ManiT Final Report
COMS 4115

Akiva Dollin - acd2164
Irwin Li - izl2000
Seungmin Lee - sl3254
Dong Hyeon (Paul) Seo - ds3457

Contents

ManiT Final Project

1. Introduction

1.1 Motivation

The goal of ManiT is to design a language that allows for the user to easily create and manipulate complex data types. ManiT will allow the user to easily define data types in the form of structs, manipulate files, and manage executables using a library of built-in functions.

1.2 Language Description

ManiT is a general purpose, object oriented programming language. ManiT draws inspiration from C and Python, with the aim of creating user defined datatypes in the form of structs allowing for ease of use in the form of type inference. Syntactically, ManiT is a mixture of Python and C. Like Python, ManiT does not have a defined entry point, such as the main function in C, and instead will execute the code sequentially.

1.3 Source Code

The source code is available at

2. Tutorial

2.1 Environment

The ManiT compiler was built and tested in OS X, Ubuntu 14.04, and Ubuntu 16.04 virtual machine. The compiler translates ManiT source code into LLVM, a portable assembly-like language.
The compiler can be found in:
GitHub: https://github.com/akdollin/ManiT

To download the compiler, simply go to the terminal and type:
$ git clone https://github.com/akdollin/ManiT

If the following fails and you do not have git in your terminal, then for:
Mac OS X,download and install the file in: https://git-scm.com/download/mac
Ubuntu, simply go to the terminal and type:
$ sudo apt-get install git

As ManiT was written using OCaml programming language and LLVM library, it is necessary that the system running ManiT has the two.
For more information about OCaml and LLVM:
http://www.ocaml.org/docs/install.html
http://llvm.org/docs/GettingStarted.html

Inside the ManiT project folder, there will be a subdirectory called “src”, which contains the source code for the ManiT programming language. Go into the folder and type “make”. This will create the ManiT compiler called “manit.native”. The compiler takes in a file in ManiT language and outputs a file containing LLVM byte code. The ManiT file extension if “.mt”. To generate LLVM byte code for a specific file, type:

```
./manit.native < <manit_file_name.mt> > <output_file_name.ll>
```

In order to execute the LLVM bytecode, pass it through the LLVM interpreter using:

```
lli <output_file_name.ll>
```

Running and Testing

Once all the prerequisites (OCaml and LLVM) are installed and the compiler has been cloned to the local system, open the terminal and trace to the compiler folder.
Then type the following on the terminal:
$ cd src

$ make
This should run all OCaml modules into the compiler.

To run the test suite file, simply type the following on the terminal upon doing the steps above to run all OCaml modules into the compiler.
$ ./testshall.sh

Alternatively, the following will also run the test suite:
$ make test

To create and run your own program, save the code in an *.mt file.
A simple method of running your own program is upon saving the *.mt file:
$ ./manit.native < *.mt > output.ll
$ lli output.ll
in which *.mt is the filename of the your program.

2.3 Program Structure
A ManiT program is a list of statements. A statement can broken down into one of the three categories: can be broken down into three segments.
1. Function definition. Function definitions are similar to C, except that the keyword def is needed before the return type. The types of formal parameters and the return type must be specified in function definition.
2. Struct definition. Struct definitions are also similar to C. The types of the members of the struct must be specified in struct definition.
3. Other statements.

ManiT does not have a defined entry point (main function), and the list of statements are executed sequentially from top to bottom. Function definition must occur above the corresponding function calls and struct definition must occur above creating an instance of the struct type. A variables that are declared outside of a scope is a global variable.

Type Inference
ManiT has partial type inference.
Function definitions and struct definitions need to be specified. An instance of a struct type is declared by specifying the type in a declaration statement. Other variables are declared when it is assigned a value. The type of a variable is determined by the type of the assigned literal, which can be one of the primitive literals or an array literal. The type of a variable cannot change in the scope that it is declared. Another variable with same name and different type can be declared outside of the scope.

Built-in Functions

ManiT supports several built-in functions that are helpful for file manipulation and forking. ManiT supports the following built-in functions:

- `open()` opens a file. It takes two string arguments, the filename and the mode to open the file in ("w" for write, "r" for read, "a" for append, ...). `open()` returns a string pointer type that contains information about the file pointer. Example:

```
f = open("fname", "w");
```

- `close()` closes a file. close() takes one argument, which is a string pointer type containing information about the file pointer. It will return an integer 0 if successfully closed and an EOF if there is an error. Example:

```
/* open a file first */
f = open("fname", "w");

/* close the file */
close(f);
```

- `write()` writes to file. Write takes four arguments. The first argument is a string which contains the data to write to file. The second argument is an integer for the length of the string, or how many characters the user wants to write to file. The third argument is an integer for the size of each element. For a character, the size is 1. The final argument is a string pointer for the file. Write will return an integer for the number of elements written. Example:

```
/* open a file first */
f = open("fname", "w");

/* write to file */
write("hello world", 1, 1, f);

/* close the file */
close(f);
```

- `fork()` splits the running process into a child process and a parent process. The `fork()` function will not take any arguments and will return 0 if the current process is the child process and will return the parent process id if the current process is the parent process. The child process will continue its execution after the fork call. Example:

```
pid = fork();
```

- `execlp()` will replace the current line of execution with the called process. `execlp()` takes four arguments (3 strings and 1 integer). The first argument is the process to execute, the second argument is the name of the process, the

third argument will take in an argument, and the final argument will the integer 0 to act as a null terminator. It will -1 if there is an error. Example:

```
execlp("echo", "echo", "hello world", 0);
```

- `sleep()` will sleep for a specified number of seconds. Sleep take an integer for the number of seconds to sleep. Example:

```
/* sleep for 5 seconds */
sleep(5);
```

- `len()` will return the length of a string. It takes one argument, which is a string. Example:

```
A = len("hello"); /* A will be 5 */
```

2.4 Basics

2.4.1 Primitives

ManiT supports the following primitives:
- int
- float
- boolean
- string

2.4.2 Arrays

Since ManiT is to compiled into LLVM, an array of size n and type t is an allocated block of memory that holds n contiguous values all of type t. Arrays are allocated on stack. The values that an array contains must be of the same type.

2.4.3 Structs

Structs are just like in C. The types of the members must be declared in struct definition. All members of a struct are public by default. Unlike array, an instance of a struct type is declared by specifying the struct as the type of the variable.

2.4.4 Operators

Airthmetic: +, -, *, ^, \
Logical: &&, ||
Relational: ==, <, <=, !=, >, >=

2.4.5 Control Flow

ManiT supports standard control flow constructs, such as if-else statements, for loop, and while loops.

3. Language Reference Manual

Introduction

ManiT is a programming language inspired from Python and C that compiles into LLVM. ManiT includes partial type-inference in addition to a number of built-in functions for manipulating files and working with processes.

3.1 Lexical Conventions

This section will describe how ManiT code will be processed and how tokens are generated.

Identifiers:

Identifiers are used to name variables, functions, and user defined struct type names as in programming languages such as C and Python. An identifier can be any letter followed by any sequence of letters, numbers, or underscores. The letters may be either lowercase or capitalized. The set of keywords, which will be listed later, cannot be used as identifiers and are reserved. Here is the regular expression for identifiers in ManiT:

```
Id = ['a'-'z' 'A'-'Z']['a'-'z' 'A'-'Z' '0'-'9' '_']*
```

Float literals consist of an integer part,

Types:

When declaring variables, users will not have to specify types.

Type	**Description**	**Syntax**
Boolean	A one byte number 0 or 1. 0 corresponds to false and 1 corresponds to true.	a = true; b = false;

Integer	Numbers without a decimal point (also known as Integers)	a = 5;
		b = 3;
		c = 89;

Float	Number that contains a decimal point.	a = 5.1;
		b = 6.3;
		c = 89.99909;

String	A sequence of characters inside a pair of “”.	string1 = “hello”; a = “world”;

		test = “plt”;
Array	A container to hold multiple data of the same datatype. An array is denoted by square brackets. [ ].	a = [1.1,2.2]; b = [1,2,3];

Integers are 32-bit signed integers as defined by two’s complement. All integers are utilized within ManiT programs using base 10. Strings are zero or more unicode characters surrounded by double quotes. They utilize at most the same amount of memory as characters in the string. Both integers, floats, and strings are immutable and cannot be changed once initialized. When changing the value of one of these types, underlying storage is collected and replaced. Arrays are mutable and therefore utilize references to their values in storage. When arrays are passed or returned from functions or structs, they are passed by reference.

Lexical Conventions:

When processing a program written in the ManiT language, the program is reduced to a sequence of tokens. There are five classes of tokens: identifiers, keywords, literals, operators, and other separators. In our language, spaces, tabs, newlines, single and multi-line comments are considered to be white-space. In general, white-space is ignored by program. Some white-space, however, is required to separate otherwise adjacent identifiers, keywords and constants.

***Comments*:**

The ManiT language supports multi-line commenting. Comments do not nest, and cannot exist within a string or character literals. Code placed inside the /* and */ are regarded as comments and ignored by the ManiT compiler.

/* ... */	Multi-line comment

***Identifiers*:**

An identifier is a sequence of letters or digits that identifies some data that the programmer will interact with. Identifiers can be any length and use any combination of letters and numbers, but must start with a letter.

Keywords:

ManiT Final Project

Keywords are a set of reserved words that serve a specific purpose in the ManiT language, and may not be used by the user. The list of keywords is as follows:

```
if                      int
else                    string
return                  bool
while                   def
for                     true
void                    false
struct
```

The following keywords are used specifically to specify the type of a value:

```
int                     bool
string                  float
void
```

Each one of these keywords has a specific meaning that will be elaborated upon below.

***Literals*:**

Any sequence of one or more digits in decimal is treated as an integer literal or constant. A negative sign is used to specify a negative integer (-1). The integer literal corresponds to the int type. Special characters such as the newline character can be specified using the backslash (\) character. The following escape sequences may be used in string literals:

Newline	NL	\n
Horizontal tab	HT	\t
Backslash	\	\\\\
Double quote	"	\"
Single quote	'	\'

Integers literals consist a sequence of one or more digits in decimal. Negative integers are prefixed by a negative or a minus sign.

Floating literals consist of a integer part, decimal part, and a fraction part. The integer and fraction portions consists of a sequence of one or more digits. The decimal

portion delimits the integer and fraction portions and is specified using the period character (.). A floating literal may be written as 1.5. Floats may be represented through the following regex:

```
[+-] ? ([0-9] * [.]) ? [0-9] +
```

A string literal is written as a sequence of zero or more ASCII characters or escape sequences surrounded by double quotes. Special characters such as the newline character may be defined using the same escape sequences used for character literals. The string literal is of the string type. The following is a regex to accept string literals in ManiT:

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

Boolean literals are explicitly the identifiers true and false. The former represents the logical true and the latter represents the logical false. These identifiers are reserved.

Separators:

Separators are used in separating tokens. Separators in ManiT language include the following:

```
{}[];,.()
```

Operators:

ManiT consists of the following operators:

Operator	Name	Associativity
=	Assign	Right
==	Equal to	-
!=	Not equal to	-
>	Greater than	-
>=	Greater than or equal to	-
<=	Less than or equal to	-
<	Less than	-
+	Addition	Left

-	Subtraction	Left
*	Multiplication	Left
/	Division	Left
.	Dot/access	Left
&&	Logical AND	Left
||	Logical OR	Left
!	Logical NOT	Right
-	Negative	Right

The precedence for the operators is as follows:

```
.
! - (negative)
* /
+ - (subtraction)
> < >= <=
== !=
&&
||
=
```

Unary Operators:

The negative and logical not operators are unary operators in ManiT.

Expressions:

An expression is composed of one of the following in ManiT:

- One of the literals described in the literals section
- Unary and binary operations between expressions
- Identifier
- Assign expressions
- Function calls
- Array definition
- Array access
- Struct access

Assignment expressions:
Assignment expressions consist of a variable name, assignment operator, and an expression. If the variable name is undeclared inside a given scope, assignment expressions evaluates the expression and declares the variable with the type of the expression, and initializes the variable with the expression. If the variable name has been declared previously inside the scope, assignment expression changes the value of the variable to that of the expression. The entire assignment expression is evaluated to the value of the right-hand-side expression.

Function calls:
Function call consist of a function name, opening parenthesis, comma-separated sequence of actual parameters, and a closing parenthesis. The function of the specified name must be defined prior to the call, and the type and the number of arguments must match those of the function definition.

Array definition:
Array definition consist of an opening bracket, a comma-separated sequence of literals, and a closing bracket. The sequence must consist of at least one literal and each literal in the sequence must have the same type.

Array access:
Array access consist of an outer expression, opening bracket, an index expression, and a closing bracket. The outer expression must evaluate to a previously defined array variable and the index expression must evaluate to an integer between 0 and length of the array minus one. The Array access expression evaluates to the value and the type of the indexed element in the array.

Struct access:
Struct access consist of an expression, a dot operator, and an identifier. The expression must evaluate to a previously defined instance of a struct type, and the identifier must be the name of one of the members of the struct type. Because all members of struct are public by default, all members of a struct can be accessed.

Statements:
ManiT executes all statements in sequence. As ManiT does not allow for classes, a program in ManiT is consisted of a list of statements. These statements are executed in sequence. A statement in ManiT can be any one of the following:

- An expression followed by a semicolon
- A list of statements prefixed by a "`{`" and followed by a "`}`"
- An if statement with an else or without an else
- A For or a while loop
- Struct declarations
- A struct definition

End of Statement:

All statements must be closed by semicolons:
';'

Expression Statement:

An expression statement is an expression followed by a semicolon. An expression statement causes the expression in the statement to be evaluated. The syntax for expression statements are:

expression;

Control Flow Statement:

The `if` statement is used to execute the block of statements in the if-clause when the specified condition is met. If the specified condition is not met, then the statement is skipped over. If there is an else statement, and the condition is not met, then the code within the else statement will be executed.

*If*s and *Elses* are chained to create conditional statements. Expressions are given for each case as well as statements to be executed. If the expression evaluates to true, the statements provided are executed. If the expression evaluates to false, then the else's provided statement is executed is there is an else. Control flow statements may or may not have an else statement. If there is no else statement, then the statements within the if statement will be ignored.

Here is an example of control flow:

```
if (expr) {
     stmts;
}
```

ManiT may also have control flow with else statements:

```
if (expr) {
     stmts;
} else {
     stmts;
```

```
}
```

In order to have multiple conditions, ManiT supports the following:

```
if (expr) {
     stmts;
} else if (expr) {
     stmts;
} else {
     stmts;
}
```

For Statement:

The *for* structure is similar to C and Java. Three expressions must be provided. An initializer, a condition, and an increment. The *for* loop executes until the condition evaluates to false which is evaluated at the beginning of each loop. At the end of each execution, the increment is evaluated. During the loop, the provided statement is executed:

```
for (expr; expr; expr) {
     stmts;
}
```

The first expression is served to initialize the iterator counter. The second expression is the condition for the loop and the final expression is used to increment the iterator counter.

```
for (expressionInit; expressionCondition; expressionIncre){
     stmts;
}
```

While Statement:

While expressions are evaluated after each execution of the provided statement. The statement is executed for as long as the expression evaluates to true:

```
while (expr) {
     stmts;
}
```

Return Statement:

return statements exits functions and returns to the function call. If an expression is given, it is evaluated and then returned.

```
return expr;
```

Function Definitions:

A function definition defines executable code that can be invoked, passing a fixed number of values as formal parameters. ManiT uses the `def` keyword for function definition. After the `def` keyword, ManiT requires the user to pass another keyword to specify the return type. Type keywords include `int, float, bool, string, void`. This is followed by an identifier that serves as the function name. It is followed by a list of formal parameters listed inside parentheses. The body of the function is contained between braces after the list of formal parameters.

A function definition is specified using the following parsing rule:

```
def typ ID ( formal_list ) {
     stmts;
}
```

Each formal in the list must be accompanied by its type. The following is an example function definition:

```
def int function_name (int param_a, int param_b) {
     /* code here */
     return 0;
}
```

Scope:

Scope refers to which variables, functions, and structs are accessible at a given point of a program. Broadly speaking, variables may be declared in four different places:

1. **Local Variables** are declared inside a function or a block using the assignment operator. They can only be used or accessed from within the function or block that they are defined in.
2. **Formal Parameters** declared within a function definition. The scope of formal parameters is limited to the function block in which they are declared. Formal parameters may be used throughout their corresponding functions.
3. **Global Variables** are declared outside all functions and will be available throughout the entire program.
4. **Struct Variables** are specified within a user defined struct type and may be used only with reference to accessing an instance of the user defined struct type.

Struct Definition:

Struct definitions define a user-defined struct type that is used to define an instance of struct. ManiT uses the `struct` keyword to denote the beginning of struct definition. Struct definition contains a non-empty list of variable declarations, which specify the members of a struct. Unlike variable declarations outside of struct definition, the

member variable declarations must specify the type of each member. All members are defined as public by default.

Structs are defined in the following manner:

```
struct_decl:
  ID LBRACE vdecl_list RBRACE SEMI
```

Where `vdecl_list` is defined as follows:

```
vdecl_list:
  vdecl_list vdecl

vdecl:
  any_typ_not_void ID SEMI
```

ManiT uses `struct` keyword

in a similar manner to functions but do not utilize the parenthesis after the identifier. Within the curly braces of the struct is a sequence of member declarations defining the structure. These declarations are the members of the struct.

Struct definitions allow
Structs are declared in the following manner:

```
struct_decl:
  ID LBRACE vdecl_list RBRACE SEMI
```

Where vdecl_list are defined as follows:

```
vdecl_list:
  vdecl_list vdecl

vdecl:
  any_typ_not_void ID SEMI
```

Struct Variable Declarations:
The variable declarations inside each struct must be set with an type identifier (e.g. `int, string, float, ...`). Each of one these type identifiers are reserved keywords.

Variable Declarations/Struct Declarations:

Variable declarations are a statement in the ManiT language. The only type of variable declaration that ManiT allows without assignment for as statements is the declaration of an instance of a user-defined struct type. Struct declarations are specified using the `struct` keyword followed by the user specified identifier to describe the user-defined struct type. After this identifier is a variable identifier, which is used to specify the name of the struct instance. This is specified in the following:

```
STRUCT ID ID;
```

The first identifier is the name of the user-defined struct type that is defined in the struct definition. The second identifier is the name of the struct instance. Here is an example of a struct declaration:

```
/* struct definition */
struct pt {
     int x;
     int y;
};

/* struct declaration */
struct pt a;

/* struct access and assign */
a.x = 1;
a.y = 2;
```

Grammar:

```
program:
  stmts EOF

stmts:
    /* nothing */  { [] }
  | stmts stmt

stmt:
    expr SEMI
  | RETURN SEMI
  | RETURN expr SEMI
  | LBRACE stmts RBRACE
```

```
  | IF LPAREN expr RPAREN stmt %prec NOELSE
  | IF LPAREN expr RPAREN stmt ELSE stmt
  | FOR LPAREN expr_opt SEMI expr SEMI expr_opt RPAREN stmt
  | WHILE LPAREN expr RPAREN stmt
  | func
  | STRUCT struct_decl
  | vdecl

expr_opt:
    /* nothing */ { [] }
  | expr

struct_typ:
  | STRUCT ID

any_typ_not_void:
  | STRING
  | FLOAT
  | INT
  | BOOL
  | struct_typ

any_typ:
  | any_typ_not_void
  | VOID

vdecl_list:
  { [] }
  | vdecl_list vdecl

vdecl:
    any_typ_not_void ID SEMI

func:
   DEF any_typ ID LPAREN formals_opt RPAREN LBRACE stmts RBRACE

struct_decl:
  ID LBRACE vdecl_list RBRACE SEMI
```

```
formals_opt:
    /* nothing */ { [] }
  | formal_list

formal_list:
    any_typ_not_void ID
  | formal_list COMMA any_typ_not_void ID

expr:
    INTLIT
  | FLOATLIT
  | STRINGLIT
  | TRUE
  | FALSE
  | ID
  | expr PLUS   expr
  | expr MINUS  expr
  | expr TIMES  expr
  | expr DIVIDE expr
  | expr EQ     expr
  | expr NEQ    expr
  | expr LT     expr
  | expr LEQ    expr
  | expr GT     expr
  | expr GEQ    expr
  | expr AND    expr
  | expr OR     expr
  | MINUS expr %prec NEG
  | NOT expr
  | LPAREN expr RPAREN
  | ID ASSIGN expr
  | ID LPAREN exprs_opt RPAREN
  | expr DOT ID
  | LBRACK exprs_list RBRACK
  | expr LBRACK expr RBRACK

exprs_opt:
    /* nothing */ { [] }
  | exprs_list
```

```
exprs_list:
    expr
  | exprs_list COMMA expr
```

The grammar uses the following precedence rules:

```
%token SEMI LPAREN RPAREN LBRACE RBRACE COMMA
%token LBRACK RBRACK
%token PLUS MINUS TIMES DIVIDE ASSIGN NOT
%token EQ NEQ LT LEQ GT GEQ TRUE FALSE AND OR
%token RETURN IF ELSE FOR WHILE
%token DEF GLOBAL STRUCT DOT

%token INT FLOAT BOOL STRING VOID
/* token VOID */

/* Literals */
%token <int> INTLIT
%token <float> FLOATLIT
%token <string> STRINGLIT
%token <string> ID
%token EOF

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

4. Project Plan

Overview

As we did not have any prior experience with OCaml, we began our project with MicroC as our base. Hence, our compiler has its basic framework similar to that of MicroC with files such as ast, parser, scanner, semant, and codegen.
After understanding how MicroC worked, we began modifying the code in MicroC to add in features such as structs, arrays, and type inference. Upon finishing each sections of the ManiT, we would compile the code to test the code through writing test code in the test folder. We would then see the output of the test file and kept repeating this method until we were satisfied with the outputs from our language. When our tests did result in errors such as parser error, we would continuously patch our code.

Planning

We set milestones by different subfeatures. We began with basic scanner, parser, and codegen from MicroC. After adding rudimentary semant and SAST for partial type inference, we iteratively added arrays, structs, built_in functions to our language.
Our team met on regular team meetings once a week. The group meeting was every Friday afternoon with the TA, Graham Gobieski.

Challenges and Changes

Our project initially began with the goal to modify any large numbers with ease. However, with the help of our TA, we slowly shifted our plan to recreate a language based off of C and Python.
The main challenge came from redesigning the front end for partial time inference and to parse the program as a list of statements. ManiT does not have a defined entry point (main function) and parses a program as a list of statements, whereas MicroC does not have type inference and parses a program as a tuple of global variables and functions (including main function). As a result, we had to redesign the semantics checker to recursively annotate types to each expression from AST and generate an SAST, while maintaining the semantics checking from MicroC. Additionally, having a list of statements required us to maintain scope of the global and local variables differently. Redesigning much of the front end to tackle these complexities was the main challenge of our project.
The other challenges came from understanding the LLVM interface. For example, using LLVM functions to define and assign variables of string type and to access struct members using the index of the member caused minor difficulties.

Roles and Responsibilities:

ManiT Final Project

Roles:

Manager: Akiva Dollin
System Architect: Seungmin Lee
Language Guru: Irwin Li
Tester: Dong Hyeon Seo

Team Member	Files
Akiva Dollin	Parser, AST, SAST, Semant, CodeGen, Test Suite, Demo
Seungmin Lee	Scanner, Parser, Semant, AST, SAST, CodeGen, Docs
Irwin Li	Scanner, Parser, AST, Semant, SAST, CodeGen, Test Suite, Demo, Docs
Dong Hyeon Seo	Scanner, Parser, AST, Docs

Team Member	Breakdown of Contributions
Akiva Dollin	Initial docs. AST, SAST, Codegen, and Semant for struct declaration, struct assign, and array assign. Test suite lead. Demo. Stress testing. Error handling.
Seungmin Lee	Initial docs. AST, SAST, Codegen, and Semant for all functions, arrays, and struct access. Partial type inference lead. Final docs.
Irwin Li	Initial docs. Demo lead. Stress testing lead. Codegen lead. AST, SAST, Codegen, and Semant for printing, built-in functions, and struct assignment. AST, SAST, Semant bug handling. Final docs.
Dong Hyeon Seo	Initial docs. Initial Scanner, Parser, and AST. Final docs.

Team Responsibilities:

Though we initially split our roles to the project into different sections, we were willing to also work upon different roles throughout the project. We tried our best meeting every Friday with the TA and we initially began the project working together on one or two computers. As we began struggling with the language, we decided to slowly split tasks to pairs or alone to finish the project in time.

Tools Used:

Unix:

All code was ran on UNIX environment machines. All four of us utilized Ubuntu with two using PLT official virtual image through VirtualBox. This was done to ensure hardware consistency in our project.

LLVM:

We used the most recent LLVM that was compatible with our machine for the project.

Github:

Github was utilized to ensure our codes were up to date and to store codes securely throughout the project. However, we were hesitant throughout the semester with pushing our work as much of the code failed to run, and we did not want to modify the master branch until absolutely necessary.

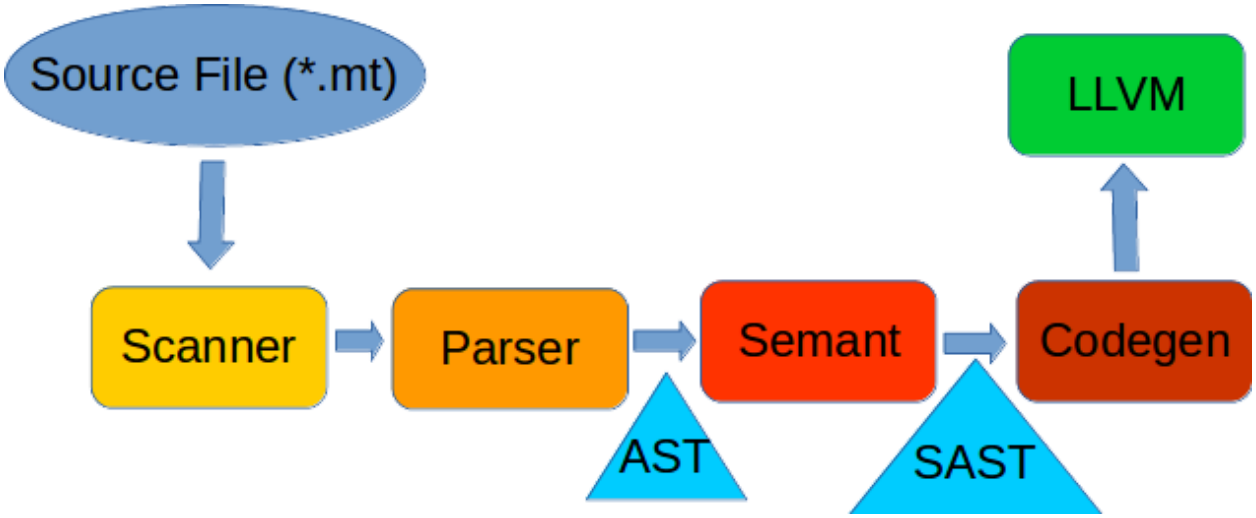
Google Drive:

We stored our project notes from the TA, proposal, LRM, final report, and presentation to Google Drive so that all the team members could both view and edit the above files listed.

Sublime Text / Atom / Vim:

We used these three open source text editors for our codes. We did not feel any standardization for this as they were all just ‘text editors’ at the end of the day.

5. Architectural Design



The figure above is the overview of our Compiler Architectural design.
The ManiT compiler has overall a standard structure that was learned during class. The front end contains the lexical Scanner and parser while the back end contains the rest including Semant and Codegen before the LLVM IR code generation.

Scanner.mll
This is the start of the front end of the ManiT programming language. It reads a ManiT Source File (*.mt) and parses it into tokens.

Parser.mly
This takes in the tokens from the Scanner module and then converts the tokens into an Abstract Syntax Tree (AST). If the Parser recognized the tokens from the Scanner module is not defined, then the Parser terminates as it would then imply the Source File has a syntax error in the code.

Semant.ml
This takes in the AST and then produces and SAST (Semantic Checking AST). The Semant checks for the semantic errors of the program itself. In other words, the Semant checks for errors such as type mismatch, valid function calls, etc.

Codegen.ml
The Codegen, also known as Code Generator, takes in the SAST and translates it into an LLVM IR.

6. Test Plan

Test cases in our compiler can be found in ManiT/src/tests. The test cases test for many of the basics of Java, C, and Python which can be figured out from the test file names. E.g.:

Test file for array access would be named “accessArray1.mt” or any filename with a *.mt file.

The output of the test files are then compared with a *.out file. For instance, “accessArray1.mt” output would be compared to “accessArray1.out”. If the outputs are the same, the test suite prints an “OK”. Otherwise, the test file fails and hence prints “FAILED” to the terminal.

The following file, **testall.sh**, runs all the tests in the ManiT/src/tests folder.
The test files will also be listed in the **Full Code Listing**. This file can be ran through two methods:

1. $ ./testall.sh
2. $ make tests

Testall.sh

```
#!/bin/sh

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

# Path to the LLVM compiler
LLC="llc"

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

# Set time limit for all operations
ulimit -t 30

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

keep=0

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

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

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

```
}

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

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

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

    echo -n "$basename..."

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

    generatedfiles=""

    generatedfiles="$generatedfiles ${basename}.ll ${basename}.s ${basename}.exe
${basename}.out" &&
    Run "$MANIT" "<" $1 ">" "${basename}.ll" &&
    Run "$LLI" "${basename}.ll" ">" "${basename}.out" &&
    Compare ${basename}.out ${reffile}.out ${basename}.diff

    # Report the status and clean up the generated files

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

CheckFail() {
    error=0
    basename=`echo $1 | sed 's/.*\\///
                             s/.mt//'`
    reffile=`echo $1 | sed 's/.mt$//'`
```

```
    basedir="`echo $1 | sed 's/\/[^\/]*$//'`/."

    echo -n "$basename..."

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

    generatedfiles=""

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

    # Report the status and clean up the generated files

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

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

shift `expr $OPTIND - 1`

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

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

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

for file in $files
do
    Check $file 2>> $globallog
done
```

```
rm -f *.ll *.s *.out *.err
exit $globalerror
```

Test Suite Log

The following is the test suite log which contains tests for the various features in our compiler:

Tests

a.mt

```
struct test {
        int tmp;
};

struct test a;
```

a.out

```
```

accessArray1.mt

```
a = [1,23,3];
b = a[0];
c = a[2];
d = a[0];
```

accessArray1.out

```

```

accessArray2.mt

```
def void foo() {
        a = [1,23,3];
        b = a[0];
        c = a[2];
        d = a[0];
}

foo();
```

accessArray2.out

```
```

accessArray3.mt

```
def int foo() {
        a = [1,23,3];
        b = a[0];
        c = a[2];
        d = a[0];
        return d;
}

temp = foo();
print(temp);
```

accessArray3.out

```
1
```

accessArray4.mt

```
def float foo() {
	a = [1.1,1.0,1.000001];
	b = a[0];
	c = a[2];
	d = a[0];
	return d;
}

temp = foo();
```

accessArray4.out

```
```

accessArray5.mt

```
def float foo() {
        arra = [1.0,23.1,0.2];
        e = arra[0];
        g = arra[0];
        arra[0] = 7.9;
        return arra[0];
}

tempFloat = foo();
print(tempFloat);
```

accessArray5.out

```
7.900000
```

add.mt

```
a = 2 + 2;
print(a);
```

add.out

```
4;
```

and.mt

```
if (true && true) {
        print("hi");
}

if (true && false) {
        print("no seen");
}
```

and.out

```
hi
```

array1.mt

```
a = [1,2];
b = [1,2,3,4,5];
```

array1.out

```
```

array2.mt

```
b = [1];
d = [1.1,2.0];
```

array2.out

```
```

array3.mt

```
def void foo() {
        a = [1,2];
}

foo();
```

array3.out

```

```

arrayArray.mt

```
a = [1];
b = [2,1];
c = [1.1,1.2,1.3];
```

arrayArray.out

```
```

arrayDec.mt

```
a = [1,2];
```

arrayDec.out

```

```

arrayMani.mt

```
a = [1,2,3,4];
```

arrayMani.out

```

```

arrayStruct.mt

```
struct test {
        int a;
        string b;
};

struct test temp;

struct c {
        int d;
};

struct c temp2;
```

arrayStruct.out

```
```

arraytest.mt

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

arraytest.out

```

```

arraytest2.mt

```
a = [1,2,3,4];
b = a[3];
print(b);
print(4);
```

arraytest2.out

```
4
4
```

assign.mt

```
a = 4;
b = "hi";
c = true;

print(a);
print(b);
print(c);
```

assign.out

```
4
hi
1
```

assigntest.mt

```
a = 2;
b = a;
```

assigntest.out

```
```

b.mt

```
a = "hello";
```

b.out

```
```

bintest.mt

```
a = 1+1;
b = 1-1;
c = 1*1;
d = 1/1;
```

bintest.out

```
```

bool.mt

```
temp = false;
temp2 = true;
```

bool.out

```
```

boollit.mt

```
a = true;
b = false;
```

boollit.out

```

```

calltest.mt

```
def void foo() {
        a = 2;
        b = "hello";
        if(a == 2) {
                b = "test";
        }
}

foo();
```

calltest.out

```

```

close.mt

```
f = open("tests/close.out", "w");

close(f);
```

close.out

```

```

comments.mt

```
/*Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt
ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco
laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in
voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat
cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.*/

/*test text*/

print("Comment test succesful");
```

comments.out

```
Comment test succesful
```

comments2.mt

```
def int foo() {
        a = [1,23,3];
        b = a[0];
        c = a[2];
        /*Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor
incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud
exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor
in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur
sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est
laborum.*/
        d = a[0];
        return d;
}

temp = foo();
print(temp);
```

comments2.out

```
1
```

comments3.mt

```
def int foo() {
        a = [1,23,3];
        b = a[0];
        c = a[2];
        d = a[0];
        return d;
}

temp = foo();
/*Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt
ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco
laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in
voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat
cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.*/
print(temp);
/*Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt
ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco
laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in
voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat
cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.*/
```

comments3.out

```
1
```

divid.mt

```
a = 4/2;
b = 1/2;

print(a);
print(b);
```

divid.out

```
2
0
```

equal.mt

```
a = 4;
b = 4;
if(a == b) {
        print("OK");
}
```

equal.out

```
OK
```

execlp1.mt

```
execlp("echo", "echo", "hello world\n", 0);
```

execlp1.out

```
hello world
```

execlp2.mt

```
execlp("cat", "cat", "tests/execlp2.out", 0);
```

execlp2.out

```
Test
```

float.mt

```
a = 1.1;
b = 0.01;
c = 11.0;
d = 1.000000000000000001;
```

float.out

```
```

floatlit.mt

```
a = 1.1;
b = 2.1;
c = 0.3;
```

floatlit.out

```
```

for.mt

```
i = 0;
for(i = 0; i<5; i=i+1) {
        a = 4;
}
```

for.out

```

```

forkexec.mt

```
/* fork into execlp */

pid = fork();

if (pid != 0)
{
        execlp("echo", "echo", "hi", 0);
}
```

forkexec.out

```
hi
```

forktest.mt

```
pid = fork();

if (pid != 0) {
        print("forked");
}
```

forktest.out

```
forked
```

funcall.mt

```
def void foo() {
        print("OK");
}

foo();
```

funcall.out

```
OK
```

globalasstest.mt

```
a = 5;
def int foo() {
        a = 6;
        return a;
}

c = foo();
print(c);
```

globalasstest.out

```
6
```

great.mt

```
a = 4;
b = 2;

if( a > b) {
        print("OK");
}
```

great.out

```
OK
```

greator.mt

```
a = 4;
b = 4;

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

greator.out

```
OK
```

helloworld.mt

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

helloworld.out

```
Hello World
```

idtest.mt

```
a = 1;
b = "string";
```

idtest.out

```
```

if.mt

```
if(1 == 1) {
        a = 4;
}
```

if.out

```

```

int.mt

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

int.out

```
```

intlit.mt

```
a = 1;
b = 2;
```

intlit.out

```
```

len.mt

```
s = "hello world";

length = len(s);

print(length);
```

len.out

```
11
```

less.mt

```
a = 4;
b = 2;

if( b < a) {
        print("OK");
}
```

less.out

```
OK
```

lessor.mt

```
a = 2;
b = 2;

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

lessor.out

```
OK
```

ManiT Final Project

math.mt

```
a = 2 + 2;
b = 2 - 2;
c = 2 * 2;
d = 2/2;
e = (2+2) * 2;

print(a);
print(b);
print(c);
print(d);
print(e);
```

math.out

```
4
0
4
1
8
```

minus.mt

```
a = 4;
b = 3;
c = a - b;

if( c == 1) {
        print("OK");
}
```

minus.out

```
OK
```

neg.mt

```
a = -6;
b = 2;

if( a + b == -4) {
        print("OK");
}
```

neg.out

```
OK
```

nequal.mt

```
a = 4;
b = 2;

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

nequal.out

```
OK
```

not.mt

```
a = false;

if(!a) {
        print("OK");
}
```

not.out

```
OK
```

open.mt

```
/* open test */
f = open("tests/open.mt", "r");
close(f);
print("opened");
```

open.out

```
opened
```

or.mt

```
a = 2;
b = 3;

if(a == 3 || b == 3) {
        print("OK");
}
```

or.out

```
OK
```

print.mt

```
print("Hello");
print(2);
print(true);
print(false);
a = 2;
b = 2.2;
c = true;
d = false;
print(a);
print(c);
print(d);
```

print.out

```
Hello
2
1
0
2
1
0
```

sleep.mt

```
sleep(1);

print("slept");
```

sleep.out

```
slept
```

stringlit.mt

```
a = "string";
```

stringlit.out

```
```

strings.mt

```
a = "1";
b = "12";
c = "123";
d = "1234";
e = "12345";
f = "123456";
g = "1234567";
h = "12345678";
i = "123456789";
j = "1234567890";
k = "123456789010";
l = "a";
m = "aa";
n = "aaaaaaaaaaaaaaaa";
o = "q1q1q1q1q1q1q1q1q1q";
p = "1.01.101.10.10.10.10''s";
```

strings.out

```

```

struct1.mt

```
struct hello {
        int a;
        string b;
        float c;
        int d;
        string e;
};

struct temp {
        int a;
};

struct temp1 {
        int a;
};

struct temp b;

struct temp1 c;
```

struct1.out

```

```

struct2.mt

```
struct hello {
        int a;
        string b;
        float c;
        int d;
        string e;
};

struct temp {
        int a;
};

struct temp1 {
        int a;
};
```

struct2.out

```
```

struct3.mt

```
struct hello {
        int a;
        string b;
        float c;
        int d;
        string e;
};

struct temp {
        int a;
};

struct temp1 {
        int a;
};

struct temp b;

struct temp1 c;

b.a = 4;
print(b.a);
```

struct3.out

```
4
```

structAccess.mt

```
struct test {
        int a;
};

struct test tester;
b = tester.a;
```

structAccess.out

```

```

structArray.mt

```
struct test {
        int a;
        string b;
};

struct test temp;

struct c {
        int d;
};

struct c temp2;

temp2.d = 10;

a = temp2.d;
```

structArray.out

```

```

structCreate.mt

```
struct test {
        int a;
};

struct test tester;
```

structCreate.out

```

```

structDec.mt

```
struct test {
        int a;
        float b;
        string c;
        bool e;
};
```

structDec.out

```
```

structStruct.mt

```
struct test {
        int a;
        string b;
};

struct test temp;
```

structStruct.out

```

```

structtest.mt

```
struct tester {
        int a;
        string c;
        bool d;
};

struct tester b;
b.a = 1;
```

structtest.out

```

```

structtype.mt

```
struct a {
        int b;
};
```

structtype.out

```
```

ManiT Final Project

textPrint.mt

```
a = " _______ _________ _______  _______          _______  _         _______  ______
_______  _______  _______  _______";
b = "(  ____ \\\\\\\\\__   __/(  ____ \\\\\(  ____ )|\\\\     /|(  ____ \\\\\( (    /|  (  ____
\\\\\(  __  \\\\ |\\\\     /|(  ___  )(  ____ )(  __  \\\\ (  ____ \\\\\";
c = "| (    \\\\\/   ) (   | (    \\\\\/| (    )|| )   ( || (    \\\\\/|  \\\\\  ( |  | (
\\\\\/| (  \\\\\  )| )   ( || (   ) || (    )|| (  \\\\\  )| (    \\\\\/";
d = "| (_____    | |   | (__    | (____)|| (___) || (__    |   \\\\\ | |  | (__    | |   ) ||
| _ | || (___) || (____)|| |   ) || (_____";
e = "(_____  )   | |   |  __)   |  _____)|  ___  ||  __)   | (\\\\\ \\\\\) |  |  __)   | |   |
|| |( )| ||  ___  ||     __)| |   | |(_____  )";
f = "      ) |   | |   | (      | (      | (   ) || (      | | \\\\\   |  | (      | |   ) ||
|| || || (   ) || (\\\\\ (   | |   ) |      ) |";
g = "/\\\\\____) |   | |   | (____/\\\\\| )      | )   ( || (____/\\\\\| )  \\\\\  |  |
(____/\\\\\| (__/  )| () () || )   ( || ) \\\\\ \\\\\__| (__/  )/\\\\\____) |";
h = "\\\\\_______)   )_(   (_______/|/       |/     \\\\\|(_______/|/    )_)
(_______/(______/ (_______)|/     \\\\\||/   \\\\\__/(______/ \\\\\_______)";

print(a);
print(b);
print(c);
print(d);
print(e);
print(f);
print(g);
print(h);
```

textPrint.out

```
_______ _________ _______  _______          _______  _         _______  ______
_______  _______  _______  _______
(  ____ \\__   __/(  ____ \(  ____ )|\     /|(  ____ \( (    /|  (  ____ \(  __  \ |\
/|(  ___  )(  ____ )(  __  \ (  ____ \
| (    \/   ) (   | (    \/| (    )|| )   ( || (    \/|  \  ( |  | (    \/| (  \  )| )   (
|| (   ) || (    )|| (  \  )| (    \/
| (_____    | |   | (__    | (____)|| (___) || (__    |   \ | |  | (__    | |   ) || | _ |
|| (___) || (____)|| |   ) || (_____
(_____  )   | |   |  __)   |  _____)|  ___  ||  __)   | (\ \) |  |  __)   | |   | || |( )|
||  ___  ||     __)| |   | |(_____  )
      ) |   | |   | (      | (      | (   ) || (      | | \   |  | (      | |   ) || || ||
|| (   ) || (\ (   | |   ) |      ) |
/\____) |   | |   | (____/\| )      | )   ( || (____/\| )  \  |  | (____/\| (__/  )| () ()
|| )   ( || ) \ \__| (__/  )/\____) |
\_______)   )_(   (_______/|/       |/     \|(_______/|/    )_)  (_______/(______/
(_______)|/     \||/   \__/(______/ \_______)
```

times.mt

```
a = 4;
b = 2;
c = a * b;
print(c);
```

times.out

```
8
```

typeinfer.mt

```
i = 1;
f = 1.11;
s = "String Test";
bt = true;
bf = false;

print(i);
print(s);
print(bt);
print(bf);
```

typeinfer.out

```
1
String Test
1
0
```

uoptest.mt

```
a = 1;
b = a+1;

c = 2;
d = c-1;
```

uoptest.out

```

```

while.mt

```
i = 0;
while(i < 10) {
        i = i+1;
}
```

while.out

```
```

write.mt

```
/* write test */

f = open("tests/write.out", "w");

/* first argument is the data, */
write("hello", 5,1,f);
close(f);

print("hello");
```

write.out

```
hello
```

7. Conclusions
Lessons Learned

Akiva Dollin

As Professor Edwards states in class, OCaml sucks until you understand it. As Professor Edwards states in class, start early. Professor Edwards knows what he's talking about. The most difficult part of this class is getting to the point where Ocaml makes sense. This cannot be done by looking at past semesters code. This cannot be done by asking the TA for help. The only way to accomplish this is by sitting down and failing miserably for a few weeks. After this period, everything is much more manageable. Additionally, starting early will save your life. If we had the chance to do it over again, we would complete the basic functionality at least a month before the deadline. This would allow us to spend much more time on the additional functionality.

To conclude, OCaml is actually pretty cool.

Seungmin Lee

Whereas large design choices should be made carefully with foresight, features should be added incrementally by testing small bits at a time. Because ManiT has type inference and parses the program as a list of statements, the architectural design of the compiler had to be changed significantly from MicroC. Because we had initially made some invalid assumptions regarding the design of our compiler, we had to redesign large portions of the codebase to accommodate the design changes multiple times throughout the project. On the other hand, I often tried to add multiple subfeatures and test them at once using more general test case for faster development. Ironically, this significantly lengthened the debugging process, especially because I was less familiar with Ocaml and LLVM errors throughout the earlier half of the semester. Consequently, I believe that I learned an important lesson about software design in general; first understand and design the layers correctly with foresight and then fill in the layers incrementally by utilizing test suites that test most specific behaviors.

Irwin Li

Something that we really struggled with as a group was time management. Every member of our group had difficult course loads and it was very difficult to schedule meetings where we could all meet. We needed to schedule regular meeting times so that we could proceed smoothly through the project. The benchmarks that Professor Edwards set are really meant to be met. A lot of the information related to building a compiler have to be self-learned. Once you get the micro-C compiler, you need to start immediately. While the class may help you learn some of the information, the majority of

the project is self contained and largely unrelated to the course. The class is intended to give you general information related to languages and compilers. The project is intended to make you learn the specifics.

Another problem that we struggled with was communication. We used facebook as our communication platform. Although we were able to organize some meetings in the end, I soon realized that facebook was not the most reliable platform. Not all of our members would actively check for facebook messages, and sometimes group members would be unresponsive.

Finally, I learned that you should not be afraid to move away from the structure of micro-C. I believe that Professor Edwards intended for us to use micro-C to help understand how compilers may be built in the general sense. The structure and conventions set by the micro-C compiler like are helpful, but that doesn't mean that you can blindly copy the code and expect results. Building a compiler is learned.

Dong Hyeon Seo

OCaml has a high learning curve without a doubt. Unlike many other languages like Java, it is a language that takes time for the language to be somewhat intuitive. However, the language seems to be powerful with the right approach; the ability to pattern match while working recursively in a function was something I had not experienced in coding with other languages like Java and C.

That said, as of the project itself, I felt it was a humbling experience. I got to work with phenomenal team members who were greatly supportive throughout the semester. And as tough as the project itself was at times, we managed to pull off towards our goal.

A few notes I would have liked to message my pre-PLT semester project deadline self are: setting earlier deadline and learning to utilize Github more efficiently. Deadlines I noticed could not always be kept when utilizing a programming language that had great differences from many of the common C based languages. I felt that creating earlier deadlines could have been better due to this as typing codes with a language I was not familiar with could imply that finishing certain parts of the project would take longer than anticipated. As for Github usage, I felt that I lacked the necessary knowledge to be more efficient with it. Merge conflicts and branches on top of branches were quite a nuisance in the beginning of the project and I felt that having the knowledge of Github prior to this course could have saved me some headache.

Having concluded these notes, I came to have more respect towards the concept of compilers. The notion of having an orderly system of Scanner, Parser, Semantic Checker, and Code Generator to compile a code into LLVM was definitely a neat surprise. Overall, I have enjoyed the course and am thankful to have had the opportunity to work with these awesome team members throughout the project.

ManiT Final Project

Full Code Listing:

Scanner.mll

```
(* Ocamllex scanner for ManiT *)

{
    open Parser
    let unescape s = Scanf.sscanf ("\"" ^ s ^ "\"") "%S%!" (fun x -> x)
}

let digit = ['0'-'9']
let digits = digit+


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

let float = ['+' '-']? (digits '.' ['0'-'9']* | '.' digits) (['e' 'E'] (['+' '-']? digits))?


rule token = parse
(* recursive call to eat white space *)
  [' ' '\t' '\r' '\n'] { token lexbuf } (* Whitespace *)
| "/*"     { comment lexbuf }           (* Comments *)

| '('      { LPAREN }
| ')'      { RPAREN }
| '{'      { LBRACE }
| '}'      { RBRACE }
| ';'      { SEMI }
| ','      { COMMA }
| '['      { LBRACK }
| ']'      { RBRACK }

(* Operators *)
| '+'      { PLUS }
| '-'      { MINUS }
| '*'      { TIMES }
| '/'      { DIVIDE }
| '='      { ASSIGN }
| "=="     { EQ }
| "!="     { NEQ }
| '<'      { LT }
| "<="     { LEQ }
| ">"      { GT }
| ">="     { GEQ }
| "&&"     { AND }
```

```
| "||"     { OR }
| "!"      { NOT }
| "."      { DOT }


(* branch control *)
| "if"     { IF }
| "else"   { ELSE }
| "for"    { FOR }
| "while"  { WHILE }
| "return" { RETURN }

(* half-way type inf *)
| "int"    { INT }
| "bool"   { BOOL }
| "float"  { FLOAT }
| "string" { STRING }
| "void"   { VOID }

| "def"    { DEF }    (* keyword for func decl. see parser.*)
| "global" { GLOBAL } (* keyword for global assignment. see python *)
| "struct" { STRUCT }

(* Literals for each type.
Order matters if same token matches two regexes
need regex for types: char, float, array,
*)
| "true"   { TRUE }
| "false"  { FALSE }
| string   { STRINGLIT(unescape s) }
| digits as lxm { INTLIT(int_of_string lxm) }
| float as lxm { FLOATLIT (float_of_string lxm) }
| ['a'-'z' 'A'-'Z']['a'-'z' 'A'-'Z' '0'-'9' '_']* as lxm { ID(lxm) }
| eof { EOF }
| _ as char { raise (Failure("illegal character " ^ Char.escaped char)) }

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

Parser.mly

```
/* Ocamlyacc parser for ManiT */

%{
    open Ast;;
    let unescape s = Scanf.sscanf ("\"" ^ s ^ "\"") "%S%!" (fun x -> x)
```

```
%}

%token SEMI LPAREN RPAREN LBRACE RBRACE COMMA
%token LBRACK RBRACK
%token PLUS MINUS TIMES DIVIDE ASSIGN NOT
%token EQ NEQ LT LEQ GT GEQ TRUE FALSE AND OR
%token RETURN IF ELSE FOR WHILE
%token DEF GLOBAL STRUCT DOT

%token INT FLOAT BOOL STRING VOID
/* token VOID */

/* Literals */
%token <int> INTLIT
%token <float> FLOATLIT
%token <string> STRINGLIT
%token <string> ID
%token EOF

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

%start program
%type <Ast.program> program

%%

program:
  stmts EOF { List.rev $1 }

stmts:
    /* nothing */  { [] }
  | stmts stmt { $2 :: $1 }

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

```
  | FOR LPAREN expr_opt SEMI expr SEMI expr_opt RPAREN stmt
     { For($3, $5, $7, $9) }
  | WHILE LPAREN expr RPAREN stmt { While($3, $5) }
  | func { Func($1) }
  | STRUCT struct_decl { Struc($2) }
  | vdecl { Vdecl($1) }

expr_opt:
  expr          { $1 }

struct_typ:
  | STRUCT ID { $2 }

any_typ_not_void:
  | STRING  { String }
  | FLOAT   { Float }
  | INT     { Int }
  | BOOL    { Bool }
  | struct_typ    { Struct_typ($1) }

any_typ:
  | any_typ_not_void { $1 }
  | VOID  { Void }

/*only used for structs*/
vdecl_list:
  { [] }
  | vdecl_list vdecl { $2 :: $1 }

vdecl:
    any_typ_not_void ID SEMI { ($1, $2) }

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

struct_decl:
  ID LBRACE vdecl_list struct_fdecls RBRACE SEMI
  { { sname = $1;
      vdecls = List.rev $3;
      fdecls  = List.rev $4 } }

struct_fdecls:
    /* nothing */ { [] }
  | func struct_fdecls { $1::$2 }
```

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

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

expr:
    INTLIT          { IntLit($1) }
  | FLOATLIT        { FloatLit($1) }
  | STRINGLIT   { StringLit(unescape $1) }
  | TRUE             { BoolLit(true) }
  | FALSE            { BoolLit(false) }
  | ID               { Id($1) }
  | expr PLUS   expr { Binop($1, Add,   $3) }
  | expr MINUS  expr { Binop($1, Sub,   $3) }
  | expr TIMES  expr { Binop($1, Mult,  $3) }
  | expr DIVIDE expr { Binop($1, Div,   $3) }
  | expr EQ     expr { Binop($1, Equal, $3) }
  | expr NEQ    expr { Binop($1, Neq,   $3) }
  | expr LT     expr { Binop($1, Less,  $3) }
  | expr LEQ    expr { Binop($1, Leq,   $3) }
  | expr GT     expr { Binop($1, Greater, $3) }
  | expr GEQ    expr { Binop($1, Geq,   $3) }
  | expr AND    expr { Binop($1, And,   $3) }
  | expr OR     expr { Binop($1, Or,    $3) }
  | MINUS expr %prec NEG { Unop(Neg, $2) }
  | NOT expr         { Unop(Not, $2) }
  | LPAREN expr RPAREN { $2 }
  | expr ASSIGN expr   { Assign($1, $3) }
  | ID LPAREN exprs_opt RPAREN { Call($1, $3) }
  /* structs and arrays */
  | expr DOT ID { Struct_access($1, $3) }
  | LBRACK exprs_list RBRACK { Array_create($2) }
  | expr LBRACK expr RBRACK { Array_access($1,$3) }

exprs_opt:
    /* nothing */ { [] }
  | exprs_list  { List.rev $1 }

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

Ast.ml

```
(* Abstract Syntax Tree. Contains Ocaml types so that parser can generate these types from
tokens *)

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

type uop = Neg | Not

type typ = Int | Bool | Float | String | Void | Struct_typ of string | Array_typ of typ *
int

type bind = typ * string

type expr =
    IntLit of int
  | FloatLit of float
  | BoolLit of bool
  | StringLit of string
  | Id of string
  | Binop of expr * op * expr
  | Unop of uop * expr
  | Assign of expr * expr
  | Call of string * expr list (*fname and actuals*)
  | Array_create of expr list
  | Array_access of expr * expr
  | Struct_access of expr * string

type stmt =
    Block of stmt list
  | Expr of expr
  | Return of expr
  | If of expr * stmt * stmt
  | For of expr * expr * expr * stmt
  | While of expr * stmt
  | Func of func
  | Struc of strc
  | Vdecl of bind

and
func = {
    typ : typ;
    fname : string;
    formals : (typ * string) list;
    body : stmt list;
}

and
strc = {
  sname : string;
```

```
  vdecls : bind list;
  fdecls : func list;
}

type program = stmt list
```

Semant.mll

```
(* Semantic checking for the ManiT compiler.
Checks semantics of AST and returns SAST. *)

open Sast
module A = Ast
module StringMap = Map.Make(String)

let built_in = [("print", A.String, A.Int)]

let global_env = {
(*     define standard library functions*)
    funcs = [
        { typ = A.String; fname = "open"; formals = [(A.String,"a"); (A.String,"b")]; body =
[] };
        { typ = A.Int; fname = "write"; formals = [(A.String,"a"); (A.Int,"b"); (A.Int,"c");
(A.String,"d")]; body = [] };
        { typ = A.String; fname = "fgets"; formals = [(A.String,"a"); (A.Int,"b");
(A.String,"c")]; body = [] };
        { typ = A.Int; fname = "len"; formals = [(A.String,"a")]; body = [] };
        { typ = A.Int; fname = "close"; formals = [(A.String,"a")]; body = [] };
        { typ = A.Int; fname = "fork"; formals = []; body = [] };
        { typ = A.Int; fname = "sleep"; formals = [(A.Int,"a")]; body = [] };
        { typ = A.String; fname = "execlp"; formals = [(A.String,"a"); (A.String,"b");
(A.String,"c");(A.Int,"d")]; body = [] };
    ]
}


let structs_hash:(string, A.strc) Hashtbl.t = Hashtbl.create 10

(* whether t2 is assignable to t1. Add rules as necessary *)
let is_assignable t1 t2 = match t1, t2 with
      t1, t2 when t1 = t2 -> true
      (* add tables *)
    | _ -> false
(* let is_assignable t1 t2 = if t1 = t2 then true else false *)

let all_the_same = function
```

```
    [] -> true
  | lst -> let hd = (List.hd lst) in List.for_all ((=) hd) lst

(* finds var in scope *)
let rec find_var scope name = try
  (*List.find ('a -> bool) -> a' list
    finds first element in a' list that satisfies predicate (a' -> bool) *)
  List.find (fun (s, _) -> s = name) scope.variables with Not_found ->
  (*if not found in our scope, try parent's scope or raise not found *)
  match scope.parent with
    (* parent is also a scope. if parent is None, do nothing. *)
      Some(parent) -> find_var parent name
    | _ -> raise Not_found

let find_built_in name = try
  List.find (fun (id, _ , _ ) -> id = name) built_in with Not_found -> raise Not_found

let find_func name = try
  List.find (fun f -> f.fname = name) global_env.funcs with Not_found -> raise Not_found

let exist_func name = try
  ignore(List.find (fun f -> f.fname = name) global_env.funcs); true with Not_found -> false

let check_duplicate_struct strctName =
(* Hashtbl.find structs_hash strct; true with Not_found -> false *)
  try ignore(Hashtbl.find structs_hash strctName); true with Not_found -> false

(* Helper function to check for dups in a list *)
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)

(*check_expr: core type-matching function that recursively annotates type of each expr. *)
let rec check_expr (env : environment) = function
  (* literals *)
    Ast.IntLit(value) -> IntLit(value), A.Int
  | Ast.FloatLit(value) -> FloatLit(value), A.Float
  | Ast.StringLit(value) -> StringLit(value), A.String
  | Ast.BoolLit(value) -> BoolLit(value), A.Bool

  (* Variable access *)
  | Ast.Id(name) -> let (name, typ) = try find_var env.scope name with
      Not_found -> raise (Failure("undeclared identifier! " ^ name)) in
      Id(name), typ

  (* Assignment(string, expr)
```

```
  checks expr of R.H.S, and compares type of expr to that of L.H.S from its declaration.
  populates scope's variable if not found. *)
  | Ast.Assign(lhs, expr) ->
    let check = match lhs with
      A.Id(name) ->
        let (expr, right_typ) = check_expr env expr in (* R.H.S typ *)
        let sast_assign = (* (n, (e, e's typ)), n's typ *)
        try let (name, left_typ) = find_var env.scope name in
          if left_typ <> right_typ (* type mismatch. depends on rule. *)
          then raise (Failure (" type mismatch "))
          else Assign((Id(name), left_typ), (expr, right_typ)), right_typ
        with Not_found -> (* new name. declaration. *)
          let decl = (name, right_typ) in
          env.scope.variables <- (decl :: env.scope.variables);
          Assign((Id(name), right_typ), (expr, right_typ)), right_typ
        in sast_assign
      | A.Array_access( _, _ ) ->
        let (expr, right_typ) = check_expr env expr in (* R.H.S typ *)
        let (arr, left_typ) = check_expr env lhs in
          if left_typ <> right_typ (* type mismatch. depends on rule. *)
          then raise (Failure (" type mismatch in array assign"))
          else Assign((arr, left_typ), (expr, right_typ)), right_typ
      | A.Struct_access( _, _ ) ->
        let (expr, right_typ) = check_expr env expr in (* R.H.S typ *)
        let (strct, left_typ) = check_expr env lhs in
          if left_typ <> right_typ (* type mismatch. depends on rule. *)
          then raise (Failure (" type mismatch in struct assign"))
          else Assign((strct, left_typ), (expr, right_typ)), right_typ
      | _ -> raise(Failure("Not a valid assign: We only allow id, struct, and array
assign"))

    in check
  (* Binop(expr, op, expr)
  checks types of L.H.S and R.H.S*)
  | Ast.Binop (e1, op, e2) ->
    let e1 = check_expr env e1
    and e2 = check_expr env e2 in

    let _, t1 = e1
    and _, t2 = e2 in

    let binop_type t1 op t2 = match op with
        A.Add | A.Sub | A.Mult | A.Div ->
        ( match t1, t2 with
          A.Int, A.Int -> A.Int
        | A.Float, A.Float -> A.Float
        | _, _ -> raise(Failure("binary op type mismatch")))
      | A.Less | A.Leq | A.Greater | A.Geq ->
        (match t1, t2 with
```

```
          A.Int, A.Int -> A.Bool
        | A.Float, A.Float -> A.Bool
        | _, _ -> raise(Failure("binary op type mismatch")))
      | A.Equal | A.Neq ->
        (match t1, t2 with
          A.Int, A.Int -> A.Bool
        | A.Float, A.Float -> A.Bool (* float comparison ok? *)
        | A.Bool, A.Bool -> A.Bool
        | _, _ -> raise(Failure("binary op type mismatch")))
      | A.And | A.Or ->
        (match t1, t2 with
          A.Bool, A.Bool -> A.Bool
        | _, _ -> raise(Failure("binary op type mismatch")))

    in let typ = binop_type t1 op t2 in
    Binop(e1, op, e2), typ

  (* Unop(uop, expr)
  uop is either Neg or Not*)
  | Ast.Unop(uop, e) ->
      let (e, typ) = check_expr env e in (
      match uop with
        A.Neg ->
          (if typ != A.Int && typ != A.Float
          then raise(Failure("unary minus opeartion does not support this type ")));
          Unop(uop, (e, typ)), typ
      | A.Not ->
          (if typ != A.Bool
          then raise(Failure("unary not operation does not support this type ")));
          Unop(uop, (e, typ)), typ
      )

  (* Function Call *)
  | Ast.Call(name, actuals) -> (
    (* check types to each actuals and get types of formals from fdecl. *)
    let typed_actuals = List.map (fun e -> (check_expr env e)) actuals in
    match name with
      | "print" -> Call("print", typed_actuals), A.Int
      | _ -> (* non-print functions *) (
        let func = try find_func name with Not_found ->
          raise(Failure("undefined function was called.")) in

        let match_types formals actuals = match formals, actuals with
          | (ftyp, _) :: _, ( _ , atyp) :: _ ->
            if not(is_assignable ftyp atyp) then raise(Failure("typ of actuals do not match
those of formals"));
            if not(List.length formals = List.length actuals) then
            raise(Failure("number of actuals and formals do not match")); ()
          | _, _ -> if not(List.length formals = List.length actuals) then
```

```
                    raise(Failure("number of actuals and formals do not match"))

        in match_types func.formals typed_actuals;
        Call(name, typed_actuals), func.typ (* return name and f_typ from fdecl *)
      ))
  | A.Struct_access(var_expr, attr) ->
      (* convert to str *)
      let var = (match var_expr with
          A.Id(s) -> s
        | _ -> raise(Failure("struct access: complex vars not supported as of now."))) in

      let (name,temp) = find_var env.scope var in  (* find the instance of struct that was
declared in current scope *)
      let structName = match temp with
        A.Struct_typ(typTemp) -> typTemp
        | _ -> raise(Failure("unknown struct access")) in
      let strc = Hashtbl.find structs_hash structName in (* find strc type definition *)
      let (typ, _) = List.find (fun (_, id) -> id = attr) strc.A.vdecls in (* typ of attr *)
      (* find index of attr in struct. this index is used in codegen *)
      let rec index_of_list x l = (match l with
          hd::tl -> let (_,id) = hd in if id = x then 0 else 1 + index_of_list x tl
        | _ -> raise(Failure("index_of_list failed"))) in
      let index = index_of_list attr strc.A.vdecls in
      Struct_access(name, attr, index), typ

  | A.Array_create(expr_list) ->
      let length = List.length expr_list in
      let checked_expr_list = List.map (fun expr -> check_expr env expr) expr_list in
      let typs = List.map (fun (_,typ) -> typ) checked_expr_list in
      (match all_the_same typs with
        false -> raise(Failure("typs elements in array are not coherent"))
      | true -> Array_create(checked_expr_list), Ast.Array_typ(List.hd typs, length))

  | A.Array_access(var_expr, index_expr) ->
      (* convert to str *)
      let var = (match var_expr with
          A.Id(s) -> s
        | _ -> raise(Failure("array access: complex vars not supported as of now."))) in
      (* find var first *)
      let (var, temp) = try find_var env.scope var with Not_found ->
        raise(Failure("array variable not found")) in

      let (var_typ1, length1) = match temp with
        A.Array_typ(var_typ, length) -> var_typ, length
        | _ -> raise(Failure("unknown array access")) in

      (* need to check if arr typ *)
      (* check index expr *)
      let (index_expr, index_expr_typ) = check_expr env index_expr in
```

```
      if index_expr_typ != A.Int then raise(Failure("array access requires int arg"))
      (* need separate function to evaluate the expr.*)
      else match index_expr with
          IntLit(index) ->
            if index < 0 || index > length1 - 1
            then raise(Failure("access out of bounds"))
            else Array_access(var, index), var_typ1
        | _ -> raise(Failure("array access: only int lit allowed for now"))

(* gets return types from checked stmts with typed expressions *)
let rec get_return_types typ_list stmt = match stmt with
    Return((_, t)) -> t :: typ_list
  | Block(sl) -> List.fold_left get_return_types typ_list sl
  | If(_, s1, s2) -> (get_return_types typ_list s1) @ (get_return_types typ_list s2)
  | While (_, s) -> (get_return_types typ_list s)
  | For (_, _, _, s) -> (get_return_types typ_list s)
  | _ -> typ_list

(* checks typ of func from fdecl with those in fbody *)
let check_return_types func_typ func_body =
  let ret_typs = List.fold_left get_return_types [] func_body in
  List.iter (fun each_ret_typ -> (if (each_ret_typ != func_typ)
  then raise(Failure("return types in fbody do not match with fdecl")));  ) ret_typs

(* check_stmt *)
let rec check_stmt env = function
  | Ast.Expr(e) -> Expr(check_expr env e)
  | Ast.Return(e) -> Return(check_expr env e)
  | Ast.Block(stmtlist) ->
    (* sets a new scope to scope passed in *)
    let new_scope = { parent = Some(env.scope); variables = [] } in
    let new_env = { scope = new_scope } in
    (* populates variables and annotates exprs by calling check_stmt *)
    (* adds new env to all stmts *)
    let stmtlist = List.map (fun s -> check_stmt new_env s) stmtlist in
    (* setting to *)
    new_env.scope.variables <- List.rev new_scope.variables;
    Block(stmtlist) (* new_env *)

  (* Func.
  checks env. checks if all return types match with fdecl. adds fdecl to env. *)
  | Ast.Func(func) ->
    (* add fdecl to global env if it hasn't declared previously. *)
    ( match exist_func func.A.fname with
      false ->
        (* make new scope and env with formals *)
        let flipped_formals = List.map (fun (t, id) -> (id, t)) func.A.formals in
        let new_scope = { parent = Some(env.scope); variables = flipped_formals } in
        let new_env = { scope = new_scope } in
```

```
        (* iterate thru stmtlist like a block *)
        let sast_fbody = List.map (fun stmt -> check_stmt new_env stmt) func.A.body in
        let sast_func = { typ = func.A.typ; fname = func.A.fname; formals = func.A.formals;
body = sast_fbody } in
        global_env.funcs <- (sast_func :: global_env.funcs);
        (* check return typs within fbody and ftype. calls check_expr again on return stmts.
*)
        check_return_types func.A.typ sast_fbody;
        Func(sast_func)
    | true -> raise(Failure("cannot redeclare function with same name")); )
  (* struct stmt *)
  | Ast.Struc(strc) ->
    (*  ignore(check_duplicate_struct strc); *)
    (match check_duplicate_struct strc.A.sname with
      false ->
        ignore(report_duplicate (fun n -> "duplicate struct field " ^ n) (List.map (fun n ->
snd n) strc.A.vdecls));
        let struct_sast = { sname = strc.A.sname; vdecls = strc.A.vdecls } in
        Hashtbl.add structs_hash strc.A.sname strc;
        Struc(struct_sast)
    | true -> raise(Failure("cannot redeclare struct with same name")))

  | Ast.Vdecl(typ, name) ->
    (match typ with (* check if struct typ *)
      A.Int |A.Bool|A.Float|A.String| A.Void|A.Array_typ(_,_)-> raise(Failure("ManiT is type
inferred"))
      | A.Struct_typ(strc_name) ->
        try ignore(find_var env.scope name); raise(Failure("Cannot redeclare an existing
variable name!")) with
        Not_found ->
            (match check_duplicate_struct strc_name with (* check if struct typ exists *)
              true -> (* add to scope variables and return Vdecl *)
                let decl = (name, typ) in
                env.scope.variables <- (decl :: env.scope.variables);
                Vdecl(typ,name)
            | false -> raise(Failure("Struct not declared!")))
    |_-> raise(Failure("ManiT is type inferred, you dun messed up!")))


  (* conditionals *)
  | Ast.If(e, s1, s2) ->
      let (e, typ) = check_expr env e in
      (if typ != A.Bool then raise (Failure ("If stmt does not support this type")));
      If((e, typ), check_stmt env s1, check_stmt env s2)
  | Ast.While(e, s) ->
      let (e, typ) = check_expr env e in
      (if typ != A.Bool then raise (Failure ("While stmt does not support this type")));
      While((e, typ), check_stmt env s)
  | Ast.For(e1, e2, e3, s) ->
```

```
      let (e1, typ1) = check_expr env e1 (*need to have empty expr *)
      and (e2, typ2) = check_expr env e2
      and (e3, typ3) = check_expr env e3 in
      (if typ2 != A.Bool then raise(Failure("For stmt does not support this type")));
      For((e1, typ1), (e2, typ2), (e3, typ3), check_stmt env s)

(* environment is a record with scope and return type.
scope is subrecord with parent and variables.*)
let init_env =
  let init_scope  = {
    parent = None;
    variables = [];
  }
  in { scope = init_scope; }

(* outter-most function that is called in manit.ml
in: (bind_global list, functions, statements)
out: same triple in SAST types, semantically checked.
*)

let check_program program =
  let env = init_env in
  List.map (fun stmt -> check_stmt env stmt) program
```

Sast.ml

```
(* semantically checked AST.
semant.ml takes AST and produces SAST while checking semantics
*)

type bind = Ast.typ * string

type expr_det =
    IntLit of int
  | FloatLit of float
  | BoolLit of bool
  | StringLit of string
  | Id of string
  | Binop of expr_t * Ast.op * expr_t
  | Unop of Ast.uop * expr_t
  | Call of string * expr_t list
  | Assign of expr_t * expr_t
  | Array_create of expr_t list  (* Ast.typ holds length info *)
  | Array_access of string * int (* var, index *)
  | Struct_access of string * string * int (* var, attr, index, type *)

  and expr_t = expr_det * Ast.typ (* typ comes first to match use in codegen *)
```

```
type stmt_t =
    Block of stmt_t list
  | Expr of expr_t
  | Return of expr_t
  | If of expr_t * stmt_t * stmt_t
  | For of expr_t * expr_t * expr_t * stmt_t
  | While of expr_t * stmt_t
  | Func of func_t
  | Struc of strc_t
  | Vdecl of bind

  and
  func_t = {
    typ : Ast.typ;
    fname : string;
    formals : (Ast.typ * string) list;
    body : stmt_t list; (* need typed statements *)
  }

  and strc_t = {
    sname : string;
    vdecls : bind list;
  }

type symbol_table = {
  parent : symbol_table option;
  mutable variables: (string * Ast.typ) list
}

type environment = {
  scope: symbol_table;
}

type global_environment = {
  mutable funcs: func_t list;
}

type program = stmt_t list
```

Codegen.ml

```
(* codgen.ml
takes SAST and generates LLVM IR *)

module L = Llvm
```

```
module S = Sast
module A = Ast
module StringMap = Map.Make(String)

let context = L.global_context ()
let the_module = L.create_module context "ManiT"

(* each is lltype *)
(* and i64_t  = L.i64_type  context *)
let d_t  = L.double_type context
let i64_t  = L.i64_type  context
let i32_t  = L.i32_type  context
let i8_t   = L.i8_type   context
let i1_t   = L.i1_type   context
let void_t = L.void_type context
let str_t = L.pointer_type i8_t

(* struct types *)
let struct_types:(string, L.lltype) Hashtbl.t = Hashtbl.create 10

(* globals are initially empty *)
let globals:(string, L.llvalue) Hashtbl.t = Hashtbl.create 50

(* finds struct typ *)
let find_struct_typ name = try Hashtbl.find struct_types name
  with Not_found -> raise(Exceptions.InvalidStruct name)

(* Ast type to llvm type *)
let rec ltype_of_typ = function
    A.Int -> i32_t
  | A.Float -> d_t
  | A.String -> str_t
  | A.Bool -> i1_t
  | A.Void -> void_t
  | A.Struct_typ(sname) -> find_struct_typ sname (* assume that all struct typs all already
made *)
  | A.Array_typ(elem_typ, length) -> L.array_type (ltype_of_typ elem_typ) length

(* Ast type to llvm value. *)
let rec lvalue_of_typ typ = match typ with
    A.Int | A.Bool | A. Void  -> L.const_int (ltype_of_typ typ) 0
  | A.Float -> L.const_float (ltype_of_typ typ) 0.0
  | A.String -> L.const_pointer_null  (ltype_of_typ typ)
  | A.Struct_typ(sname) -> L.const_named_struct (find_struct_typ sname) [||]
  | A.Array_typ(elem_typ, length) -> L.const_array (ltype_of_typ elem_typ) (Array.make
length (lvalue_of_typ elem_typ))


(* declares struct typ *)
```

```
let declare_struct_typ s =
  let struct_t = L.named_struct_type context s.S.sname in
  Hashtbl.add struct_types s.S.sname struct_t

(* builds the body of struct typ *)
let define_struct_body s =
  let struct_typ = try Hashtbl.find struct_types s.S.sname
    with Not_found -> raise(Exceptions.ErrCatch "undefined struct typ") in
  let vdecl_types = List.map (fun (typ, _) -> typ) s.S.vdecls in
  let vdecl_lltypes = Array.of_list (List.map ltype_of_typ vdecl_types) in
  L.struct_set_body struct_typ vdecl_lltypes false


(* ********* FUNCTIONS BEGIN HERE ********* *)
(* translates all functions to llvm IR *)
let translate_functions functions the_module =

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

(* file open and close *)
(*
 *  * fopen takes 2 arguments, a filename, which is a string, and a mode (e.g. rw)
 *   * It returns a file pointer on success.
 *    *)
let open_file_t = L.function_type str_t [| str_t ; str_t |] in
let open_file_func = L.declare_function "fopen" open_file_t the_module in

let close_file_t = L.function_type i32_t [| str_t |] in
let close_file_func = L.declare_function "fclose" close_file_t the_module in

let fputs_t = L.function_type i32_t [| i32_t ; str_t |] in
let _ = L.declare_function "fputs" fputs_t the_module in

(*Args: str, num of chars to copy, file pointer*)
let fgets_t = L.function_type str_t [| str_t; i32_t; str_t |] in
let fgets_func = L.declare_function "fgets" fgets_t the_module in

let fwrite_t = L.function_type i32_t [| str_t; i32_t; i32_t; str_t |] in
let fwrite_func = L.declare_function "fwrite" fwrite_t the_module in
(* ****************** END FILE READ WRITE ****************** *)


let strlen_t = L.function_type i32_t [| str_t |] in
let strlen_func = L.declare_function "strlen" strlen_t the_module in

(* forking *)
let fork_t = L.function_type i32_t [||] in
```

```
let fork_func = L.declare_function "fork" fork_t the_module in
(* sleep *)
let sleep_t = L.function_type i32_t [|i32_t|] in
let sleep_func = L.declare_function "sleep" sleep_t the_module in
(* execlp *)
let execlp_t = L.function_type i32_t [|str_t ; str_t; str_t; i32_t |] in
let execlp_func = L.declare_function "execlp" execlp_t the_module in


(* build function prototypes *)
let prototypes =
  let build_proto m fdecl =
    let name = fdecl.S.fname
    and formal_types =
    Array.of_list (List.map (fun (t, _) -> ltype_of_typ t) fdecl.S.formals) in
    let ftype = L.function_type (ltype_of_typ fdecl.S.typ) formal_types in
  StringMap.add name (L.define_function name ftype the_module, fdecl) m in
List.fold_left build_proto StringMap.empty functions in

(* format strings for printing. only in main_func *)
let (main_func, _) = try StringMap.find "main" prototypes
with Not_found -> raise(Exceptions.ErrCatch "main function does not exist") in
let builder = L.builder_at_end context (L.entry_block main_func) in
let int_format_str = L.build_global_stringptr "%d\n" "fmt" builder
and float_format_str = L.build_global_stringptr "%f\n" "fmt" builder
and string_format_str = L.build_global_stringptr "%s\n" "fmt" builder in

(* Core method that build llvm IR for fbody *)
let build_function fdecl =

  (* search prototype and get builder *)
  let (the_function, _ ) = StringMap.find fdecl.S.fname prototypes in
  let builder = L.builder_at_end context (L.entry_block the_function) in

  (* create formals. *)
  let formals =
    let add_formal m (t, id) param = L.set_value_name id param;
    (* allocate the formal and store param *)
    let formal = L.build_alloca (ltype_of_typ t) id builder in
    ignore (L.build_store param formal builder);
    StringMap.add id formal m in
  (* expr below evaluates to a map. see fold_left2 *)
  List.fold_left2 add_formal StringMap.empty fdecl.S.formals
  (Array.to_list (L.params the_function)) in

  (* at start, formals are the only locals. added extra step to use hashtbl *)
  let locals:(string, L.llvalue) Hashtbl.t = Hashtbl.create 50 in
  let _ = StringMap.iter (fun id formal -> Hashtbl.add locals id formal) formals in
```

```
  (* original lookup: Return the value for a variable or formal argument *)
  let find_var id = try Hashtbl.find locals id with Not_found -> Hashtbl.find globals id in


  (* Allocates lhs when assignment is declaration *)
  let alloc_expr id typ in_block builder =
    let init = match typ with
        A.Int | A.Float | A.Bool | A.String | A.Array_typ(_,_) -> lvalue_of_typ typ
      | _ -> raise(Failure("cannot alloc for exprs of these typs"))
    in

    (* if not in main and not in block, global. else, local *)
    (if ("main" = fdecl.S.fname) && not in_block
    (* delcare and add to map *)
    then let global = L.define_global id init the_module in Hashtbl.add globals id global
    else let local =  L.build_alloca (ltype_of_typ typ) id builder in Hashtbl.add locals id
local);
    builder in

  (* Allocates lhs in vdecl stmt *)
  let alloc_stmt id typ builder in_block =
    let init = match typ with
        A.Struct_typ(_) -> lvalue_of_typ typ
      | _ -> raise(Failure("cannot alloc for stmts of these typs")) in

    (* same as above *)
    (if ("main" = fdecl.S.fname) && not in_block
    then let global = L.define_global id init the_module in Hashtbl.add globals id global
    else let local =  L.build_alloca (ltype_of_typ typ) id builder in Hashtbl.add locals id
local);
    builder in

  (* Returns addr of expr. used for lhs of assignment expr *)
  let addr_of_expr expr typ in_b builder = match expr with
    S.Id(id) ->
      (* allocate space for lhs *)
      let var = try find_var id with Not_found ->
        (ignore(alloc_expr id typ in_b builder); find_var id) in var
    | S.Struct_access (var, _, index) ->
        let llvalue = find_var var in (* llvalue from build_alloca *)
        let addr = L.build_struct_gep llvalue index "tmp" builder in addr
    | S.Array_access(arrayName,index) ->
        let llvalue = try find_var arrayName with Not_found ->
          raise(Failure("Cannot assign to array that doesnt exist - Codegen")) in
        let addr = L.build_gep llvalue [| L.const_int i32_t 0; L.const_int i32_t index |]
"array" builder in addr
    | _ -> raise(Failure("cannot get addr of LHS")) in

  (* Construct code for an expression; return its value *)
```

```
  let rec build_expr builder in_b = function
      S.IntLit i, _ -> L.const_int i32_t i
    | S.FloatLit f, _ -> L.const_float d_t f
    | S.BoolLit b, _ -> L.const_int i1_t (if b then 1 else 0)
    | S.StringLit s, _ -> L.build_global_stringptr s "" builder
    | S.Id s, _ -> L.build_load (find_var s) s builder (* R.H.S lookup *)
    | S.Binop (e1, op, e2), _ ->
        let e1' = build_expr builder in_b e1
        and e2' = build_expr builder in_b e2 in
        (match op with
          A.Add     -> L.build_add
        | A.Sub     -> L.build_sub
        | A.Mult    -> L.build_mul
        | A.Div     -> L.build_sdiv
        | A.And     -> L.build_and
        | A.Or      -> L.build_or
        | A.Equal   -> L.build_icmp L.Icmp.Eq
        | A.Neq     -> L.build_icmp L.Icmp.Ne
        | A.Less    -> L.build_icmp L.Icmp.Slt
        | A.Leq     -> L.build_icmp L.Icmp.Sle
        | A.Greater -> L.build_icmp L.Icmp.Sgt
        | A.Geq     -> L.build_icmp L.Icmp.Sge
        ) e1' e2' "tmp" builder
    | S.Unop(op, e), _ ->
        let e' = build_expr builder in_b e in
        (match op with
          A.Neg     -> L.build_neg
        | A.Not     -> L.build_not) e' "tmp" builder
    | S.Assign ((lhs, _), rhs), typ ->
       let l_val = (addr_of_expr lhs typ in_b builder)
      in
      (* build rhs and store *)
      let e' = build_expr builder in_b rhs in
        ignore (L.build_store e' l_val builder); e'
    | S.Call ("print", [(e, expr_t)]), _ ->
      let var = build_expr builder in_b (e,expr_t) in
      (match expr_t with
        A.Int ->
          L.build_call printf_func [| int_format_str ; (var) |]
      | A.Float ->
          L.build_call printf_func [| float_format_str ; (var) |]
      | A.Bool ->
          L.build_call printf_func [| int_format_str ; (var) |]
      | A.String ->
          L.build_call printf_func [| string_format_str ; (var) |]
      | A.Void ->
          L.build_call printf_func [| string_format_str ; (L.build_global_stringptr "" ""
builder) |]
      | _ -> raise(Failure("Call semant failed"))) "printf" builder
```

```
    (* built in functions *)
    | S.Call ("open", e), _ ->
        let actuals = List.rev (List.map (build_expr builder in_b) (List.rev e)) in
        L.build_call open_file_func (Array.of_list actuals) "fopen" builder
    | S.Call ("execlp", e), _ ->
        let actuals = List.rev (List.map (build_expr builder in_b) (List.rev e)) in
        L.build_call execlp_func (Array.of_list actuals) "execlp" builder
    | S.Call ("fgets", e), _ ->
        let actuals = List.rev (List.map (build_expr builder in_b) (List.rev e)) in
        L.build_call fgets_func (Array.of_list actuals) "fgets" builder
    | S.Call ("write", e), _ ->
        let actuals = List.rev (List.map (build_expr builder in_b) (List.rev e)) in
        L.build_call fwrite_func (Array.of_list actuals) "fwrite" builder
    | S.Call ("len", e), _ ->
        let actuals = List.rev (List.map (build_expr builder in_b) (List.rev e)) in
        L.build_call strlen_func (Array.of_list actuals) "strlen" builder
    | S.Call ("close", e), _ ->
        let actuals = List.rev (List.map (build_expr builder in_b) (List.rev e)) in
        L.build_call close_file_func (Array.of_list actuals) "fclose" builder
    | S.Call ("fork", _), _ ->
        L.build_call fork_func (Array.of_list []) "fork" builder
    | S.Call ("sleep", e), _ ->
        let actuals = List.rev (List.map (build_expr builder in_b) (List.rev e)) in
        L.build_call sleep_func (Array.of_list actuals) "sleep" builder
    | S.Call (f, act), _ ->
       let (fdef, fdecl) = StringMap.find f prototypes in
       let actuals = List.rev (List.map (build_expr builder in_b) (List.rev act)) in
       let result = (match fdecl.S.typ with A.Void -> ""
                                          | _ -> f ^ "_result") in
       L.build_call fdef (Array.of_list actuals) result builder

    (* build array literal *)
    | S.Array_create (expr_list), arr_typ  ->
       (match arr_typ with
          A.Array_typ(typ, length) ->
            let revElem_list = List.rev (expr_list) in
             let elems = Array.of_list (List.map (fun expr -> build_expr builder in_b expr)
revElem_list) in
             let each_type = ltype_of_typ typ in
             let array_type = L.array_type each_type length in
             L.const_array array_type elems
        | _ -> raise(Failure("non-array types in create")))

    | S.Array_access (arr, index), _ ->
       let arr_lvalue = find_var arr in
       ignore(L.build_load arr_lvalue "loaded" builder);
       let elem_ptr = L.build_gep arr_lvalue (*loaded_lvalue*) [|L.const_int i32_t 0;
L.const_int i32_t index|] "arr addr" builder in
       L.build_load elem_ptr "array_access" builder
```

```
    | S.Struct_access (var, _ , index), _ ->
       let llvalue = find_var var in
       let addr = L.build_struct_gep llvalue index "tmp" builder in
       L.build_load addr "struct_access" builder
  in

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

  (* Build the code for the given statement; return the builder for
     the statement's successor *)
  let rec build_stmt builder in_b = function
    | S.Expr e -> ignore (build_expr builder in_b e); builder
    | S.Return e -> ignore (match fdecl.S.typ with
        A.Void -> L.build_ret_void builder
      | _ -> L.build_ret (build_expr builder in_b e) builder); builder

    (* if entering a block, need to keep track *)
    | S.Block sl -> List.fold_left (fun builder stmt -> build_stmt builder (*true*) in_b
stmt) builder sl
    | S.If (predicate, then_stmt, else_stmt) ->
       let bool_val = build_expr builder true predicate in
       let merge_bb = L.append_block context "merge" the_function in

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

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

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

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

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

```
        let pred_builder = L.builder_at_end context pred_bb in
        let bool_val = build_expr pred_builder true predicate in

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

    | S.For (e1, e2, e3, body) -> build_stmt builder true
          ( S.Block [S.Expr e1 ; S.While (e2, S.Block [body ; S.Expr e3]) ] )

    (* vdecl for structs. type checked in semant *)
    | S.Vdecl(typ, id) -> alloc_stmt id typ builder in_b
    | _ -> raise(Failure("Something went bad in codegen checkStatement"))
  in

  (* build code for each stmt in body.*)
  let builder = build_stmt builder false (S.Block fdecl.S.body) in

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

List.iter build_function functions;
the_module
(* ******* translate function ends here ****** *)


(* function to split fdecls and stmts. store stmts in main's body *)
let split stmts =
  let split1 (fdecls, sdecls, main_body) stmt = match stmt with
    S.Func(fdecl) -> fdecls@[fdecl], sdecls, main_body
  | S.Struc(sdecl) -> fdecls, sdecls@[sdecl], main_body
  | _ -> fdecls, sdecls, main_body@[stmt]
  and init = ([],[],[]) in
  List.fold_left split1 init stmts

(* tranlsate *)
let translate (stmts) =
  let (fdecls, sdecls, main_body) = split stmts in

  (* main_func *)
  let main_func = {
    S.fname = "main";
    S.typ = A.Int;
    S.formals = [];
    S.body = main_body
  } in
```

```
  (* functions *)
  let functions = [main_func]@fdecls in
  let structs = sdecls in

  let _ = List.iter declare_struct_typ structs in
  let _ = List.iter define_struct_body structs in
  let the_module = translate_functions functions the_module in
the_module
```

Manit.ml

```
(* Top-level of the ManiTcompiler: scan & parse the input,
   check the resulting AST, generate LLVM IR, and dump the module *)

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


(* version with options. commented out due to errors in prettyprinter
type action = Ast | LLVM_IR | Compile

let _ =
  (* command line options. *)
  let action = if Array.length Sys.argv > 1 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 *)

  (* w/o command line options *)
  else Compile in
  let lexbuf = Lexing.from_channel stdin in (* set input stream *)

  (* scan input to tokens, parse the tokens to get AST *)
  let ast = Parser.program Scanner.token lexbuf in

  (* semantic checker takes AST, checks input, produces SAST *)
  let sast = Semant.check_program ast in

  (* depends on command line options. default is Compile *)
  match action with
```

```
    (* Ast.string_of_program is the pretty-print func in AST. *)
    Ast -> print_string (Prettyprint.string_of_program sast)
  | LLVM_IR -> print_string (Prettyprint.string_of_llmodule (Codegen.translate sast))

    (* Codegen translates SAST (or AST) to IR of type module.
       run that module through LLVM analyzer for debugging.
       if valid module, translate module to string and syscall print. *)
  | Compile -> let m = Codegen.translate sast in
    Llvm_analysis.assert_valid_module m;
    print_string (Llvm.string_of_llmodule m)
*)
```

Makefile

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

# Easiest way to build: using ocamlbuild, which in turn uses ocamlfind

.PHONY : manit.native

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

# "make clean" removes all generated files

.PHONY : clean
clean :
	ocamlbuild -clean
	rm -rf testall.log *.diff manit scanner.ml parser.ml parser.mli
	rm -rf *.cmx *.cmi *.cmo *.cmx *.o pic*

#to test for shift/reduce conflicts and other stuff
.PHONY : parse
parse :
	ocamlyacc -v parser.mly

#run all tests
.PHONY : test
test :
	./testall.sh

#run hello world test
.PHONY : test2
test2 :
```

```
	./manit.native < helloworld.mt > output.ll
	lli output.ll

#run demo
.PHONY : demo
demo :
	./manit.native < demo.mt > output.ll
	lli output.ll

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

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

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

scanner.ml : scanner.mll
	ocamllex scanner.mll

parser.ml parser.mli : parser.mly
	ocamlyacc -v parser.mly

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

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

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

### Generated by "ocamldep *.ml *.mli" after building scanner.ml and parser.ml
ast.cmo :
ast.cmx :
codegen.cmo : ast.cmo
codegen.cmx : ast.cmx
manit.cmo : semant.cmo scanner.cmo parser.cmi codegen.cmo ast.cmo
manit.cmx : semant.cmx scanner.cmx parser.cmx codegen.cmx ast.cmx
parser.cmo : ast.cmo parser.cmi
parser.cmx : ast.cmx parser.cmi
scanner.cmo : parser.cmi
scanner.cmx : parser.cmx
semant.cmo : ast.cmo
semant.cmx : ast.cmx
parser.cmi : ast.cmo
```