

ChartLan Final Report

Yibo Zhu (yz2486)

Xiuming Dou (xd2138)

Xiao Xu (xx2165)

Ziyue Chen (zc2239)

Xiang Ma (xm2151)

December 18th 2012

Catalogue

1.	Introduction to ChartLan	4
1.1	Motivation.....	4
1.2	Overview	4
2.	Language Tutorial.....	6
3.	Language Reference Manual	10
3.1	Lexical conventions	10
3.2	Declarations	11
3.3	Expression	11
3.4	Operators	11
3.5	Statements.....	14
3.6	Scope Rules	16
3.7	Sample programs	16
4.	Project Plan	18
4.1	Processes.....	18
4.2	Programming Style Guide	19
4.3	Project Timeline	19
4.4	Roles and Responsibilities.....	20
4.5	Software Development Environment	21
4.6	Project log	21
5.	Architecture Design	23
5.1	Translator Construction	23
5.2	Components and Interfaces.....	23
6.	Test Plan.....	25
6.1	Representative program code:	25
6.2	Test suites	29
6.3	Test detail.....	31
7.	Lessons Learned	32
7.1	Xiao Xu	32
7.2	Xiuming Dou.....	32

7.3	Yibo Zhu	32
7.4	Xiang Ma	33
7.5	Ziyue Chen.....	33
8.	Appendix	33
8.1	Makefile	33
8.2	Scanner.....	34
8.3	Parser	36
8.4	AST	41
8.5	Semantics Checking.....	45
8.6	Compiler.....	57
8.7	Bytecode	65
8.8	Bytecode Interpreter	69

1. Introduction to ChartLan

1.1 Motivation

Today, we live in a society composed of large quantities of data. Therefore, data processing is extremely important in everyday life. However, in many programming languages, there are not many features designed for convenience of data processing. Hence, programmers may need a large block codes to deal with a single group of data. Our programming language, ChartLan, is proposed for easier manipulation of data. With the help of new data type and new defined operators, users can simplify the process of inserting, modifying and deleting data based on C-style syntax. In ChartLan, the new data type is somewhat like array in Java or C++, but it has its own unique definitions and characteristics.

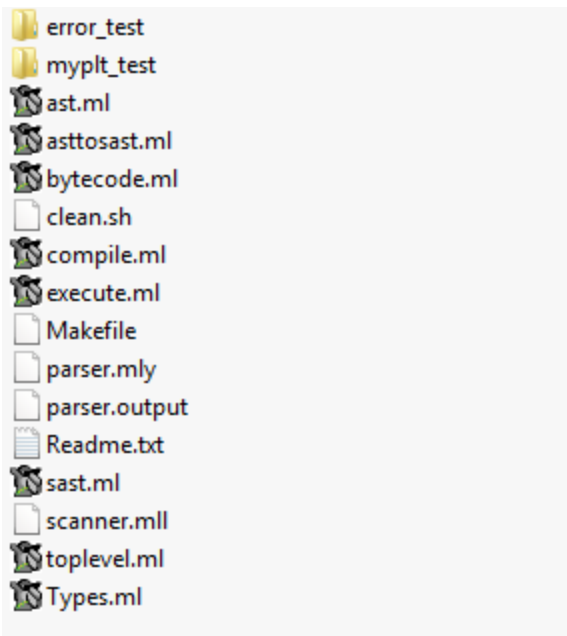
1.2 Overview

ChartLan is an imperative language with a C-like structure and it is mainly designed to enhance the ability of dealing with groups of data. It incorporates constants, variables, expressions, blocks, conditionals (if...else...), loops (while) and built-in or user-defined functions. The new data type we add for data processing is Array, which is omitted in the syntax of micro-c language. With ChartLan, users can manage a group of related data by creating array, inserting the data into array, modifying data, deleting unwanted items and conducting operations between arrays. The function of indexing can also help users quickly pinpoint the data they need.

The structure of ChartLan strictly follows micro-c while it has its own style, semantics and syntax. In ChartLan, we add a process in the compiler so it can check errors and bugs which can't be easily captured by Scanner and Parser like data type, the logical order of data, etc.

So far, we finish ChartLan mainly by writing Scanner, Parser, AST, Semantics Checking Part, Compiler, Bytecode and Bytecode Interpreter. This will be further specified in the following sections.

The list of our program files:



2. Language Tutorial

ChartLan was designed to manipulate array in a more convenient way. Each program can be written in a plain text file. ChartLan is C-like language. The main difference is as follows:

- When we define a function we should use the keyword, “def”, to declare a function:

```
def int main() {.....}
```

- The type and return type should int, string or array.
- There is no Boolean or Bool type in ChartLan. We use “1” or “0” standing for “true” or “false”.
- The declaration should be placed in the front of a function, and the action of assign and other statement should be after the declaration. :

```
def int main()
{
    int x;
    x=1;
    print(x);
    return 2;
}
```

- According to the type which you want to print, you should choose different print function.

```
def int main()
{
    printstring("haha");
    print(1);
    printarray(%(1,2,3)%)
    return 2;
}
```

- The variable of string type should define the length of string during the declaration period. The double quotes take For example:

```
def int main()
{
    string[6] x;
    x="haha";
    print(x);
    return 2;
}
```

- The type of array in ChartLan must be int. One constant of a array should start from “%(” and end with “)%”, for example:

```
intarray[3] x;
def int main()
{
    x=%(1,2,3)%;
    printarray(x);
    return 2;
}
```

- We can manipulate the array in a more efficient and more convenient way. For example, we can use “+” to insert a value to the head or end of a array.

```
intarray[4] x;
def int main()
{
    x=12+%(3,4,5)%;
    printarray(x);
    return 2;
}
```

The above program will output %(12,3,4,5).

```

intarray[4] x;
def int main()
{
    x=%(3,4,5)%+12;
    printarray(x);
    return 2;
}

```

The above program will output %(3,4,5,12).

- There is no “for loop” in ChartLan. We can only use “while” as loop function.

```

def int main()
{
    int i;
    i= 3;
    while(i>0)
    {
        print(i+5);
        i=i-1;
    }
    return 2;
}

```

- We can compile the language by the following instruction(test.txt is the name of file of program):

```
./toplevel -c < test.txt
```

There is also some useful instruction:

- make:

```
make
```

- Show the Ast result

```
./toplevel -a < test.txt
```

- Show the Ast to Sast result


```
./toplevel -s < test.txt
```

- Show the Bytecode result

```
./toplevel -b < test.txt
```

- If we want to recompile the compiler, we should use the clean instruction before make.

```
./clean.sh
```

3. Language Reference Manual

3.1 Lexical conventions

(1) Comments

The characters `#~` introduce a comment, which terminates with the characters `~#`.

(2) Identifiers

An identifier is a sequence of letters and digits; the first character must be alphabetic. Upper and lower case letters are considered different.

(3) Keywords

The following identifiers are reserved for use as keywords, and may not be used otherwise:

<i>Int</i>	Integer number
<i>String</i>	Serial of characters
<i>Array</i>	List of integers
<i>If...else...</i>	Conditional control
<i>Def</i>	Function definition
<i>While</i>	Loop identifier
<i>intarray</i>	integer array constructor
<i>Return</i>	Return a value
<i>Print</i>	Print a integer
<i>Printstring</i>	Print a string
<i>Printarray</i>	Print an array

(4) Constants

<i>Integer constants</i>	...-2 -1 0 1 2...
<i>Boolean constants</i>	integer 1 for Boolean true, 0 for Boolean false
<i>String constants</i>	"abcd" or "@\$%^&" not including any " in between
<i>Array constants</i>	%(integer constants separated by ",")% e.g. %(1,2,3)%

3.2 Declarations

General rule: Type-specifier identifier;

Where type-specifier can be int, string[numofchars including double quotes] , intarray[length]

Eg: int x; string[6] c; (later can be assigned to "haha") intarray[3] x;

3.3 Expression

- (1) Identifier : Its type is specified by its declaration.
- (2) Constants : A integer, string, array, index is an expression.
- (3) (expression): A parenthesized expression is a primary expression whose type and value are identical to those of the unadorned expression.
- (4) expression binary operator expression
binary operators can be: +, -, *, /, &&, ||, .+, .-, .* , ./
- (5) Identifier [expression]: The intuitive meaning is that of a subscript, the subscript expression is int.

3.4 Operators

(1) Addition operators:

The addition operator + and group left to right

Expression + expression

The binary '+' operator indicates addition.

Integer + integer gives integer addition value.

Intarray + integer inserts the integer at the end of array

Integer + Intarray inserts the integer at the head of array

(2)Subtraction operator:

The addition operator - and group left to right, and it is like the inverse operation of addition.

Expression - expression

The binary - operator indicates subtraction.

Integer - integer gives integer subtraction value.

(3)Multiplication operator:

The multiplication operators * group left to right

*Expression * expression*

The binary * operator indicates multiplication.

Integer * integer gives integer multiplication value.

Intarray1 * intarray2 gives the concatenation of two arrays.

(4)Division operator:

The division operators / group left to right, the division is like the inverse operation of multiplication.

Expression/expression

The binary / operator indicates division.

The integer data type can perform division as commonly use, and divisor cannot be zero.

(5)Arithmetic operator:

Intarray .+ integer

Intarray .- integer

Intarray .*integer

Intarray ./ integer

The arithmetic operators are used when users want to do the operation to the every element in array or table.

E.g.: $\%(2,3,1)\% + 3 = \%(5,6,4)\%$;
 $\%(2,3,4)\% - 1 = \%(1,2,3)\%$;

(6)Equality operator:

Expression == expression

returns 1 (Boolean constants true) if the two expressions are identical and false otherwise.

Expression != expression

returns 0 (Boolean constants false) if the two expressions are identical and true otherwise.

Expression1 >(>=) expression2

returns 1 (Boolean constants true) if expression1 is greater (not less than) expression2 and false otherwise.

Expression <(<=) expression

returns 1 (Boolean constants true) if expression 1 is less (not greater than) expression2 and false otherwise.

(7)Assignment operator:

Identifier = expression.

Assigns an expression's value to the identifier.

(8)Logical operator:

Expression && expression

returns 1 (Boolean constants true) if the two expressions are both true and false otherwise.

Expression || expression

returns 0 (Boolean constants true) if the one of the two expressions is true and false otherwise.

3.5 Statements

(1) Expression statement:

Most statements are expression statements, which have the form *expression*;

(2) Compound statement:

So that several statements can be used where one is expected, the compound statement is provided:

compound statement

{statement-list}

statement-list:

statement

statement statement-list

(3) Conditional statement:

The two forms of the conditional statement are

if (expression) {statement}

if (expression) {statement1} else {statement2}

In both cases the expression is evaluated and if it is nonzero, the statement1 is executed. In the second case, the statement2 is executed if the expression is 0.

(4) While statement:

The while statement has the form

while (expression) statement

The statement is executed repeatedly so long as the value of the expression remains nonzero.

The test takes place before each execution of the statement.

(5)Return statement:

A function returns to its caller by means of the return statement, which has one of the forms

return ;

return (expression);

(6)Function Statement:

The function-statement is a compound statement which has declarations at the start.

functionstatement:

{ variable declarationlist statementlist }

*Def typespecifier functionname (typespecifier identifier1, typespecifier identifier2 ...)
{ function body}*

Where typespecifier indicates the return type of the function which could be either int, string, array.

Arguments consists of arguments of the function which are identifiers with their type specified in the front.

Def is the keyword that specifies a function definition.

Functionname is a combination of any letters or integers but the first must be a letter.

Eg: Def int max(int a, int b){ #~gives back the bigger one of the two integers~#

```
If ( a > b ) {  
    Return a;  
}  
Else{
```

```
    Return b;
```

```
    }
```

Some of built-in functions:

Print(x): print a integer

Printarray(x): print an integer array

Printstring(x): print a string

3.6 Scope Rules

An identifier's lexical scope, which is essentially the region of a program during which the identifier can be used directly without drawing "undefined identifier" diagnostics. There are two kinds of identifiers in ChartLan's lexical scope.

Global variable is a variable that is accessible in every scope. That is to say, in our language, a global variable can be used anywhere in the program once it is defined.

Local variable is a variable that is given local scope. Such a variable is accessible only from the function or block in which it is declared.

3.7 Sample programs

(1)GCD function

```
#~GCD Test~#
```

```
def int gcd(int a, int b){
```

```
    while (a != b) {
```

```
        If (a > b) a = a-b;
```

```
        else b = b-a;
```

```
    }
```

```
    return a;
```

```
}
```



```
def int main(){ int x; x=gcd(5,10); print(x); return 2;}
```

(2) Dot operation function

```
#~Dot Test~#
```

```
def int main() {  
    intarray[3] x;  
    intarray[3] y;  
    intarray[3] z;  
    x=%(1,2,3)%;  
    y=x.+2;  
    z=y.*2;  
    printarray(z);  
    return 1;  
}
```

4. Project Plan

4.1 Processes

The ChartLan team is made up of 5 members and we held the first regular group meeting right after the class in which the professor released the project. Regularly, we met weekly on Friday morning at CS lounge. Yibo Zhu became the group leader after the first meeting.

In the planning period, on weekly meeting, everyone would give a short description of his or her workload of the week before. Usually, we might concentrate on several attracting points and kept a more detailed discussion. If no point was admired by all team members, brainstorm was also introduced to generate other novel concepts. Actually, Chartlan combines the thoughts of several proposals and was revised by every team member.

After we submitted the project proposal and got comments from the professor and TA, we continued to write specifications (a.k.a. LRM) and make more detailed design of our language. In this stage, we also met weekly and the topics usually followed language style, syntax and semantics. Meanwhile, as part of our team concluded all the points into LRM, other members initiated our programming assignment by writing the first version of scanner and parser.

We developed the scanner, parser and AST around Mid-term. In this period, we met in CLIC to do these coding assignments two or three times a week. At the beginning of November, our fixed weekly meeting paused for two weeks because of the Hurricane Sandy and Mid-term exams. These files were finished without syntax error before December. The works were relatively elementary and could be seen as the introduction to the translator of the Ocaml version.

Development was the core of the whole project. Due to the superiority of the number of team members, we separated our tasks into three parts: AST to SAST, Compiler and Bytecode interpreter. Each member was responsible for the comprehension of one part referring to the slides provided by the professor. We conducted our programming work directed by the team members in charge of that part and our team became more efficient due to this mode.

While the main structure was finally done, Chartlan was tested in different scenarios. Three members verified ChartLan with various test cases while the other two men began the conclusion of the whole-semester-long project and started the final report.

4.2 Programming Style Guide

As we make up a team, all the members comply with the following guides when we carry on team work:

(1) Each paper or electronic material must be submitted with the signature of the drafter, and the other members must view the material and make comments on it before it is finally approved. Double-check is needed for the final-version of all documents.

(2) For each block of reference code, team member should make comments on it for the convenience of other members, using formal comment format. Each modification of the source code should be saved as a new version, with the team-member name as part of the file name, like "filename-3rd version-member name.xx".

(3) We share files and blocks of code using Google Drive with a common account. We also use instant online-contact software to keep in touch with each other, or hold temporary meeting.

(4) Neat and Style coding is needed. As we write all codes using Ocaml, every block of code must keep consistent with the Ocaml style. We all use the same programming environment to guarantee the minimum of text-style conflicts with each other.

(5) Since ChartLan is based on micro-c syntax, we also make the style of ChartLan similar to that of micro-c for the convenience of the users. Also we leave our distinct grammars and make proper modifications from the degree of developing.

(6) We make a set of regulations for the naming of constants, variables and function names. Thus, every can easily take a better comprehension of that block of code.

4.3 Project Timeline

ID	Assignment	Beginning Time	Finish Time	Duration (weeks)	2012 / 10				2012 / 11				2012 / 12			
					9/30	10/7	10/14	10/21	10/28	11/4	11/11	11/18	11/25	12/2	12/9	12/16
1	Proposal	09/19/2012	09/28/2012	1.43	■											
2	LRM	10/03/2012	10/26/2012	3.43	■	■	■	■								
3	Scanner	10/15/2012	10/27/2012	1.86			■	■								
4	Parser	10/20/2012	10/29/2012	1.43			■	■								
5	AST	11/02/2012	11/16/2012	2.14					■	■						
6	AST to SAST	11/21/2012	12/08/2012	2.57							■	■				
7	Compiler	11/21/2012	12/12/2012	3.14							■	■	■			
8	Bytecode Interpreter	11/26/2012	12/14/2012	2.71								■	■	■		
9	Test	12/14/2012	12/17/2012	.57												■
10	Final Report	12/13/2012	12/18/2012	.86												■

4.4 Roles and Responsibilities

- Yibo Zhu performs as the team leader, and really has done outstanding job from designing to programming.
- Proposal was drafted by Yibo Zhu, and was finally decided with everybody's view.
- LRM was open discussed in group meeting and was summarized, revised and finished by Ziyue Chen.
- Scanner, Parser, AST were completed together as group study cases.
- Semantics Checking (AST to SAST), SAST were preprocessed by Ziyue Chen and Yibo Zhu.
- Compiler was preprocessed by Xiang Ma and Xiao Xu.
- Yibo Zhu wrote the main part of Semantics Checking with the assist of Ziyue Chen, and also wrote Compiler with the assist of Xiang Ma.
- Xiuming Dou was fully responsible for the programming and debugging of Bytecode and Bytecode Interpreter.
- Debugging and Testing involved everyone, conducted by Yibo Zhu. (Xiuming individually responsible for Bytecode Interpreter)
- Arrangements of group meeting, preservation and updates of all kinds of files are done by Ziyue Chen.
- Project Log, Presentation materials and Final Report were kept and finished by Xiao Xu, assisted by Xiang Ma and Ziyue Chen, with everybody's comments.

4.5 Software Development Environment

ChartLan is built on a Window 7-compatible environment. We include a Makefile to ensure all files can be compiled automatically. The configuration and all the files related to the translator are written in Ocaml. So, `ocamlc`, `ocamllex` and `ocamlyacc` from OCaml were the main tools we used when we developed Chartlan. All members used Eclipse with an OCaml plugin as the only code editor to make the code style compatible and consistent. All codes were tested under 64-bit computer with Windows 7 environment.

4.6 Project log

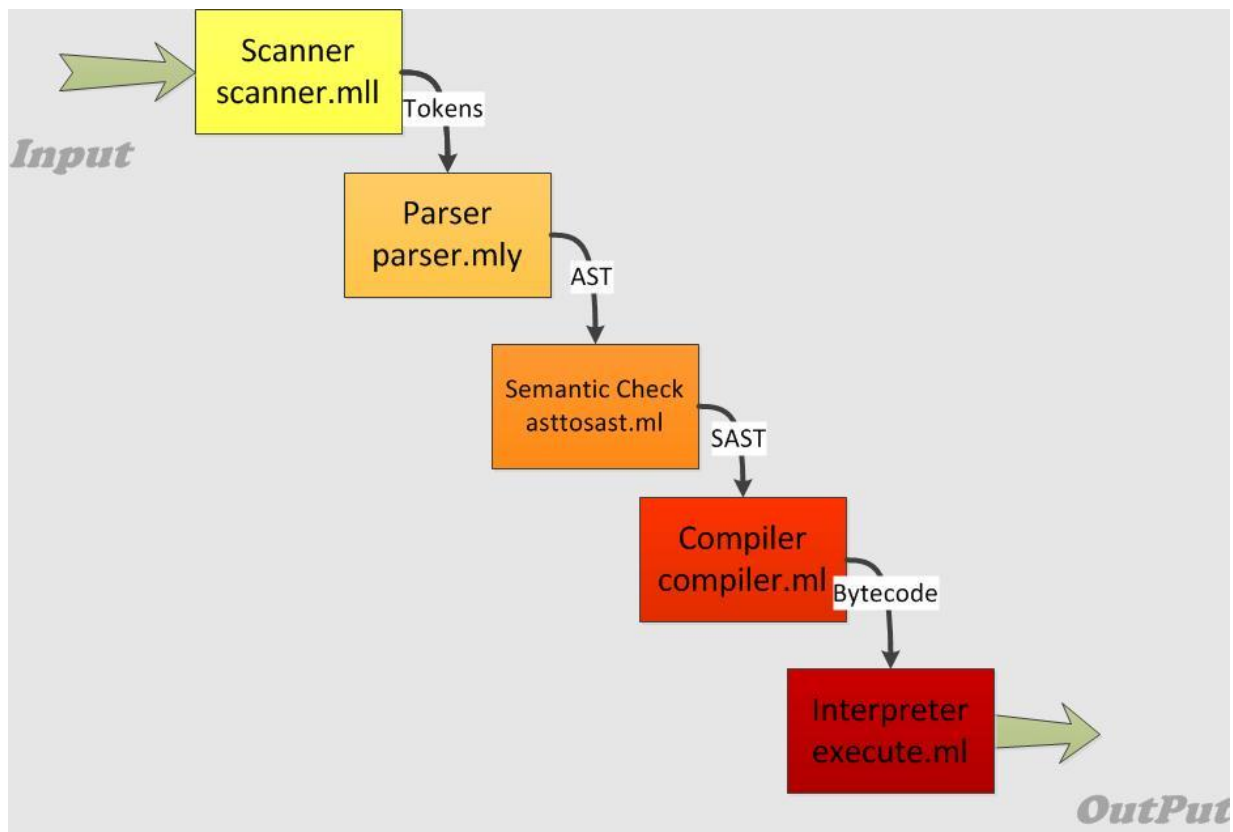
The project log keeps all the main events of our project, as described in the following chart.

Date	Event
09/19/2012	ChartLan Team Founded.
09/19/2012	First group meeting, discussed the general concepts of a language construction and some novel concepts about the language design.
09/21/2012	Second group meeting, named our language as ChartLan, finished the first version on language proposal.
09/28/2012	Final version of proposal finished
10/05/2012	Fourth group meeting, modified proposal according to the feedback of professor and TA
10/12/2012	Fifth group meeting, discussed the data type, key words, expressions and operators about ChartLan.
10/19/2012	Sixth group meeting, formed the whole configuration of ChartLan, made comments on the first version of LRM, and studied the code of scanner in the slides.
10/26/2012	Seventh group meetin, LRM submitted, debugged scanner, and kept on with parser.
10/29/2012	Scanner and parser were completed with no conflicts and syntax errors.
11/16/2012	Eighth group meeting, The configuration of AST was finished and we seperated the remaining workload into three parts. Every member kept track one of the three.
11/21/2012	Semantic-check part(AST to SAST) and Compiler part began.
11/23/2012	Nineth group meeting, Discussed problems each member faced when reading or writing the codes.
11/30/2012	Tenth group meeting, regularly meeting, shared experience along with problems regarding Ocaml.
12/08/2012	Semantic-check part(AST to SAST) finished with coding and began debugging
12/12/2012	Compiler finished with coding and begin debugging.
12/14/2012	Bytecode Interpreter finished, debugging finished, and the main construction of translator finished. Started test and final report.
12/17/2012	All works were nearly finished.
12/18/2012	Final Presentation
	*We met daily as possible as we could in December.

5. Architecture Design

5.1 Translator Construction

The following graph shows the construction of our translator:



5.2 Components and Interfaces

- Scanner: The input is a sequence of ChartLan sentences (think these like you type a series of C commands). The scanner will grasp the meaningful parts of these sentences, conducts lexical analysis and give a stream of tokens. Whitespaces and comments will be removed in this process. The streams of tokens are the interface between scanner and parser.
- Parser: Parser accepts the list of tokens given by scanner and then parses these tokens and generates an abstract syntax tree (AST) with the format specified in the ast file we

have formerly written. The abstract syntax tree is the interface between parser and semantics checking part.

- **Semantics Checking:** This part accepts the abstract syntax tree passed by parser, and checks the type of the components in the AST. For example, it checks whether the function or variable has a declaration, whether the value of a variable has the same type as declaration, whether indexing can be performed. Finally it gives SAST and passes it to compiler.
- **Compiler:** Compiler accepts the SAST and generates a stack-based bytecode representation of code. The format and the commands of bytecode was previously setup by a Bytecode file and this list of bytecode commands will be passed to Bytecode Interpreter.
- **Bytecode Interpreter:** Bytecode Interpreter accepts the bytecode commands generated by the compiler, runs and executes them, then, it will output the results and gives back to the user.
- As was described in the previous section, Scanner, Parser and AST were completed together in October and November as group study cases. Semantics Checking was written by Yibo Zhu and Ziyue Chen. Compiler was written by Yibo Zhu, assisted by Xiang Ma and Xiao Xu. Xiuming Dou was responsible for Bytecode and Bytecode Interpreter.

6. Test Plan

6.1 Representative program code:

(1)GCD function

```
#~GCD Test~#
```

```
def int gcd(int a, int b){  
    while (a != b) {  
        if (a > b) a = a-b;  
        else b = b-a;  
    }  
    return a;  
}  
  
def int main(){ int x; x=gcd(5,10); print(x); return 2;}
```

Ast output(./toplevel -a < gcd.txt):

```
$ ./toplevel -a < gcd.txt  
  
int main()  
{  
int x = 1;  
x=gcd(5, 10);  
print(x);  
return 2;  
}  
  
int gcd(int a = 1;  
, int b = 1;  
)  
{  
while (a != b) {  
if (a > b)  
a=a - b;  
else  
b=b - a;  
}  
return a;  
}
```

Sast output(./toplevel -s < gcd.txt):

```
$ ./toplevel -s < gcd.txt
Function: main = int
Formals:
Locals:
    Variable: x = int
Function: gcd = int
Formals:
    Variable: b = int
    Variable: a = int
Locals:
Function: printarray = int
Formals:
    Variable: val = array
Locals:
Function: print = int
Formals:
    Variable: val = int
Locals:
Function: printstring = int
Formals:
    Variable: val = string
Locals:
```

Target language program(./toplevel -b <gcd.txt):

```

0 slots to store global variables
0 Jsr 2
1 Hlt
2 Ent 1
3 Litin 10
4 Litin 5
5 Jsr 13
6 Sfp 1
7 Lfp 1
8 Jsr -1
9 Litin 2
10 Rts 0
11 Litin 0
12 Rts 0
13 Ent 0
14 Bra 14
15 Lfp -3
16 Lfp -2
17 Greater
18 Beq 6
19 Lfp -3
20 Lfp -2
21 Sub
22 Sfp -3
23 Bra 5
24 Lfp -2
25 Lfp -3
26 Sub
27 Sfp -2
28 Lfp -3
29 Lfp -2
30 Neg
31 Bne -16
32 Lfp -3
33 Rts 2
34 Litin 0
35 Rts 2
36 Ent 0
37 Litin 0
38 Rts 1
39 Ent 0
40 Litin 0
41 Rts 1
42 Ent 0
43 Litin 0
44 Rts 1

```

Program output(./toplevel -c <gcd.txt):

```

$ ./toplevel -c < gcd.txt
5

```

(2) Dot operation function

#~Dot Test~#

```

def int main() {
    intarray[3] x;
    intarray[3] y;
    intarray[3] z;
    x=%(1,2,3)%;
    y=x.+2;
    z=y.*2;
    printarray(z);
    return 1;
}

```

}

Ast output(./toplevel -a < functiondot.txt)

```

$ ./toplevel -a < functiondot.txt

int main()
<
array x = 3;
array y = 3;
array z = 3;
x=(1,2,3);
y=x .+ 2;
z=y .* 2;
printarray(z);
return 1;
>

```

Sast output(./toplevel -s < functiondot.txt)

```

$ ./toplevel -s < functiondot.txt

Function: main = int
Formals:
Locals:
    Variable: z = array
    Variable: y = array
    Variable: x = array

Function: printarray = int
Formals:
    Variable: val = array

Locals:
Function: print = int
Formals:
    Variable: val = int

Locals:
Function: printstring = int
Formals:
    Variable: val = string

Locals:

```

Target language program (./toplevel -b < functiondot.txt):

```

$ ./toplevel -b < functiondot.txt
0 slots to store global variables
0 Jsr 2
1 Hlt
2 Ent 9
3 Litin 3
4 Litin 2
5 Litin 1
6 Litin 3
7 Sfpa 9
8 Litin 3
9 Lfpa 9
10 Litin 3
11 Litin 2
12 Dadd
13 Litin 3
14 Sfpa 6
15 Litin 3
16 Lfpa 6
17 Litin 3
18 Litin 2
19 Dmult
20 Litin 3
21 Sfpa 3
22 Litin 3
23 Lfpa 3
24 Litin 3
25 Jsr -2
26 Litin 1
27 Rts 0
28 Litin 0
29 Rts 0
30 Ent 0
31 Litin 0
32 Rts 1
33 Ent 0
34 Litin 0
35 Rts 1
36 Ent 0
37 Litin 0
38 Rts 1

```

Program output(./toplevel -c < functiondot.txt):

```

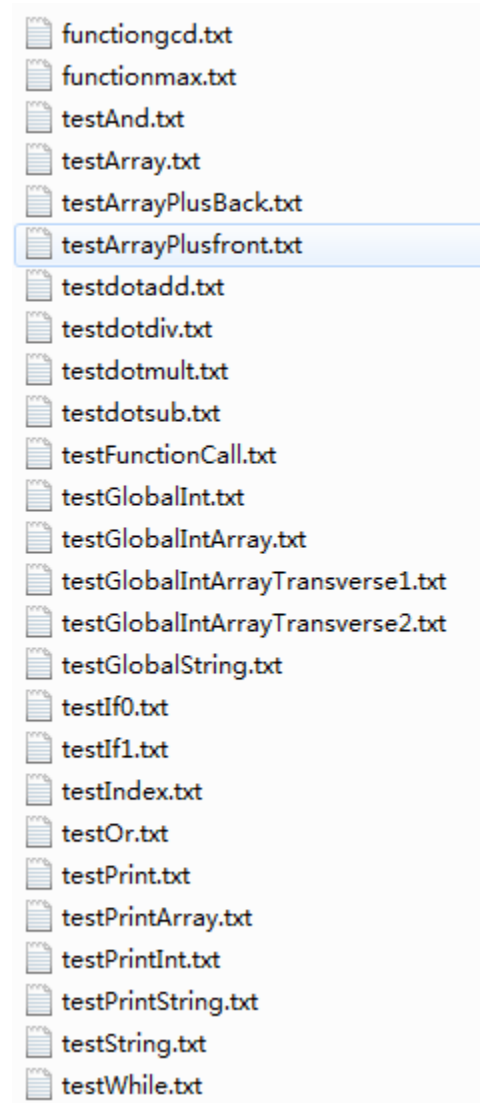
$ ./toplevel -c < functiondot.txt
6
8
10

```

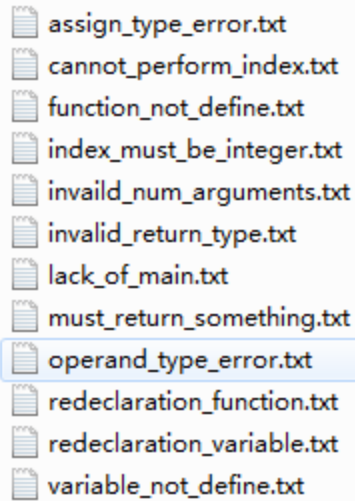
6.2 Test suites

Our tests are broken up into two main groups: functionality tests and semantic tests. Functionality tests (located in the /myplt_test folder) are used to test the functionality of our code. Each of these tests should pass. Semantic tests are used to ensure that code that does not pass the semantics required by the LRM do not get through to the compilation stage. All these tests should fail and print out the corresponding error message(located in the /error_test).

Functionality tests:



Semantic tests:



- assign_type_error.txt
- cannot_perform_index.txt
- function_not_define.txt
- index_must_be_integer.txt
- invaield_num_arguments.txt
- invalid_return_type.txt
- lack_of_main.txt
- must_return_something.txt
- operand_type_error.txt
- redeclaration_function.txt
- redeclaration_variable.txt
- variable_not_define.txt

6.3 Test detail

We choose our functionality tests based on compiler which includes all the functionalities our language should support; we choose our semantic tests based on asttosast.ml which performs intensive type checking.

We use shell script to automate our tests.

Our test cases were written by all members, separating by parts. They covered primitive type operations, statement control flow, array operations, and expressions. The test cases are as simple and independent as possible, so that we can identify where the problem is.

7. Lessons Learned

7.1 Xiao Xu

I feel lucky to pick this hard but always meaningful and cheerful class this semester. For the first time, I experienced a huge project all along the semester and I followed the project from the birth of it until it was finally done, rather than concentrate on some parts of it.

OCaml is a powerful language. I learn a lot about OCaml, about how a programming language is generated through the project. Here, I must feel exceedingly thankful to my teammates. We really make up an extraordinary team and have a great job. I enjoy the time we worked together, helped and trust each other.

7.2 Xiuming Dou

I learned a lot from the project. Listening carefully in class helped us understand how to do the project quickly. The examples in slides give us a good start. Good understanding of Ocaml will speed up the project. And I think doing homework 2 is effective way to understand Ocaml.

We communicated a lot in the meeting, this brain storm not only help everyone to understand the project thoroughly, but also boost some good ideas to solve problems. The work division to each member accelerates the project efficiently.

7.3 Yibo Zhu

We would be able to implement much more functionalities if we start earlier. Well, to start early is itself challenging. Since our understanding of the project is shallow and unknowns are lot. If I am to take this course again, I would look ahead of the lectures and try the most difficult parts (semantic checking/compiler) as soon as possible.

Another thing is that designing a suitable language for the project is crucial. It largely determines how much work you might do: static typed, scoped ... stuffs are generally easier then dynamic stuffs; Strongly typed requires intensive type checking ...

7.4 Xiang Ma

Through the course of this project, there are several things that I learned. First, I have an insight understanding of how scanner and parser work together to build the Ast tree. Second, I learned how to type checking to generate Sast tree from Ast. Third, I know how compile and interpreter implement. Forth, the skill of programming in ocaml has been improved largely. I have a deep understanding of recursive.

To sum up, by the project, I know how a program is solved by compiler. I think it is very useful for me in the future.

7.5 Ziyue Chen

It is my first time to design a language and write a compiler for that. And Ocaml is a new language for me. Although, it is hard, I find it very interesting and powerful. I learn that I should practice more with Ocaml before I start to write our code. During the project, I gradually learned how the whole compiler works and start to write Ast and semantic check for our language with my teammate. I think semantic check, compiler and bytecode interpreter are most challenging part. Overall, I feel great to finish this project with my teammates.

8. Appendix

8.1 Makefile

(*Written by Yibo Zhu*)

```
OBJS = Types.cmo ast.cmo sast.cmo asttosast.cmo parser.cmo scanner.cmo bytecode.cmo  
compile.cmo execute.cmo toplevel.cmo
```

```
TARFILES = scanner.mll parser.mly \
```

```
    ast.ml bytecode.ml compile.ml execute.ml toplevel.ml
```

```
toplevel : $(OBJS)
```

```
ocamlc -g -o toplevel $(OBJS)
```

```
scanner.ml : scanner.mll
```

```
    ocamllex scanner.mll
```

```
parser.ml: parser.mli
```

```
    ocamlc -c parser.mli
```

```
parser.mli: parser.mly
```

```
    ocamlyacc -v parser.mly
```

```
%.cmo : %.ml
```

```
    ocamlc -g -c $<
```

```
%.cmi : %.mli
```

```
    ocamlc -g -c $<
```

```
oplevel.tar.gz : $(TARFILES)
```

```
    cd .. && tar czf toplevel.tar.gz $(TARFILES:%=oplevel/%)
```

8.2 Scanner

(*Written by team as group study case*)

```
{ open Parser }
```

```
let letter = ['a'-'z' 'A'-'Z']
```

```
let digit = ['0'-'9']
```

```
rule token = parse
```

```
  [' '\t' '\r' '\n']      {token lexbuf}
```

```
  | '='   { ASSIGN }
```

```
  | '+'   { PLUS }
```

```
  | '-'   { MINUS }
```

| '*' { TIMES }
| '/' { DIVIDE }
| ".+" { DADD }
| ".-" { DSUB }
| ".*" { DMULT }
| "./" { DDIV }
| "&&" { AND }
| "||" { OR }
| "==" { EQ }
| "!=" { NEQ }
| ">" { GT }
| "<" { LT }
| ">=" { GEQ }
| "<=" { LEQ }
| '(' { LPAREN }
| ')' { RPAREN }
| '[' { LBRACKET }
| ']' { RBRACKET }
| '{' { LBRACE }
| '}' { RBRACE }
| ';' { SEMI }
| ',' { COMMA }
| '%' { PERCENT }
| \"\"[^\"]*\" as lxm { STRING(lxm) }

```

| "if" { IF}
| "while" { WHILE}
| "else" { ELSE}
| "int" { INT }
| "Creatarray"{CREATEARRAY}
| "string" {STR}
| "intarray" {INTARRAY}
(* | "strarray" {STRARRAY} *)
| "def" {DEF}
| "return" {RETURN}
| letter(letter|digit|'_' )* as id {ID(id)}
| digit+ as lit {INTEGER(int_of_string lit)}
| "#~" {comment lexbuf}
| eof {EOF}
| _ as invaildchar {raise (Failure("illegal character "^ Char.escaped invaildchar))}
and comment =
    parse "~#" {token lexbuf}
    | _ {comment lexbuf}

```

8.3 Parser

(*Written by team as group study case*)

```
%{ open Ast%}
```

```
%{ open Types%}
```

```
%token ASSIGN LPAREN RPAREN LBRACKET RBRACKET LBRACE RBRACE SEMI COMMA PERCENT  
DEF EOF
```

```
%token IF WHILE ELSE INT STR INTARRAY RETURN
```

```
%token PLUS MINUS TIMES DIVIDE DADD DSUB DMULT DDIV AND OR EQ NEQ GT LT GEQ LEQ
```

```
%token INTARRAY CREATEARRAY
```

```
%token <int> INTEGER
```

```
%token <string> STRING ID
```

```
%nonassoc NOELSE
```

```
%nonassoc ELSE
```

```
%right ASSIGN
```

```
%left EQ NEQ
```

```
%left OR
```

```
%left AND
```

```
%left LT GT LEQ GEQ
```

```
%left PLUS MINUS DADD DSUB
```

```
%left TIMES DIVIDE DMULT DDIV
```

```
%start program
```

```
%type <Ast.program> program
```

```
%%
```

```
program:
```

```
/* nothing */ { [], [] }
```

```
| program vdecl { ($2 :: fst $1), snd $1 }
```

```
| program fdecl { fst $1, ($2 :: snd $1) }
```

```
fdecl:
```

```
DEF types ID LPAREN formals_opt RPAREN LBRACE vdecl_list stmt_list RBRACE
{ { fname = $3;
formals = $5;
locals = List.rev $8;
body = List.rev $9;
returntype = $2; } }
formals_opt:
/* nothing */ { [] }
| formal_list { List.rev $1 }

formal_decl:
    types ID {($2, $1)}

formal_list:
    formal_decl { [{vname=fst $1; vtype=snd $1;vsize=1;}] }
| formal_list COMMA formal_decl { {vname= fst $3; vtype=snd $3;vsize=1;>::$1}

types:
    INT {Types.Int}
| STR {Types.Str}
| INTARRAY {Types.Arr}

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

vdecl:

```

INT ID SEMI { {vname = $2; vtype=Types.Int; vsize=1;} }
| STR LBRACKET INTEGER RBRACKET ID SEMI { {vname = $5; vtype=Types.Str; vsize=$3;} }
| INTARRAY LBRACKET INTEGER RBRACKET ID SEMI {{vname = $5; vtype=Types.Arr;vsize = $3;}}
/* | INTARRAY ID ASSIGN listvalue SEMI { {vname = $2; vtype=Types.Arr; vvalue=Array(fst
$4);vsize = snd $4;} }*/

```

stmt_list:

```

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

```

stmt:

```

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

```

expr:

```

INTEGER { Integer($1) }
| STRING { String ($1)}
| 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) }
| ID ASSIGN expr { Assign($1, $3) }
| ID LPAREN actuals_opt RPAREN { Call($1, $3) }
| LPAREN expr RPAREN { $2 }
| listvalue {Array(List.rev(fst $1))}

/*| ID LBRACKET INTEGER RBRACKET {Index($1,Integer($3))} */ /*match index*/

| ID LBRACKET expr RBRACKET {Index($1,$3)}

/*our version*/

| expr DADD expr {Binop($1, Dadd, $3)}
| expr DSUB expr {Binop($1, Dsub, $3)}
| expr DMULT expr {Binop($1, Dmult, $3)}
| expr DDIV expr {Binop($1, Ddiv, $3)}
| expr AND expr {Binop($1, And, $3)}
| expr OR expr {Binop($1, Or, $3)}

listvalue:

PERCENT LPAREN listintelement RPAREN PERCENT{{fst $3,snd $3}} /*int array*/

```


listintelement:

| INTEGER {[Integer(\$1),1]}

| listintelement COMMA INTEGER {(Integer(\$3)::(fst \$1),snd \$1 + 1)}

actuals_opt:

/* nothing */ { [] }

| actuals_list { List.rev \$1 }

actuals_list:

expr { [\$1] }

| actuals_list COMMA expr { \$3 :: \$1 }

8.4 AST

(*Written by team as group study case*)

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

type expr =

Integer of int

| String of string

| Array of expr list

| Id of string

| Binop of expr * op * expr

| Assign of string * expr

```
| Call of string * expr list
| Index of string * expr
| Noexpr
(*type constant_type =
    Integer of int
    |String of string
    | Array of expr list
*)
```

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

```
type variable_decl = {
    vname: string;
    vtype: Types.t;
    (*value: expr;*)
    vsize: int;
}
```

```
type func_decl = {  
  fname : string;  
  formals : variable_decl list;  
  locals : variable_decl list;  
  body : stmt list;  
  returntype : Types.t;  
}  
  
type program = variable_decl list * func_decl list
```

(*test if the parser and ast is correct*)

```
let string_of_op = function
```

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

```
| Dadd ->"+"
```

```
| Dsub -> "-"
```

```
| Dmult ->".*"
```

```
| Ddiv ->"/"
```

```
let rec string_of_expr = function
```

```
    Integer(i) -> string_of_int i
```

```
  | String(s) -> s
```

```
  | Array(a) -> "(" ^ String.concat "," (List.map string_of_expr a) ^ ")"
```

```
  | Id (i) -> i
```

```
  | Binop(e1,o,e2) -> string_of_expr e1 ^ " " ^ string_of_op o ^ " " ^ string_of_expr e2
```

```
  | Assign(l, e) -> l ^ "=" ^ string_of_expr e
```

```
  | Call( f, e) -> f ^ "(" ^ String.concat "," (List.map string_of_expr e) ^ ")"
```

```
  | Index(i, e) -> i ^ "[" ^ string_of_expr e ^ "]"
```

```
  | Noexpr ->""
```

```
let rec string_of_obj_type t =match t with
```

```
    Types.Int-> "int"
```

```
  | Types.Str ->"string"
```

```
  | Types.Arr -> "array"
```

```
let string_of_vdecl id =
```

```
    string_of_obj_type id.vtype ^ " " ^ id.vname ^ " = " ^ string_of_int id.vsize ^ ";\n"
```

```
let rec string_of_stmt = function
```

Block(stmts) ->

```
"{\n" ^ String.concat "" (List.map string_of_stmt stmts) ^ "}\n"
| Expr(e) -> string_of_expr e ^ ";\n";
| Return(e) -> "return " ^ string_of_expr e ^ ";\n";
| If(e, s1, s2) -> "if (" ^ string_of_expr e ^ ")\n" ^
  string_of_stmt s1 ^ "else\n" ^ string_of_stmt s2
| While(e, s) -> "while (" ^ string_of_expr e ^ ") " ^ string_of_stmt s
```

let string_of_fdecl fdecl =

```
string_of_obj_type fdecl.returntype ^ " " ^
fdecl.fname ^ "(" ^ String.concat ", " (List.map string_of_vdecl
fdecl.formals) ^ ")\n{\n" ^
String.concat "" (List.map string_of_vdecl fdecl.locals) ^
String.concat "" (List.map string_of_stmt fdecl.body) ^
"}\n"
```

let string_of_program (vars, funcs) =

```
String.concat "" (List.map string_of_vdecl vars) ^ "\n" ^
String.concat "\n" (List.map string_of_fdecl funcs)
```

8.5 Semantics Checking

(*Written by Yibo Zhu, assisted by Ziyue Chen*)

open Ast

open Sast

open Types

```
type symbol_table = {  
  parent : symbol_table option;  
  variables : Sast.variable_decl list;  
  functions: Sast.function_decl list;  
}
```

```
type trans_env = {  
  scope : symbol_table;  
}
```

```
let rec find_variable (scope : symbol_table) name =  
  try  
    List.find (fun v -> v.v_name = name) scope.variables  
  with Not_found ->  
    match scope.parent with  
    Some(parent) -> find_variable parent name  
    | _ -> raise (Failure("variable not defined"))
```

```
let var_exists scope name =  
  List.exists (fun v -> v.v_name = name) scope.variables
```

```
let rec find_function (scope : symbol_table) name =
```

```
try
  List.find (fun f -> f.fname = name) scope.functions
with Not_found ->
  match scope.parent with
  Some(parent) -> find_function parent name
  | _ -> raise (Failure("function not defined"))

let func_exists scope name =
  List.exists (fun f -> f.fname = name) scope.functions

let assign_allowed lt rt =
  lt = rt

let can_assign (lt:Types.t) rval =
  let (_, rt) = rval in
  if assign_allowed lt rt then
    rval
  else
    raise (Failure("Assign Type Error" ))

let rec check_int_array b = match b with
[] -> true
| h::t -> if snd h = Types.Int then check_int_array t else false
```

```
let can_op lval op rval =  
  let (_, lt) = lval  
  and (_, rt) = rval in  
  let type_match = (lt = rt) in  
  let int_int = (lt = Types.Int && rt = Types.Int) in  
    let int_array = (lt = Types.Int && rt = Types.Arr) in  
    let str_array = (lt = Types.Str && rt = Types.Arr) in  
  
    let array_int = (lt = Types.Arr && rt = Types.Int) in  
    let array_str = (lt = Types.Arr && rt = Types.Str) in  
    let array_array = (lt = Types.Arr && rt = Types.Arr) in  
  
  let array_int = (lt = Types.Arr && rt = Types.Int) in  
  let result = match op with  
    | Ast.Add   -> (type_match | | array_int | | int_array), (if (int_array) then rt else lt)  
    | Ast.Sub   -> (int_int), lt  
    | Ast.Mult  -> if (int_int | | array_array) then (true, lt) else (int_array | | str_array), rt  
    | Ast.Div   -> (int_int), lt  
    | Ast.Equal -> (true, Types.Int)  
    | Ast.Neq   -> (true, Types.Int)  
    | Ast.Less  -> (int_int), Types.Int  
    | Ast.Leq   -> (int_int), Types.Int  
    | Ast.Greater -> (int_int), Types.Int  
    | Ast.Geq   -> (int_int), Types.Int
```



```
| Ast.And   -> (int_int), Types.Int
```

```
| Ast.Or    -> (int_int), Types.Int
```

```
| Ast.Dadd  -> ( array_int), It
```

```
| Ast.Dsub  -> ( array_int), It
```

```
| Ast.Dmult -> ( array_int), It
```

```
| Ast.Ddiv  -> ( array_int), It
```

```
in if fst result then
```

```
  snd result
```

```
else
```

```
  raise (Failure("operand type miss match"))
```

```
let translate (globals, funcs) =
```

```
  let rec trans_expr env = function
```

```
    Ast.Integer(i) -> Sast.Integer(i), Types.Int
```

```
  | Ast.String(s) -> Sast.String(s), Types.Str
```

```
  | Ast.Id(n) ->
```

```
    let vdecl = (find_variable env.scope n) in
```

```
    Sast.Id(vdecl), vdecl.v_type
```

```
  | Ast.Binop(e1, op, e2) ->
```

```
    let e1 = trans_expr env e1
```

```
    and e2 = trans_expr env e2 in
```

```
      let rtype = can_op e1 op e2 in
```

```
      Sast.Binop(e1, op, e2), rtype
```

```
  | Ast.Call(n, a) ->
```

```
let fdecl = (find_function env.scope n) in
let types =
  List.map (fun v -> v.v_type) (List.rev fdecl.fformals) in
let args =
  List.map (fun s -> (trans_expr env s)) a in

let checked_args = try
  List.map2 can_assign types args
with Invalid_argument(x) ->
  raise (Failure("invalid number of arguments")) in
Sast.Call(fdecl, checked_args), fdecl.freturntype
| Ast.Assign(n, e) ->
  let vdecl = (find_variable env.scope n) in
  let aval = (trans_expr env e) in
  Sast.Assign(vdecl, (can_assign vdecl.v_type aval)), vdecl.v_type
| Ast.Noexpr ->
  Sast.Noexpr, Types.Int
| Ast.Array(a) -> let b = List.map (fun v -> trans_expr env v) a in

  if check_int_array b then

    Sast.Array(b), Types.Array

  else
```

```

    raise (Failure("Int array elements must all be int"))

    | Ast.Index(a,b) -> let a = (find_variable env.scope a)
                                                                    and b
= trans_expr env b in

    if snd b <> Types.Int then

        raise (Failure("index must be an integer!"))

                                                                    else

        if a.v_type = Types.Str then

            Sast.Index(a,b),Types.Str

        else

            if a.v_type = Types.Arr then

                Sast.Index(a,b), Types.Int

            else

                raise(Failure("can not perform indexing on this type of variable"))

    in let rec trans_stmt env = function
Ast.Block(s) ->
    let scope' = {parent = Some(env.scope); variables = []; functions = []}
    in let env' = {env with scope = scope'}
    in let s' = List.map (fun s -> trans_stmt env' s) s

```

```

    in Sast.Block(s')
  | Ast.Expr(e) ->
    Sast.Expr(trans_expr env e)
  | Ast.Return(e) ->
    Sast.Return(trans_expr env e)
  | Ast.If (e, s1, s2) ->
    let e' = trans_expr env e
    in Sast.If(can_assign Types.Int e', trans_stmt env s1, trans_stmt env s2)
  | Ast.While (e, s) ->
    let e' = trans_expr env e
    in Sast.While(can_assign Types.Int e', trans_stmt env s)
in let add_local env v =
  let evaluate = match (var_exists env.scope v.vname) with
    true -> raise (Failure("redeclaration of variable"))
    | false -> 0
  in let new_v = {
    v_name = v.vname;
    v_type = v.vtype;
    v_size = v.vsize;
  }
  in let vars = new_v :: env.scope.variables
  in let scope' = {env.scope with variables = vars}
  in {env with scope = scope'}

```

```

    in let add_func env f =
let new_f = match ((var_exists env.scope f.fname) || (func_exists env.scope f.fname)) with
  true -> raise (Failure("redeclaration of function"))
| false -> {
  freturntype = f.returntype;
  ffname = f.fname;
  fformals = [];
  flocales = [];
  fbody = [];
  parsed = false;
}

in let funcs = new_f :: env.scope.functions

in let scope' = {env.scope with functions = funcs}

in {env with scope = scope'}

in let trans_func env (f:Ast.func_decl) =
  let sf = find_function env.scope f.fname
  in let functions' = List.filter (fun f -> f.ffname != sf.ffname) env.scope.functions
  in let scope' = {parent = Some(env.scope); variables = []; functions = []}
  in let env' = {env with scope = scope'}
  in let env' = List.fold_left add_local env' (f.formals)
  in let formals' = env'.scope.variables
  in let env' = List.fold_left add_local env' (f.locals)
  in let remove v =

```

```
    not (List.exists (fun fv -> fv.v_name = v.v_name) formals')
in let locals' = List.filter remove env'.scope.variables
in let body' = List.map (fun f -> trans_stmt env' f) (f.body)
in let new_f = {
  sf with
  fformals = formals';
  flocales = locals';
  fbody = body';
  parsed = true;
}
in let funcs = new_f :: functions'
in let scope' = {env.scope with functions = funcs}
in {env with scope = scope'}
in let validate_func f =
let is_return = function
  Sast.Return(e) -> true
  | _ -> false
in let valid_return = function
  Sast.Return(e) -> if assign_allowed f.freturntype (snd e) then
    true
  else
    raise (Failure( "Invalid return type "
    ))
```

```
| _ -> false

in let returns = List.filter is_return f.fbody

in let returns_valid = List.for_all valid_return returns

in let return_count = List.length returns

in if (return_count = 0 && f.ffmpeg <> "print" && f.ffmpeg <> "printarray" && f.ffmpeg <>
"printstring" ) then

    raise (Failure( " must return something" ))

else

    f

in let make_print t =

{

    freturntype = Types.Int;

    ffmpeg = if (t = Types.Str) then "printstring" else "print";

    fformals = [{

        v_name = "val";

        v_type = t;

                                v_size=1;

    }];

    flocls = [];

    fbody = [];

    parsed = false;

}
```

```
in let global_scope = {
  parent = None;
  variables = [];
  functions = [{
    freturntype = Types.Int;
    ffname = "printarray";
    fformals = [{
      v_name = "val";
      v_type = Types.Arr;
      v_size=1;
    }];
    flocls = [];
    fbody = [];
    parsed = false;
  }] @ (List.map make_print [Types.Int; Types.Str;]);
}

  in let genv = {
    scope = global_scope;
  }

in let genv = List.fold_left add_local genv (List.rev globals)
in let genv = List.fold_left add_func genv (List.rev funcs)
in let genv = List.fold_left trans_func genv (List.rev funcs)
in (genv.scope.variables, List.map validate_func genv.scope.functions)
```


8.6 Compiler

(*Written by Yibo Zhu, assisted by Xiang Ma and Xiao Xu*)

open Sast

open Bytecode

module StringMap = Map.Make(String)

(* Symbol table: Information about all the names in scope *)

type env = {

function_index : int StringMap.t; (* Index for each function *)

global_index : int StringMap.t; (* "Address" for global variables *)

local_index : int StringMap.t; (* FP offset for args, locals *)

}

let rec remove_print = function

 []->[]

 | hd::tl->if (hd.ffmpeg="print" | |

hd.ffmpeg="printarray" | |hd.ffmpeg="printstring") then (remove_print tl) else
(hd::(remove_print tl))

(* val enum : int -> 'a list -> (int * 'a) list *)

let rec enum stride n = function

 [] -> []

 | hd::tl ->

 if stride > 0 then

 match hd.v_type with

```

Types.Int -> (n, hd.v_name) :: enum stride (n + 1) tl
|
Types.Str-> (* Here's the question: how many slots need to be allocated
to string? *)

(* To make it simpler, allocate 30 slots for it *)
(n + hd.v_size-1, hd.v_name) :: enum stride (n + hd.v_size) tl
|Types.Arr -> (n + hd.v_size-1, hd.v_name) :: enum stride (n+ hd.v_size) tl
|
_ -> raise(Failure ("Undefined type with variable" ^ hd.v_name))
else
match hd.v_type with
Types.Int -> (* Allocate global storage space for an int *)
(n, hd.v_name) :: enum stride (n + stride) tl
|
Types.Str -> (* Here's the question: how many slots need to be allocated
to string? *)

(* To make it simpler, allocate 30 slots for it *)
(n, hd.v_name) :: enum stride (n + stride * hd.v_size) tl
|
Types.Arr ->
(n, hd.v_name) :: enum stride (n+stride * hd.v_size) tl
|
_ -> raise(Failure ("Undefined type with variable " ^ hd.v_name))

```

```
let rec enum_func stride n = function
```

```
  [] -> []
```

```
  | hd::tl -> (n, hd) :: enum_func stride (n+stride) tl
```

```
let get_vari_size a vlist =
```

```
  List.fold_left (fun a b -> a + (match b.v_type with
```

```

Types.Int -> 1
| Types.Str ->
b.v_size*1
| Types.Arr -> b.v_size*1
| _ -> raise(Failure("Error in
get_vari_size !!"))
)) 0 vlist

```

```
(* val string_map_pairs StringMap 'a -> (int * 'a) list -> StringMap 'a *)
```

```
let string_map_pairs map pairs =
```

```
List.fold_left (fun m (i, n) -> StringMap.add n i m) map pairs
```

```
(** Translate a program in AST form into a bytecode program. Throw an
exception if something is wrong, e.g., a reference to an unknown
variable or function *)
```

```
let translate (globals, functions) =
```

```
(* Allocate "addresses" for each global variable *)
```

```
let global_indexes = string_map_pairs StringMap.empty (enum 1 0 globals) in
```

```
(* Assign indexes to function names; built-in "print" is special *)
```

```
let built_in_functions = StringMap.add "print" (-1) StringMap.empty in
```

```
let built_in_functions = StringMap.add "printarray" (-2) built_in_functions in
```

```

let built_in_functions = StringMap.add "printstring" (-3) built_in_functions in
let function_indexes = string_map_pairs built_in_functions
  (enum_func 1 1 (List.map (fun f -> f.fname) (remove_print functions))) in

(* Translate a function in AST form into a list of bytecode statements *)
let translate env fdecl =
  (* Bookkeeping: FP offsets for locals and arguments *)
  let num_formals = get_vari_size 0 fdecl.fformals
  and num_locals = get_vari_size 0 fdecl.flocals
  and local_offsets = enum 1 1 fdecl.flocals
  and formal_offsets = enum (-1) (-2) fdecl.fformals in
  let env = { env with local_index = string_map_pairs
    StringMap.empty (local_offsets @ formal_offsets) } in

let rec expr = function
  Integer i -> [Litin i]
  | String s -> [Litsh s]
  | Array a -> let rec f l = match l with
    [] -> []
    | hd::tl -> expr (fst hd)@f tl in List.rev (f a)

  | Index (x,y) -> (if (x.v_type = Types.Arr || x.v_type = Types.Str) then
    (try [Litin x.v_size]@[Lfpa
  (StringMap.find x.v_name env.local_index)]

```

```

with Not_found -> try [Litin x.v_size]@[Loda (StringMap.find x.v_name env.global_index)]
with Not_found -> raise (Failure ("undeclared variable " ^ x.v_name)))
                                else
                                                raise(Failure ("Indexing
performed on wrong type"))                                )@expr (fst y)@[GetC]
| Id s -> if s.v_type = Types.Int then
    (try [Lfp (StringMap.find s.v_name env.local_index)]
with Not_found -> try [Lod (StringMap.find s.v_name env.global_index)]
with Not_found -> raise (Failure ("undeclared variable " ^ s.v_name)))
                                else
                                                if (s.v_type = Types.Arr || s.v_type =
Types.Str) then
                                                (try [Litin s.v_size]@[Lfpa
(StringMap.find s.v_name env.local_index)]
with Not_found -> try [Litin s.v_size]@[Loda (StringMap.find s.v_name env.global_index)]
with Not_found -> raise (Failure ("undeclared variable " ^ s.v_name)))
                                                else
                                                raise(Failure ("Wrong type of
variable"))

| Binop (e1, op, e2) -> if ((snd e1) = Types.Arr&& op= Ast.Add) then
    expr (fst e2)@expr (fst e1)
    else if ((snd e2) = Types.Arr&& op= Ast.Add) then
    expr (fst e2)@ expr (fst e1)

```

```

Types.Arr) then
    else if ((snd e1) = Types.Arr && op = Ast.Mult && (snd e2) =
Types.Arr) then
        expr (fst e2) @ expr (fst e1)
    else if (op = Ast.Dmult | | op = Ast.Dadd | | op = Ast.Dsub | | op = Ast.Ddiv)
then
        expr (fst e1) @ [Litin (match (fst e1) with
                                | s -> s.v_size
                                | Array a -> List.length a)] @ expr (fst
e2) @ [Bin op]
        else
            expr (fst e1) @ expr (fst e2) @ [Bin op]

| Assign (s, e) -> expr (fst e) @ (if (s.v_type = Types.Arr | | s.v_type = Types.Str) then
                                (try [Litin
s.v_size] @ [Sfpa (StringMap.find s.v_name env.local_index)]
                                with Not_found -> try [Litin s.v_size] @ [Stra (StringMap.find
s.v_name env.global_index)]
                                with Not_found -> raise (Failure ("undeclared
variable " ^ s.v_name))))
        else
            if s.v_type = Types.Int then
                (try [Sfp (StringMap.find s.v_name env.local_index)]
                with Not_found -> try [Str (StringMap.find s.v_name env.global_index)]
                with Not_found -> raise (Failure ("undeclared variable " ^ s.v_name)))

```

```

else
    raise(Failure("can not
assign such type"))
)
| Call (funcdel, actuals) -> if (funcdel.ffname = "printarray") then
    expr (fst (List.hd actuals)) @[Litin (match (fst
(List.hd actuals)) with
    Array a -> List.length a
    | Id s -> s.v_size)
    ]@[Jsrf (StringMap.find
funcdel.ffname env.function_index)]
    else if (funcdel.ffname = "printstring" && (snd (List.hd actuals)) =
Types.Str) then
    expr (fst (List.hd actuals))@[Litin (match (fst
(List.hd actuals)) with
    String s -> String.length s
    | Id x -> x.v_size)]@[Jsrf (-3)]
    else
    (try
    (List.concat (List.map expr (List.map fst (List.rev actuals)))) @

```

```

    [Jsr (StringMap.find funcdecl.ffname env.function_index) ]
with Not_found -> raise (Failure ("undefined function " ^ funcdecl.ffname)))
| Noexpr -> []

```

```

in let rec stmt = function

```

```

    Block sl  -> List.concat (List.map stmt sl)
| Expr e    -> expr (fst e) (*@ [Drp]*)
| Return e  -> expr (fst e) @ [Rts num_formals]
| If (p, t, f) -> let t' = stmt t and f' = stmt f in
    expr (fst p) @ [Beq(2 + List.length t')] @
    t' @ [Bra(1 + List.length f')] @ f'
| While (e, b) ->
    let b' = stmt b and e' = expr (fst e) in
    [Bra (1+ List.length b')] @ b' @ e' @
    [Bne (-(List.length b' + List.length e'))]

```

```

in [Ent num_locals] @ (* Entry: allocate space for locals *)
stmt (Block fdecl.fbody) @ (* Body *)
[Litin 0; Rts num_formals] (* Default = return 0 *)

```

```

in let env = { function_index = function_indexes;
              global_index = global_indexes;
              local_index = StringMap.empty } in

```



```

(* Code executed to start the program: Jsr main; halt *)

let entry_function = try

  [Jsr (StringMap.find "main" function_indexes); Hlt]

with Not_found -> raise (Failure ("no \"main\" function"))

in

(* Compile the functions *)

let func_bodies = entry_function :: List.map (translate env) functions in

(* Calculate function entry points by adding their lengths *)

let (fun_offset_list, _) = List.fold_left

  (fun (l,i) f -> (i :: l, (i + List.length f))) ([],0) func_bodies in

let func_offset = Array.of_list (List.rev fun_offset_list) in

{ size_globals = get_vari_size 0 globals;

  (* Concatenate the compiled functions and replace the function

  indexes in Jsr statements with PC values *)

  text = Array.of_list (List.map (function

    Jsr i when i > 0 -> Jsr func_offset.(i)

  | _ as s -> s) (List.concat func_bodies))

}

```

8.7 Bytecode

(*Written by Xiuming Dou*)

open Ast

type bstmt =

(*push commands*)

| Litin of int (* Push a literal *)

| Litsh of string (*push string*)

| Drp (* Discard a value the bytecode interpreter will handle the different types that can be dropped*)

| Bin of Ast.op (* Perform arithmetic on top of stack *)

(*copy of global with id of int to stack top*)

| Lod of int (* puts global variable on top of stack *)

(*store stack object in global variables given id*)

| Str of int (* create global variable from top of stack *)

(*these stay the same from micro C*)

| Lfp of int (* Load frame pointer relative *)

| Sfp of int (* Store frame pointer relative *)

| Jsr of int (* Call function by absolute address *)

| Ent of int (* Push FP, FP -> SP, SP += i *)

| Rts of int (* Restore FP, SP, consume formals, push result *)

| Beq of int (* Branch relative if topofstack is zero *)

```

    | Bne of int (* Branch relative if topofstackis nonzero*)
    | Bra of int (* Branch relative *)
| Lfpa of int (* This is the start index of this array variable. Index is evaluated and
                put on top of stack in an int structure. *)

| Sfpa of int
| Loda of int
| Stra of int
    | Hlt (* Terminate *)
    | GetC (*get value specified by int on top of stack *)
type prog = {
    size_globals : int; (* Number of global variables *)
    text : bstmt array; (* Code for all the functions *)
}

```

```

let string_of_stmt = function
    Litin(i) -> "Litin " ^ string_of_int i
  |   Drp           -> "Drp"
  |   Bin(Ast.Add) -> "Add"
  | Bin(Ast.Sub) -> "Sub"
  | Bin(Ast.Mult) -> "Mult"
  | Bin(Ast.Div) -> "Div"
  | Bin(Ast.Equal) -> "Equal"
  | Bin(Ast.Neq) -> "Neq"
  | Bin(Ast.Less) -> "Less"

```

- | Bin(Ast.Leq) -> "Leq"
- | Bin(Ast.Geq) -> "Geq"
- | Bin(Ast.Greater) -> "Greater"
- | Bin(Ast.Dmult)->"Dmult"
 - | Bin(Ast.Dsub) ->"Dsub"
 - | Bin(Ast.Dadd) ->"Dadd"
 - | Bin(Ast.Ddiv) ->"Ddiv"
- | Bin(Ast.And) ->"And"
- | Bin(Ast.Or) ->"Or"
 - | Lod(i) -> "Lod " ^ string_of_int i
 - | Str(i) -> "Str " ^ string_of_int i
 - | Lfp(i) -> "Lfp " ^ string_of_int i
 - | Sfp(i) -> "Sfp " ^ string_of_int i
 - | Jsr(i) -> "Jsr " ^ string_of_int i
 - | Ent(i) -> "Ent " ^ string_of_int i
 - | Rts(i) -> "Rts " ^ string_of_int i
 - | Bne(i) -> "Bne " ^ string_of_int i
 - | Beq(i) -> "Beq " ^ string_of_int i
 - | Bra(i) -> "Bra " ^ string_of_int i
 - | Litsh(i) -> "Litsh" ^ i
 - | GetC -> "GetC"
 - | Loda(i) -> "Loda " ^ string_of_int i
 - | Stra(i) -> "Stra " ^ string_of_int i
 - | Lfpa(i) -> "Lfpa " ^ string_of_int i

```

      |      Sfpa(i) -> "Sfpa " ^ string_of_int i
|      Hlt->"Hlt"

```

```

let string_of_prog p =
  string_of_int p.size_globals ^ " slots to store global variables\n" ^
  let funca = Array.mapi
    (fun i s -> string_of_int i ^ " " ^ string_of_stmt s) p.text
  in String.concat "\n" (Array.to_list funca)

```

8.8 Bytecode Interpreter

(*Written by Xiuming Dou*)

```
open Ast
```

```
open Bytecode
```

```
let execute_prog prog =
```

```
let stack = Array.make 8192 0
```

```
and globals = Array.make 8192 0 in
```

```
let rec exec fp sp pc = match prog.text.(pc) with
```

```

Litin i -> stack.(sp) <- i ; exec fp (sp+1) (pc+1)

```

```

|Litsh str -> (let rec push_str_elements str' sl =

```

```

          if sl >= 0 then ( stack.(sp+(String.length str)-1-sl) <- int_of_char str'.[sl];

```

```

                                push_str_elements str' (sl-1);

    in push_str_elements str ((String.length str)-1);

    exec fp (sp+(String.length str)) (pc+1)

| Drp -> exec fp (sp-1) (pc+1)

| Bin op -> if (op=Add
| |op=Sub| |op=Mult| |op=Div| |op=Equal| |op=Neq| |op=Less| |op=Leq| |op=Greater| |op=Geq| |op=And| |op=Or) then

    (let op1 = stack.(sp-2) and op2 = stack.(sp-1) in

    stack.(sp-2) <- (let boolean i = if i then 1 else 0 in

    match op with

    Add -> op1 + op2

    | Sub -> op1 - op2

    | Mult -> op1 * op2

    | Div -> op1 / op2

    | Equal -> boolean (op1 = op2)

    | Neq -> boolean (op1 != op2)

    | Less -> boolean (op1 < op2)

    | Leq -> boolean (op1 <= op2)

    | Greater -> boolean (op1 > op2)

    | Geq -> boolean (op1 >= op2)

    | And -> if (op1<>0 &&op2<>0) then 1 else 0

    | Or -> if (op1=0 && op2 =0) then 0 else 1);

    exec fp (sp-1) (pc+1)

```

```
)  
else  
((match op with  
Dmult -> (let rec d_mult offset =  
            if offset>=0 then (stack.(sp-3-offset) <- stack.(sp-3-offset) * stack.(sp-1);  
                                d_mult (offset-1))  
            in d_mult (stack.(sp-2)-1))  
  
|Dadd-> (let rec d_add offset =  
            if offset>=0 then (stack.(sp-3-offset) <-  
stack.(sp-3-offset) + stack.(sp-1);  
                                d_add (offset-1))  
            in d_add (stack.(sp-2)-1))  
  
|Dsub-> (let rec d_sub offset =  
            if offset>=0 then (stack.(sp-3-offset) <-  
stack.(sp-3-offset) - stack.(sp-1);  
                                d_sub (offset-1))  
            in d_sub (stack.(sp-2)-1))  
  
|Ddiv-> (let rec d_div offset =  
            if offset>=0 then (stack.(sp-3-offset) <-  
stack.(sp-3-offset) / stack.(sp-1);  
                                d_div (offset-1))  
            in d_div (stack.(sp-2)-1))  
  
);exec fp (sp-2) (pc+1))
```

```
|Lod i-> stack.(sp) <- globals.(i) ; exec fp (sp+1) (pc+1)
```

```
|Loda i -> ( let length=stack.(sp-1) in
```

```
    let rec load_arr_globals i' offset=
```

```
        if offset >= 0 then (stack.(sp-2+length-offset) <- globals.(i'-offset);
```

```
            load_arr_globals i' (offset-1))
```

```
    in (load_arr_globals i (stack.(sp-1)-1);exec fp (sp+length-1) (pc+1)))
```

```
|Str i-> globals.(i) <- stack.(sp-1) ; exec fp sp (pc+1)
```

```
|Stra i -> (let rec push_arr_globals i' offset =
```

```
    if offset >= 0 then (globals.(i'-offset) <- stack.(sp-2-offset);
```

```
        push_arr_globals i' (offset-1))
```

```
    in push_arr_globals i (stack.(sp-1)-1));
```

```
    exec fp (sp-1) (pc+1);
```



```
|Lfp i-> stack.(sp) <- stack.(fp+i) ; exec fp (sp+1) (pc+1)
```

```
|Lfpa i -> ( let length=stack.(sp-1) in
              let rec load_arr_elements i' offset=
                  if offset >= 0 then (stack.(sp-2+length-offset) <- stack.(fp+i'-offset);
                                      load_arr_elements i' (offset-1))
              in (load_arr_elements i (stack.(sp-1)-1);exec fp (sp+length-1) (pc+1)))
```

```
|GetC -> stack.(sp) <- stack.(sp-1-stack.(sp-1)-1); exec fp (sp+1) (pc+1)
```

```
|Sfp i-> stack.(fp+i) <- stack.(sp-1) ; exec fp sp (pc+1)
```

```
|Sfpa i-> (let rec push_arr_elements i' offset =
            if offset >= 0 then (stack.(fp+i'-offset) <- stack.(sp-2-offset);
                                push_arr_elements i' (offset-1))
          in push_arr_elements i (stack.(sp-1)-1));
   exec fp (sp-1) (pc+1);
```

```
|Js(-1)-> print_endline (string_of_int stack.(sp-1)) ; exec fp sp (pc+1)
```

```
|Js(-2) -> (let rec print_array offset =
```

```

        if offset >= 0 then (print_endline (string_of_int (stack.(sp-2)-(stack.(sp-1)-1-
offset))));

        print_array (offset-1)

    in print_array (stack.(sp-1)-1) );

exec fp sp (pc+1);

|Jsr(-3) -> (let rec print_string offset =

        if offset >= 0 then (print_char (char_of_int (stack.(sp-2)-(stack.(sp-1)-1-offset))));

        print_string (offset-1)

    in print_string (stack.(sp-1)-1) );

exec fp sp (pc+1);

| Jsr i -> stack.(sp) <- pc + 1 ; exec fp (sp+1) i

| Ent i -> stack.(sp) <- fp ; exec sp (sp+i+1) (pc+1)

| Rts i -> let new_fp = stack.(fp) and new_pc = stack.(fp-1) in stack.(fp-i-1) <- stack.(sp-1) ;exec new_fp
(fp-i) new_pc

| Beq i -> exec fp (sp-1) (pc + if stack.(sp-1) = 0 then i else 1)

| Bne i -> exec fp (sp-1) (pc + if stack.(sp-1) != 0 then i else 1)

| Bra i -> exec fp sp (pc+i)

| Hlt -> ()

in exec 0 0 0

```