编译原理实验三

MiniC 编译器前端设计与实现

西北工业大学 计算机学院

曾雷杰

2024 年 3 月

目录

第一章 分支与循环实现 **7**
1.1.1 关系表达式翻译 7
1.1.2 布尔表达式翻译 8
1.1.3 分支语句的翻译 12
1.1.4 While 语句的翻译 14
1.2 MiniC 编译器的功能验证 **17**
1.2.1 线下测试 18
1.2.2 线上测试 19
第二章 附录 MiniC 文法 **20**
第三章 比赛用的 SysY 语言文法 **25**
第四章 附录 C 语言的 BNF 范式 **27**
第五章 附录 DragonIR 格式说明 **33**
5.1 标识符 **33**
5.2 常量或字面量 **33**
5.3 类型 **33**
5.4 基本变量定义 **34**
5.5 注释 **34**
5.6 变量赋值指令 **35**
5.6.1 全局变量的取值与设值 35
5.6.2 局部变量的取值与设值 35
5.6.3 临时变量的取值与设值 36

5.7 指针操作或内存操作 **36**
5.7.1 从内存取值 36
5.7.2 把值保存到内存中 36

5.8 表达式运算指令 **37**
5.8.1 算术运算 37
5.8.2 关系运算 38
5.8.3 逻辑运算 38

5.9 跳转与标签 IR 指令 **38**
5.9.1 Label 指令 38
5.9.2 无条件指令 38
5.9.3 有条件指令 39

5.10 数组定义与操作指令 **39**
5.10.1 数组定义 39
5.10.2 一维数组元素访问 40
5.10.3 多维数组元素访问 41

5.11 函数定义 **43**
5.11.1 函数返回值变量 43
5.11.2 函数入口指令 44
5.11.3 函数出口指令 44
5.11.4 return 语句翻译 44
5.11.5 函数形参处理 44
5.11.6 函数翻译举例 44

5.12 函数调用 **46**
5.12.1 有函数值返回的函数调用 46
5.12.2 调用没有返回值的函数调用 48

5.13 内置函数 **49**

第六章 附录 DragonIR 翻译举例与分析 ***53***

6.1 表达式语句翻译 **53**
6.1.1 源代码 53
6.1.2 生成的 IR 54

6.2 if-else 语句翻译 **54**
6.2.1 源代码 54
6.2.2 生成的 IR 55

6.3 if 语句翻译 **55**
6.3.1 源代码 55
6.3.2 生成的 IR 56

6.4 逻辑与的翻译 **56**
6.4.1 源代码 56
6.4.2 生成的 IR 56

6.5 逻辑或的翻译 **57**
6.5.1 源代码 57
6.5.2 生成的 IR 57

6.6 逻辑非的翻译 **58**
6.6.1 源代码 58
6.6.2 生成的 IR 58

6.7 while 语句翻译 **59**
6.7.1 源代码 59
6.7.2 生成的 IR 60

6.8 函数调用 **60**
6.8.1 源代码 60
6.8.2 生成的 IR（不使用 arg 指令） 61
6.8.3 生成的 IR（使用 arg 指令） 63

6.9 一维数组的样例 **65**
6.9.1 源代码 65
6.9.2 生成的 IR 66

6.10 多维数组的样例 **67**
6.10.1 源代码 67
6.10.2 生成的 IR 69

6.11 函数 return 语句翻译样例 **75**

6.11.1 源代码 75
6.11.2 生成的 IR 75

第一章 分支与循环实现

1.1.1 关系表达式翻译

关系表达式指的是含有大于、小于、大于等于、小于等于、等于和不等于运算符的表达式。这里也要考虑不同类型变量的隐式转换，类似算术表达式的处理，这里不做过多的阐述。

在进行翻译前，需要关系运算符的父亲节点传真假出口的 Label 指令给关系运算符节点，用于有条件跳转指令的目标位置设置。

如关系表达式 a<b 的翻译结果为：

```
%t1 = cmp lt %l1, %l2          ; %l1 代表变量 a，%l2 代表变量 b
bc %t1, label .L0, label .L1    ; .L0 代表真出口，.L1 代表假出口
```

其中 cmp lt 代表小于指令，bc 代表条件跳转指令。

如图 5 是关系表达式的 AST，关系运算符的具体值保存在关系运算节点的属性中。先遍历左孩子，产生 IR 指令，其运算结果赋值给 t1，再遍历右孩子，产生 IR 指令，其运算结果赋值给 t2，然后产生比较操作的 IR 指令，最后产生有条件跳转指令。

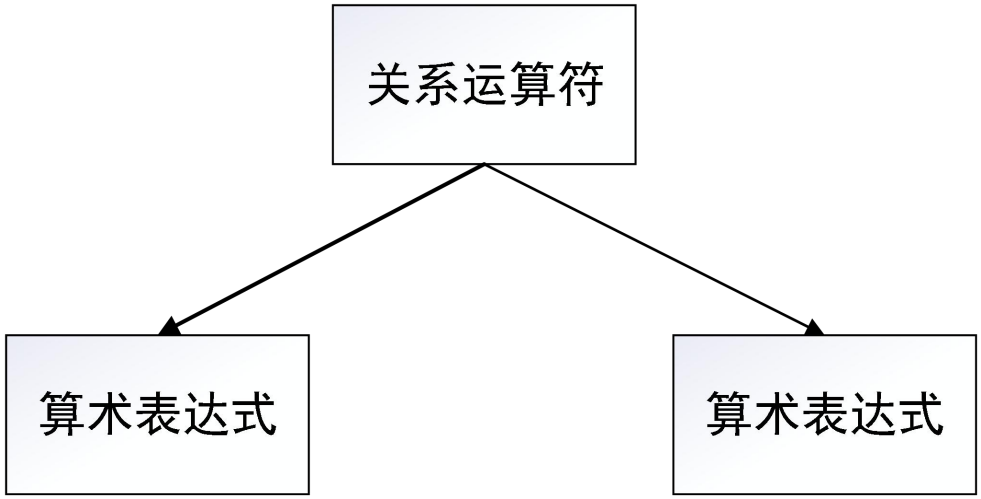


图 5 关系运算符的 AST

例如(a + b) < (c + d)的翻译结果为：

```
%t1 = %l1 + %l2        ; %l1 代表变量 a，%l2 代表变量 b
%t2 = %l3 + %l4        ; %l3 代表变量 a，%l4 代表变量 b
%t3 = cmp lt %t1, %t2
bc %t3, label .L0, Label .L1
```

1.1.2 布尔表达式翻译

主要关注对逻辑与、逻辑或和逻辑非的翻译，其实现的思想是短路求值，转换成有条件转移和无条件转移语句。在翻译的过程中，需要引入真出口或者假出口，通过真假出口的拉链与回填等技术实现。

下面针对逻辑与和逻辑非的翻译进行介绍。逻辑或操作类似逻辑与，请参考逻辑与进行实现。

如图 6 所示，表示的是 C 程序表达式 BoolExpr0 && BoolExpr01 所对应的抽象语法树。AND 表示逻辑与运算，BoolExpr0 和 BoolExpr01 代表布尔表达式。布尔表达式可以是关系表达式或者其它的逻辑表达式等。

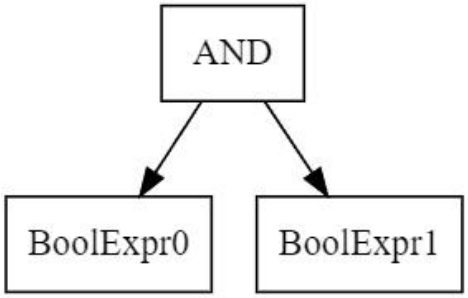


图 6 逻辑与的 AST

在翻译的过程中，仍然采用深度优先遍历，自左往右逐步产生线性 IR，由于是自顶而下的遍历，逻辑与整体的真假出口处置可通过两种方式来进行，一种方式是在遍历孩子节点前由参数传入，外部创建真假出口 Label 指令，在翻译的过程中，不断的向下传递使用；另外一种是不传入参数，真假出口未知，在翻译过程中要把真假出口进行拉链，等知道出口时用真实的出口进行回填即可。

本实验建议大家使用第一种方式进行实现。如图 7 所示，假定传入的真假出口为两个 Label 指令，假定为 L1 和 L2。

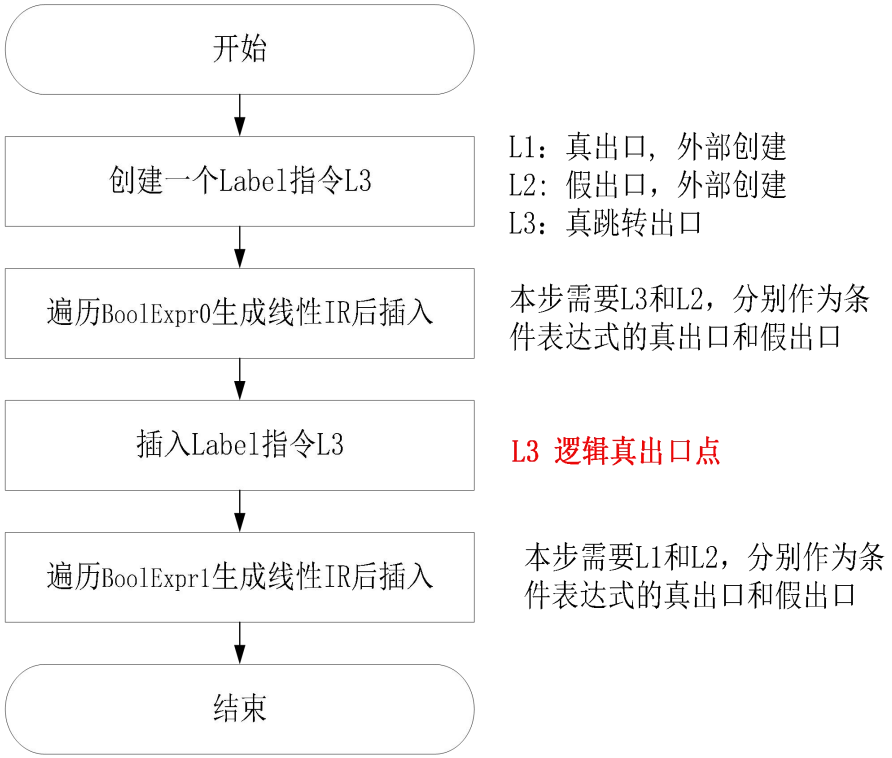


图 7 逻辑与转换线性 IR

逻辑非的 AST 如所示，NOT 节点只有一个孩子 BoolExpr，BoolExpr 表示布尔表达式。

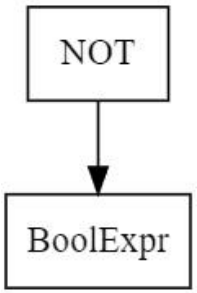


图 8 NOT 的 AST

逻辑非 AST 到线性 IR 的转换非常简单，只要把真假出口调换一下然后传入作为 BoolExpr 的真假出口即可，如图 9 所示。

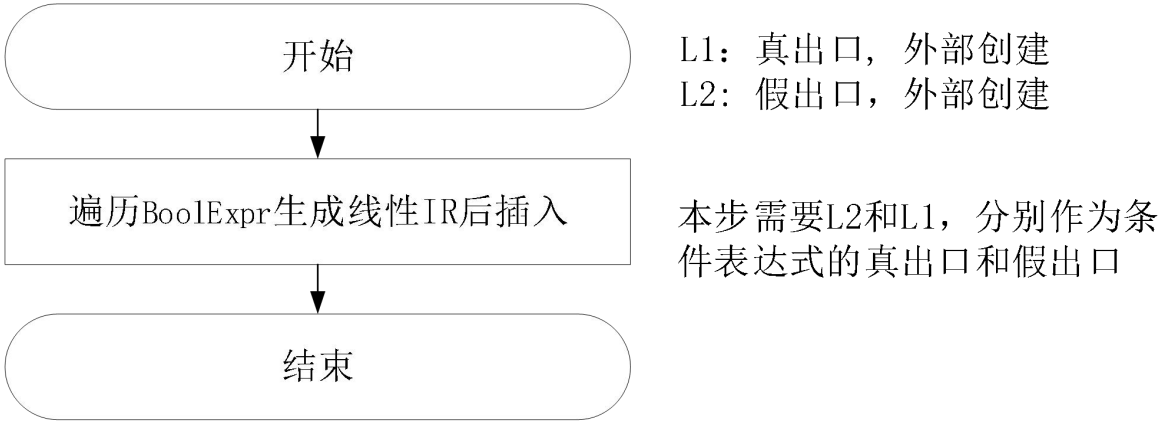


图 9 NOT 到线性 IR 的转换算法

1.1.3 分支语句的翻译

分支语句对应的 AST 树如图 10 所示，其中 CondExpr 表示 if 语句的条件，TrueBlock 表示真语句块，FalseBlock 表示假语句块。若语句块没有用大括号括住时，TrueBlock 或 FalseBlock 节点类型是语句。若 if 语句没有 else，则 FalseBlock 孩子节点不存在。

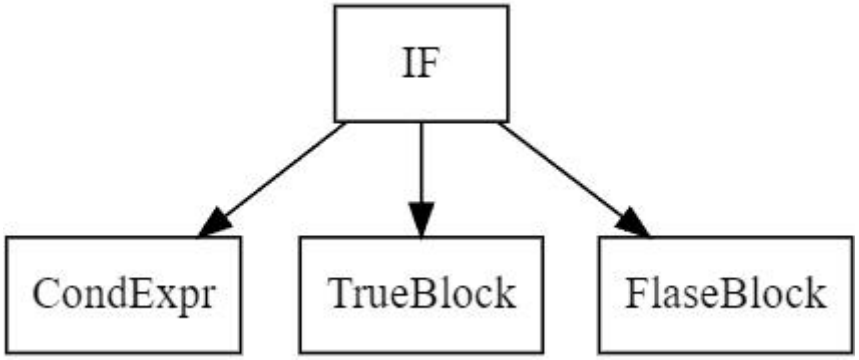


图 10 IF 语句的抽象语法树

AST 到线性 IR 的翻译算法如图 11 所示，首先创建三个出口点，分别代表真出口、假出口、和分支语句出口，然后采用深度优先遍历算法遍历 IF 节点的三个孩子然后产生相应的线性 IR，然后放到 IF 语句对应的线性 IR 中，不过在孩子节点遍历时，需要在适当的地方加入真出口点、假出口点以及一些无条件跳转指令，具体步骤按照图 11 所示进行。请注意，FalseBlock 节点可能不存在，对于 if-then 语句，若存在则是 if-then-else 语句。

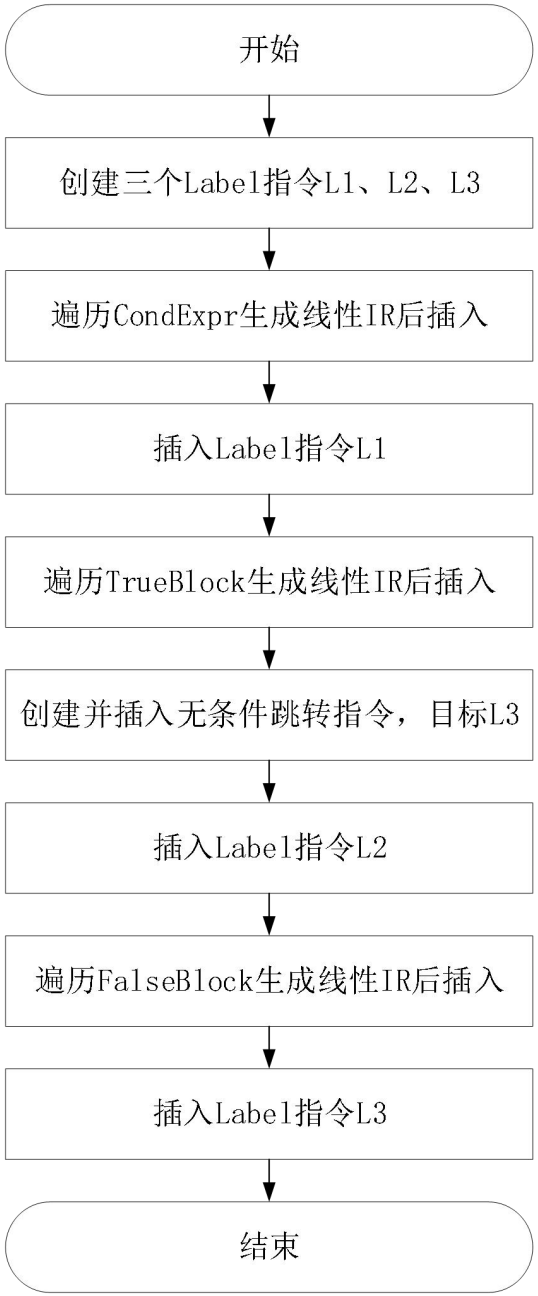

开始
创建三个Label指令L1、L2、L3
L1：真出口
L2：假出口
L3：分支语句出口
遍历CondExpr生成线性IR后插入
本步需要L1和L2，分别作为条件表达式的真出口和假出口
插入Label指令L1
L1真出口点
遍历TrueBlock生成线性IR后插入
创建并插入无条件跳转指令，目标L3
插入Label指令L2
L2假出口点
遍历FalseBlock生成线性IR后插入
插入Label指令L3
L3分支语句出口点
结束

图 11 分支语句转换线性 IR 翻译算法

例如下面的 C 语言程序：

```
if(a > 2) {
    b = 0;
} else {
    b = 1
}
```

其翻译后的结果如下图所示：

```
    %t1 = icmp gt %l1, 2          ; %11 代表变量 a
    bc %t1, label .L1, label .L2
.L1:                              ; if 语句的真出口
    %l2 = 0                       ; %12 代表变量 b
    br label .L3
.L2:                              ; if 语句的假出口
    %l2 = 1
.L3:                              ; if 语句的出口
```

1.1.4 While 语句的翻译

分支语句对应的 AST 树如图 12 所示，CondExpr 表示循环语句的条件，BodyBlock 表示循环的循环体。若循环体没有用大括号括住时，BodyBlock 可能是语句节点。

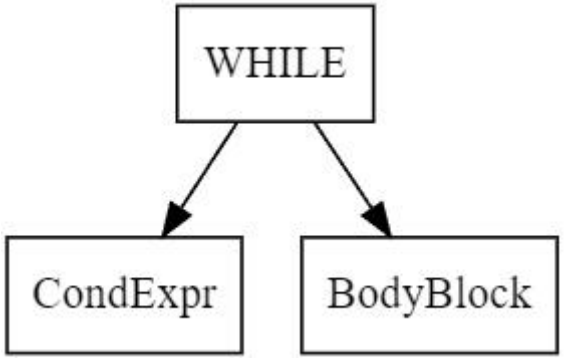
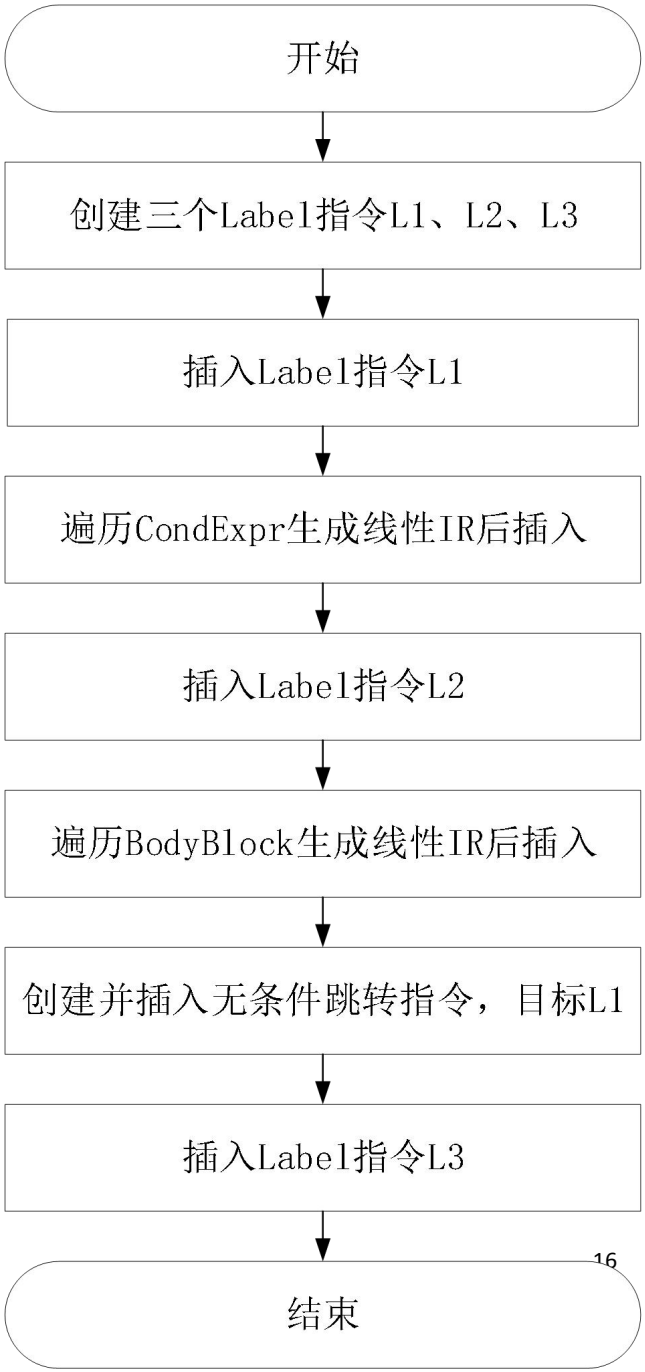


图 12 While 语句的抽象语法树

AST 到线性 IR 的翻译算法如图 13 所示，首先创建三个出口点，分别代表循环入口、循环体入口、和循环语句出口，然后采用深度优先遍历算法遍历 IF 节点的两个孩子然后产生相应的线性 IR，然后放到 While 语句对应的线性 IR 中，不过在孩子节点遍历时，需要在适当的地方加入循环入口、循环体入口、和循环语句出口点以及一些无条件跳转指令，具体步骤按照图 13 所示进行。



L1：循环入口
L2：循环体入口
L3：循环语句出口

L1循环入口点

本步需要L2和L3，分别作为条件表达式的真出口和假出口

L2循环体入口

L3循环语句出口点

图 13 While 语句转换线性 IR 翻译算法

例如下面的 C 语言程序：

```
while(a < 10) {
    b = b + 1;
    a = a + 1;
}
```

其翻译后的结果如下图所示：

```
.L1:                                    ; 循环体的入口
     %t1 = icmp lt %l1, 2               ; %11 代表变量 a
     bc %t1, label .L2, label .L3
.L2:                                    ; 循环体的入口
     %t2 = %l2 + 1                      ; %12 代表变量 b
     %l2 = %t2
     %t3 = %l1 + 1                      ; %11 代表变量 a
     %l1 = %t3
     br label .L1
.L3:                                    ; 循环体的出口
```

1.2 MiniC 编译器的功能验证

如生成的是 LLVM 的 IR，请用 llc 等工具编译后执行。如果是本教材定义的 DragonIR，请按照如下的方式进行测试验证。

1.2.1 线下测试

1.2.1.1 测试用例和 IR 虚拟机工具

测试用例和 IR 虚拟机工具放置在如下的网址：

http://10.69.45.39:30080/publicprojects/commonclasstestcases

测试用例在 function 文件夹下。IR 编译运行器在 IRCompiler 文件夹下，根据平台选择对应的 IR 编译运行器。

IRCompiler 可能会存在问题，如有问题，请在 QQ 群中报告。在报告时，请给出源代码程序以及产生的 IR 文件。

1.2.1.2 使用前准备

请通过 git 把测试用例以及 IR 编译运行器克隆到本地进行。具体使用和要求请查阅 Git 中的 Readme.md 文件。

在运行前请把本实验所实现的 MiniC 编译器可执行程序 minic（Windows 上后缀为.exe）拷贝到与 readme.md 文件同一级目录下。

1.2.1.3 单测试用例手动指定验证

下面以 Windows 系统为例进行说明。

以 000_main.c 的测试用例为例说明，该文件不接收输入和输出

第一步，执行如下的命令，输出 IR 内容到文件 000_main.ir 中

./minic.exe -i -o 000_main.ir 000_main.c

第二步，执行如下的命令对 IR 进行编译与运行

./IRCompiler.exe -R 000_main.ir

如果有输出，则可看到输出内容。

如果测试用例要求输入，则可如下方式执行：

./IRCompiler.exe -R 046_op_priority4.ir < 046_op_priority4.in

第三步，查看程序指令的结果

echo $?

第四步，查看第二步输出内容以及第三步输出内容是否和 000_main.out 的内容一致。如果一致，则功能正常，否则不正常。

1.2.1.4 测试用例自动运行验证

确保本实验实现的 minic 编译器程序拷贝到与 readme.md 同一层文件夹下。

单个用例测试执行,假定执行第 3 个用例，则 sh minicrun.sh 3 即可

多个用例测试执行，假定从第 2 个到第 6 个，则 sh minicrun.sh 2 6 即可。

如果有错误，可在 function 目录下查看.ir 文件和.result 文件。

1.2.2 线上测试

请利用学校内的编译在线评测平台进行评测。

平台网址：http://10.69.45.39:18081/

第二章 附录 MiniC 文法

采用 EBNF 方式来描述 MiniC 文法。

G[<program>]:

<program> → { }

 → <type> <def>　　全局变量或者函数名定义或声明

<type> → 'int' | 'void'　　基本类型，int 或 void

<def> → ident (<varrdef> <deflist> | '(' <para> ')' <functail>)　　全局变量或者函数名定义

<varrdef> → { '[' num ']' }　　数组元素纬度指定，只允许无符号常量

<defdata> → ident <varrdef>　　下标变量或简单变量

<deflist> → ',' <defdata> <deflist> | ';'　　同类型多个变量名列表

<functail> → <blockstat> | ';'　　只有分号函数声明，含有语句块则是函数定义

<para> → <onepara> { ',' <onepara> } | ε　　函数参数

<para> → <paralist> | ε　　函数参数　　文法改造举例：修改成左递归更容易创建 AST

<paralist> → <paralist> ',' <onepara> | <onepara>

<onepara> → <type> <paradata>　　函数形参定义，<type>不能为 void

<paradata> → ident <paradatatail>　　形参为数组形式的变量或简单变量

<paradatatail> → '[' num? ']' { '[' num ']' } | ε

<localdef> → <type> <defdata> <deflist>　　本地定义语句，<type>不能为 void

<breakstat> → 'break' ';'　　break 语句

<continuestat> → 'continue' ';'　　continue 语句

<returnstat> → 'return' [<expr>] ';'　　返回语句

<assignstat> → <expr> ';'　　赋值语句或者表达式语句

<onestatement> → <localdef> | <statement>　　局部变量定义或者语句

<compoundstatement> → { <onestatement> }　　复合语句

<compoundstatement> → <statementlist> | ε　　文法改造举例：修改成左递归更容易创建 AST

<statementlist> → <statementlist> <onestatement> | <onestatement>

<blockstat> → '{' <compoundstatement> '}'　　语句块

<emptystat> → ';'　　空语句

```
<whilestat> → 'while' '(' <expr> ')' <statement>  while 循环语句

<ifstat> → 'if' '(' <expr> ')' <statement> [ 'else' <statement> ]   分支语句

<statement> → <whilestat>
    | <ifstat>
    | <breakstat>
    | <continuestat>
    | <returnstat>
    | <blockstat>
    | <assignstat>
    | <emptystat>
<expr> → <assexpr>                          表达式

<lval> → ident { '[' <expr> ']' }             左值

<assexpr> → <orexpr> | <lval> '=' <assexpr>    赋值表达式

<orexpr> → <andexpr> { '||' <andexpr> }        逻辑或表达式
```

<andexpr> → <cmpexpr> { '&&' <cmpexpr> }　　逻辑与表达式

<cmpexpr> → <aloexpr> { <cmps> <aloexpr> }　　关系表达式

<cmps> → '>=' | '>' | '<' | '<=' | '==' | '!=' 关系运算符

<aloexpr> → <item> { <addsub> <item> }　　加减表达式

<addsub> → '+' | '-'　　加减运算符

<item> → <factor> { <muldiv> <factor> }　　乘除余表达式

<muldiv> → '*' | '/' | '%'　　乘除余运算符

<factor> → <lop> <factor> | <selfexpr> | <funccall> | <elem>　一元运算，函数调用，自增自减运算

<lop> → '!' | '-'　　求负和逻辑非运算符

<funccall> → ident '(' <realarg> ')'　　函数调用

<selfexpr> → <selfop> <lval> | <lval> <selfop> 自增自减运算

<selfop> → '++' | '--'　　自增自减运算符

<elem> → <lval> | num | '(' <expr> ')'　　右值：变量，数组，常量，表达式

<realarg> → <expr> {',' <expr>} | ε　　实际参数列表

非终结符为<>括起来的符号

终结符是由单引号括起的串，或者 ident 、num 等没有用<>括起来的符号。

ident 代表标识符，num 代表整型常量

对于单字符的终结符，flex 可直接返回字符的 ASCII 值即可，而对于多字符的终结符，需要定义 Token 符号。

第三章 比赛用的 SysY 语言文法

SysY 语言的文法采用扩展的 Backus 范式（EBNF，Extended Backus-Naur Form）表示，其中：

符号[...]表示方括号内包含的为可选项

符号{...}表示花括号内包含的为可重复 0 次或多次的项

终结符或者是由单引号括起的串，或者是 Ident、InstConst 这样的记号。

SysY 语言的文法表示如下，其中 CompUnit 为开始符号：

编译单元 CompUnit → [CompUnit] (Decl | FuncDef)

声明 Decl → ConstDecl | VarDecl

常量声明 ConstDecl → 'const' BType ConstDef { ',' ConstDef } ';'

基本类型 BType → 'int'

常数定义 ConstDef → Ident { '[' ConstExp ']' } '=' ConstInitVal

常量初值 ConstInitVal → ConstExp

| '{' [ConstInitVal { ',' ConstInitVal }] '}'

变量声明 VarDecl → BType VarDef { ',' VarDef } ';'

变量定义 VarDef → Ident { '[' ConstExp ']' }

| Ident { '[' ConstExp ']' } '=' InitVal

变量初值 InitVal → Exp | '{' [InitVal { ',' InitVal }] '}'

函数定义 FuncDef → FuncType Ident '(' [FuncFParams] ')' Block

函数类型 FuncType → 'void' | 'int'

函数形参表 FuncFParams → FuncFParam { ',' FuncFParam }

函数形参 FuncFParam → BType Ident ['[' ']' { '[' Exp ']' }]

语句块 Block → '{' { BlockItem } '}'

语句块项 BlockItem → Decl | Stmt

语句 Stmt → LVal '=' Exp ';' | [Exp] ';' | Block

| 'if' '(Cond ')' Stmt ['else' Stmt]

| 'while' '(' Cond ')' Stmt

| 'break' ';' | 'continue' ';' | 'return' [Exp] ';'

表达式 Exp → AddExp 注：SysY 表达式是 int 型表达式

条件表达式 Cond → LOrExp

左值表达式 LVal → Ident {'[' Exp ']'}

基本表达式 PrimaryExp → '(' Exp ')' | LVal | Number

数值 Number → IntConst

一元表达式 UnaryExp → PrimaryExp

| Ident '(' [FuncRParams] ')' | UnaryOp UnaryExp

单目运算符 UnaryOp → '+' | '-' | '!' 注：'!'仅出现在条件表达式中

函数实参表 FuncRParams → Exp { ',' Exp }

乘除模表达式 MulExp → UnaryExp | MulExp ('*' | '/' | '%') UnaryExp

加减表达式 AddExp → MulExp | AddExp ('+' | '-') MulExp

关系表达式 RelExp → AddExp | RelExp ('<' | '>' | '<=' | '>=') AddExp

相等性表达式 EqExp → RelExp | EqExp ('==' | '!=') RelExp

逻辑与表达式 LAndExp → EqExp | LAndExp '&&' EqExp

逻辑或表达式 LOrExp → LAndExp | LOrExp '||' LAndExp

常量表达式 ConstExp → AddExp 注：使用的 Ident 必须是常量

第四章 附录 C 语言的 BNF 范式

<translation-unit> ::= {<external-declaration>}*

<external-declaration> ::= <function-definition>
| <declaration>

<function-definition> ::= {<declaration-specifier>}* <declarator> {<declaration>}*
<compound-statement>

<declaration-specifier> ::= <storage-class-specifier>
| <type-specifier>
| <type-qualifier>

<storage-class-specifier> ::= auto
| register
| static
| extern
| typedef

<type-specifier> ::= void
| char
| short
| int
| long
| float
| double
| signed
| unsigned
| <struct-or-union-specifier>
| <enum-specifier>
| <typedef-name>

<struct-or-union-specifier> ::= <struct-or-union> <identifier> { {<struct-declaration>}+ }
| <struct-or-union> { {<struct-declaration>}+ }
| <struct-or-union> <identifier>

<struct-or-union> ::= struct
| union

<struct-declaration> ::= {<specifier-qualifier>}* <struct-declarator-list>

<specifier-qualifier> ::= <type-specifier>
| <type-qualifier>

<struct-declarator-list> ::= <struct-declarator>
| <struct-declarator-list> , <struct-declarator>

<struct-declarator> ::= <declarator>
| <declarator> : <constant-expression>
| : <constant-expression>

<declarator> ::= {<pointer>}? <direct-declarator>

<pointer> ::= * {<type-qualifier>}* {<pointer>}?

<type-qualifier> ::= const
| volatile

<direct-declarator> ::= <identifier>
| (<declarator>)
| <direct-declarator> [{<constant-expression>}?]
| <direct-declarator> (<parameter-type-list>)
| <direct-declarator> ({<identifier>}*)

<constant-expression> ::= <conditional-expression>

<conditional-expression> ::= <logical-or-expression>
| <logical-or-expression> ? <expression> : <conditional-expression>

<logical-or-expression> ::= <logical-and-expression>
| <logical-or-expression> || <logical-and-expression>

<logical-and-expression> ::= <inclusive-or-expression>
| <logical-and-expression> && <inclusive-or-expression>

<inclusive-or-expression> ::= <exclusive-or-expression>
| <inclusive-or-expression> | <exclusive-or-expression>

<exclusive-or-expression> ::= <and-expression>
| <exclusive-or-expression> ^ <and-expression>

<and-expression> ::= <equality-expression>
| <and-expression> & <equality-expression>

<equality-expression> ::= <relational-expression>
| <equality-expression> == <relational-expression>
| <equality-expression> != <relational-expression>

<relational-expression> ::= <shift-expression>
| <relational-expression> < <shift-expression>

```
                                    | <relational-expression> > <shift-expression>
                                    | <relational-expression> <= <shift-expression>
                                    | <relational-expression> >= <shift-expression>

<shift-expression> ::= <additive-expression>
                                | <shift-expression> << <additive-expression>
                                | <shift-expression> >> <additive-expression>

<additive-expression> ::= <multiplicative-expression>
                                    | <additive-expression> + <multiplicative-expression>
                                    | <additive-expression> - <multiplicative-expression>

<multiplicative-expression> ::= <cast-expression>
                                            | <multiplicative-expression> * <cast-expression>
                                            | <multiplicative-expression> / <cast-expression>
                                            | <multiplicative-expression> % <cast-expression>

<cast-expression> ::= <unary-expression>
                                | ( <type-name> ) <cast-expression>

<unary-expression> ::= <postfix-expression>
                                | ++ <unary-expression>
                                | -- <unary-expression>
                                | <unary-operator> <cast-expression>
                                | sizeof <unary-expression>
                                | sizeof <type-name>

<postfix-expression> ::= <primary-expression>
                                    | <postfix-expression> [ <expression> ]
                                    | <postfix-expression> ( {<assignment-expression>}* )
                                    | <postfix-expression> . <identifier>
                                    | <postfix-expression> -> <identifier>
                                    | <postfix-expression> ++
                                    | <postfix-expression> --

<primary-expression> ::= <identifier>
                                    | <constant>
                                    | <string>
                                    | ( <expression> )

<constant> ::= <integer-constant>
                        | <character-constant>
                        | <floating-constant>
                        | <enumeration-constant>

<expression> ::= <assignment-expression>
                            | <expression> , <assignment-expression>
```

<assignment-expression> ::= <conditional-expression>
| <unary-expression> <assignment-operator> <assignment-expression>

<assignment-operator> ::= =
| *=
| /=
| %=
| +=
| -=
| <<=
| >>=
| &=
| ^=
| |=

<unary-operator> ::= &
| *
| +
| -
| ~
| !

<type-name> ::= {<specifier-qualifier>}+ {<abstract-declarator>}?

<parameter-type-list> ::= <parameter-list>
| <parameter-list> , ...

<parameter-list> ::= <parameter-declaration>
| <parameter-list> , <parameter-declaration>

<parameter-declaration> ::= {<declaration-specifier>}+ <declarator>
| {<declaration-specifier>}+ <abstract-declarator>
| {<declaration-specifier>}+

<abstract-declarator> ::= <pointer>
| <pointer> <direct-abstract-declarator>
| <direct-abstract-declarator>

<direct-abstract-declarator> ::= (<abstract-declarator>)
| {<direct-abstract-declarator>}? [{<constant-expression>}?]
| {<direct-abstract-declarator>}? ({<parameter-type-list>}?)

<enum-specifier> ::= enum <identifier> { <enumerator-list> }

```
                        | enum { <enumerator-list> }
                        | enum <identifier>

<enumerator-list> ::= <enumerator>
                        | <enumerator-list> , <enumerator>

<enumerator> ::= <identifier>
                    | <identifier> = <constant-expression>

<typedef-name> ::= <identifier>

<declaration> ::=   {<declaration-specifier>}+ {<init-declarator>}* ;

<init-declarator> ::= <declarator>
                        | <declarator> = <initializer>

<initializer> ::= <assignment-expression>
                    | { <initializer-list> }
                    | { <initializer-list> , }

<initializer-list> ::= <initializer>
                        | <initializer-list> , <initializer>

<compound-statement> ::= { {<declaration>}* {<statement>}* }

<statement> ::= <labeled-statement>
                | <expression-statement>
                | <compound-statement>
                | <selection-statement>
                | <iteration-statement>
                | <jump-statement>

<labeled-statement> ::= <identifier> : <statement>
                        | case <constant-expression> : <statement>
                        | default : <statement>

<expression-statement> ::= {<expression>}? ;

<selection-statement> ::= if ( <expression> ) <statement>
                        | if ( <expression> ) <statement> else <statement>
                        | switch ( <expression> ) <statement>

<iteration-statement> ::= while ( <expression> ) <statement>
                        | do <statement> while ( <expression> ) ;
                        | for ( {<expression>}? ; {<expression>}? ; {<expression>}? )
<statement>
```

<jump-statement> ::= goto <identifier> ;
| continue ;
| break ;
| return {<expression>}? ;

第五章 附录 DragonIR 格式说明

本章节对 MiniC 编译器前端所产生的中间 IR 进行定义和说明。

5.1 标识符

自定义 IR 主要包含四种标识符，它们是全局变量、局部变量或形式参数、内部生成的临时变量、内部生成的无条件跳转的标签和函数名。

全局变量：标识符要求%g 或者@开头，后面紧跟原始的全局变量名。建议全局变量按@开头。**全局变量可赋值多次，即可做左值，也可做右值。要求全局变量全局唯一，不可重复。**

局部变量：标识符要求以%l 开头，后面为数字或字母组成的符号串，建议数字。需要注意%后面的是小写字母 l（对应的大写为 L），不是数字 1。**局部变量可赋值多次，即可做左值，也可做右值。**

临时变量：标识符以%t 开头，后面为数字或字母组成的符号串，建议数字，为编译器内部生成的临时变量。**临时变量只能赋值一次，以后只能用作右值，不能左值。**

标签：标识符以.L 开头的，后面为数字或字母组成的符号串，建议数字，用于条件或者无条件跳转指令跳转目标。**在同一个函数内部，标签不能重复。**

函数名：标识符要求@开头，后面紧跟原始的函数名。要求函数名全局唯一，不可重复。

以@开头的符号成为全局符号，要求全局唯一，不可多次定义。

5.2 常量或字面量

本章定义的常量只能是十进制形式的非负的整型字面量，如 123。

5.3 类型

基本类型主要有 i32、i8、i1 和 void，支持指针。
详细如下：

i32 表示 32 位 int 类型，相当于 C 语言的 int 类型

i8 表示 8 位 int 类型，相当于 C 语言的 char 类型

i1 表示 1 位 int 类型，相当于布尔类型

void 表示 C 语言的 void 类型，主要用于表示函数无返回值。

i32 *代表指向 i32 的指针变量类

i8 *代表指向 i8 的指针变量。

5.4 基本变量定义

变量定义要以 declare 开头，后面是类型，最后是标识符。变量要遵从先定义后使用的规则，否则语义错误。

例如：

declare i32 @f2 全局变量的声明，i32 为类型，@开头的是全局变量，变量名为 f2。

declare i32 %gf2 全局变量的声明，i32 为类型，%g 开头的是全局变量，变量名为 f2。

declare i32 %l2 局部变量的声明，需要在栈内分配空间

declare i32 %t10 临时变量的声明，不需要在栈内分配空间，切记此类变量只能定值一次。

全局变量的定义需放在 IR 文件的头部，第一个函数定义的前面。

函数内的局部变量和编译器生成的临时变量在函数体的开始部分声明，且在第一条入口指令的前面。也就是说，这部分是函数内的变量符号表显示。

5.5 注释

本 IR 只支持单行注释，以分号(;)开始。

以分号(;)开始到行末的所有内容为注释。

5.6 变量赋值指令

5.6.1 全局变量的取值与设值

全局变量可作为左值和右值。

例如，假定 a 和 b 为全局变量，下面是赋值指令

@a=3 常量赋值

@a=%t1 临时变量赋值

@a=%l1 局部变量赋值，建议先取到临时变量后赋值给全局变量

@a=@b 全局变量赋值，建议先取到临时变量后赋值给全局变量

%t0=@a 全局变量设值给临时变量

%l0=@a 全局变量赋值局部变量，建议先取到临时变量后赋值局部变量

@b=@a 全局变量赋值全局变量，建议先取到临时变量后赋值全局变量

5.6.2 局部变量的取值与设值

局部变量可作为左值和右值。

例如，假定 a 和 b 为全局变量，下面是赋值指令

%l0=3 常量赋值

%l0=%t1 临时变量赋值

%l0=%l1 局部变量赋值，建议先取到临时变量后赋值给局部变量

%l0=@b 全局变量赋值，建议先取到临时变量后赋值给局部变量

%t0=%l0 局部变量赋值临时变量

%l0=%l1 局部变量赋值局部变量，建议先取到临时变量后赋值给局部变量

@b=%l0 局部变量赋值全局变量，建议先取到临时变量后赋值给全局变量

5.6.3 临时变量的取值与设值

临时变量只能设值一次，之后可多次使用。约定临时变量不能赋值给临时变量，常量不能赋值给临时变量。

%t0=@a 全局变量赋值给临时变量

%t0=%l0 局部变量赋值给临时变量

临时变量可用作右值，可多次使用，请参考局部变量或全局变量的取值以及表达式运算中源操作数的使用。

5.7 指针操作或内存操作

5.7.1 从内存取值

形式定义：变量 = * 变量

功能：从右侧变量（指针变量）所指向的内存中获取值赋值给左侧的变量（如左值的临时变量）。要求右侧的变量要是指针类型，并且左侧与右侧指向的类型一致。

5.7.2 把值保存到内存中

形式定义：*变量 = 变量

功能：把右侧的变量值保存到左侧变量（指针变量）所指向的内存中。要求左侧的变量要是指针类型，并且左侧与右侧指向的类型一致。

5.8 表达式运算指令

表达式运算后的结果赋值给临时变量，一般不直接赋值给局部变量或全局变量。表达式右侧的源操作数变量可以是可右值操作的所有变量，常量亦包含在内。

5.8.1 算术运算

格式：变量 = 算术运算符 变量, 变量

算术运算符 可以是 add 、 sub 、 mul 、 div 、 mod

这三个变量的类型一般要求要一致，但是 add 和 sub 指令还允许右侧变量一个为指针类型变量，一个为基本类型变量，结果为指针类型变量。

add：加法运算

sub：减法运算

mul：乘法运算

div：求商运算

mod：求余运算

特别说明，add 或 sub 指令可用于类似指针的加整型变量的偏移获取元素地址的功能，这时，本 IR 要求 add 或 sub 指令的第一个源操作数（逗号左侧的变量）是指针型变量或者数组首地址，第二个源操作数（都好右侧的变量）是整型的临时变量或常量，局部变量不建议。

5.8.1.1 一元运算

格式：变量 = neg 变量

neg：求负运算

两个变量都是基本类型变量。

5.8.2 关系运算

格式：变量 = cmp 比较运算符 变量, 变量

比较运算符可以为 le、 lt、 gt、 ge 、ne 、eq

说明：比较结果（左侧变量）要求为 bool 类型，即 i1。

le: 小于等于

lt：小于

gt：大于

ge：大于等于

ne：不等于

eq：等于

5.8.3 逻辑运算

在语义分析的过程中，逻辑运算采用短路求值，转换为有条件或者无条件指令实现。因此中间 IR 不定义逻辑运算指令。具体参考后续的 IR 举例。

5.9 跳转与标签 IR 指令

5.9.1 Label 指令

格式：标识符:

说明：标识符用于定义 Label 名称，以.L 开头的标识符。要求函数内唯一，不可重复。

主要用于指令的跳转目标，类似与 C 语言的 label 语句。

5.9.2 无条件指令

格式：br label 标识符

说明：无条件跳转到标识符所代表的位置，标识符是前面 Label 指令所定义的 Label 名，其中关键字 label 可省略。注意：Label 名可在本条指令的前面，也可在本条指令的后面。

5.9.3 有条件指令

bc condvar, label X, label Y

condvar 条件临时变量，类型为 i1； X 为真跳转标签，Y 假跳转标签；本指令 s 表示若 condvar 为真，则跳转到 X 处执行，否则跳转到 Y 处执行。

bt condvar, label X

condvar 条件临时变量，类型为 i1； X 为真跳转标签；本指令表示若 condvar 为真，则跳转到 X 处执行，否则顺序执行下一条指令。

bf condvar, label X

condvar 条件临时变量，类型为 i1； X 为真跳转 Label 名；本指令表示若 condvar 为假，则跳转到 X 处执行，否则顺序执行下一条指令。

其中 label 为关键字，可省略，说明后面的标识符是 Label 名字，用做跳转指令的目标。

建议使用 bc 指令，不要使用 bt 或者 bf 指令。

5.10 数组定义与操作指令

5.10.1 数组定义

函数内的局部数组变量定义类似与 C 语言的数组定义，只是把类型 int 修改为 i32，变量名采用%l 开头的标识符。例如

int a[10][15];

翻译为:

declare i32 %l1[10][15]

请注意，对于全局或局部的数组定义，数组的维度必须大于 0，但是对于形式参数数组，翻译后的数组的第一个维要为 0，其它维大于 0。

形参形式的数组变量，在编译器内部实际是按指针进行处理的，并且忽略第一维度的数值。例如对于数组 int a[6][5]与 int a[][5]是等价的，因此这里形参数组可翻译为 i32 %t0[0][5]，第一维度的数值设置为 0，形参名修改为以%t 开头的变量，同时会分配同样类型的局部变量，declare i32 %l1[0][5]。具体可参考函数定义中形参的翻译。

5.10.2 一维数组元素访问

5.10.2.1 数组元素取值

假设%l6 代表变量 m，%l2 代表数组 a，%l5 代表变量 k，数组元素类型为 int，如下的

m = a[k];

翻译为：

```
%t10 = mul %l5, 4        ；元素偏移转换成字节偏移
%t11 = add %l2, %t10     ；数组元素首地址+偏移得到元素的字节位置
%l6 = *%t11    ；从内存中取值，假定取 4 个字节的数据
```

第一行指令实现 k 元素相对数组首地址的字节偏移

第二行指令实现 a[k]元素的地址，数组元素首地址+字节偏移，赋值给%t11

第三行指令实现从内存%t11(指针类型)获取值，赋值给局部变量 m 中

5.10.2.2 数组元素设值

a[k] = m;

翻译为：

```
%t10 = mul %l5, 4
%t11 = add %l2, %t10
```

*%t11=%l6

第一行指令实现 k 元素相对数组首地址的字节偏移

第二行指令实现 a[k]元素的地址，数组元素首地址+字节偏移，赋值给%t11

第三行指令实现把局部变量 m 的值保存到内存%t11(指针类型)所指向的内存中。

5.10.3 多维数组元素访问

5.10.3.1 数组元素取值

源代码片段：

```
int a[10][10];
int m, k, t;

t = a[m][k];
```

翻译后的 IR 片段

```
declare i32 %l1[10][10]        数组 a
declare i32 %l2                  变量 m
declare i32 %l3                  变量 k
declare i32 %l4                  变量 t

declare i32 %t5
declare i32 %t6
```

declare i32 %t7 保存 a[m][k]相对数组首地址的偏移，单位字节

declare i32* %t8 数组元素的位置，类型为指向 i8 的指针

```
%t5 = mul %l2, 10
%t6 = add %t5, %l3
%t7 = mul %t6, 4        (m*10+k)*sizeof(int)
%t8 = add %l1, %t7      数组元素首地址+元素的字节偏移后的指针
%l4 = *%t8                从内存中获取 i32 类型的元素值
```

5.10.3.2 数组元素设值

源代码片段：

```
int a[10][10];
int m, k, t;

t = a[m][k];
```

翻译后的 IR 片段

declare i32 %l1[10][10] 数组 a

declare i32 %l2 变量 m

declare i32 %l3 变量 k

declare i32 %l4 变量 t

```
declare i32 %t5
declare i32 %t6
declare i32 %t7
```

declare i32* %t8 数组元素的位置，类型为指向 i8 的指针

%t5 = mul %l2, 10

%t6 = add %t5, %l3

%t7 = mul %t6, 4 (m*10+k)*sizeof(int)

%t8 = add %l1, %t7 数组元素首地址+元素的字节偏移后的指针

*%t8 = %l4 把 t 的值设置到%t8 所指向的内存中

5.11 函数定义

要事先创建一个用于保存返回值的局部变量（称为返回值局部变量）、一个函数入口 Label 指令和一个函数出口标签指令。

函数入口标签指令位于函数的开头，其后紧跟一个函数入口 entry 指令。

函数出口标签指令放在函数的尾部，紧跟一个函数出口指令。该出口标签指令用于 return 语句的翻译的跳转目标，保持函数只有一个出口。

函数定义的翻译后的基本格式如下所示:

declare 指令 用于函数内所有变量的类型说明

函数入口标签指令 优化时可删除

函数入口指令

其它 IR 指令

函数出口标签指令 优化时可删除

函数出口指令

5.11.1 函数返回值变量

在函数翻译时，如果返回的返回值类型不是 void，则需要在函数的开头申请一个返回值局部变量，用于保存函数的返回值。

含有返回值的 return 语句的翻译可先把返回值保存到返回值局部变量中，然后再调整到跳转到函数出口标签。

5.11.2 函数入口指令

格式：entry

功能：函数的入口。每个函数有且只有一个。必须存在，用于表示函数的第一条有效指令。

放置在 declare 语句指令的后面，其它指令的前面。

5.11.3 函数出口指令

格式：exit 变量

格式：exit 常量

格式：exit

功能：函数的出口指令。每个函数有且只有一个出口指令。存在于函数的最尾部。这里的变量一般指的是翻译时创建的返回值局部变量。

5.11.4 return 语句翻译

对于含有返回值的 return 语句翻译，先把返回值保存到返回值局部变量中，然后通过 br 指令跳转到函数出口标签。

对于没有返回值的 return 语句翻译，直接通过 br 指令跳转到函数出口标签。

5.11.5 函数形参处理

对于函数的形参，首先通过 declare 分配与形参对应的局部变量（将来用于在栈内分配空间），其次把原始的形参变成临时变量类型，在 entry 指令后把形参的值（临时变量）保存到对应的局部变量中。后续对形参的所有引用修改为对应的局部变量引用。

5.11.6 函数翻译举例

int test(int a, int b)

```
    {
    ...
            return 0;
    ...
            return 1;
    }
```

转换为中间 IR 形式：

```
define i32 @test(i32 %t0, i32 %t1) {
        declare i32 %l2          ;返回值局部变量
        declare i32 %l3          ;形参 a 对应的局部变量
        declare i32 %l4          ; 形参 b 对应的局部变量
        ...
.L1:                             ; 函数入口标签指令
        Entry            ; 函数入口指令
        %l3 = %t0        ;a 赋值给 a 对应的局部变量
        %l4 = %t1        ;b 赋值给 b 对应的局部变量
                         后续所有对 a 和 b 的操作修改成对应的局部变量操作
        ...
        %l2 = 0           ; return 0 的翻译
        br .L1
        ...
        %l2 = 1          ; return 1 的翻译
        br .L1
.L2:                     ;函数出口标签
        exit %l2         ; 函数出口指令
}
```

其中 define 为关键字，函数名以@开头，要求全局唯一，后面跟 C 语言的函数名

局部变量%l2 为函数返回值在栈内分配的局部变量

局部变量%l3 为形参 a 在栈内分配的局部变量

局部变量%l4 为形参 b 在栈内分配的局部变量

entry 为函数的入口，将来对应函数内寄存器保存以及栈空间分配等

%l3 = %t0 对应把形参 a 的值保存到栈内对应的变量%l3 上

%l4 = %t1 对应把形参 b 的值保存到栈内对应的变量%l4 上

.L1 对应函数的入口 Label。

.L2 对应函数的出口 Label，所有的函数出口都跳转到该位置处，并设置返回值到变量%l2 上

%l2 = 0 和 br .L1 指令对应 return 0，先把返回值保存到返回值局部变量上，然后跳转到出口标签.L1 上。

exit %l2 带有返回值的退出指令。当然， %l2 变量的值也可以先暂存到临时变量中，然后通过指令 exit 临时变量也可实现同样的功能。

5.12 函数调用

C 语言的函数调用时函数参数的运算根据不同的编译器实现方式不同实现也不同。本 IR 规定函数实参按照自左往右运算，实参结果按自右往左依次入栈。对于个别后端 CPU，如允许通过寄存器传值，则实参结果可按照约定进行寄存器传值，之后仍旧按自右往左依次入栈。

5.12.1 有函数值返回的函数调用

使用 call 指令实现函数调用，或者借助 arg 指令与 call 指令完成。

例 1：

e = test(c, d);

转换为

%t1 = %l1

%t2 = %l2

%t3 = call i32 @test(i32 %t1,i32 %t2)

%l3 = %t3

或者

%t3 = call i32 @test(i32 %l1,i32 %l2)

%l3 = %t3

其中%l1 代表局部变量 c，%l2 代表局部变量 d，%l3 代表局部变量 e

%t1,%t2,%t3 为编译器内部生成的临时变量

例 2：

e = test2(c + d, c * d);

转换为

%t1 = %l1

%t2 = %l2

%t3 = add %t1, %t2

%t4 = %l1

%t5 = %l2

%t6 = mul %t4, %t5

%t7 = call i32 @test2(i32 %t3, i32 %t6)

%l3 = %t7

或者

%t1 = add %l1, %l2

%t2 = mul %l1, %l2

%t7 = call i32 @test2(i32 %t1, i32 %t2)

%l3 = %t7

其中%l1 代表局部变量 c，%l2 代表局部变量 d，%l3 代表局部变量 e

%t1~%t7 为编译器内部生成的临时变量

例 3：（不建议使用）

e = test(c, d);

转换为

arg %l1 ；自左往右传递参数，传递第一个实参

arg %l2 ；传递第二个实参

%t1 = call i32 @test() ；这里不写实参，只写调用

%l3 = %t1

其中%l1 代表局部变量 c，%l2 代表局部变量 d，%l3 代表局部变量 e

%t1 为编译器内部生成的临时变量

5.12.2 调用没有返回值的函数调用

函数调用 putint(c)转换后的 IR 为：

%t1 = %l1 ; %l1 代表变量 c

call void @putint(i32 %t1)

或者

call void @putint(i32 %l1)

5.13 内置函数

IR 虚拟机内置了一些支持标准终端的输入与输出功能的函数功能。这些函数作为标准的库函数使用，自定义函数不能使用这些函数名。在翻译中可直接调用这些函数，不用定义和声明。

每个函数的功能请参考内部实现的源代码，这些代码非常简单，这里不再解释说明。

```
int getint();
int getch();
int getarray(int a[]);

void putint(int a);
void putch(int a);
void putarray(int n,int a[]);
```

内部实现源代码如下：

```
#include <stdio.h>

int getint()
{
    int d;

    scanf("%d", &d);

    return d;
}
```

```
int getch()
{
	char d;

	scanf("%c", &d);

	return d;
}

int getarray(int a[])
{
	int n, i;

	// 获取元素个数
	scanf("%d",&n);

	// 获取元素内容
	for(i = 0; i < n; ++i) {
		scanf("%d",&a[i]);
	}

	return n;
}

void putint(int k)
{
	printf("%d", k);
```

```
}

void putch(int c)
{
	printf("%c", (char)c);
}

void putarray(int n, int * d)
{
	int k;

	// 输出元素个数
	printf("%d:", n);

	// 输出元素内容，空格分割
	for(k = 0; k < n; k ++) {
		printf(" %d", d[k]);
	}

	// 输出换行符
	printf("\n");
}

void putstr(char * str)
{
	printf("%s", str);
}
```

第六章 附录 DragonIR 翻译举例与分析

请利用 http://10.69.45.39:30080/publicprojects/commonclasstestcases.git 提供的 IR 编译器工具对输入的程序进行输出的验证。

假定输入的源代码为 A.c，则通过执行 IRCompiler -S -I -o A.ir A.c

输出中间 IR，IR 会输出到文件 A.ir 中。

如果要执行优化，删除一些不必要的 label 等，可加入-O2 参数。

这里列举了一些用例和输出，可自行追加查看输出效果。

6.1 表达式语句翻译

6.1.1 源代码

```
int g1;

int main()
{
    int a, b, c;
    int sum;

    a = getint();
    b = getint();
    c = 3;

    sum = -(a + b * c) + a;

    putint(sum);

    return 0;
}
```

6.1.2 生成的 IR

```
declare i32 @g1
define i32 @main() {
        declare i32 %l0
        declare i32 %l1 ; variable: a
        declare i32 %l2 ; variable: b
        declare i32 %l3 ; variable: c
        declare i32 %l4 ; variable: sum
        declare i32 %t5
        declare i32 %t6
        declare i32 %t7
        declare i32 %t8
        declare i32 %t9
        declare i32 %t10
.L1:
        entry
        %t5 = call i32 @getint()              ; 函数调用  a = getint()
        %l1 = %t5
        %t6 = call i32 @getint()              ; 函数调用  b = getint()
        %l2 = %t6
        %l3 = 3                               ; c = 3
        %t7 = mul %l2, %l3                    ; sum = -(a + b * c) + a;
        %t8 = add %l1, %t7
        %t9 = neg %t8
        %t10 = add %t9, %l1
        %l4 = %t10
        call void @putint(i32 %l4)          ; putint(sum);
        %l0 = 0
        br label .L2
.L2:
        exit %l0
}
```

6.2 if-else 语句翻译

6.2.1 源代码

```
int main()
{
    int a, b;
```

```
    if (a > 2) {
        b = 0;
    } else {
        b = 1;
    }

    return b;
}
```

6.2.2 生成的 IR

```
define i32 @main() {
     declare i32 %l0
     declare i32 %l1 ; variable: a
     declare i32 %l2 ; variable: b
     declare i1 %t3
.L1:
     entry
     %t3 = icmp gt %l1, 2
     bc %t3, label .L3, label .L4
.L3:                                ; if 语句的真出口
     %l2 = 0
     br label .L5                   ; 跳转到出口
.L4:                                ; if 语句的假出口
     %l2 = 1
     br label .L5                   ; 跳转到出口，可省略
.L5:                                ; if 语句的出口
     %l0 = %l2
     br label .L2
.L2:
     exit %l0
}
```

6.3 if 语句翻译

6.3.1 源代码

```
int main()
{
    int a, b;

    b = 1;

    if (a > 2) {
        b = 0;
    }
```

```
    return b;
}
```

6.3.2 生成的 IR

```
define i32 @main() {
      declare i32 %l0
      declare i32 %l1 ; variable: a
      declare i32 %l2 ; variable: b
      declare i1 %t3
.L1:
      entry
      %l2 = 1
      %t3 = icmp gt %l1, 2
      bc %t3, label .L3, label .L4
.L3:                                    ; if 语句的真出口
      %l2 = 0
.L4:                                    ; if 语句的假出口
.L5:                                    ; if 语句的出口
      %l0 = %l2
      br label .L2
.L2:
      exit %l0
}
```

6.4 逻辑与的翻译

6.4.1 源代码

```
int main()
{
    int a, b;

    if (a > 2 && a < 10) {
        b = 0;
    } else {
        b = 1;
    }

    return b;
}
```

6.4.2 生成的 IR

```
define i32 @main() {
      declare i32 %l0
      declare i32 %l1 ; variable: a
```

```
      declare i32 %l2 ; variable: b
      declare i1 %t3
      declare i1 %t4
.L1:
      entry
      %t3 = icmp gt %l1, 2              ; a > 2
      bc %t3, label .L6, label .L4      ; 短路求值
.L6:
      %t4 = icmp lt %l1, 10
      bc %t4, label .L3, label .L4
.L3:                                    ; if 语句的真出口
      %l2 = 0
      br label .L5
.L4:                                    ; if 语句的假出口
      %l2 = 1
      br label .L5
.L5:                                    ; if 语句的出口
      %l0 = %l2
      br label .L2
.L2:
      exit %l0
}
```

6.5 逻辑或的翻译

6.5.1 源代码

```
int main()
{
    int a, b;

    if (a > 2 || a < 0) {
        b = 0;
    } else {
        b = 1;
    }

    return b;
}
```

6.5.2 生成的 IR

```
define i32 @main() {
      declare i32 %l0
      declare i32 %l1 ; variable: a
      declare i32 %l2 ; variable: b
      declare i1 %t3
      declare i1 %t4
```

```
.L1:
      entry
      %t3 = icmp gt %l1, 2              ; a > 2
      bc %t3, label .L3, label .L6     ; 短路求值
.L6:
      %t4 = icmp lt %l1, 0              ; a < 0
      bc %t4, label .L3, label .L4
.L3:
      %l2 = 0
      br label .L5
.L4:
      %l2 = 1
      br label .L5
.L5:
      %l0 = %l2
      br label .L2
.L2:
      exit %l0
}
```

6.6 逻辑非的翻译

6.6.1 源代码

```
int main()
{
    int a, b;

    if ((a > 2 && a < 10)) {
        b = 0;
    } else {
        b = 1;
    }

    return b;
}
```

6.6.2 生成的 IR

```
define i32 @main() {
      declare i32 %l0
      declare i32 %l1 ; variable: a
      declare i32 %l2 ; variable: b
      declare i1 %t3
      declare i1 %t4
.L1:
      entry
```

```
      %t3 = icmp gt %l1, 2           ; a > 2
      bc %t3, label .L6, label .L3  ; 假出口变成真出口
.L6:
      %t4 = icmp lt %l1, 10          ; a < 10
      bc %t4, label .L4, label .L3  ; 真假出口翻转
.L3:                                  ; if 语句的真出口
      %l2 = 0
      br label .L5
.L4:                                 ; if 语句的假出口
      %l2 = 1
      br label .L5
.L5:                                 ; if 语句的出口
      %l0 = %l2
      br label .L2
.L2:
      exit %l0
}
```

6.7 while 语句翻译

6.7.1 源代码

```
int g1;


int main()
{
    int a = 1;
    int sum = 0;


    while(a < 101) {
        sum = sum + a;
        a = a + 1;
    }


    putint(sum);
```

```
    return 0;
}
```

6.7.2 生成的 IR

```
declare i32 @g1
define i32 @main() {
        declare i32 %l0
        declare i32 %l1 ; variable: a
        declare i32 %l2 ; variable: sum
        declare i1 %t3
        declare i32 %t4
        declare i32 %t5
.L1:
        entry
        %l1 = 1
        %l2 = 0
.L3:                                    ; while 循环的入口
        %t3 = icmp lt %l1, 101
        bc %t3, label .L4, label .L5
.L4:                                    ; while 循环体入口
        %t4 = add %l2, %l1
        %l2 = %t4
        %t5 = add %l1, 1
        %l1 = %t5
        br label .L3                    ; 跳转到循环的入口
.L5:                                    ; while 循环的出口
        call void @putint(i32 %l2)
        %l0 = 0
        br label .L2
.L2:
        exit %l0
}
```

6.8 函数调用

6.8.1 源代码

```
int test(int a, int b)
{
    int c;

    c = a + b;

    return c;
}

int main()
{
    int a, b;
    int sum;

    a = getint();
    b = getint();

    sum = test(a, b);

    putint(sum);

    return 0;
}
```

6.8.2 生成的 IR（不使用 arg 指令）

```
define i32 @test(i32 %t0, i32 %t1) {
        declare i32 %l2 ; variable: a
        declare i32 %l3 ; variable: b
```

```
        declare i32 %l4
        declare i32 %l5 ; variable: c
        declare i32 %t6
.L1:
        entry
        %l2 = %t0
        %l3 = %t1
        %t6 = add %l2, %l3
        %l5 = %t6
        %l4 = %l5
        br label .L2
.L2:
        exit %l4
}
define i32 @main() {
        declare i32 %l0
        declare i32 %l1 ; variable: a
        declare i32 %l2 ; variable: b
        declare i32 %l3 ; variable: sum
        declare i32 %t4
        declare i32 %t5
        declare i32 %t6
.L1:
        entry
        %t4 = call i32 @getint()
        %l1 = %t4
        %t5 = call i32 @getint()
```

```
        %l2 = %t5
        %t6 = call i32 @test(i32 %l1,i32 %l2)    ; sum = test(a, b)
        %l3 = %t6
        call void @putint(i32 %l3)
        %l0 = 0
        br label .L2
.L2:
        exit %l0
}
```

6.8.3 生成的 IR（使用 arg 指令）

```
define i32 @test(i32 %t0, i32 %t1) {
        declare i32 %l2 ; variable: a
        declare i32 %l3 ; variable: b
        declare i32 %l4
        declare i32 %l5 ; variable: c
        declare i32 %t6
.L1:
        entry
        %l2 = %t0
        %l3 = %t1
        %t6 = add %l2, %l3
        %l5 = %t6
        %l4 = %l5
        br label .L2
.L2:
        exit %l4
}
```

```
define i32 @main() {
        declare i32 %l0
        declare i32 %l1 ; variable: a
        declare i32 %l2 ; variable: b
        declare i32 %l3 ; variable: sum
        declare i32 %t4
        declare i32 %t5
        declare i32 %t6
.L1:
        entry
        %t4 = call i32 @getint()
        %l1 = %t4
        %t5 = call i32 @getint()
        %l2 = %t5
        arg %l1                    ; 自左往右传递参数，传递第一个实参
        arg %l2                    ; 传递第二个实参
        %t6 = call i32 @test()     ; 这里不写实参，只写调用
        %l3 = %t6
        arg %l3
        call void @putint()
        %l0 = 0
        br label .L2
.L2:
        exit %l0
}
```

6.9 一维数组的样例

6.9.1 源代码

```
int sum(int a[10], int n)
{
    int k, sum;

    k = 0;
    sum = 0;
    while(k < n) {
        sum = sum + a[k];
        k = k + 1;
    }

    return sum;
}

int main()
{
    int n;
    int a[10];

    n = getarray(a);

    int t;
    t = sum(a, n);

    return t;
```

```
}
```

6.9.2 生成的 IR

```
; 一维数组作为形参，忽略第一维，因此类型为 i32 [0]
define i32 @sum(i32 %t0[0], i32 %t1) {
        declare i32 %l2[0] ; variable: a          ; 一维数组对应的栈内局部变量
        declare i32 %l3 ; variable: n
        declare i32 %l4
        declare i32 %l5 ; variable: k
        declare i32 %l6 ; variable: sum
        declare i1 %t7
        declare i32* %t8
        declare i32 %t9
        declare i32 %t10
        declare i32 %t11
        declare i32 %t12
        declare i32 %t13
.L1:
        entry
        %l2 = %t0
        %l3 = %t1
        %l5 = 0
        %l6 = 0
.L3:                                    ; 循环入口
        %t7 = icmp lt %l5, %l3
        bc %t7, label .L4, label .L5
.L4:                                    ; 循环体入口
 ; 下面三行对应 a[k]的翻译，其值保存在%t11 中
        %t9 = mul %l5, 4
        %t8 = add %l2, %t9
        %t11 = *%t8
        %t12 = add %l6, %t11        ;%t12 = sum + %t11
        %l6 = %t12                   ;sum = %12
        %t13 = add %l5, 1           ; %t13 = k + 1
        %l5 = %t13                  ; k = %t13
        br label .L3                ; 跳转到循环的入口
.L5:                                ; 循环出口
        %l4 = %l6
        br label .L2
.L2:
```

```
        exit %l4
}
define i32 @main() {
        declare i32 %l0
        declare i32 %l1 ; variable: n
        declare i32 %l2[10] ; variable: a
        declare i32 %l3 ; variable: t
        declare i32 %t4
        declare i32 %t5
.L1:
        entry
            ; 一维数组实参传递

        %t4 = call i32 @getarray(i32 %l2[10])
        %l1 = %t4
         ; 一维数组和整型的实参传递

        %t5 = call i32 @sum(i32 %l2[10],i32 %l1)
        %l3 = %t5
        %l0 = %l3
        br label .L2
.L2:
        exit %l0
}
```

6.10 多维数组的样例

6.10.1 源代码

```
int getarray(int a[]);

void putint(int);


int test(int a[10][5]) {

    int sum = 0;

    int k, m;


    k = 0;

    while (k < 10) {
```

```
            m = 0;
            while (m < 5) {

                sum = sum + a[k][m];
                m = m + 1;
            }

            k = k + 1;
        }

        return sum;
}

int main() {
        int temp;
        int a[10][10][5];
        int m, k, n;

        k = 0;
        while (k < 10) {

            m = 0;
            while (m < 10) {

                n = 0;
                while (n < 5) {
```

```
                a[k][m][n] = m * k * n;
                n = n + 1;
            }

            m = m + 1;
        }

        k = k + 1;
    }

    k = 0;
    while (k < 10) {

        temp = test(a[k]);
        putint(temp);

        k = k + 1;
    }

    return 0;
}
```

6.10.2 生成的 IR

```
define i32 @test(i32 %t0[0][5]) {
    declare i32 %l1[0][5] ; variable: a
    declare i32 %l2
```

```
    declare i32 %l3 ; variable: sum
    declare i32 %l4 ; variable: k
    declare i32 %l5 ; variable: m
    declare i1 %t6
    declare i1 %t7
    declare i32 %t8
    declare i32 %t9
    declare i32* %t10
    declare i32 %t11
    declare i32 %t12
    declare i32 %t13
    declare i32 %t14
    declare i32 %t15
    declare i32 %t16
.L1:
    entry
    %l1 = %t0
    %l3 = 0
    %l4 = 0
.L3:
    %t6 = icmp lt %l4, 10
    bc %t6, label .L4, label .L5
.L4:
    %l5 = 0
.L6:
    %t7 = icmp lt %l5, 5
    bc %t7, label .L7, label .L8
```

```
.L7:
        %t8 = mul %l4, 5
        %t9 = add %t8, %l5
        %t11 = mul %t9, 4
        %t10 = add %l1, %t11
        %t13 = *%t10
        %t14 = add %l3, %t13
        %l3 = %t14
        %t15 = add %l5, 1
        %l5 = %t15
        br label .L6
.L8:
        %t16 = add %l4, 1
        %l4 = %t16
        br label .L3
.L5:
        %l2 = %l3
        br label .L2
.L2:
        exit %l2
}
define i32 @main() {
        declare i32 %l0
        declare i32 %l1 ; variable: temp
        declare i32 %l2[10][10][5] ; variable: a
        declare i32 %l3 ; variable: m
        declare i32 %l4 ; variable: k
```

```
    declare i32 %l5 ; variable: n
    declare i1 %t6
    declare i1 %t7
    declare i1 %t8
    declare i32 %t9
    declare i32 %t10
    declare i32 %t11
    declare i32 %t12
    declare i32* %t13
    declare i32 %t14
    declare i32 %t15
    declare i32 %t16
    declare i32 %t17
    declare i32 %t18
    declare i32 %t19
    declare i32 %t20
    declare i1 %t21
    declare i32 %t22
    declare i32 %t23[10][5]
    declare i32 %t24
    declare i32 %t25
    declare i32 %t26
.L1:
    entry
    %l4 = 0
.L3:
    %t6 = icmp lt %l4, 10
```

```
    bc %t6, label .L4, label .L5
.L4:
    %l3 = 0
.L6:
    %t7 = icmp lt %l3, 10
    bc %t7, label .L7, label .L8
.L7:
    %l5 = 0
.L9:
    %t8 = icmp lt %l5, 5
    bc %t8, label .L10, label .L11
.L10:
    ; a[k][m][n] = m * k * n，正确是先右侧的乘法后赋值给数组元素
    ; IR 虚拟机输出参考先左后右是错误的，需注意。
    %t16 = mul %l3, %l4
    %t17 = mul %t16, %l5
    %t9 = mul %l4, 10
    %t10 = add %t9, %l3
    %t11 = mul %t10, 5
    %t12 = add %t11, %l5
    %t14 = mul %t12, 4
    %t13 = add %l2, %t14
    *%t13 = %t17
    %t18 = add %l5, 1
    %l5 = %t18
    br label .L9
.L11:
```

```
        %t19 = add %l3, 1
        %l3 = %t19
        br label .L6
.L8:
        %t20 = add %l4, 1
        %l4 = %t20
        br label .L3
.L5:
        %l4 = 0
.L12:
        %t21 = icmp lt %l4, 10
        bc %t21, label .L13, label .L14
.L13:
        ; 下面五行是 temp = test(a[k])语句的翻译
        ; a[k]的类型是 i32 [10][5]
        %t22 = mul %l4, 50
        %t24 = mul %t22, 4
        %t23 = add %l2, %t24
        %t25 = call i32 @test(i32 %t23[10][5])
        %l1 = %t25
        call void @putint(i32 %l1)
        %t26 = add %l4, 1
        %l4 = %t26
        br label .L12
.L14:
        %l0 = 0
        br label .L2
```

```
.L2:
        exit %l0
}
```

6.11 函数 return 语句翻译样例

6.11.1 源代码

```
int main()
{
    int a;

    a = getint();

    if(a > 0) {
        return a + 1;
    } else if(a < 0) {
        return a - 1;
    }

    return 0;
}
```

6.11.2 生成的 IR

```
define i32 @main() {
        declare i32 %l0
        declare i32 %l1 ; variable: a
        declare i32 %t2
        declare i1 %t3
        declare i32 %t4
        declare i1 %t5
        declare i32 %t6
.L1:
        entry
        %t2 = call i32 @getint()
        %l1 = %t2
        %t3 = icmp gt %l1, 0
        bc %t3, label .L3, label .L4
.L3:
; return a + 1    先把 a+1 的结果保存到返回值局部变量，后跳转函数出口
        %t4 = add %l1, 1
```

```
        %l0 = %t4
        br label .L2
        br label .L5
.L4:
        %t5 = icmp lt %l1, 0
        bc %t5, label .L6, label .L7
.L6:
; return a - 1    先把 a-1 的结果保存到返回值局部变量，后跳转函数出口
        %t6 = sub %l1, 1
        %l0 = %t6
        br label .L2
.L7:
.L8:
        br label .L5
.L5:
; return 0  先把 0 保存到返回值局部变量，后跳转函数出口
        %l0 = 0
        br label .L2
;  函数出口和函数退出指令并带返回值
.L2:
        exit %l0
}
```