浙江大学

本科实验报告

实验名称:	一个类 C 语言编译器的设计
课程名称:	《编译原理》
开课学院:	计算机学院
组　　长:	汪俊威
组　　员:	康锦辉，林成楠
学　　号:	3180105099　3170105970　3150102362
指导老师:	王强

2020 年　5　月　31　日

Catalog

Ⅰ. Introduction

1. Description

Our work is a compiler of a C-like language. We have implemented some basic characteristics of the C language, including but not limited to variable, function, loop statement, conditional statement, assignment statement.

- **Variable declaration**:

  Syntax: $type\ var\_name;$

  e.g.

```
int x;
double y;
char c;
```

  Three basic types are supported: int, double, char.

- **Function declaration**:

  Syntax: $\$return\_type\$\ \$function\_name\$\ (\$type_1\$\ \$\arg_1\$, \$type_2\$\ \$\arg_2\$, \cdots);$

  e.g.

```
int func(int a, int b);
```

- **Function definition**:

  Syntax:
  $return\_type\ function\_name\ (\$type_1\$\ \$\arg_1\$, \$type_2\$\ \$\arg_2\$, \cdots)\{\ statements\ \}$

  e.g.

```
int SumOfTwoInt(int a, int b){
    int c;
    c=a+b;
    return c;
}
```

- **Loop statement**:

  "for" loop:

  Syntax: $for(statement; expression; statement)\ statement$

  e.g.

```
for(int i=0;i<3;i++) printf("Wa\n");
for(int i=0;i<3;i++) {
    printf("W");
    printf("a");
```

```
        printf(" Hello, world!\n");
    }
```

"while" loop:

Syntax: $while\ (\$expression\$)\ statement$

e.g.

```
int a;
a=0;
while(a<3) a++;
while(a<6){
    a++;
    printf("Wa\n");
}
```

- **Conditional statement**:

  "if-else" statement:

  Syntax:

  $if\ (\$expression\$)\ \$statement\$$
  or $if\ (\$expression\$)\ \$statement\$\ else\ \$statement\$$

  e.g.

```
int a; a=0;
if(a<0) a=1;
else a=2;

if(a<2){a=3;a++;}
else{a=4;}
```

- **Assignment statement**:

  Syntax: $\$variable\$\ \$assign\_operator\$\ \$expression\$;$

  e.g.

```
int a;
a=1;
a+=1; a-=1;     a/=1; a*=1; a%=1;
a<<=1; a>>=1;
a^=1; a|=1; a&=1;
```

2. **Files**

   Necessary files:

   `Scanner.l` `parser.y` `grammartreenode.h` `grammartreenode.cpp`

func.h cmm.out

Makefile demo.cpp and some other test files

Instructor files:

README.h AST.md Show.txt

Helpful files:

demo.cpp demo.cpp.ll demo.cpp.s demo.cpp.out

t1.c t2.c t3.c t4.c

3. Jobs assignment

Lexical： 汪俊威，康锦辉，林成楠

Grammar： 汪俊威，康锦辉

Semantic： 汪俊威，林成楠

IR codes ： 汪俊威，康锦辉

Machine codes： 汪俊威

Report ： 康锦辉，林成楠，汪俊威

PPT： 康锦辉

II . lexical Analysis

The task of lexical analysis is to scan the source program from left to right,Identify each type of word according to the lexical rules of the language and generate the corresponding token.

First, we give a set of formal definitions and state definitions to simplify the lexical rules that follow.

```
D       [0-9]                       //Digit
L       [a-zA-Z]                    //letter
H       [a-fA-F0-9]                 //hexadecimal
E       ([Ee][+-]?{Digit}+)         //Scientific count
```

P ([Pp][+-]?{Digit}+) //Binary scientific counting

FS (f|F|l|L) //Data type definition

IS ((u|U)|(u|U)?(l|L|ll|LL)|(l|L|ll|LL)(u|U)) //Data type definition

Second, identify the identifier and fixed symbol, generate Numbers

identifier							
auto	sizeof	goto	return	do	while	for	continue
break	switch	default	case	if	else	float	double
char	void	int	long	bool	const	short	signed
unsigned	static	extern	inline	typedef	struct	enum	union

Fixed symbols							
...	>>=	<<=	+=	-=	*=	/=	%=
&=	^=	|=	>>	<<	++	--	->
&&	||	<=	>=	==	!=	;	,
:	=	.	&	!	~	-	+
*	/	%	<	>	^	|	?
(	)	[	]	{	}		

Third, lexical rules are used to define variables and constants

{L}({L}|{D})* //variable

0[xX]{H}+{IS}? //int constant

0[0-7]*{IS}? //intconstant

[1-9]{D}*{IS}? //int constant

{D}+{E}{FS}? //double constant

{D}*"."{D}+{E}?{FS}? //double constant

{D}+"."{D}*{E}?{FS}? //double constant

Fourth, skip the comments and Spaces.

"//"[^\n]* //Handle "//" class comments

"/*" //handle /**/ class comments with mutiline_comment() function

[\t\v\n\f] //Ignore the blank space

III. Parsing

1. LALR(1)

An LALR parser or a Look-Ahead LR parser is a simplified version of a canonical LR parser, using bottom-up method, to parse (separate and analyze) a text according to a set of production rules specified by a formal grammar for a programming language. And LALR(1) is a specific version of LALR, which means looking ahead for one word.

2. Yacc

We use Yacc to generate a syntax analyzer. Yacc is a LALR(1) parsing generator. We define derivation rules and corresponding behaviors in a Yacc file, and after processing, it would generate a C header file and a C++ source file which include tools, or functions, which can be used to parse the syntax. In our work, we use Bison as a distro of Yacc.

3. Abstract syntax tree

The result of parsing would be stored in an abstract syntax tree (AST).

The structure of an AST node:

```
class ASTnode{
public:
    std::string content;
    std::string name;
    int line_no;
    int intval;
```

```
        double douval;
        float floval;
        std::string vtype;
        bool booval;
        ASTnode* left;
        ASTnode* right;

        ASTnode();
    };
```

The $name$ attribute is the type of node.

The $content$ attribute is the specific content of node, like the name of some variable, some constant, or some symbol.

The $line\_no$ attribute is the corresponding line number of the node.

The left child of a node points to its first sub-node at the next level, and the right child of a node points to its neighbored node.

We construct the AST when parsing. As we use Yacc to generate a syntax analyzer, we define the construction process in the behaviors that would be executed when some derivation is matched. In fact, the construction process also occurs when analyzing lexical, since tokens are terminals in derivations and leaf nodes in the abstract syntax tree.

The detail of derivations can be seen in source files.

```
program   <@1>
| translation_unit   <@1>
| | external_declaration   <@1>
| | | function_definition   <@1>
| | | | declaration_specifiers   <@1>
| | | | | type_specifier   <@1>
| | | | | | INT:int
| | | | declarator   <@1>
| | | | | direct_declarator   <@1>
| | | | | | direct_declarator   <@1>
| | | | | | | IDENTIFIER:main
| | | | | | | (   <@1>
| | | | | | | )   <@1>
| | | | compound_statement   <@1>
| | | | | {   <@1>
| | | | | block_item_list   <@2>
| | | | | | block_item_list   <@2>
| | | | | | | block_item_list   <@2>
| | | | | | | | block_item_list   <@2>
| | | | | | | | | block_item   <@2>
| | | | | | | | | | declaration   <@2>
| | | | | | | | | | | declaration_specifiers   <@2>
| | | | | | | | | | | | type_specifier   <@2>
| | | | | | | | | | | | | INT:int
| | | | | | | | | | | init_declarator_list   <@2>
| | | | | | | | | | | | init_declarator   <@2>
```

```
| | | | | | | | | | | | declarator   <@2>
| | | | | | | | | | | | | direct_declarator   <@2>
| | | | | | | | | | | | | | IDENTIFIER:i
| | | | | | | | | | | | | =   <@2>
| | | | | | | | | | | | | initializer   <@2>
| | | | | | | | | | | | | | assignment_expression   <@2>
| | | | | | | | | | | | | | | conditional_expression   <@2>
| | | | | | | | | | | | | | | | logical_or_expression   <@2>
| | | | | | | | | | | | | | | | | logical_and_expression   <@2>
| | | | | | | | | | | | | | | | | | inclusive_or_expression   <@2>
| | | | | | | | | | | | | | | | | | | exclusive_or_expression   <@2>
| | | | | | | | | | | | | | | | | | | | and_expression   <@2>
| | | | | | | | | | | | | | | | | | | | | equality_expression   <@2>
| | | | | | | | | | | | | | | | | | | | | | relational_expression   <@2>
| | | | | | | | | | | | | | | | | | | | | | | shift_expression   <@2>
| | | | | | | | | | | | | | | | | | | | | | | | additive_expression   <@2>
| | | | | | | | | | | | | | | | | | | | | | | | | multiplicative_expression   <@2>
| | | | | | | | | | | | | | | | | | | | | | | | | | cast_expression   <@2>
| | | | | | | | | | | | | | | | | | | | | | | | | | | unary_expression   <@2>
| | | | | | | | | | | | | | | | | | | | | | | | | | | | postfix_expression   <@2>
| | | | | | | | | | | | | | | | | | | | | | | | | | | | | primary_expression   <@2>
| | | | | | | | | | | | | | | | | | | | | | | | | | | | | | CONSTANT_INT:0
| | | | | | | | | | | ;   <@2>
| | | | | | | | block_item   <@3>
| | | | | | | | | declaration   <@3>
| | | | | | | | | | declaration_specifiers   <@3>
```

IV. Semantic Analysis

We analyze semantics from the root of the abstract syntax tree. We analyze the type of the node we meet, and decide the specific semantic according to the structure of the node and its child-nodes, until meeting a node without child-node, which means the mode we meet represents a terminal.

1. Attributes computing

We use the attributes between an AST node and its child-nodes to compute and pass the attributes.

In our consideration, only expressions have attributes to pass and compute. For simplicity, we present the our methods to compute attributes with derivations simplified and abbreviated.

$expr \rightarrow assign\_expr$

$$\$\$.val = \$1.val, \qquad \$\$.type = \$1.type$$

$assign\_expr \rightarrow cond\_expression$

$$\$\$.val = \$1.val, \qquad \$\$.type = \$1.type$$

$assign\_expr \rightarrow unary\_expr\ assign\_op\ assign\_expr$

$$if(assign\_op == "=")\ \$1.val = \$2.val$$

$$if(assign\_op == "+=")\ \$1.val = \$1.val + \$2.val$$

$$\ldots\ldots$$

$$\$\$.val = \$1.val, \qquad \$\$.type = \$1.type$$

$cond\_expr \rightarrow logical\_or\_expr$

$$\$\$.val = \$1.val,$$

$$if(\$1.type == bool)\ \$\$.type = \$1.type$$

$logical\_or\_expr \rightarrow logical\_and\_expr$

$$\$\$.val = \$1.val, \qquad \$\$.type = \$1.type$$

$logical\_or\_expr \rightarrow logical\_or\_expr\ OP\_OR\ logical\_and\_expr$

$$\$\$.val = \$1.val\ or\ \$2.val, \qquad \$\$.type = bool$$

$logical\_and\_expr \rightarrow inclu\_or\_expr$

$$\$\$.val = \$1.val, \qquad \$\$.type = \$1.type$$

$logical\_and\_expr \rightarrow logical\_and\_expr\ OP\_AND\ inclu\_or\_expr$

$$\$\$.val = \$1.val\ and\ \$2.val, \qquad \$\$.type = bool$$

$inclu\_or\_expr \rightarrow exclu\_or\_expr$

$$\$\$.val = \$1.val, \qquad \$\$.type = \$1.type$$

$inclu\_or\_expr \rightarrow inclu\_or\_expr \mid exclu\_or\_expr$

$$\$\$.val = \$1.val \mid \$2.val$$

$$if(\$1.type == int\ \&\&\ \$2.type == int)\ \$\$.type = int$$

$exclu\_or\_expr \rightarrow and\_expr$

$$\$\$.val = \$1.val, \qquad \$\$.type = \$1.type$$

$exclu\_or\_expr \rightarrow exclu\_or\_expr\ \hat{}\ and\_or\_expr$

$$\$\$.val = \$1.val\ \hat{}\ \$2.val$$

$$if(\$1.type == int\ \&\&\ \$2.type == int)\ \$\$.type = int$$

$and\_expr \rightarrow eq\_expr$

$$\$\$.val = \$1.val, \qquad \$\$.type = \$1.type$$

$and\_expr \rightarrow and\_expr\ \&\ eq\_expr$

$$\$\$.val = \$1.val\ \&\ \$2.val$$

$$if(\$1.type == int \;\&\&\; \$2.type == int)\ \$\$.type = int$$

$eq\_expr \rightarrow rela\_expr$

$$\$\$.val = \$1.val, \qquad \$\$.type = \$1.type$$

$eq\_expr \rightarrow eq\_expr\ OP\_EQ\ rela\_expr$

$$if(\$1.val == \$2.val)\ \$\$.val = true$$

$$if(\$1.val \neq \$2.val)\ \$\$.val = false$$

$$\$\$.type = bool$$

$eq\_expr \rightarrow eq\_expr\ OP\_NE\ rela\_expr$

$$if(\$1.val == \$2.val)\ \$\$.val = false$$

$$if(\$1.val \neq \$2.val)\ \$\$.val = true$$

$$\$\$.type = bool$$

$rela\_expr \rightarrow shift\_expr$

$$\$\$.val = \$1.val, \qquad \$\$.type = \$1.type$$

$rela\_expr \rightarrow rela\_expr < shift\_expr$

$$if(\$1.val < \$2.val)\ \$\$.val = true$$

$$if(\$1.val \geq \$2.val)\ \$\$.val = false$$

$$\$\$.type = bool$$

$rela\_expr \rightarrow rela\_expr > shift\_expr$

$$if(\$1.val > \$2.val)\ \$\$.val = true$$

$$if(\$1.val \leq \$2.val)\ \$\$.val = false$$

$$\$\$.type = bool$$

$rela\_expr \rightarrow rela\_expr\ OP\_LE\ shift\_expr$

$$if(\$1.val \leq \$2.val)\ \$\$.val = true$$

$$if(\$1.val > \$2.val)\ \$\$.val = false$$

$$\$\$.type = bool$$

$rela\_expr \rightarrow rela\_expr\ OP\_GE\ shift\_expr$

$$if(\$1.val \geq \$2.val)\ \$\$.val = true$$

$$if(\$1.val < \$2.val)\ \$\$.val = false$$

$$\$\$.type = bool$$

$shift\_expr \rightarrow add\_expr$

$$\$\$.val = \$1.val, \qquad \$\$.type = \$1.type$$

$shift\_expr \rightarrow shift\_expr\ OP\_LSHITF\ add\_expr$

$$if(\$1.type == int\ \&\&\ \$2.type == int)\ \$\$.type = int$$

$$\$\$.val = \$1.val \ll \$2.val$$

$shift\_expr \rightarrow shift\_expr\ OP\_RSHITF\ add\_expr$

$$if(\$1.type == int\ \&\&\ \$2.type == int)\ \$\$.type = int$$

$$\$\$.val = \$1.val \gg \$2.val$$

$add\_expr \rightarrow mul\_expr$

$$\$\$.val = \$1.val, \qquad \$\$.type = \$1.type$$

$add\_expr \rightarrow add\_expr + mul\_expr$

$$if(\$1.type == \$2.type)\ \$\$.type = \$1.type$$

$$\$\$.val = \$1.val + \$2.val$$

$add\_expr \rightarrow add\_expr - mul\_expr$

$$if(\$1.type == \$2.type)\ \$\$.type = \$1.type$$

$$\$\$.val = \$1.val - \$2.val$$

$mul\_expr \rightarrow cast\_expr$

$$\$\$.val = \$1.val, \qquad \$\$.type = \$1.type$$

$mul\_expr \rightarrow mul\_expr * cast\_expr$

$$if(\$1.type == \$2.type)\ \$\$.type = \$1.type$$

$$\$\$.val = \$1.val * \$2.val$$

$mul\_expr \rightarrow mul\_expr / cast\_expr$

$$if(\$1.type == \$2.type)\ \$\$.type = \$1.type$$

$$\$\$.val = \$1.val / \$2.val$$

$mul\_expr \rightarrow mul\_expr \%\ cast\_expr$

$$if(\$1.type == \$2.type)\ \$\$.type = \$1.type$$

$$\$\$.val = \$1.val \% \$2.val$$

$cast\_expr \rightarrow unary\_expr$

$$\$\$.val = \$1.val, \qquad \$\$.type = \$1.type$$

$unary\_expr \rightarrow postfix\_expr$

$$\$\$.val = \$1.val, \qquad \$\$.type = \$1.type$$

$unary\_expr \rightarrow unary\_op\ cast\_expr$

$$if(\$1 == "!")\ \$\$.type = bool$$

$$if(\$1 == "\sim"\ \&\&\ \$2.type == int)\ \$\$.type = int$$

$$if(\$1 == "*")\ \$\$.type = ptr\ of\ \$2.type$$

$$if(\$1 == "\&")\ \$\$.type = addr$$

$$if(\$1 == +"\ ||\ \$1==" - ")\ \$\$.type = \$2.type$$

$$if(\$1 == "!")\ \$\$.val = (\$2.val == 0)\ ?\ false : true$$

$$if(\$1 == "\sim")\ \$\$.val = \sim\$2.val$$

$$if(\$1 == "+")\ \$\$.val = \$2.val$$

$$if(\$1 == "-")\ \$\$.val = -\$2.val$$

$$if(\$1 == "*")\ \$\$.val = *\ \$2.val$$

$$if(unary\_op == "\&")\ \$\$.val = addr\ of\ \$2.val$$

$unary\_expr \rightarrow OP\_INC\ unary\_expr$

$$if(\$2.type \neq int)\ error$$

$$\$\$.type = \$2.type, \qquad \$2.val = \$2.val + 1, \qquad \$\$ = \$2.val$$

$unary\_expr \rightarrow OP\_DEC\ unary\_expr$

$$if(\$2.type \neq int)\ error$$

$$\$\$.type = \$2.type, \qquad \$\$.val = \$2.val - 1$$

$postfix\_expr \rightarrow prim\_expr$

$$\$\$.val = \$1.val, \qquad \$\$.type = \$1.type$$

$postfix\_expr \rightarrow postfix\_expr$ [$expr$]

$$\$\$.val = *\ (\$1.val + \$3.val), \qquad \$\$.type = \$1.arr\_type$$

$postfix\_expr \rightarrow postfix\_expr$ (arg_expr_list)

$$\$\$.val = \$1(\$3.val), \qquad \$\$.type = \$1.rtype$$

$postfix\_expr \rightarrow postfix\_expr\ OP\_INC$

$$if(\$2.type \neq int)\ error$$

$$\$\$.type = \$2.type, \qquad \$\$.val = \$2.val, \qquad \$2.val = \$2.val + 1$$

$postfix\_expr \rightarrow postfix\_expr\ OP\_DEC$

$$if(\$2.type \neq int)\ error$$

$$\$\$.type = \$2.type, \qquad \$\$.val = \$2.val, \qquad \$2.val = \$2.val - 1$$

2. Symbol table

We use symbol tables to store the maps between identifiers and variables. When it comes to a new nest, we create a new table based on the current table, like the structure of a stack. When we need to lookup the symbol tables to find some variable, we search the most recent table, or the table on the top of table stack first, and then the second recent table, and so on. If all tables are searched but no matched variable found, we process this situation as an error.

The special structure of the symbol table:

LLVM::Module::getValueSymbolTable()

Std::map<std::string,LLVM::Value*>

3. Type check

We check the type of variables at a lot of positions, like function calls, variables operations, and function definitions. We here display some examples about our type check work with pseudo codes, and more details can be seen in the source code files.

Function calls:

We check the return type and parameter type when we process a function call. If the return type doesn't match the caller, or a parameter type doesn't match the declaration of the function, we consider the situation an error.

```
if(node->name=="postfix_expr"
&& node->left->name=="identifier"
&& node->left->right->name=="("
&& node->left->right->right->name=="arg_expr_list"){
    //postfix_expr -> identifier '(' arg_expr_list ')'
    func = lookup_func_table(node->left);
    args = get_args(node->left->right);
    check(args, func.paras){
        if(unmatch) error;
    }
    ......
}
```

Variable operations:

We check the types of variables when making variable operations. For example, when we make bit shift operations, we need to check whether the two operands are both integer and report an error if not. In fact, this is a part of work in the attribute computing.

e.g.

$shift\_expr \rightarrow shift\_expr\ OP\_LSHITF\ add\_expr$

```
if(node->name=="shift_expr"
&& node->left->name=="shift_expr"
&& node->left->right->name==OP_LSHIFT
&& node->left->right->right->name=="add_expr"){
    //shift_expr -> shift_expr OP_LSHIFT add_expr
    tmp1 = compute(node->left);
    tmp2 = compute(node->left->right->right);
    if(tmp1.type!=int || tmp2.type!=int) error;
    ......
}
```

Function definitions:

We check the parameter types when we process a function definition by comparing them with the declaration of the function and report an error if there is any unmatched parameter type.

```
function process_func_def(ASTnode* cur_node){
    paras = get_paras(cur_node);
    func_name = get_func_name(cur_node);
    decl = get_decl(func_name);
    check(decl.paras, paras){
        if(unmatch) error;
    }
    ......
}
```

V. Optimization Considerations

The LLVM IR can automatically do some optimization like the following translation:

Main.cpp:

```
int main(){
    int i = 23 + 14*3 - 12/2%2
    return 0;
}
```

Main.ll

```
def i32 @main(){
entry:
    %i = alloca i32 , align 4
    Store i32 63, i32* %i, align 4
    Ret i32 0
```

```
}
```

Automatically calculating the results of 23 + 14*3 – 12/2%2.

VI. Code Generation

1. Intermediate codes

We generate the intermedia codes when analyzing the semantics. We use LLVM to generate intermediate codes, which is a is a set of compiler and toolchain technologies, and can be used to develop a front end for any programming language and a back end for any instruction set architecture.

LLVM is designed around a language-independent intermediate representation (IR) that serves as a portable, high-level assembly language that can be optimized with a variety of transformations over multiple passes.

The standard of intermediate codes should be LLVM IR since we deicide to use the frame of LLVM to finish the work from intermediate codes to assembly codes, and the finial executable file. LLVM IR is something like the three-address code, and has a huge amount of specifications. To ensure the specifications of the intermediate codes we would generate, we apply some auxiliary tools given by LLVM to our code generator, like $llvm::IRBuilder$, $llvm::value$, $llvm::Value$, $llvm::Function$, $llvm::Type$, $llvm::BasicBlock$, $llvm::LLVMContext$, and so on.

We create a new translation module at the beginning of generating process, and insert functions or some declarations into the module. When we obtain the attributes of some statement from computing, we find an appropriate location and insert codes after sorting out the attributes and the information of nodes. For example, when we process an AST node of additive expression type, we compute the attributes of its child-nodes, insert the codes about the processing procedure, and return the result.

```
Value* additive_exp(node head = nullptr, Module* m = nullptr){
    if(head->left->right){
        Value* l = additive_exp(head->left,m);
        Value* lval = isptr?builder.CreateLoad(l):l;
        Value* r = multiplicative_exp(head->left->right->right);
        Value* rval = isptr?builder.CreateLoad(r):r;
        string type = head->vtype;
        isptr =false;
        if(head->left->right->name=="+"){
            if(type=="int")
                return builder.CreateAdd(lval,rval);
            else
                return builder.CreateFAdd(lval,rval);
        }else if(head->left->right->name=="-"){
            if(type=="int")
```

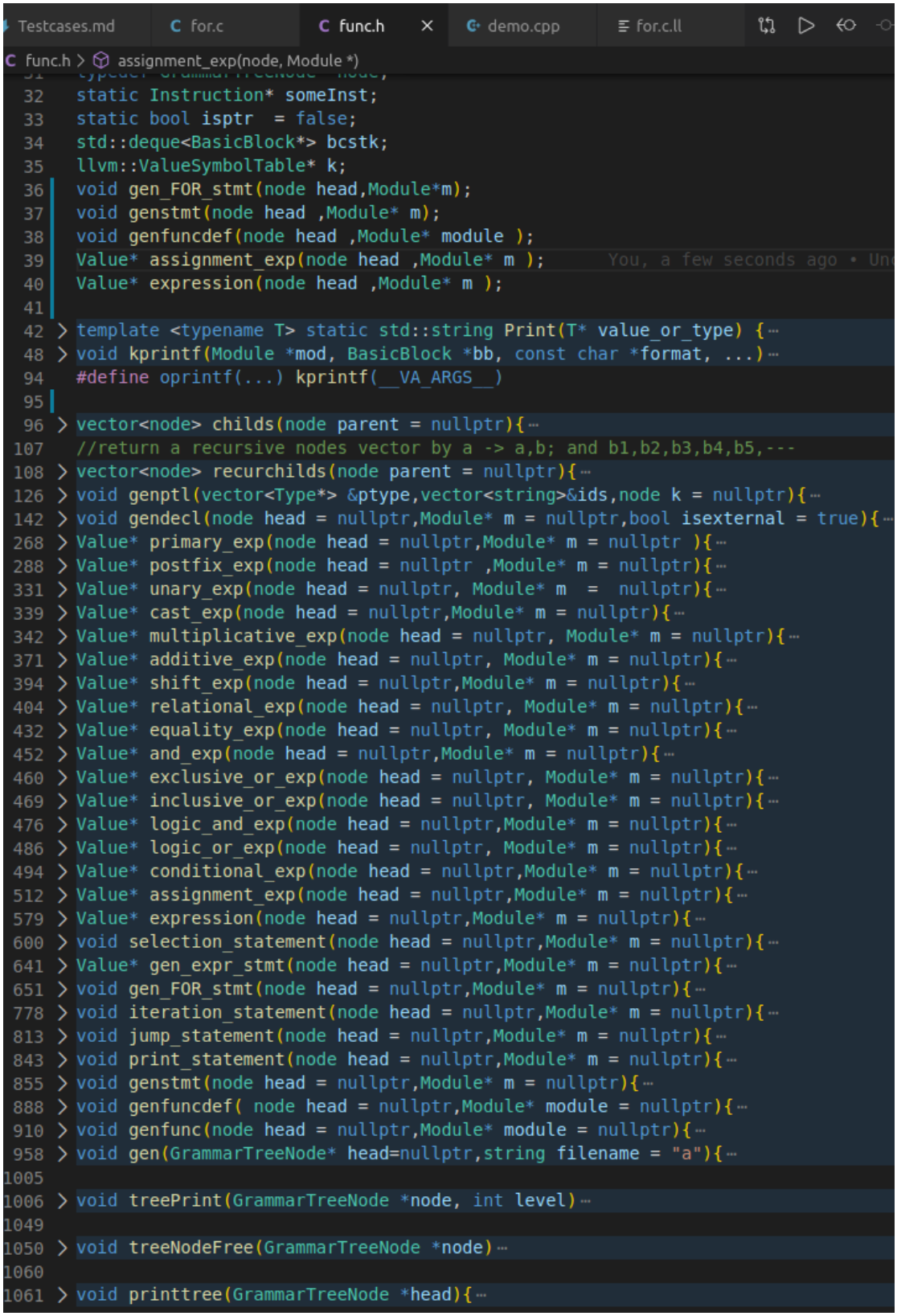
```
                return builder.CreateSub(lval,rval);
            else
                return builder.CreateFSub(lval,rval);
        }
    }else{
        return multiplicative_exp(head->left,m);
    }
}
```

More detail can be seen in the source code files. Here is a catalog:

```
int main(int argc, char* argv[]){
    if(argc>1){
        yyin=fopen(argv[1],"r");
    }else{
        yyin=stdin;
    }
    yyparse();
    printf("\n");
    treePrint(root,0);
    gen(root,argv[1]);
    std::string cmd = "llc "+argv[1]+" -o "+ argv[1]+".s & gcc "+argv[1]
                    +".s -o"+argv[1]+".out"
    system(cmd.c_str());
    fclose(yyin);
    return 0;
}
```

The function main() is the entry of source codes to target codes or IR codes .

The function gen() helps with translating AST to LLVM IR codes , whose definition is located in func.h file. And the gen() starts to call necessary functions to help navigate from the rootNode of the AST to the leafNode . meaning the Program -> external_exp -> --- -> terminal symbols .



The code generation follows a Top-Down policy , that is generating from the root node of the AST tree.

gen(root,argv[1]) is the enter of the whole generating procedure.

And then following all the possible ways to the teminal symbols , while calling functions. For examples,

a. gen(root,argv[1]
b. genfunc(node->left)
c. genfuncdef(node->left)
d. genstmt(node->left)
e. expression(node->left)
f. assignment_exp(node->left)
g. conditional->exp(node->left)
h. …
i. primary_exp(node->left)

You can view the func.h file int the src codes for more details

2. Assembly codes

Fortunately, LLVM provides a quick way to generate assembly codes from LLVM IR by **llc** src.ll -o src.s

Example:

```
        .text
        .file   "for.c"
        .globl  main                    # -- Begin function main
        .p2align 4, 0x90
        .type   main,@function
main:                                   # @main
        .cfi_startproc
# %bb.0:                                # %entry
        movl    $0, -4(%rsp)
        movl    $0, -8(%rsp)
        xorl    %eax, %eax
        retq
.Lfunc_end0:
        .size   main, .Lfunc_end0-main
        .cfi_endproc
                                        # -- End function
        .section ".note.GNU-stack","",@progbits
```

3. Executive codes

Any C compiler gcc or clang can compile asm codes (.s files) to target machine codes.

By **clang src.s -o src.out**

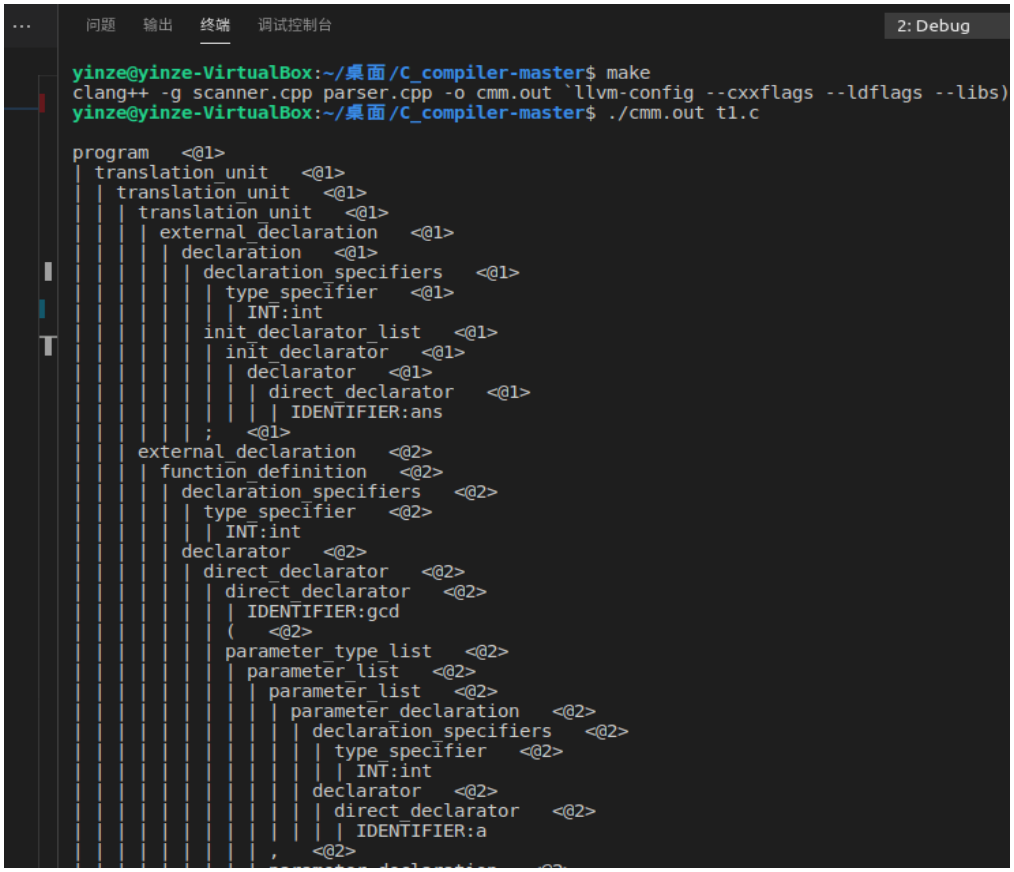
VII. Test Cases

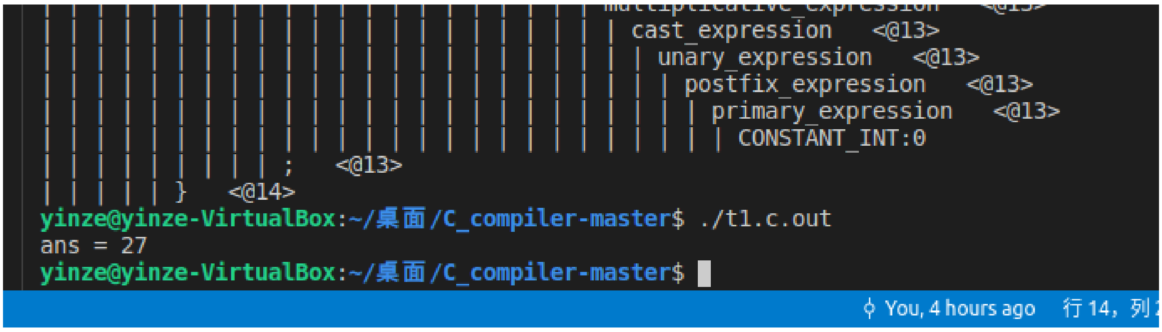
Case 1: t1.c

Source code:

```
int ans;int gcd(int a,int b){
    if(b==0){
        return a;
    }else{
        return gcd(b,a%b);
    }
}
int main(){
    ans = gcd(9,36)*gcd(3,6);
    println("ans = %d",ans);
    return 0;
}
```

Result:





Case 2: t2.c

Source code:

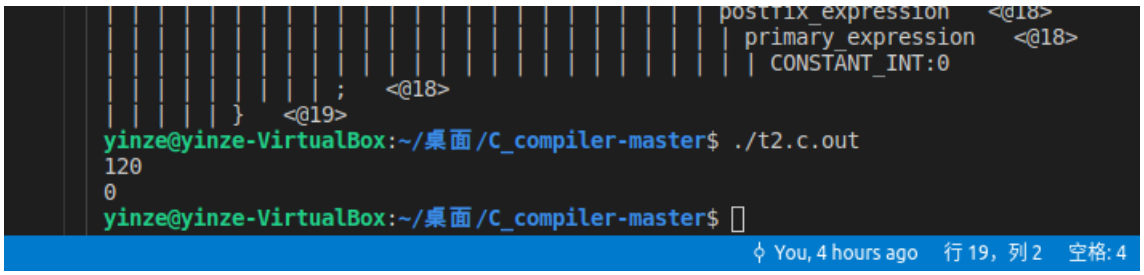
```
int f,k;

int go(int b,int a){
    int fk;
    double t;
    if(a>0){
        return a*go(b,a-1);
    }else{
        return 1;
    }
}

int main(){
    k = 0;
    f = go(k,5);

    println("%d\n",f);
    println("%d\n",k);
    return 0;
}
```

Result:



Case 3: t3.c

Source code:

```
int main(){

    int a = 13%5;

    a += 3+2*2-12/4;

    println("a = %d",a);

    int b = a==4 ? 1 : 2;

    a+= a,b,1,2,3,4;

    println("a = %d",a);

    while(a<100){

        b+=a;

        a+=1;

    }
```

```
    println("a = %d",a);

    println("b = %d",b);

    return 0;

}
```

Result: