Small-C Compiler Report

I. Introduction

In this project, I implement a code generator to translate the intermediate representation, which is produced by the syntax analyzer implemented in project 1, into LLVM instructions. The code generator returns a LLVM assembly program, which can be run on LLVM. After finishing this project, I get a compiler, which can translate small-C source programs to LLVM assembly programs.

Section II introduces the data structure of this compiler. Section III introduces semantic analysis I have done, and lists all semantic error that this compiler can detect. Section IV introduces optimization. Section V introduces instruction selection, register allocation method and how I handle I/O.

II. Data Structure

1、Parsing Tree Node

TreeNode is a class that describes a node in the parsing tree, meanwhile every node in parsing tree is represented by TreeNode to make the problem easier. The code of class TreeNode is shown below:

```
class TreeNode
{
public:
    TreeNode* child;
    TreeNode* sibling;

    int lineno;
    string node_type;//can be dec,stmt,exp,for,if etc
    string type;//can be int, CONST, array, struct

    int iVal;
    string name;

    TreeNode(int l=-1):child(nullptr),sibling(nullptr)
    {
        lineno = l;
        node_type = type = "";
    }
    ~TreeNode() {}
};
```

2、Abstract Parsing Tree

Class ParsingTree is the class that stores abstract parsing tree(AST). It has a root node named root. In the process of reducing, it creates a new TreeNode and links it with its children.

3、Code generation

CodeGenContext is the class that generates LLVM intermediate representation(IR). The basic idea of generating target code is that always deal with the left-most child first. Class CodeGenContext accepts the root of AST as input, and recursively deals with the left-most child (like post-order traverse). For example, in the beginning of code generation, the program will call generateCode(); in the body of generateCode(), it will call EXTDEFS_genCode(); in the body of EXTDEFS_genCode(), it will call EXTDEF_genCode() and EXTDEFS_genCode(); so on and so forth.

III. Semantic analysis

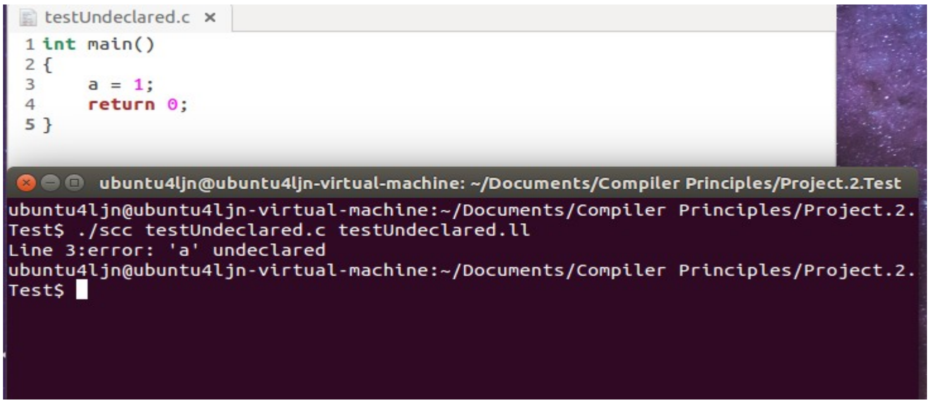
In this section I will introduce all semantic analysis I have done in this project.

1、Variables and functions should be declared before use. Given the input file below:

```
int main()
{
    a = 1;
    return 0;
}
```

the compiler will print error message:

Line 3:error: 'a' undeclared

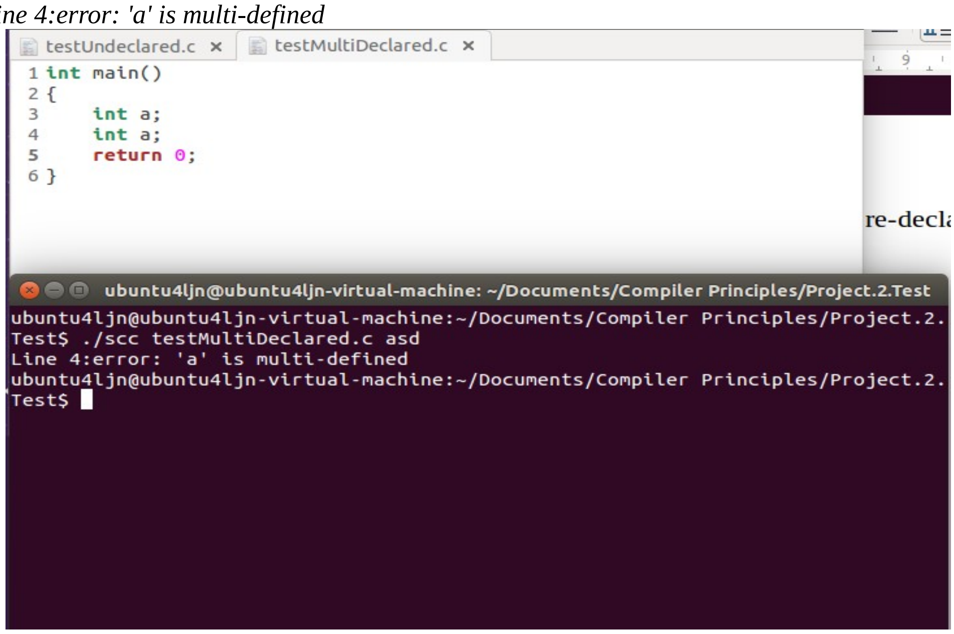


2、 Variables and functions should not be re-declared. Given the input file below:

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

the compiler will print error message:

Line 4:error: 'a' is multi-defined

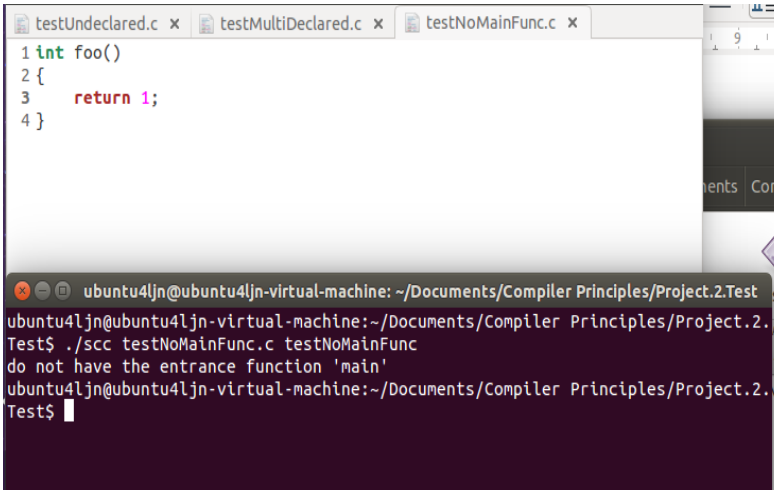


3、 Program must contain a function int main() to be the entrance. Given the code below:

```
int foo()
{
	return 1;
}
```

the compiler will print error message:

do not have the entrance function 'main'



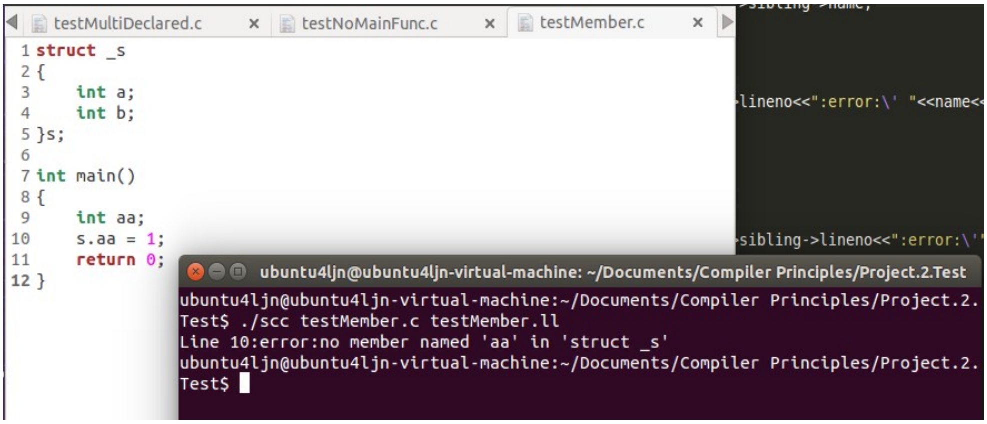
4、Operator. can only be used to an object of a struct. Given the code below:

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

int main()
{
        int aa;
        s.aa = 1;
        return 0;
}
```

the compiler will print error message:

Line 10:error:no member named 'aa' in 'struct _s'



5、Left-value cannot be an expression. Given the code below:

```
int a=1;
int main()
{
        a+1 = 1;
        return 0;
}
```

the compiler will print error message:

Line 4: lvalue cannot be an expression

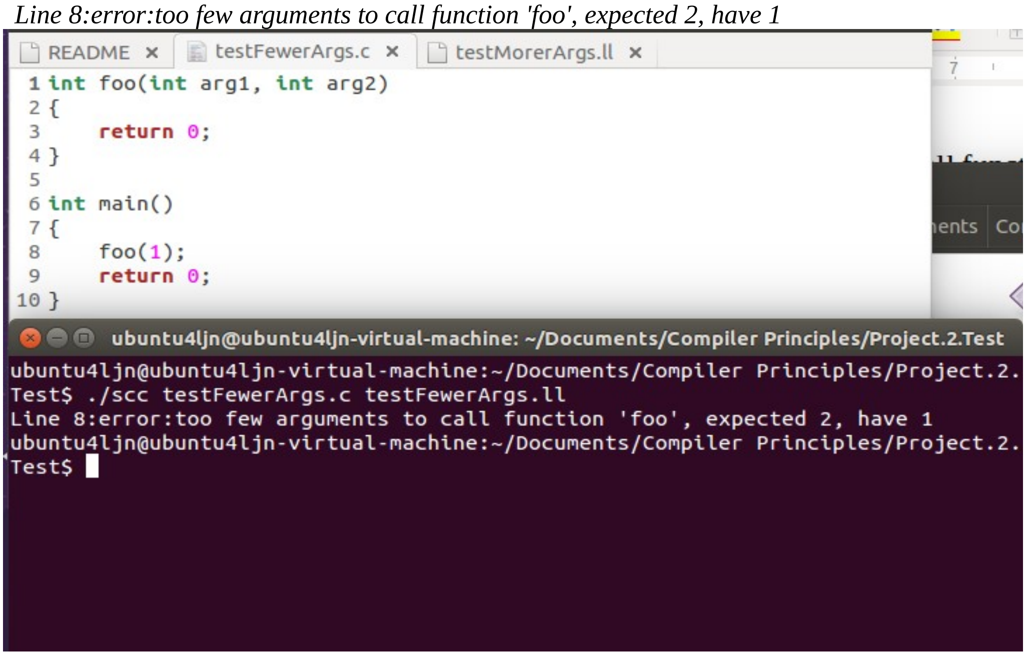
6、 Number of arguments in a function call should be exactly equal to the function prototype. Given the code below:

```
int foo(int arg1, int arg2)
{
        return 0;
}

int main()
{
        foo(1);
        return 0;
}
```

the compiler will print error message:

Line 8:error:too few arguments to call function 'foo', expected 2, have 1



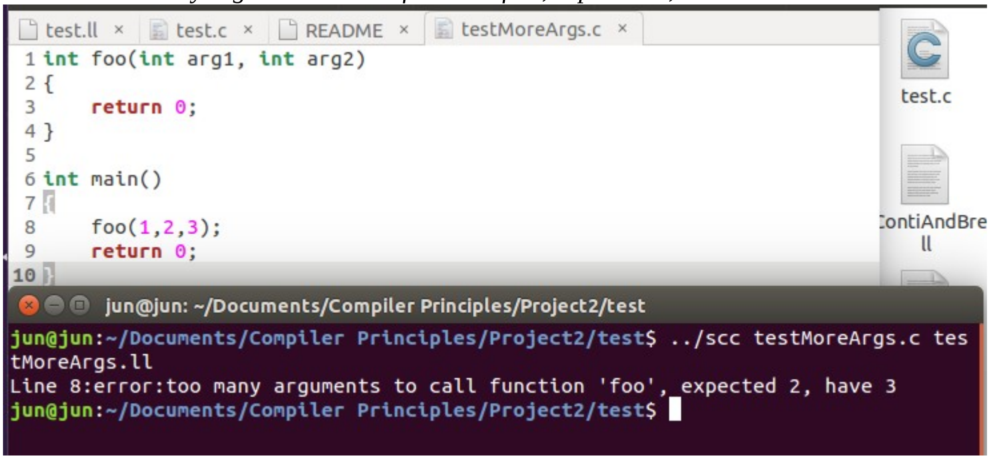
Given the code:

```
int foo(int arg1, int arg2)
{
        return 0;
}

int main()
{
        foo(1,2,3);
        return 0;
}
```

the compiler will print error message:

Line 8:error:too many arguments to call function 'foo', expected 2, have 3

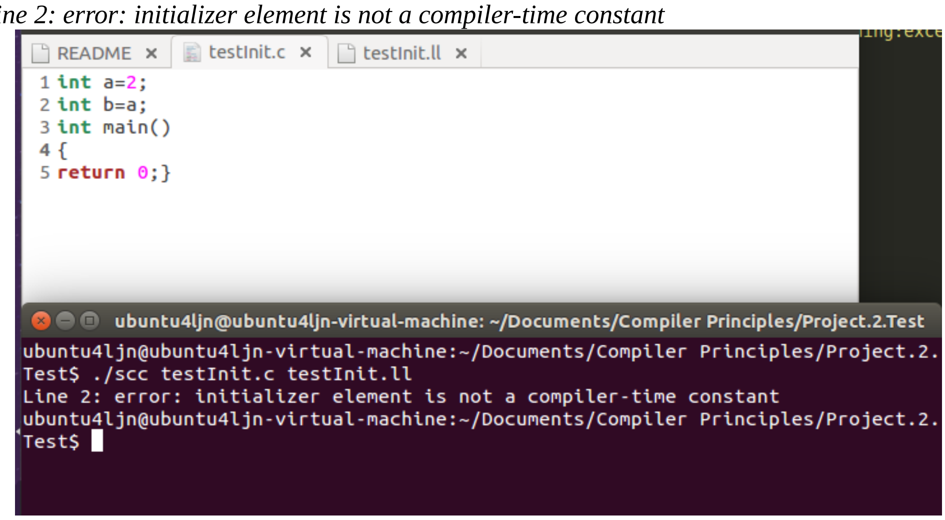


7、Initializer element should be a compiler-time constant. Given the code below:

int a=2;
int b=a;
int main()
{return 0;}

the compiler will print error message:

Line 2: error: initializer element is not a compiler-time constant



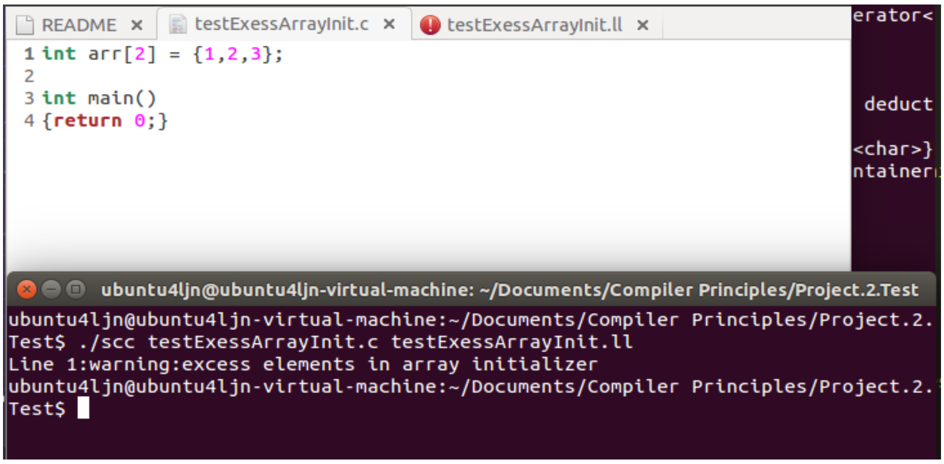
8、Number of elements in array initializer should not excess the length of array. Given the code below:

int arr[2] = {1,2,3};

int main()
{return 0;}

the compiler will print warning message:

Line 1:warning:excess elements in array initializer



9、 Argument of function read() should be a variable. Given the code below:

```
int main()
{
        int a;
        read(a);
        read(a+1);
        return a;
}
```

the compiler will print the error message:

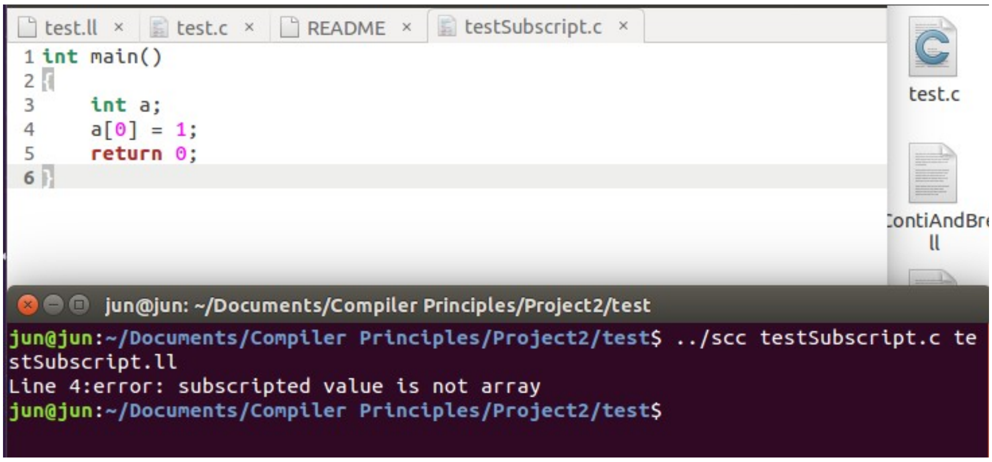
Line 5:error: args of function 'read' should be a variable

10、 Using operator[] to a non-array variable is not allowed. Given the code below:

```
int main()
{
        int a;
        a[0] = 1;
        return 0;
}
```

the compiler will print the error message:

Line 4:error: subscripted value is not array

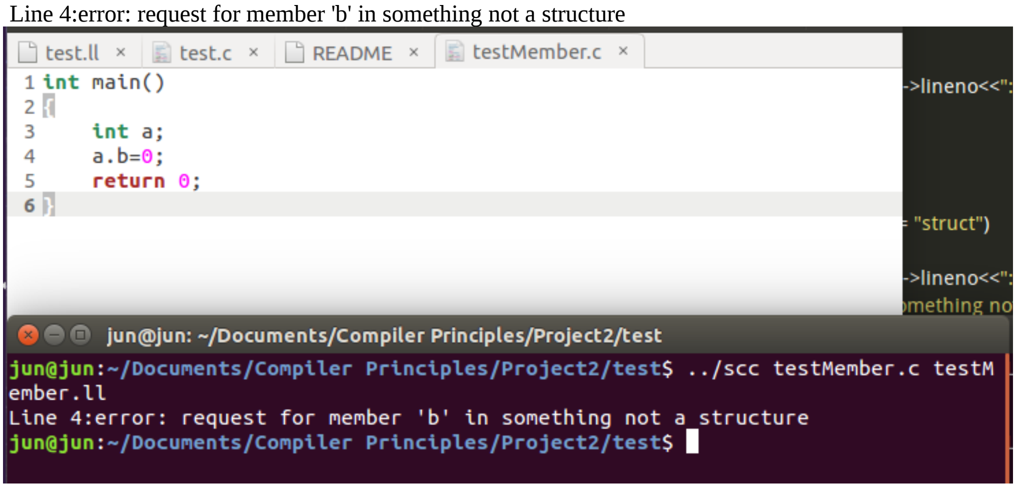


11、 Operator. can only be used to an object of a struct. Given the code below:

```
int main()
{
	int a;
	a.b=0;
	return 0;
}
```

the compiler will print error message:

Line 4:error: request for member 'b' in something not a structure



12、 a

IV. Optimization

Since LLVM IR is not three-address representation nor other intermediate representations introduced in textbook, I do no optimization in this project.

V. Machine-Code generation

1、 Instruction selection

LLVM official website offers a large amount of documents to help LLVM developers to learn how to use LLVM IR. After finishing reading LLVM language reference manual(http://llvm.org/docs/LangRef.html), I figure out a practical method to translate small c code to LLVM IR.

1) declaration

a) Suppose a is a global int variable and is initiated by 1, that is

int a = 1;

the IR is:

@a = global i32 1, align 4

if a has no initializer, the default value is 0

b) Suppose b is a local int variable and has no initializer, that is

int b;

the IR is:

%b = alloca i32, align 4

if b is initialized by 1, that is

int b = 1;

there is an additional IR instruction

store i32 1, i32* %b

c) Suppose c is a global array, that is

int c[4]={1,2};

the IR is:

c = global [4 x i32] [i32 1, i32 2, i32 0, i32 0], align 16

d) Suppose d is a local array and is partially initialized, that is

int d[3] = {1,2};

the IR is:

```
%d = alloca [3 x i32], align 4
%arrayIdx0 = getelementptr inbounds [3 x i32], [3 x i32]* %d, i32 0, i32 0
%arrayIdx1 = getelementptr inbounds [3 x i32], [3 x i32]* %d, i32 0, i32 1
%arrayIdx2 = getelementptr inbounds [3 x i32], [3 x i32]* %d, i32 0, i32 2
store i32 1, i32* %arrayIdx0, align 4
store i32 2, i32* %arrayIdx1, align 4
store i32 0, i32* %arrayIdx2, align 4
```

e) Suppose e is a structure with no tag, that is

struct {int a;int b;} s;

the IR is

```
%struct.anon1 = type { i32 , i32 }
%e = common global %struct.anon1 zeroinitializer, align 4
```

if e has tag _e, that is

struct _e {int a;int b;}e;

the IR is:

```
%struct._e = type { i32, i32 }
%e = common global %struct._e zeroinitializer, align 4
```

2) assignment

Suppose a and b are both local int variables. The small-c code

a = b

will be translated as:

```
%t1 = load i32, i32* %b, align 4
store i32 %t1, i32* %a, align 4
```

Suppose c[4] is a array, the small-c code

a = c[0]

will be translated as:

```
%t1 = getelementptr inbounds [4 x i32], [4 x i32]* %c, i32 0, i32 0
%t2 = load i32, i32* %t1, align 4
store i32 %t2, i32* %a, align 4
```

the small-c code

c[1] = a

will be translated as:

```
%t1 = load i32, i32* %a, align 4
%t2 = getelementptr inbounds [4 x i32], [4 x i32]* %c, i32 0, i32 1
%t3 = load i32, i32* %t2, align 4
store i32 %t1, i32 %t2
```

Actually the third instruction is no use. This is because the left value and right value of array indexing is different, but in order to simplify the problem, I apply the same method to both situation.

3) arithmetic expression

The section will use three temporary registers %t1, %t2, and %t3 as example.

Small C code	LLVM IR
t1=t2+t3	%t1 = add nsw i32 %t2, i32 %t3
t1=t2-t3	%t1 = sub nsw i32 %t2, i32 %t3
t1=t2*t3	%t1 = mul nsw i32 %t2, i32 %t3
t1=t2/t3	%t1 = div nsw i32 %t2, i32 %t3
t1=t2%t3	%t1 = srem i32 %t2, i32 %t3
t1=t2&t3	%t1 = and i32 %t2, i32 %t3
t1=t2|t3	%t1 = or i32 %t2, i32 %t3
t1=t2^t3	%t1 = xor i32 %t2, i32 %t3
t1=t2<<t3	%t1 = shl i32 %t2, i32 %t3
t1=t2>>t3	%t1 = ashr i32 %t2, i32 %t3

4) relationship expression

Suppose a and b are both local int variable, op can be 'eq', 'ne', 'slt', 'sle', 'sgt', 'sge' for logic relationship such as equal, not equal, less, less or equal, greater and greater or equal respectively,

the IR is

```
%t1 = load i32* %a, align 4
%t2 = load i32* %b, align 4
%t3 = icmp op i32 %t1, i32 %t2
```

5) logic expression

In c language, logic symbol can be &&, || and !, which combine boolean expressions. This compiler will calculate the value of each boolean expressions, and apply bit and/or/not to these boolean values. If the boolean expression is a variable or a constant, the boolean value is 1 if the value of the variable or constant is not 0; the boolean value is 0 if the value of the variable or constant is 0.

2、register allocation

Since the number of registers in LLVM IR is infinite, the register name of a variable can be arbitrarily chosen as long as it is up to the standard. In order to make the translated IR easy to read, I will use meaningful register name which I will explain below:

Temporal register name is %ti, where i starts from 0.

Register name of a global variable such as a is @a.

Register name of a local variable such as b is %a.

Register name of a structure such as c is %struct.c; if the structure has no name, the register name is %struct.anoni, where i starts from 0.

3、input and output

read() and write() functions are two special function in small-c language, and are similar to scanf() and printf() of c language. Using the command

```
clang -emit-llvm fileName -S
```

clang compiler will print the translation of IR of printf() and scanf().

Declare a string constant in the beginning of the target file

```
@.str = private unnamed_addr constant [3 x i8] c"%d\00", align 1
```

and add declaration in the end of the target file

```
declare i32 @__isoc99_scanf(i8*, ...)
declare i32 @printf(i8*, ...)
```

then the IR of

```
read(a);
```

is(a is a local variable):

```
%t1 = call i32 (i8*, ...) @__isoc99_scanf(i8* getelementptr inbounds ([3 x i8], [3 x i8]* @.str, i32 0, i32 0), i32* %a)
```

the IR of

```
write(a);
```

is:

```
%t1 = load i32, i32* %a, align 4
%t2 = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([3 x i8], [3 x i8]* @.str, i32 0, i32 0), i32 %t1)
```

VI. Conclusion

This compiler can pass all the test cases, and is also able to detect plenty of semantic errors, though there are still many semantic errors that the compiler cannot detect.