl(t, s1, e1) -> string_of_typ t ^" " ^s1 ^ " = " ^
string_of_sexpr e1 ^ ";\n"
  | SArrayDecl(t, v, e1, e) -> string_of_typ t ^ " " ^ v ^  "[" ^
string_of_sexpr e1 ^ "];\n"
  | SIf(e, s1, s2) ->  "if (" ^ string_of_sexpr e ^ ")\n" ^
string_of_sstmt s1 ^ "else\n" ^ string_of_sstmt s2
  | SFor(e1, e2, e3, s) ->
      "for (" ^ string_of_sstmt e1  ^ " ; " ^ string_of_sexpr e2 ^ " ; "
^ string_of_sexpr e3  ^ ") " ^ string_of_sstmt s
  | SWhile(e, s) -> "while (" ^ string_of_sexpr e ^ ") " ^
string_of_sstmt s
  | SReturn(e) -> "return " ^ string_of_sexpr e

let string_of_sfdecl fdecl =
  string_of_typ fdecl.styp ^ " " ^
  fdecl.sfname ^ "(" ^ String.concat ", " (List.map snd fdecl.sformals)
```

```
^
  ")\n{\n" ^
  String.concat "" (List.map string_of_sstmt fdecl.sfstmts) ^
  "}\n"

let string_of_sstructs sdecl =
  "struct " ^ sdecl.ssname ^ "{\n" ^
  String.concat "\n" (List.map snd sdecl.smembers) ^
  "\n};\n"

let string_of_sprogram (structs, funcs) =
  String.concat "" (List.map string_of_sstructs structs) ^ "\n" ^
  String.concat "\n" (List.map string_of_sfdecl funcs)
```

sement.ml

```
(* Semantic checking for the PartialC compiler *)

open Ast
open Sast

module StringMap = Map.Make(String)

let check (structs, functions) =
  let add_struct map sd =
    let dup_err = "struct dup error"
    and make_err er = raise (Failure er)
    and n = sd.sname
    in match sd with
      _ when StringMap.mem sd.sname map -> make_err dup_err
    | _ -> StringMap.add n sd map
  in
  let check_struct struc =
    let symbols = List.fold_left (fun m (ty, name) -> StringMap.add name
(ty, name, 0) m) StringMap.empty struc.members
    in
      {
        ssname = struc.sname;
        smembers = struc.members;
    }
  in
  (* Add function name to symbol table *)
  let add_func map fd =
    let built_in_err = "function " ^ fd.fname ^ "may not be redefined"
    and dup_err = "duplicate function " ^ fd.fname
```

```
    and make_err er = raise (Failure er)
    and n = fd.fname
    in match fd with
         _ when StringMap.mem n map -> make_err dup_err
       | _ ->  StringMap.add n fd map
  in
  (*build in fucntions*)
  let built_in_funcs = List.fold_left add_func StringMap.empty [
      {typ = Void; fname = "prints"; formals = [(String, "args")];
fstmts = [] };
      {typ = Void; fname = "printi"; formals = [(Int, "args")];  fstmts
= [] };
      {typ = Void; fname = "printf"; formals = [(Float, "args")];
fstmts = [] };
      {typ = Int;  fname = "sizeof"; formals = [(Array(Int), "args")];
fstmts = [] }
      ]
  in
  (* Collect all other function names into one symbol table *)
  let function_decls = List.fold_left add_func built_in_funcs functions
  in

  (* Return a function from our symbol table *)
  let find_func s =
    try StringMap.find s function_decls
    with Not_found -> raise (Failure ("unrecognized function " ^ s))
  in
  let _ = find_func "main" in (* Ensure "main" is defined *)
  let check_function func =
    let add_var map (tp, name, len) =
        let dup_err = "Variable with name " ^ name ^" is a duplicate."
in
        match (tp, name) with
          _ when StringMap.mem name map -> raise (Failure dup_err)
        | _ -> StringMap.add name (tp, name, len) map
    in

    let find_var map name =
        try StringMap.find name map
        with Not_found -> raise( Failure("Undeclared variable: " ^
name))
    in
    let check_assign lvaluet rvaluet err =
       if lvaluet = rvaluet then lvaluet else raise (Failure err)
    in
    let type_of_identifier s symbols =
```

```
      let (ty, _, _) = try StringMap.find s symbols with Not_found ->
raise( Failure("ID not found: " ^ s))
    in ty in
    let rec check_expr map e = match e with
        IntLit  l -> (Int, SIntLit l, map)
      | FloatLit l -> (Float, SFloatLit l, map)
      | BoolLit l  -> (Bool, SBoolLit l, map)
      | StringLit l -> (String, SStringLit l, map)
      | ArrayLit(l) ->
        let array_body = List.map (check_expr map) l in
        let array_type, _, _ = List.nth array_body 0 in
            (Array array_type, SArrayLit(List.map (fun (t, sx, _) ->
(t,sx)) array_body), map)
      | ArrayIndex(name, idx) ->
        let stringName = match name with
            Id i -> i
          | _ -> raise(Failure("Invalid identifier for array: " ^
string_of_expr name)) in
        let (typ, sid, map1) = check_expr map name
        in
        let (idx_type, sindex, map2) = check_expr map1 idx in
        let _ = match sindex with
          SIntLit l ->
            let (_, _, size) = StringMap.find stringName map in
            if l >= size && size != 0 then raise(Failure("Array Index
out ouf bound: " ^ string_of_int l))
            else l
          | _ -> 0
        in
        let element_type = match typ with
            Array(t) -> t
          | _ -> raise(Failure("Type is not expected: " ^ string_of_typ
typ))
        in
        (element_type, SArrayIndex((typ, sid), (idx_type, sindex)),
map2)

      | StructAccess(v, m) ->
        let stringName = match v with
            Id i -> i
          | _ -> raise(Failure("Invalid identifier for struct: " ^
string_of_expr v)) in
        let lt, vname, map1 = find_name v map "assignment error" in
      (Int, SStructAccess((lt, vname), m), map1)
      | Noexpr(ty) -> (ty, SNoexpr(ty), map)
      | Id s       -> (type_of_identifier s map, SId s, map)
```

```
      | AssignOp(v, e)->
        let lt, vname, map1 = find_name v map "assignment error" in
        let rt, ex, map2 = check_expr map1 e in
        (check_assign lt rt "type miss match", SAssignOp((lt, vname),
(rt, ex)), map2)

      | ArrayAssignOp(v, i, e)->
        let stringName = match v with
            Id i -> i
          | _ -> raise(Failure("Invalid identifier for array: " ^
string_of_expr v)) in
        let lt, vname, map1 = find_name v map "assignment error" in
        let rt, ex, map2 = check_expr map1 e in
        let it, ix, map3 = check_expr map2 i in
        let (idx_type, sindex, _) = check_expr map i in
          let _ = match sindex with
            SIntLit l ->
              let (_, _, size) = StringMap.find stringName map in
              if l >= size && size != 0 then raise(Failure("Array Index
out ouf bound: " ^ string_of_int l))
              else l
            | _ -> 0
          in
        let element_type = (match lt with
            Array(t) -> t
            | _ -> raise (Failure ("got " ^ string_of_typ lt))
            )
        in
        (check_assign element_type rt "array type miss match",
SArrayAssignOp((lt, vname), (it, ix),(rt, ex)), map3)

      | StructAssignOp(v, m, e)->
        let stringName = match v with
            Id i -> i
          | _ -> raise(Failure("Invalid identifier for array: " ^
string_of_expr v)) in
        let lt, vname, map1 = find_name v map "assignment error" in
        let rt, ex, map2 = check_expr map1 e in
      (check_assign Ast.Int Ast.Int "array type miss match",
SStructAssignOp((lt, vname), m, (rt, ex)), map2)

      | Call(fname, args) as call ->
          let fd = find_func fname in
          let param_length = List.length fd.formals in
          if List.length args != param_length then
            raise (Failure ("expecting " ^ string_of_int param_length ^
```

```
                              " arguments in " ^ string_of_expr call))
          else let check_call (ft, _) e =
            let (et, e', map') = check_expr map e in
            let err = "illegal argument found " ^ string_of_typ et ^
              " expected " ^ string_of_typ ft ^ " in " ^ string_of_expr
e
            in (check_assign ft ft err, e')
            (* Hack for struct *)
          in
          let args' = List.map2 check_call fd.formals args
          in (fd.typ, SCall(fname, args'), map)

      | Not(e) as notEx-> let (t, e', map') = check_expr map e in
        if t != Bool then
          raise (Failure ("expecting bool expression in " ^
string_of_expr notEx))
        else (Bool, SNot((t, e')), map')
      | Binop(e1, op, e2) as ex ->
        let (t1, e1', map') = check_expr map e1
        in let (t2, e2', map'') = check_expr map' e2
        in
        let same = t1 = t2 in
        let ty =
        match t1 with
        (* | ArrayList inner -> (match op with
                  Add -> t1
                  | _ -> make_err ("Illegal binary operation, cannot
perform "^string_of_expr ex^" on lists.")) *)
        _ -> match op with
                Add | Sub | Mul | Div | Mod    when same && t1 = Int
-> Int
              | Add | Sub | Mul | Div    when same && t1 = Float ->
Float
              | Add                      when same && t1 = String ->
String
              | Eq | Neq                 when same               -> Bool
              | Lt | Leq | Gt | Geq      when same && (t1 = Int || t1 =
Float) -> Bool
              | And | Or                 when same && t1 = Bool -> Bool
              | _ -> raise (Failure ("Illegal binary operator " ^
string_of_typ t1 ^ " " ^ string_of_op op ^ " " ^ string_of_typ t2 ^ " in
" ^ string_of_expr ex))
        in (ty, SBinop((t1, e1'), op, (t2, e2')), map'')

    and find_name (name) map err = match name with
        Id _ -> check_expr map name
```

```
        | _ -> raise (Failure ("find name error"))
    in
    let check_bool_expr map e =
      let (t', e', map') = check_expr map e
      and err = "expected Boolean expression in " ^ string_of_expr e
      in if Bool != Bool then raise (Failure err) else (t', e')
      (* Hack for struct *)
    in
    (* Return a semantically-checked statement i.e. containing sexprs *)
    let rec check_stmt map st = match st with
        Expr e -> let (ty, sexpr, new_map) = check_expr map e in (SExpr
(ty, sexpr), new_map)
      | VarDecl(tp, id, e) ->
        let (right_ty, sexpr, map') = check_expr map e  in
        let err = "illegal argument found." in
        let len = match e with
            Ast.ArrayLit t ->  List.length t
          | _ -> 0 in
        let new_map = add_var map' (tp, id, len) in
        let right = (right_ty, sexpr) in
        (SVarDecl(tp, id, right), new_map)
      (* A block is correct if each statement is correct and nothing
         follows any Return statement.  Nested blocks are flattened. *)
      | ArrayDecl(t, id, e1, e) ->
        let (ty', e1', _) = check_expr map e1 in
          if ty' != Ast.Int then raise ( Failure ("Integer is expected
instead of " ^ string_of_typ t))
        else
          let len = match e1 with
            Ast.IntLit t -> t
          in
          let new_map = add_var map (t, id, len) in
          let (t2, sx2, map') = check_expr map e in
          let r2 = (t2, sx2) in
          (SArrayDecl(t, id, (ty', e1'), r2), new_map)
      | Return e -> let (t, e', map') = check_expr map e in
        if t = func.typ then (SReturn (t, e'), map' )
        else raise ( Failure ("return gives " ^ string_of_typ t ^ "
expected " ^
       string_of_typ func.typ ^ " in " ^ string_of_expr e))

      | Block sl ->
        let rec check_stmt_list map sl = match sl with
            [Return _ as s] -> ([fst (check_stmt map s)], map)
          | Return _ :: _    -> raise (Failure "nothing may follow a
return")
```

```
          | Block sl :: ss  -> check_stmt_list map (sl @ ss) (* Flatten
blocks *)
          | s :: ss         -> let cs, m' = check_stmt map s in
                              let csl, m'' = check_stmt_list m' ss in
                              (cs::csl, m'')
          | []              -> ([], map)
        in (SBlock(fst (check_stmt_list map sl)), map)
      | While(cond, stmtList) -> SWhile(check_bool_expr map cond, fst
(check_stmt map stmtList)), map
      | For(e1, e2, e3, stmtList) -> let (st1, m') = check_stmt map e1
in
                                     let (ty3, sx3, m'') = check_expr m'
e3 in
                                     SFor(st1, check_bool_expr m'' e2,
(ty3, sx3), fst (check_stmt m'' stmtList)), m''
      | If(cond, s1, s2) ->
        let sthen, _ = check_stmt map s1 in
        let selse, _ = check_stmt map s2 in
        (SIf(check_bool_expr map cond, sthen, selse), map)
      | _ -> raise (Failure "Match failure")

    in (* body of check_function *)
    let symbols = List.fold_left (fun m (ty, name) -> StringMap.add name
(ty, name, 0) m) StringMap.empty func.formals
    in
      {
        styp = func.typ;
        sfname = func.fname;
        sformals = func.formals;
        sfstmts = match fst (check_stmt symbols (Block(func.fstmts)))
with
          SBlock(sl) -> sl
          | _ -> let err = "internal error: block didn't become a
block?"
          in raise (Failure err)
    }

  in
  let sfuncc = List.map check_function functions in
  let sstructs = List.map check_struct structs in
  (sstructs, sfuncc)
```

codegen.ml

```
(* Code generation: translate takes a semantically checked AST and
produces LLVM IR

LLVM tutorial: 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
open Sast

module StringMap = Map.Make(String)

(* translate : Sast.program -> Llvm.module *)
let translate (structs, functions) =
  let report_error e = raise (Failure e) in
  let context    = L.global_context () in

  (* Create the LLVM compilation module into which
     we will generate code *)
  let the_module = L.create_module context "PartialC" in

  (* Get types from the context *)
  let i32_t      = L.i32_type    context
  and i8_t       = L.i8_type     context
  and i1_t       = L.i1_type     context
  and float_t    = L.double_type context
  and char_t     = L.i8_type     context
  and void_t     = L.void_type   context
  and struct_t n  = L.named_struct_type context n in
 (* Some compond type *)
  let array_t = fun (llvm_type) -> L.struct_type context [|
L.pointer_type llvm_type; i32_t; i32_t|] in
  let string_t = L.struct_type context [| L.pointer_type char_t|] in
  let i32OF = L.const_int (L.i32_type context) in
  (* llvm type of some strcut members *)
  let rec ltype_of_struct_members = function
```

```
      A.Struct n -> struct_t n
    | A.Int -> i32_t
    | A.Float -> float_t
    | A.String -> string_t
    | A.Bool -> i1_t
  in
  (* Structs declaration*)
  let structs_decls =
    let struct_decl map sdecl =
      let name = sdecl.ssname
      and member_types = Array.of_list (List.map (fun (t,_) ->
ltype_of_struct_members t) sdecl.smembers) in
      let stype = L.struct_type context member_types in
      StringMap.add name (stype, sdecl.smembers) map in
    List.fold_left struct_decl StringMap.empty structs
  in
  let lookup_struc s =
  try StringMap.find s structs_decls
    with Not_found -> raise (Failure ("struct not found")) in
  (* llvm types for primitive types*)
  let rec ltype_of_typ = function
      A.Int   -> i32_t
    | A.Bool  -> i1_t
    | A.Float -> float_t
    | A.Void  -> void_t
    | A.String -> string_t
    | A.Array a -> array_t (ltype_of_typ a)
    | A.Struct n -> fst (lookup_struc n)
  in
  (* llvm type of array element *)
  let rec ltype_of_array_element = function
  A.Array a -> ltype_of_typ a
  in
  (* print fucntion *)
  let printf_t = L.var_arg_function_type i32_t [| (L.pointer_type
char_t)|] in
  let printf_func = L.declare_function "printf" printf_t the_module in

  (* Define each function (arguments and return type) so we can
     call it even before we've created its body *)
  let function_decls =
    let function_decl map fdecl =
      let name = fdecl.sfname
      and formal_types = Array.of_list (List.map (fun (t,_ ) ->
ltype_of_typ t ) fdecl.sformals) in
      let ftype = L.function_type (ltype_of_typ fdecl.styp) formal_types
```

```
in
      StringMap.add name (L.define_function name ftype the_module,
fdecl) map in
    List.fold_left function_decl StringMap.empty functions in

  (* Fill in the body of the given function *)
  let build_function_body fdecl =
    let (the_function, _) = StringMap.find fdecl.sfname function_decls
in
    let builder = L.builder_at_end context (L.entry_block the_function)
in
    let char_format_str = L.build_global_stringptr "%s\n" "" builder
    and int_format_str = L.build_global_stringptr "%d\n" "" builder
    and float_format_str = L.build_global_stringptr "%f\n" "" builder in
  (* loca, variables *)
    let local_vars =
      let add_formal m (t, n) p =
        L.set_value_name n p;
      let local = L.build_alloca (ltype_of_typ t) n builder in
          ignore (L.build_store p local builder);
          StringMap.add n (local, A.Void) m
      in
      List.fold_left2 add_formal StringMap.empty fdecl.sformals
          (Array.to_list (L.params the_function)) in
   (* loop up for variables *)
    let lookup map n = match StringMap.find_opt n map with
        Some (v, _) -> v
      | None -> try fst (StringMap.find n local_vars)
                with Not_found -> report_error("Could not find " ^ n)
    in
    (* Construct code for an expression; return its value *)
    let rec expr map builder ((_, expression) : sexpr) = match
expression with
        SIntLit i  -> L.const_int i32_t i, map, builder
      | SFloatLit f -> L.const_float float_t f, map, builder
      | SBoolLit b  -> L.const_int i1_t (if b then 1 else 0), map,
builder
      | SStringLit s ->
        let alloc = L.build_alloca string_t "alloc" builder in
        let strGlobal = L.build_global_string s "strGlobal" builder in
        let str = L.build_bitcast strGlobal (L.pointer_type i8_t)
"str_cast" builder in
        let str_loc = L.build_struct_gep alloc 0 "str_cast_loc" builder
in
        let _ = L.build_store str str_loc builder in
        let value = L.build_load alloc "" builder
```

```
        in (value, map, builder)

      | SArrayLit a ->
         let llvm_ty = ltype_of_typ (fst (List.hd a))in
         let ty = array_t llvm_ty in
         let alloc = L.build_alloca ty "alloc" builder in
         let data_location = L.build_struct_gep alloc 0 "data_location"
builder in
         let len_loc = L.build_struct_gep alloc 1 "" builder in
         let len = List.length a in
         let cap = len * 2 in
         let data_loc = L.build_array_alloca llvm_ty (i32OF cap)
"data_loc" builder in
         let array_iter (acc, builder) ex =
           let value, m', builder = expr map builder ex in
           let item_loc = L.build_gep data_loc [|i32OF acc |] "item_loc"
builder in
           let _ = L.build_store value item_loc builder in
           (acc+1, builder)
         in
         let _, builder = List.fold_left array_iter (0, builder) a in
         let _ = L.build_store data_loc data_location builder in
         let _ = L.build_store (i32OF len) len_loc builder in
         let value = L.build_load alloc "value" builder
       in (value, map, builder)

      | SArrayIndex(id, idx) ->
        let name = match snd id with
            SId s -> s
          | _ -> "err:cannot index non-id"
        in
        let a_addr = lookup map name in
        let data_location = L.build_struct_gep a_addr 0 "" builder in
        let data_loc = L.build_load data_location "" builder in
        let ival, _, builder = expr map builder idx in
        let i_addr = L.build_gep data_loc [| ival |] "" builder in
        let value = L.build_load i_addr "" builder in
        (value, map, builder)

      | SNoexpr(t)     -> (match t with
          A.Int -> L.const_int i32_t 0
        | A.Float -> L.const_float float_t 0.0
        | A.Bool -> L.const_int i1_t 1
        | A.String ->
          let alloc = L.build_alloca string_t "alloc" builder in
          let strGlobal = L.build_global_string "" "strGlobal" builder
```

```
in
          let str = L.build_bitcast strGlobal (L.pointer_type i8_t)
"str_cast" builder in (* Mingjie: this is crucial*)
          let str_loc = L.build_struct_gep alloc 0 "str_cast_loc"
builder in
          let _ = L.build_store str str_loc builder in
          L.build_load alloc "" builder

        | A.Array _ ->
          let llvm_ty = ltype_of_typ (fst (List.hd []))in
          let ty = array_t llvm_ty in
          let alloc = L.build_alloca ty "alloc" builder in
          let data_location = L.build_struct_gep alloc 0 "data_location"
builder in
          let len_loc = L.build_struct_gep alloc 1 "" builder in
          let len = 0 in
          let cap = len * 2 in
          let data_loc = L.build_array_alloca llvm_ty (i32OF cap)
"data_loc" builder
          in
          let _ = L.build_store data_loc data_location builder in
          let _ = L.build_store (i32OF len) len_loc builder in
          L.build_load alloc "value" builder) , map, builder

      | SId s       -> L.build_load (lookup map s) s builder, map,
builder
      | SAssignOp (v, e) ->
        let (e1, map1, builder) = expr map builder e in
          (match (snd v) with
            SId s ->
            ignore(L.build_store e1 (lookup map s) builder); e1, map1,
builder)

      | SArrayAssignOp (v, i, e) ->
        let rval, m', builder = expr map builder e in
        let name = match snd v with
            SId s -> s
        in
        let a_addr = lookup map name in
        let data_location = L.build_struct_gep a_addr 0 "" builder in
        let data_loc = L.build_load data_location "" builder in
        let ival, _, builder = expr map builder i in
        let addr = L.build_gep data_loc [| ival |] "" builder  in
        let _ = L.build_store rval addr builder in
        (rval, m', builder)
```

```
      | SStructAssignOp (v, m, e) ->
        let rval, map1, builder = expr map builder e in
        let name = match snd v with
            SId s -> s
        in
        let a_addr = lookup map1 name in
        let strcut_name = (match snd (StringMap.find name map1) with
A.Struct i -> i) in
        (* let _ = print_string strcut_name in *)

        let mname = (match m with A.Id i -> i) in

        let members = snd (lookup_struc strcut_name) in
          let rec get_idx n lst i = match lst with
            | [] -> raise (Failure( "Struct member Error"))
            | hd::tl -> if (hd=n) then i else get_idx n tl (i+1)
          in let idx = (get_idx mname (List.map (fun (_,nm) -> nm)
members) 0) in
        let ptr = L.build_struct_gep a_addr idx ("struct_p") builder in
        let _ = L.build_store rval ptr builder in
        (rval, map1, builder)

      | SStructAccess (v, m) ->
        let name = match snd v with
            SId s -> s
        in
        let a_addr = lookup map name in
        let strcut_name = (match snd (StringMap.find name map) with
A.Struct i -> i) in

        let mname = (match m with A.Id i -> i) in

        let members = snd (lookup_struc strcut_name) in
          let rec get_idx n lst i = match lst with
            | [] -> raise (Failure( "Struct member Error"))
            | hd::tl -> if (hd=n) then i else get_idx n tl (i+1)
          in let idx = (get_idx mname (List.map (fun (_,nn) -> nn)
members) 0) in
        let ptr = L.build_struct_gep a_addr idx ("struct_p") builder in
        let value = L.build_load ptr "member_v" builder in
        (value, map, builder)

      | SNot (e) ->
        let (e', _, _) = expr map builder e in
        L.build_not e' "not operation" builder, map, builder
```

```
      | SBinop ((A.Float,_ ) as e1, op, e2) ->
        let (e1', _, _) = expr map builder e1
        and (e2', _, _) = expr map builder e2 in
        (match op with
          A.Add     -> L.build_fadd
        | A.Sub     -> L.build_fsub
        | A.Mul     -> L.build_fmul
        | A.Div     -> L.build_fdiv
        | A.Eq   -> L.build_fcmp L.Fcmp.Oeq
        | A.Neq     -> L.build_fcmp L.Fcmp.One
        | A.Lt    -> L.build_fcmp L.Fcmp.Olt
        | A.Leq     -> L.build_fcmp L.Fcmp.Ole
        | A.Gt -> L.build_fcmp L.Fcmp.Ogt
        | A.Geq     -> L.build_fcmp L.Fcmp.Oge
        | A.And | A.Or ->
            raise (Failure "internal error: semant should have rejected
and/or on float")
        ) e1' e2' "float op" builder, map, builder

      | SBinop (e1, A.Mod, e2) ->
        let (e1', _, _) = expr map builder e1
        and (e2', _, _) = expr map builder e2 in
        L.const_srem e1' e2', map, builder

      | SBinop (e1, op, e2) ->
        let (e1', _, _) = expr map builder e1
        and (e2', _, _) = expr map builder e2 in
        (match op with
          A.Add     -> L.build_add
        | A.Sub     -> L.build_sub
        | A.Mul     -> L.build_mul
        | A.Div     -> L.build_sdiv
        | A.And     -> L.build_and
        | A.Or      -> L.build_or
        | A.Eq   -> L.build_icmp L.Icmp.Eq
        | A.Neq     -> L.build_icmp L.Icmp.Ne
        | A.Lt    -> L.build_icmp L.Icmp.Slt
        | A.Leq     -> L.build_icmp L.Icmp.Sle
        | A.Gt -> L.build_icmp L.Icmp.Sgt
        | A.Geq     -> L.build_icmp L.Icmp.Sge
        ) e1' e2' "general op" builder, map, builder

      | SCall ("prints", [e]) ->
        let e', _, builder = expr map builder e in L.build_call
printf_func [| char_format_str ; e' |] "printf" builder, map, builder
```

```
      | SCall ("printi", [e]) ->
        let e', _, builder = expr map builder e in L.build_call
printf_func [| int_format_str ; e' |] "printf" builder, map, builder

      | SCall ("printf", [e]) ->
        let e', _, builder = expr map builder e in L.build_call
printf_func [| float_format_str ; e' |] "printf" builder, map, builder

      | SCall ("sizeof", [e]) -> let a_addr = (match e with _, SId s ->
lookup map s) in
        let len_field_loc = L.build_struct_gep a_addr 1 "" builder in
        let value = L.build_load len_field_loc "" builder in
        value, map, builder

      | SCall ("sizeof", [e]) ->
        let (e', _, builder) = expr map builder e  in
        let length = L.array_length (L.type_of e') in
        L.const_int i32_t length, map, builder

      | SCall (f, args) ->
        let (fdef, fdecl) = StringMap.find f function_decls in
        let llargs = List.map (fun(a,b,c) -> a) (List.rev (List.map
(expr map builder) (List.rev args))) in
        let result = (match fdecl.styp with
                          A.Void -> ""
                        | _ -> f ^ "_result") in
        L.build_call fdef (Array.of_list llargs) result builder, map,
builder
    in

    (* LLVM insists each basic block end with exactly one "terminator"
       instruction that transfers control.  This function runs "instr
builder"
       if the current block does not already have a terminator.  Used,
       e.g., to handle the "fall off the end of the function" case. *)
    let add_terminal builder instr = match L.block_terminator
(L.insertion_block builder) with
            Some _ -> ()
      | None -> ignore (instr builder) in

    (* Build the code for the given statement; return the builder for
       the statement's successor (i.e., the next instruction will be
built
       after the one generated by this call) *)

    let rec stmt map builder s = match s with
```

```
        SBlock sl ->
               let b, _ = List.fold_left (fun (b, m) s -> stmt m b s)
(builder, map) sl in (b, map)
      | SReturn e -> ignore(match fdecl.styp with
                              (* Special "return nothing" instr *)
                              A.Void -> L.build_ret_void builder
                              (* Build return statement *)
                            | _ -> let e',_,_ = (expr map builder e) in
L.build_ret e' builder ); builder, map

      | SExpr e -> ignore(expr map builder e); builder, map
      | SVarDecl(ty, st, rex) ->
          (match ty with
          A.Struct s->
              let l_type = ltype_of_typ ty in
              let addr = L.build_alloca l_type st builder in
              let m' = StringMap.add st (addr, ty) map in
              (builder, m')
          | _ ->
              let l_type = ltype_of_typ ty in
              let addr = L.build_alloca l_type st builder in
              let rval, m', builder = expr map builder rex in
              let m'' = StringMap.add st (addr, A.Void) m' in
              let _ = L.build_store rval addr builder in
          (builder, m''))

      | SArrayDecl(t, v, e1, e) ->

          let llvm_ty = ltype_of_typ t in
          let addr = L.build_alloca llvm_ty v builder in
          let alloc = L.build_alloca llvm_ty "alloc" builder in
          let data_location = L.build_struct_gep alloc 0 "data_location"
builder in
          let len = (match e1 with _, SIntLit i -> i) in
          let len_loc = L.build_struct_gep alloc 1 "" builder in
          let cap = len * 2 in
          let data_loc = L.build_array_alloca (ltype_of_array_element t)
(i32OF cap) "data_loc" builder in
          let noexpr_value = (match t with
              Array Ast.Int -> L.const_int i32_t 0
            | Array Ast.Float -> L.const_float float_t 0.0
            | Array Ast.Bool -> L.const_int i1_t 1
            | Array Ast.String -> let alloc = L.build_alloca string_t
"alloc" builder in
                        let strGlobal = L.build_global_string ""
"strGlobal" builder in
```

```
                        let str = L.build_bitcast strGlobal
(L.pointer_type i8_t) "str_cast" builder in (* Mingjie: this is
crucial*)
                        let str_loc = L.build_struct_gep alloc 0
"str_cast_loc" builder in
                        let _ = L.build_store str str_loc builder in
                        L.build_load alloc "" builder)
          in
          let rec sto (acc, builder) =
            let item_loc = L.build_gep data_loc [|i32OF acc |]
"item_loc" builder in
            let _ = L.build_store noexpr_value item_loc builder in
            if acc < len then sto (acc + 1, builder) else acc, builder
          in
          let _, builder = sto (0, builder) in
          let m' = StringMap.add v (addr, A.Void) map in
          let _ = L.build_store data_loc data_location builder in
          let _ = L.build_store (i32OF len) len_loc builder in
          let value = L.build_load alloc "value" builder in
          let dl = lookup m' v in
          let _ = L.build_store value dl builder in
          (* ignore(L.build_store value (lookup m' v) builder);  *)
          (builder, m')

      | SWhile(condition, stmtList) ->
          let pred_bb = L.append_block context "while" the_function in
          ignore(L.build_br pred_bb builder);
          let body_bb = L.append_block context "while_body" the_function
in
          let body_bldr, m' = stmt map (L.builder_at_end context
body_bb) stmtList in
          add_terminal body_bldr (L.build_br pred_bb);
          let pred_builder = L.builder_at_end context pred_bb in
          let bool_val, _, _ = expr m' pred_builder condition in
          let merge_bb = L.append_block context "merge" the_function in
          ignore(L.build_cond_br bool_val body_bb merge_bb
pred_builder);
          L.builder_at_end context merge_bb, m'

      | SFor(e1, e2, e3, stmtList) -> stmt map builder ( SBlock [ e1 ;
SWhile (e2, SBlock [stmtList ; SExpr e3]) ] )
      | SIf(e, s1, s2) ->
        let bool_val, m', builder = expr map builder e in
        let merge_bb = L.append_block context "merge" the_function in
        let build_br_merge = L.build_br merge_bb in (* partial function
*)
```

```
        let then_bb = L.append_block context "then" the_function in
        let then_builder, m'' = stmt m' (L.builder_at_end context
then_bb) s1 in
          add_terminal then_builder build_br_merge;
        let else_bb = L.append_block context "else" the_function in
        let else_builder, m'' = stmt m' (L.builder_at_end context
else_bb) s2 in
          add_terminal else_builder build_br_merge;
        ignore(L.build_cond_br bool_val then_bb else_bb builder);
        L.builder_at_end context merge_bb, m'
      | _ -> report_error "No implementation"
    in

    (* Build the code for each statement in the function *)
    let builder,_= stmt StringMap.empty builder (SBlock fdecl.sfstmts)
in

    (* Add a return if the last block falls off the end *)
    add_terminal builder (match fdecl.styp with
        A.Void -> L.build_ret_void
      | A.Float -> L.build_ret (L.const_float float_t 0.0)
      | t -> L.build_ret (L.const_int (ltype_of_typ t) 0))
  in

  List.iter build_function_body functions;
  the_module
```

Appendix B -- Success test cases

B.1 Empty function

```
void main(){
}
```

Output:

```

```

B.2 Test Variable Declaration - Primitive type

```
void main(){
	int a = -8;
	printi(a);

	float b = 1.2;
	printf(b);

	bool c = true;
	bool d = false;
}
```

Output:

```
-8
1.200000
```

B.3 Test Variable Declaration - Compound type

```
struct Student{
	int sid;
	float grade;
	bool graduated;
};

void main(){
	string str_a = 'hello world';
	prints(str_a);

	arr int a = [1,2,3];

	Student x;
	x.sid = 1;
	x.grade = 4.5;
	x.graduated = false;
```

```
}
```

Output:

```
'hello world'
```

B.4 Print float number

```
void main(){
      printf(2.3);
}
```

Output:

```
2.300000
```

B.5 Print integer

```
void main(){
      printi(1);
      int a = 4;
      printi(a*3);
}
```

Output:

```
1
12
```

B.6 Print string

```
void main(){
      prints('Hello World!');
      string b = 'var';
      prints(b);
}
```

Output:

```
'Hello World!'
'var'
```

B.7 Assign Array w/o initialization

```
void main(){
      arr int a[3];

}
```

Output:

```

```

B.8 Assign Array

```
void main(){
      arr int a[3];

}
```

Output:

```

```

B.9 Assign Boolean w/o initialization

```
void main(){
      bool a;
      if(a){
            prints('correct');
      }else{

      }
}
```

Output:

```
'correct'
```

B.10 Assign Boolean

```
void main(){
      bool a = false;
      if(a){
            printi(1);
      }else{

      }
```

```
}
```

Output:

```

```

B.11 Assign Float w/o initialization

```
void main(){
	float a;
	printf(a);
}
```

Output:

```
0.000000
```

B.12 Assign Float

```
void main(){
	float a = 4.2;
	printf(a);
}
```

Output:

```
4.200000
```

B.13 Assign Integer w/o initialization

```
void main(){
	int a;
	printi(a);
}
```

Output:

```
0
```

B.14 Assign Integer

```
void main(){
	int a;
	a = 1 + 3;
	printi(a);
```

```
}
```

Output:

```
4
```

B.15 Declaration w/o Initial Value

```
void main(){
      int a;
      printi(a);
    float b;
    printf(b);
    bool c;
    if(c){
      prints('correct');
    }else{
      prints('wrong');
    }
    string d;
    prints(d);

    arr int e[2];
    e = [2,3];
    printi(e[0]);
}
```

Output:

```
0
0.000000
'correct'

2
```

B.16 Assign String

```
void main(){
      string a = 'hello';
      a = 'world';
      prints(a);
}
```

Output:

```
'world'
```

B.17 Calculation Add

```
void main(){
      int a = 4 * ( 1 + 2 );
      printi(a);
      float b = 1.2 + 3.5;
      printf(b);
}
```

Output:

```
12
4.700000
```

B.18 Calculation Divide

```
void main(){
      int a = 4 / 2;
      printi(a);
      float b = 6.2 / 3.0;
      printf(b);
}
```

Output:

```
2
2.066667
```

B.19 Calculation Minus

```
void main(){
      int a = 1 - 2;
      printi(a);
      float b = 6.2 - 13.1;
      printf(b);
}
```

Output:

```
-1
-6.900000
```

B.20 Calculation Mod

```
void main(){
	int a = 4 % 3;
	printi(a);
}
```

Output:

```
1
```

B.21 Calculation Multiplication

```
void main(){
	int a = 4 * 2;
	printi(a);
	float b = 9.3 * 0.5;
	printf(b);
}
```

Output:

```
8
4.650000
```

B.22 Negation

```
void main(){
	int a = 4 + -2;
	printi(a);
	float b = 9.3 + -0.5;
	printf(b);
}
```

Output:

```
2
8.800000
```

B.23 Logical And

```
void main(){
```

```
	bool a = true;
	bool b = false;
	if(a && b){
		prints('wrong');
	}else{
		prints('correct');
	}
	if(a && a){
		prints('correct');
	}else{
		prints('wrong');
	}
}
```

Output:

```
'correct'
'correct'
```

B.24 Logical Not

```
void main(){
	bool a = true;
	bool b = false;
	if( !b ){
		prints('correct');
	}else{
		prints('wrong');
	}
	if( !a ){
		prints('wrong');
	}else{
		prints('correct');
	}
}
```

Output:

```
'correct'
'correct'
'correct'
```

B.25 Logical Or

```
void main(){
	bool a = true;
	bool b = false;
	if(a || b){
		prints('correct');
	}else{
		prints('wrong');
	}
	if(a || a){
		prints('correct');
	}else{
		prints('wrong');
	}
	if(b || b){
		prints('wrong');
	}else{
		prints('correct');
	}
}
```

Output:

```
'correct'
'correct'
'correct'
```

B.26 Logical Equal

```
void main(){
	float a = 3.0;
	if(a == 3.0){
		prints('correct');
	}else{
		prints('wrong');
	}
	if(a == 2.9){
		prints('wrong');
	}else{
		prints('correct');
	}
}
```

Output:

```
'correct'
'correct'
```

B.27 Logical Greater or Equal to

```
void main(){
	int a = 3;
	if(a >= -2){
		prints('correct');
	}else{
		prints('wrong');
	}
	if(a >= 3){
		prints('correct');
	}else{
		prints('wrong');
	}
	if(a >= 5){
		prints('wrong');
	}else{
		prints('correct');
	}
}
```

Output:

```
'correct'
'correct'
'correct'
```

B.28 Logical Greater

```
void main(){
	int a = 3;
	if(a > 2){
		prints('correct');
	}else{
		prints('wrong');
	}
	if(a > 5){
		prints('wrong');
	}else{
		prints('correct');
	}
}
```

Output:

```
'correct'
'correct'
```

B.29 Logical Less or Equal to

```
void main(){
      float a = 3.2;
      if(a <= 5.92){
            prints('correct');
      }else{
            prints('wrong');
      }
      if(a <= 3.2){
            prints('correct');
      }else{
            prints('wrong');
      }
      if(a <= -5.0){
            prints('wrong');
      }else{
            prints('correct');
      }
}
```

Output:

```
'correct'
'correct'
'correct'
```

B.30 Logical Less than

```
void main(){
      float a = 3.0;
      if(a < 5.2){
            prints('correct');
      }else{
            prints('wrong');
      }
      if(a < 2.9){
            prints('wrong');
      }else{
            prints('correct');
```

```
	}
}
```

Output:

```
'correct'
'correct'
```

B.31 Logical Not Equal to

```
void main(){
	float a = 3.0;
	if(a != 2.0){
		prints('correct');
	}else{
		prints('wrong');
	}
	if(a != 3.0){
		prints('wrong');
	}else{
		prints('correct');
	}
}
```

Output:

```
'correct'
'correct'
```

B.32 Control Flow For

```
void main(){
	for(int a=1; a<5; a=a+1){
		printi(a);
	}
}
```

Output:

```
1
2
3
4
```

B.33 Control Flow if

```
void main(){
	int a = 3;
	if (a > 1) {
		prints('larger than 1');
	} else {
		prints('less or equal than 1');
	}
}
```

Output:

```
'larger than 1'
```

B.34 Control Flow if w/o else

```
void main(){
	int a = 3;
	// a =0;
	if (a > 1) {
		prints('larger than 1');
	}
}
```

Output:

```
'larger than 1'
```

B.35 Control Flow Return float

```
void main(){
	float a = returnfloat();
	printf(a);
}

float returnfloat(){
	return 2.0;
}
```

Output:

```
2.000000
```

B.36 Control Flow Return String

```
void main(){
	string a = returnstring();
	prints(a);
}

string returnstring(){
	return 'hey';
}
```

Output:

```
'hey'
```

B.37 Control Flow Return Int

```
void main(){
	int a = returnint();
	printi(a);
}

int returnint(){
	return 2;
}
```

Output:

```
2
```

B.38 Control Flow While

```
void main(){
	int a = 2;
	while(a > 1){
		prints('hello world');
		a = 1;
	}
	while(a == 1){
		prints('second hello world');
		a = 2;
	}
}

void main(){
```

```
        for(int a=1; a<5; a=a+1){
                printi(a);
        }
}
```

Output:

```
'hello world'
'second hello world'
```

B.39 User Defined Function w/o Arguments

```
void testcall(){
        prints('called');
        //printi(a);
}

void main(){
        testcall();
}
```

Output:

```
'called'
```

B.40 User Defined Function with 2 Arguments

```
float testcall(int b, float a){
        b = b + 1;
        printi(b);
        if(b > 5){
                return 0.0;
        }else{
                testcall(b, 1.9);
        }
        printf(a);
        return -1.9;
}

void main(){
        int i=0;

        testcall(0, 0.0);
}
```

Output:

```
1
2
3
4
5
6
1.900000
1.900000
1.900000
1.900000
0.000000
```

B.41 User Defined Function with 1 Argument

```
int  testcall(int b){
      b = b + 1;
      printi(b);
      if(b > 5){
            return 0;
      }else{
            testcall(b);
      }
      return -1;
}

void main(){
      int i=0;

      testcall(0);
}
```

Output:

```
1
2
3
4
5
6
```

B.42 Array as Arg

```
void main(){
    arr int change = [1, 2, 5];
    testarray(change, 'hi');
}

int testarray(arr int a, string b){
    printi(a[1]);
    return 0;
}
```

Output:

```
2
```

B43. Array as Arg w/o initialization

```
void main(){
    arr int change[5];
    testarray(change, 'hi');
}

int testarray(arr int a, string b){
    printi(a[1]);
    return 0;
}
```

Output:

```
0
```

B.44 Coin Change

```
// Coin Change
// Suppose we have coins of value 1, 2 and 5. Given a target of N,
return the fewest number of coins to make up N

// result[N] = min ({result[N - vi] + 1}) for N > 0 and vi = 1, 2, 5
// To make change for n cents, we are going to figure out how to make
change for every value x < n first.

void main(){
    arr int change = [1, 2, 5];
```

```
    arr int result[28]; //all values initialized to

    // mark result[1] result[2] and result[5] as 1.
    for(int i=0; i<=sizeof(change); i=i+1){
        result[change[i]] = 1;
    }

    for(int j=1; j<sizeof(result); j=j+1){
        for(int k=0; k<=sizeof(change); k=k+1){
            if(change[k] < j && (result[j] > (result[j- change[k]] +
result[change[k]]) || result[j] == 0)){
                result[j] = result[j-change[k]] + 1;
            }
        }
    }

    for(int y=1; y<sizeof(result); y=y+1){
        printi(result[y]);
    }
}
```

Output:

```
1
1
2
2
1
2
2
3
3
2
3
3
4
4
3
4
4
5
5
4
5
5
6
```

```
6
5
6
6
```

B.45 Fibonacci

```
void main(){
    int target = 9;
    arr int b[10];
    fib(b, target, 4.3);
    printi(b[9]);
}
void fib(arr int f, int t, float b){
    f[0] = 1;
    f[1] = 1;
    for(int i = 2; i <= t; i=i+1)
    {
       f[i] = f[i - 1] + f[i - 2];
    }
    printf(b);
}
```

Output:

```
55
```

B.46 Array sizeof

```
void main(){
    arr int change = [1, 2, 5];
    printi(sizeof(change));
}
```

Output:

```
3
```

B.47 Array sizeof w/o initialization

```
void main(){
    arr int change[9];
    printi(sizeof(change));
}
```

Output:

```
9
```

B.48 Array Pointers

```
void foo(arr int t){
      t[0] = 10;
}

void main(){
      arr int a[3];
      foo(a);
      printi(a[0]);
}
```

Output:

```
10
```

B.49 Struct

```
struct Test{
      int a;
      float b;
      bool c;
};

void main(){
      Test x;
      x.a = 1;
      x.b = 4.5;
      x.c = true;

      printi(x.a);
      printf(x.b);
      if(x.c){
            prints('success!');
      }
}
```

Output:

```
1
4.500000
'success!'
```

Appendix C -- Failure test cases

c.1 Array out of bound check

```
void main(){
	arr int a[8];
	a[10];
}
```

Output:

```
Fatal error: exception Failure("Array Index out ouf bound: 10")
/usr/lib/gcc/x86_64-linux-gnu/7/../../../x86_64-linux-gnu/Scrt1.o: In
function `_start':
```

c.2 Array out of bound check 2

```
void main(){
	arr int a = [1,2,3,4,5,6,7,8,9,10];
	a[10];
}
```

Output:

```
Fatal error: exception Failure("Array Index out of bound: 10")
```

c.3 Return type does not match:

```
void main(){
	test();
}

int test(){
	return 1.0;
}
```

Output:

```
Fatal error: exception Failure("return gives float expected int in 1.")
```

c.4 Expression type missmatch

```
void main(){
```

```
        int i;
        i = 4.2;
}
```

Output:

```
Fatal error: exception Failure("type miss match")
```