浙江大学

本科实验报告

课程名称：编译原理

姓名：夏文星、周皓钺、黄稼树

学院：计算机科学与技术

系：计算机科学与技术

专业：计算机科学与技术

学号：3160101801

指导教师：李莹

序言

概述

编译器就是将“高级语言”翻译为“机器语言（低级语言）”的程序。编译器的主要工作流程为：1. 读取源代码 (source code)作为输入文件。 2. 使用预处理器 (preprocessor)对源代码作预处理。预处理阶段又分为词法分析、语法分析，最终将生成一颗语法树。有时也可以直接生成三地址码等中间代码。 3. 语义分析将语法树遍历生成中间代码。（语法树也是一种中间码）4. 生成目标代码 (object code) 5. 使用链接器 (Linker)将目标代码处理成可执行程序(executables)。

文件说明

上传的文件中：

Mylexer.l 为 lex 文件，完成词法分析工作。

Myparser.y 为 yacc 文件，完成语法分析工作。

AST.h 为 mylexer.l 与 myparser.y 共有的头文件，主要包含了语法树结点的定义，在语法分析工作中作为数据结构存储形式的定义。

Myparser.tab.c 与 Myparser.tab.h 与 lex.yy.h 是编译生成的。

a.Out 为可执行文件

For.c 为测试文件 out.txt 为测试输出文件

运行方式:linux 下 flex mylexer.l bison myparser.y 生成 a.out 然后运行即可

IRgenerator.h、IRgenerator.cpp 生成中间代码

main.cpp 程序入口

LLVMtest.cpp LLVM 测试代码

array_for_test.c、if_fun_op_test.c 测试样例

运行:

make array_for_test 生成测试样例 array_for_test.c 的 LLVM IR

make array_for_testrun 运行 array_for_test.c 并输出

make if_fun_op_test 生成测试样例 if_fun_op_test.c 的 LLVM IR

make if_fun_op_testrun 运行 if_fun_op_test.c 并输出

pgcc 运行 optimize.cpp,输入想优化的.ll 文件,结果即会输出到 result.ll 中,常值优化在语义分析时实现,未在此步,此步主要是乘法的优化

分工

夏文星: 词法分析&语法分析

黄稼树: 语义分析&中间代码生成

周皓铖: 代码优化&目标代码生成

实验目的

掌握词法分析、语法分析、语义分析和代码生成方法。

实验环境

Windows 系统环境或者 Linux 环境。

1、Linux 环境下的编译和运行

（1）Linux 2.6 以上版本

（2）GCC3.4 以上版本

（3）Bison 2.2 以上版本

（4）Flex 2.5.33 以上版本

发行版可以采用 Ubantu, Gentoo, Fedora Core 等。

2、Windows 环境下的编译和运行

（1）Visual Studio 6.0

（2）Masm 6.0 以上版本

（3）ParseGenerator 4.0 （Lex 和 Yacc 的集成开发包）

实验内容

第一章 词法分析

在读取源文件后，我们需要对代码进行预处理，第一步通常是词法分析。本次实验中词法由 lex 完成。

词法分析作为对输入文件的第一道处理，主要的工作是通过正则

表达式，将输入文件的词句切分成我们定义好的标记，称为 token，以便传送给 yacc 文件做进一步的语法分析。

正则表达式

在 lex 文件中，首先给出正则表达式的定义来供 lex 匹配。

```
BLANK [ \t\n\r]+
INTNUM [0-9]+
DOUNUM [0-9]+\.[0-9]+
EXPNUM {DOUNUM}|{INTNUM}((e|E)(\+|\-)?[0-9]+)
STRING \"[^"]*\"
CHARACTER \'[a-zA-Z_]\'
COMMENT (\/\/.*)
IDENTIFIER [a-zA-Z_][a-zA-Z0-9_]*
FILENAME [a-zA-Z_]+\.[a-zA-Z_]+
```

然后再定义匹配与处理，通常处理中需要返回一个标记。匹配的字符串存储在一个 yytext 的指针类型地址中，如果需要将其保留给 yacc 文件做语法分析，可以通过 yylval 绑定给相应的 token。Yylval 默认为 int 类型，也可以自己定义数据结构类型来记录匹配的原字符串。Token 会被返回给 yacc 程序完成语法分析。

匹配

本工程中给出了一下匹配：

- 注释与空格，以及一些常见符号的匹配。

```
{COMMENT} {}
{BLANK} {}

"#" {return '#';}
"++" {return SI;}
"--" {return SD;}
"<" {return LT;}
">" {return GT;}
"<=" {return LE;}
">=" {return GE;}
"==" {return EQ;}
"!=" {return NE;}


"|" {return OR;}
"&&" {return AND;}
"{" {return '{';}
"}" {return '}';}
"(" {return '(';}
")" {return ')';}
"[" {return '[';}
"]" {return ']';}
"+" {return '+';}
"-" {return '-';}
"*" {return '*';}
"/" {return '/';}
"%" {return '%';}
";" {return ';';}
"=" {return '=';}
"," {return ',';}
"." {return '.';}
```

- 对于标识符的匹配。

标识符主要是指一些保留字符串，和自定义的变量名。

预定义字符串有"define"， "include"， "true"， "false"，"char"， "int"， "double"， "void"， "bool"， "string"， "else"，"if"， "while"， "return"， "for"，将返回相应的 token 标识。

（部分代码截图如下）

```
{IDENTIFIER} {
                                string defstr = "define";
                                string incstr = "include";
                                string truestr = "true";
                                string falsestr = "false";
                                string charstr = "char";
                                string intstr = "int";
                                string doustr = "double";
                                string voidstr = "void";
                                string boolstr = "bool";
                                string strstr = "string";
                                string elsestr = "else";
                                string ifstr = "if";
                                string whilestr = "while";
                                string returnstr = "return";
                                string forstr = "for";
```

```
if(strcmp(yytext,incstr.c_str())==0)
{
        return INCLUDE;
}else if(strcmp(yytext,defstr.c_str())==0)
{
        return DEFINE;
}else if(strcmp(yytext,truestr.c_str())==0||
        strcmp(yytext,falsestr.c_str())==0)
{
        yylval.boolnum = new BoolNum(yytext);;
        return BOOLVAL;
}else if(strcmp(yytext,charstr.c_str())==0)
{
        return CHAR;
}else if(strcmp(yytext,intstr.c_str())==0)
{
        return INT;
```

而自定义的变量名则需要返回 IDEN 标识，并存储该字符串。

```
yylval.iden = yytext;
return IDEN;
```

- 头文件名的匹配。

头文件与一般字符串不同（除了含“”外也可能含< >），又与普通 iden 不同（不含‘.’）。所以单独匹配。

```
{FILENAME} {
                yylval.filename = yytext;
                return FILENAME;
                }
```

● Int、double、char、string 等数值的匹配。（不是标识符，而是具体数据类型的数据存储。）

```
{INTNUM} {
                        yylval.intnum = new IntNum(yytext);
                        return INTNUM;
                }
{DOUNUM} {
                        yylval.dounum = new DouNum(yytext);
                        return DOUNUM;
                }
{EXPNUM} {
                        yylval.dounum = new DouNum(yytext,'e');
                        return DOUNUM;
                }
{STRING} {
                        yylval.strnode = new StrNode(yytext);
                        return STRING;
                        }
{CHARACTER} {
                        yylval.cha =new ChaNode(yytext[1]);
                        return CHARACTER;
                        }
```

最后给出 yywrap 函数的定义，此函数主要负责文件读取完成后的操作。在需要连续读取文件的情况下可以通过在其中对 yyin 定义来实现连续读取文件，但是本程序只需要处理一个文件，所以直接输出一个字符串。

```
%%

int yywrap()
{
        puts("---fileend---\n");
        return 1;
}
```

第二章 语法分析

经过 lex 词法分析后的 token 作为数据源传给 yacc 进行进一步

分语法分析，可以在这一阶段生成一颗 语法树，也可以直接生成地址码。本程序采取先生成语法树的情形。

类型定义

在 yacc 中，我们首先给出值栈的类型定义。这些定义可以改变值栈类型，（默认是 int），方便我们更好地存储数据，完成语法树的构建。

其中一些自定义的变量类型，定义在 AST.h 文件中，是一些不同类型语法树节点的结构。在后文会贴出部分源码。

（以下是部分源码）

```
%union {
        int comint;

        IntNum* intnum;
        DouNum* dounum;
        ExpNum* expnum;
        StrNode* strnode;
        BoolNum* boolnum;
        char* iden;
        char* filename;
        char* str;
        ChaNode* cha;

        DeclaListNode* decls;
        Declaration* decl;
        PreDefi* defi;
        HeadFile* incl;
        VariaDecla* vars;
        Variable* var;     ///////1
        VariaListNode* varlist;
        FuncDecla* funcdecla;
        Node* node;
        Block* blo;
        Statement* stat;
        StateListNode* stalist;
```

给出值栈定义后，分别定义终结符 token 和一些将在递归规则中出现的表达式类型。

```
%start program
%token <intnum> INTNUM
%token <dounum> DOUNUM
%token <iden> IDEN
%token <filename> FILENAME
%token <strnode> STRING
%token <boolnum> BOOLVAL
%token <cha> CHARACTER
%token DEFINE
%token INCLUDE
%token CHAR
%token _VOID
%token INT
%type <decls> declarations
%type <decl> decl
%type <defi> defi_decl
%type <incl> incl_decl
%type <comint> datatype
%type <vars> var_decl
%type <varlist> var_list paralist
%type <var> vardef para var
%type <funcdecla> func_decl
%type <blo> block
%type <stalist> stmts
%type <stat> stmt
%type <exprs> expressions args
%type <expr> expression num_exp factor unary_exp term
%type <oper> logi_op addop mulop unaryop selfop
%type <func> funcall
%type <ifstmt> if_stmt
%type <forstmt> for_stmt
%type <assign> assign_stmt
%type <whilestmt> while_stmt
%type <returnstmt> return_stmt
```

规则定义与语法树实现

规则定义

接下来就是比较关键的规则定义与语法树实现了。

不同语言编译器的规则需要根据语言本身的结构来指定。

根据 C 语言的特性，我编写了如下的规则：

```
program : declarations;

declarations: declarations decl| decl;

decl : var_decl | func_decl|defi_decl|incl_decl;

defi_decl: '#' DEFINE IDEN expression;

incl_decl:'#' INCLUDE LT FILENAME GT|'#' INCLUDE STRING|'#' INCLUDE LT IDEN GT;

var_decl : datatype var_list ';';

var_list : var_list ',' vardef ;

vardef : IDEN| IDEN '='  expression ;

func_decl : datatype IDEN '(' paralist ')' block|datatype IDEN '(' paralist ')' ';'
                        |datatype IDEN '('')' block |datatype IDEN '('')' ';'  ;
       |
paralist : paralist ',' para | para ;

para : datatype IDEN;

datatype : _VOID | INT | CHAR| DOUBLE| BOOL| STR;

block : '{' stmts '}'| '{''}' ;

stmts : stmts stmt | stmt;

stmt : declarations | expressions| assign_stmt| block | if_stmt| while_stmt |
for_stmt | return_stmt;

expressions : expressions ',' expression ';' | expression ';' | ';';

expression : num_exp logi_op num_exp | num_exp;

logi_op  : LE | LT| GT | GE | EQ  | NE |OR|AND;

num_exp : num_exp addop term | term ;

addop : '+' | '-';
```

```
term : term mulop unary_exp | unary_exp ;

mulop : '*'| '/' | '%';

unary_exp : unaryop factor | factor|selfop factor|factor selfop;

unaryop : '-' ;

selfop : SI | SD;

factor : '(' expression ')'  | var | funcall| INTNUM | DOUNUM | STRING | BOOLVAL|
CHARACTER;

var : IDEN  ;

funcall : IDEN '('')' | IDEN '('args')';

args : args ',' expression | expression ;

assign_stmt : var '=' expression ';';

if_stmt : IF '(' expression ')' stmt | IF '(' expression ')' stmt ELSE stmt ;

while_stmt :  WHILE '(' expression ')' stmt;

for_stmt : FOR '('assign_stmt expression';' expression')' stmt| FOR '(';'
expression';' expression')' stmt|FOR '('assign_stmt expression';' ')' stmt|FOR
'(';' expression';' ')' stmt;

return_stmt : RETURN ';'|RETURN expression ';' ;
```

这些规则包含了 c 语言基本语法的匹配实现。主要将 c 语言的语句类型分为了:

1. 声明（含有变量声明、函数声明、define 类型声明、include 类型声明）。

2. statement 语句（含有声明、表达式、赋值语句、代码块、if 类型、while 类型、for 类型、return 类型）。

3. 表达式。表达式可以是其他语句的一部分，比如赋值语句。有时候也可以单独作为一个语句（但没有什么实际意义）。表达式下可以处理逻辑运算、数值运算、单目运算、双目运算，根据不同运算符的优先级不同，为其设计了几个递推规则，具体可以查看代码。

4. 另外表达式会处理数值，本程序可以处理如下几种类型的数

值：int、double、bool、void、string、char、科学计数法类型数值。同时把自定义变量、函数调用也作为表达可以处理的数值之一。

整个语法递推的顶端从声明开始，因为 c 语言的通常以函数声明为入口，而函数声明中含有一个 block，block 可以递推成 statement。

语法树定义

接着就是每个规则具体语法树构建的实现。在这里需要做的就是构建语法树，并将具体数值存储到树中。

首先先介绍 AST.h 中对语法树节点的定义。

语法树的节点有 26 种，用以存储不同类型的语句节点。它们有一个共同的父类：

```
class Node
{
public:
        int height;
        string nodename;
        string content;
        Node(){}
        virtual void printNode(int h, ofstream& fout){}
};
```

其中 height 是该节点的高度，nodename 是该节点的类型信息。Content 是该节点的具体内容。

最重要的是虚函数 printNode，需要继承类自己实现，将节点信息打印到指定文件 fout 里，并且调用该类子节点的 printNode 函数。这样只需要调用根节点的 printNode，就可以完成整颗语法树的打印。

下面给出几个节点的定义代码截图。

这是一个数值类型节点，类似的还有 double、string、char、bool、void 等等。

```
class IntNum : public Node
{
public:
        int value;
        string type;
        IntNum(){}
        IntNum(char* txt) : Node()
        {
                value = atoi(txt);
                type = "INT";
                nodename = "INT NUM";
        }

        void printNode(int h, ofstream& fout){
                for(int i=0; i<h; i++) fout<<"-";
                content = "("+nodename+","+type+","+to_string(value)+")";
                fout<<content<<endl;
        }
};
```

这是一个表达式语句节点的定义。注意这个节点会有自己的子节点。但不同的表达式子节点可能为空，所以 printNode 中打印非空节点即可。

```
class Expression : public Node
{
public:
        string type;
        Node* ptrvalue;
        //Operator& oper;
        string op;
        Expression* leftnode;
        Expression* rightnode;
        Expression(){}
        void printNode(int h, ofstream& fout){
                for(int i=0; i<h; i++) fout<<"-";
                fout<<content<<endl;
                if(leftnode!=NULL)leftnode->printNode(h+1, fout);
                if(rightnode!=NULL)rightnode->printNode(h+1, fout);
                if(ptrvalue!=NULL)ptrvalue->printNode(h+1, fout);
        }
};
```

这是一个函数定义节点。可能需要打印的有参数列表、代码块。

```
class FuncDecla : public Node
{
public:
        string name;//Identifier func;
        Variable returnval;
        VariaListNode* paras;
        Block* block;
        FuncDecla(){}
        void printNode(int h, ofstream& fout){
                for(int i=0; i<h; i++) fout<<"-";
                fout<<content<<endl;
                if(paras!=NULL)paras->printNode(h+1, fout);
                if(block!=NULL)block->printNode(h+1, fout);
        }
};
```

语法树构建

接着我们回到 yacc 继续给出语法树的构建。

不同的规则下需要的构建动作不同，但主要内容即为值栈的指针分配节点空间、为节点的子节点（如果有）指定对象、给节点的值赋值。

下面给出几个规则的示例：

- Declarations 规则

```
declarations: declarations decl
                        {
                                $$ = $1;
                                $$->list.push_back(*$2);

                        }
                        | decl
                        {
                                $$ = new DeclaListNode();
                                $$->list.push_back(*$1);
                                //$$->height = $1->height+1;
                                $$->nodename = "declarations";
                                $$->content ="("+to_string(yylineno)+","+$$-
>nodename+")";
                                cout<<$$->content<<endl;
                        };
```

- vardecl 变量定义规则

```
vardef : IDEN//varname
                {
                        $$ = new Variable();
                        $$->nodename = "vardef";
                        $$->name = string($1);
                        //$$->height = 0;
                        $$->content ="("+to_string(yylineno)+","+$$->nodename+","+$
$->name+")";
                        cout<<$$->content<<endl;
                }
                | IDEN '='  expression //$3 = new Expression();
                {
                        $$ = new Variable();
                        $$->nodename = "vardef";
                        //$$->height = 0;
                        $$->name = string($1);//////////////////////
                        $$->ptrexpr = $3;
                        $$->content ="("+to_string(yylineno)+","+$$->nodename+","+$
$->name+")";
                        cout<<$$->content<<endl;
                };
```

- Expression 规则

```
expression : num_exp logi_op num_exp //$1 = new Expression();
                        {
                                $$ = new Expression();
                                $$->nodename = "expression";
                                //$$->height = max($1->height,$3->height)+1;
                                $$->leftnode = $1;
                                $$->rightnode = $3;
                                $$->op = string($2);
                                $$->content ="("+to_string(yylineno)+","+$$-
>nodename+")";
                                cout<<$$->content<<endl;
                        }
                        | num_exp
                        {
                                $$ = new Expression();
                                $$->nodename = "expression";
                                //$$->nodename = $1->height+1;
                                $$->leftnode = $1;
                                $$->content ="("+to_string(yylineno)+","+$$-
>nodename+")";
                                cout<<$$->content<<endl;
                        };
```

Main 函数

最后我们需要定义一个 main 函数，作为 yacc 程序的入口。在这个程序中，我们需要指定 yyin、yyout 作为 yacc 程序读取与输入的 io 地址。

为了方便数据的保存，我们将 yyin 与 yyout 指向两个文件。

通过调用 printNode 函数，我们将语法树的完整信息打印到了 yyout 指向的文件中。

```
int main(void)
{
        string filesrc;
        string fileobj;
        cin>>filesrc;
        cin>>fileobj;
        yyin = fopen(filesrc.c_str(),"r");
        yyout = fopen(fileobj.c_str(),"w");
        fout.open(fileobj.c_str(),ios::out);
        yyparse();
        fclose(yyin);
        fclose(yyout);
        return 0;
}
```

至此，语法分析和文法分析的实现就告一段落了。

测试

我们做几份测试。

首先编译，生成可执行文件。

```
$ flex mylexer.l
$ bison -d myparser.y
reduce conflicts [-Wconflicts-sr]
$ g++ -std=c++11 lex.yy.c myparser.tab.c
```

测试的文件内容如下：

```c
#include <stdio.h>
int main(){
        int a=5;
        int b=0;
        double c=3.14158;
        b=a*24;
        printf("%d",b);

        while(a>b){
                printf("%lf\n",c);
                b++;
        }
        return 0;
}
```

输出截图：

```
(1,declarations)
-(1,decl)
--(1,incl_decl,stdio.h>)
-(14,decl)
--(14,func_decl,main(){
        int a=5;
        int b=0;
        double c=3.14158;
        b=a*24;
        printf("%d",b);

        while(a>b){
                printf("%lf\n",c);
                b++;
        }
        return 0;
},INT)
---(14,block)
----(6,stmts)
-----(6,stmt,decl)
------(3,declarations)
-------(3,decl)
--------(3,var_decl)
---------(3,var_decl,INT)
----------(3,vardef,a=5;)
-----------(3,expression)
------------(3,num_exp)
-------------(3,term)
--------------(3,unary_exp)
```

第三章 语义分析

在语义分析阶段，编译器开始对语法树进行一次或多次的遍历，检查程序的语义规则（以类型判断为主）。语义检查的步骤和人对源代码的阅读和理解的步骤差不多，一般都是在遍历语法树的过程中，

遇到变量声明和函数声明时，则将变量名——类型、函数名——返回类型——参数数量及类型等信息保存到符号表里，当遇到使用变量和函数的地方，则根据名称在符号表中查找和检查，查找该名称是否被声明过，该名称的类型是否被正确的使用等等。

我们简单以如下几个方面为例展示语义分析的过程：

语句中的变量、函数是否被声明过

以变量名为例，在对节点 NIdentifier 的 codeGen 时首先判断是否是已经被声明过的变量名，其次用 getType 判断该变量是否为数组类型。

```
Value*  NIdentifier::codeGen(CodeGenContext &context) {
    Value*  value  =  context.getSymbolValue(this->name);
    if( !value ){
        return LogErrorV("Unknown variable name " + this->name);
    }
    if( value->getType()->isPointerTy() ){
        ……
        if( arrayPtr->getType()->isArrayTy() ){
            ……
        }
```

```
        }
        ……
    }
```

二元运算的两个操作数的类型是否匹配

指令的 L、R 必须同属一个类型，结果的类型则必须与操作数的类型相匹配：

```
    if((L->getType()->getTypeID()==Type::DoubleTyID) | | (R->getType()->getTypeID() == Type::DoubleTyID)
    {
            if( (R->getType()->getTypeID() != Type::DoubleTyID) ){
            ……
            }
            if( (L->getType()->getTypeID() != Type::DoubleTyID) ){
            ……
            }
    }
```

赋值运算符两边的操作数的类型是否匹配

此处我们定义一个 trans 函数，用于提供某个表达式的值向另一类型的转换指令。若两种类型不存在转换关系时，则输出错误信息并返回源类型的值。而能否转换则通过一个事先定义的 typeTable 来

判断,此处我们允许类型转换如下:int 到 float,int 到 double，float 到 double，float 到 int，double 到 int，bool 到 int:

context.typeCheck.trans(exp,
context.typeTable.getVarType(dstTypeStr),
context.currentBlock());

第四章 中间代码 LLVM IR 生成

LLVM IR 基本语法介绍

LLVM 汇编语言中的注解以分号“；”开始，并持续到行末。

全局标识符要以“@”字符开始。所有的函数名和全局变量都必须以“@”开始。

LLVM 中的局部标识符以百分号“%”开始。

LLVM 拥有一个强大的类型系统，这也是它的一大特性。LLVM 将整数类型定义为 iN，其中 N 是整数占用的字节数。我们可以指定 1 到 223- 1 之间的任意位宽度。

对于字符串 "Hello World!"，可以使用类型[13 x i8]，假设每个字符占用 1 个字节，再加上为 NULL 字符提供的 1 个额外字节。

我们可以对“hello-world”字符串的全局字符串常量进行如下声明：@hello = constant [13 x i8] c"Hello World!\00"。使用关键字 constant 来声明后面紧跟类型和值的常量。以 c 开始，后面紧跟放在双引号中的整个字符串（其中包括 \0 并以 0 结尾）。

LLVM 允许声明和定义函数。以关键字 define 开始，后面紧跟

返回类型，然后是函数名。返回 32 字节整数的 main 的简单定义类似于： define i32 @main() { ; some LLVM assembly code that returns i32 }。

函数声明。以 puts 方法为例，它是 printf: declare i32 puts(i8*) 的 LLVM 等同物。该声明以关键字 declare 开始，后面紧跟着返回类型、函数名，以及该函数的可选参数列表。该声明必须是全局范围的。

每个函数均以返回语句结尾。有两种形式的返回语句： “ret <type> <value> ”或 “ret void”。

LLVM 使用 “call <function return type> <function name> <optional function arguments>”来调用函数。每个函数参数都必须放在其类型的前面。返回一个 6 位的整数并接受一个 36 位的整数的函数测试的语法为 “call i6 @test(i36 %arg1)”。

LLVM IR 语法说明样例

源代码：

```
int main(int argc, char *argv[])
{
    int a = 1;
    int b = 2;
    a = a + b;
    return 0;
}
```

对应的 LLVM IR：

```
; ModuleID = 'test.c'
target datalayout = "e-m:e-i64:64-f80:128-n8:16:32:64-S128"
target triple = "x86_64-pc-linux-gnu"

; Function Attrs: nounwind uwtable
define i32 @main(i32 %argc, i8** %argv) #0 {
  %1 = alloca i32, align 4
  %2 = alloca i32, align 4
  %3 = alloca i8**, align 8
  %a = alloca i32, align 4
  %b = alloca i32, align 4
  store i32 0, i32* %1, align 4
  store i32 %argc, i32* %2, align 4
  store i8** %argv, i8*** %3, align 8
  store i32 1, i32* %a, align 4
  store i32 2, i32* %b, align 4
  %4 = load i32, i32* %a, align 4
  %5 = load i32, i32* %b, align 4
  %6 = add nsw i32 %4, %5
  store i32 %6, i32* %a, align 4
  ret i32 0
}
```

生成方式

从 AST 到 LLVM IR 意味着将每一个语义节点转换成等价的 LLVM IR 指令。LLVM 将帮助我们把这步变得非常简单，因为 LLVM 将真实的指令抽象成类似 AST 的指令。这意味着我们真正要做的事就是将 AST 转换成抽象指令。这个过程可以想象成是从我们之前已经定义的抽象语法树的根节点开始遍历每一个树上节点并产生字节码的过程。在开始生成 LLVM IR 之前，还有一些准备工作要做。首先，给每个 AST 类添加一个虚函数 codeGen，用于实现具体的代码生成：

```
class NBasic {
// 各个抽象语法树节点基类
public:
   virtual ~Node() {}
   virtual Value *codeGen() = 0;
};


class NIdentifier : public NExpression {
```

```
public:
    std::string name;
    NIdentifier(const std::string& name) : name(name) { }
    virtual llvm::Value* codeGen(CodeGenContext& context);
};
```

每种 AST 节点的 codeGen()方法负责生成该类型 AST 节点的 IR 代码及其他必要信息，生成的内容以 LLVM Value 对象的形式返回。其中 LLVM 用“Value”类表示“静态一次性赋值 (SSA, Static Single Assignment）寄存器”或“SSA 值”。SSA 值最为突出的特点就在于“固定不变”：SSA 值经由对应指令运算得出后便固定下来，直到该指令再次执行之前都不可修改。

其次，我们还需要一个“Error”方法，该方法与语法解析器里用到的报错函数类似，用于报告代码生成过程中发生的错误（例如引用了未经声明的参数）：

```
Value *ErrorV(const char *Str) { Error(Str); return 0; }
```

下面几个静态变量都是用于完成代码生成的：

```
static Module *TheModule;
static IRBuilder<> Builder(getGlobalContext());
static std::map<std::string, Value*> NamedValues;
```

其中 TheModule 是 LLVM 中用于存放代码段中所有函数和全局变量的结构。从某种意义上讲，可以把它当作 LLVM IR 代码的顶层容

器。

Builder 是用于简化 LLVM 指令生成的辅助对象。IRBuilder 类模板的实例可用于跟踪当前插入指令的位置，同时还带有用于生成新指令的方法。

NamedValues 映射表用于记录定义于当前作用域内的变量及与之相对应的 LLVM 表示即代码的符号表。

我们还在 CodeGenContext 类中使用一个语句块的栈来保存最后进入的 block(因为语句都被增加到 blocks 中)我们同样用个堆栈来保存每组语句块中的符号表。

我们设计的语言知道当前范围内的内容，要支持“全局”上下的做法，必须向上搜索整个堆栈中每一个语句块，直到最后发现所要匹配的符号。在我们进入一个语句块之前，我们需要将语句块压栈，离开语句块时将语句块出栈。

具体实现

下面介绍每种 AST 节点的 codeGen()的具体实现方法，这里我们以几个具有代表性的节点对它们的 codeGen()展开详细的介绍。

数值常量

LLVM IR 中的数值常量是由 ConstantFP 类表示的。在其内部，具体数值由 APFloat（Arbitrary Precision Float，可用于存储任意精度的浮点数常量）表示。这段代码简单来说就是新建并返回了一个 ConstantFP 对象。值得注意的是，在 LLVM IR 内部，常量都只有一份，并且是共享的。下面以 double 为例：

```
Value* NDouble::codeGen(CodeGenContext& context)
{
return
ConstantFP::get(Type::getDoubleTy(context.llvmContext),
this->value);
}
```

变量

在 LLVM 中引用变量也很简单。实际上，位于 NamedValues 映射表中的变量只可能是函数的调用参数。这段代码首先确认给定的变量名是否存在于符号表中，如果不存在，就说明引用了未定义的变量，然后返回该变量的值。

```
Value *NVar::Codegen(CodeGenContext& context) {
   Value *V = NamedValues[Name];
   return V ? V : ErrorV("Unknown variable name");
}
```

二元运算

二元运算的中间代码生成的基本思想是递归地生成代码，先处理表达式的左侧，再处理表达式的右侧，最后计算整个二元表达式的值。上述代码就 opcode 的取值用了一个简单的 switch 语句，从而为各种二元运算符创建出相应的 LLVM 指令。

利用 LLVM 的 Builder 类我们需想清楚该用哪些操作数（即此处的 L 和 R）生成哪条指令（通过调用 CreateFAdd 等方法）即可，

至于新指令该插入到什么位置，交给 IRBuilder 就可以了。

在这里，LLVM 指令需要遵循严格的约束：例如，add 指令的 L、R 必须同属一个类型，结果的类型则必须与操作数的类型相匹配。

函数定义

```
Value*              NFunDecl::codeGen(CodeGenContext&
context)
    {
        vector<const type*> argTypes;
        VariableList::const_iterator it;
        for (it = arguments.begin(); it != arguments.end(); it++)
{
            argTypes.push_back(typeOf((**it).type));
        }
        FunctionType                *ftype                =
FunctionType::get(typeOf(type), argTypes, false);
        Function *function = Function::Create(ftype,
GlobalValue::InternalLinkage,                id.name.c_str(),
context.module);
        BasicBlock *bblock = BasicBlock::Create("entry",
function, 0);
        context.pushBlock(bblock);
        for (it = arguments.begin(); it != arguments.end(); it++)
```

```
{
        (**it).codeGen(context);
    }
    block.codeGen(context);
    ReturnInst::Create(bblock);
    context.popBlock();
    std::cout << "Creating function: " << id.name << endl;
    return function;
}
```

首先需要该函数的返回值类型是“Function*”而不是“Value*”。“函数原型”描述的是函数的对外接口（而不是某表达式计算出的值），返回代码生成过程中与之相对应的 LLVM Function 自然也合情合理。

FunctionType::get 调用用于为给定的函数原型创建对应的 FunctionType 对象。随后，FunctionType::get 方法以这“N”个 double 为参数类型、以单个 double 为返回值类型，创建出一个参数个数不可变（即最后一个参数 false）的函数类型。

最后实际上创建的是与这个函数原型相对应的函数。其中包含了类型、链接方式和函数名等信息，还指定了该函数待插入的模块。“ExternalLinkage”表示该函数可能定义于当前模块之外，可以被当前模块之外的函数调用。。

中间代码生成测试

测试样例：

```
extern int printf(string format)
extern int puts(string s)


int min(int a, int b){
    int re = 0
    if( a <= b ) {
        re = a
    }
    else {re=b}
    return re
}

int main(int argc, string[1] argv){
    int i
    i=min(3,7)
    printf("i=%d",i)
    puts("")
    return 0
}
```

LLVM IR：

```
define i32 @min(i32 %a, i32 %b) {
entry:
  %0 = alloca i32
  store i32 %a, i32* %0
  %1 = alloca i32
  store i32 %b, i32* %1
  %2 = alloca i32
  store i32 0, i32* %2
  %arrayPtr = load i32, i32* %0
  %3 = load i32, i32* %0
  %arrayPtr1 = load i32, i32* %1
  %4 = load i32, i32* %1
  %cmptmp = icmp sle i32 %3, %4
  %5 = icmp ne i1 %cmptmp, false
  br i1 %5, label %then, label %else

then:                                             ; preds = %entry
  %arrayPtr2 = load i32, i32* %0
  %6 = load i32, i32* %0
  store i32 %6, i32* %2
  br label %ifcont

else:                                             ; preds = %entry
  %arrayPtr3 = load i32, i32* %1
  %7 = load i32, i32* %1
  store i32 %7, i32* %2
  br label %ifcont

ifcont:                                           ; preds = %else, %then
  %arrayPtr4 = load i32, i32* %2
  %8 = load i32, i32* %2
  ret i32 %8
}

define i32 @main(i32 %argc, i8** %argv) {
entry:
  %0 = alloca i32
  store i32 %argc, i32* %0
  %1 = alloca i8**
  store i8** %argv, i8*** %1
  %2 = alloca i32
  %calltmp = call i32 @min(i32 3, i32 7)
  store i32 %calltmp, i32* %2
  %arrayPtr = load i32, i32* %2
  %3 = load i32, i32* %2
  %calltmp1 = call i32 @printf([5 x i8]* @string, i32 %3)
  %calltmp2 = call i32 @puts([1 x i8]* @string.1)
  ret i32 0
}
```

第五章 中间代码优化

优化是编译器的一个重要组成部分，由于编译器将源程序翻译成中间代码的工作是机械的、按固定模式进行的，因此，生成的中间代码往往在时间和空间上有很大浪费。当需要生成高效目标代码时，就必须进行优化。

乘法优化

第一个是乘法的优化，在常用的体系架构中，无论是 x86 还是 MIPS,都有单独的乘法单元用于计算乘法操作，此单元效率极低，且相乘的两个数越大效率越低，因此做的第一个优化是针对乘法。

如下图所示,当乘法为 reg=reg*const 时。

```
%2 = load i32, i32* %a, align 4
%3 = mul nsw i32 %2, 24
```

、

把乘法拆解成乘法和移位两步操作，移位相较于乘法，效率极高，例如上图，原式含义为：

b=a*24。拆解转化为 a<<3，b=a*3。如下图;

```
%2 = load i32, i32* %a, align 4
%temp4697 = shl i32 %2, 3
%3 = mul i32 %temp4697, 3
```

这样一来这条指令就能更有效的执行。

常量运算优化

形如下例的指令：

```
%3 = add i32 5, 6, align 4
store i32 %3, i32* %b, align 4
```

即 b=5+6，是可以在中间指令层进行优化的，无论是加法还是

乘法，只要运算的两个加数都是 const，且紧跟的一条指令是 store，则可以把两条指令合为一条：

```
store i32 11, i32* %b, align 4
```

跳转指令优化

形如如下的跳转指令：

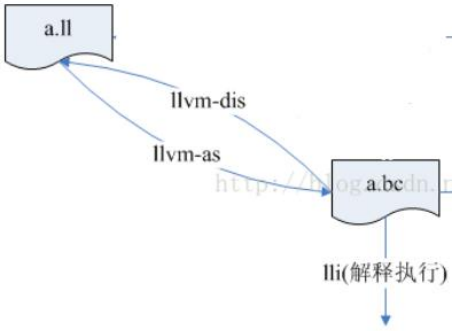
```
@.str = private unnamed_addr constant [3 x i8] c"%d\00", align 1

; Function Attrs: nounwind uwtable
define i32 @main() #0 {
  %1 = alloca i32, align 4
  %a = alloca i32, align 4
  %b = alloca i32, align 4
  store i32 0, i32* %1, align 4
  store i32 5, i32* %a, align 4
  %3 = icmp sgt i32 5, 3
  br i1 %3, label %4, label %5

; <label>:4                                       ; preds = %0
  store i32 3, i32* %b, align 4
  br label %6

; <label>:5                                       ; preds = %0
  store i32 2, i32* %b, align 4
  br label %6

; <label>:6                                       ; preds = %5, %4
  %7 = load i32, i32* %b, align 4
  %8 = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([3 x i8], [3 x i8]* @.str, i32 0, i32 0), i32 %7)
  ret i32 0
}

declare i32 @printf(i8*, ...) #1
```

当 icmp 指令比较的是两个 const，且之后紧跟着 br 指令时，因为指令必定跳转，所以删除部分永远不会访问的指令。优化后的中间代码如下：

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

; Function Attrs: nounwind uwtable
define i32 @main() #0 {
  %1 = alloca i32, align 4
  %a = alloca i32, align 4
  %b = alloca i32, align 4
  store i32 0, i32* %1, align 4
  store i32 5, i32* %a, align 4

  store i32 3, i32* %b, align 4
  br label %6


; <label>:6                                       ; preds = %5, %4
  %7 = load i32, i32* %b, align 4
  %8 = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([3 x i8], [3 x i8]* @.str, i32 0, i32 0), i32 %7)
  ret i32 0
}

declare i32 @printf(i8*, ...) #1
```

第六章 目标代码生成

目标代码生成使用 llvm 编译工具实现，llvm 中各种文件的转化指令如下图：



.ll 文件是 llvm IR 格式，.bc 是二进制格式的目标代码，可被执行。

示例源码和编译后的中间代码如下，实现了一个比较整形数大小的函数：

```
#include<stdio.h>
int min(int a , int b) {
  if (a <b )
    return a;
  return b;
}
int main(){
  int a=min(3,7)
  printf("%d\n",a);
}
```

min.c | min.ll

```
@.str = private unnamed_addr constant [4 x i8] c"%d\0A\00", align 1

define i32 @min(i32 %a, i32 %b) #0 {
  %1 = alloca i32, align 4
  %2 = alloca i32, align 4
  %3 = alloca i32, align 4
  store i32 %a, i32* %2, align 4
  store i32 %b, i32* %3, align 4
  %4 = load i32, i32* %2, align 4
  %5 = load i32, i32* %3, align 4
  %6 = icmp slt i32 %4, %5
  br i1 %6, label %7, label %9

; <label>:7                                     ; preds = %0
  %8 = load i32, i32* %2, align 4
  store i32 %8, i32* %1, align 4
  br label %11

; <label>:9                                     ; preds = %0
  %10 = load i32, i32* %3, align 4
  store i32 %10, i32* %1, align 4
  br label %11

; <label>:11                                    ; preds = %9, %7
  %12 = load i32, i32* %1, align 4
  ret i32 %12
}

define i32 @main() #0 {
  %a = alloca i32, align 4
  %1 = call i32 @min(i32 3, i32 7)
  store i32 %1, i32* %a, align 4
  %2 = load i32, i32* %a, align 4
  %3 = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([4 x i8], [4 x i8]* @.str, i32
0, i32 0), i32 %2)
  ret i32 0
}

declare i32 @printf(i8*, ...) #1
```

LLVM IR | Tab Width: 8 | Ln 29, Col 1 | INS

通过命令行调用 llvm 工具实现到目标代码的转化并运行目标代码：

```
instant2333@ubuntu:~/Downloads$ llvm-as min.ll
instant2333@ubuntu:~/Downloads$ lli min.bc
3
instant2333@ubuntu:~/Downloads$
```

再看一个实例，源码及运行结果如下：

```
#include <stdio.h>
int main(){
        int a=5;
        int b;

        b=a*24;
        printf("%d",b);
        return 0;
}
```

```
instant2333@ubuntu:~/Downloads$ llvm-as multi.ll
instant2333@ubuntu:~/Downloads$ lli multi.bc
120instant2333@ubuntu:~/Downloads$
```

可以看到命令行正确的输出了程序运行的结果 120。