

浙江大学

本科实验报告

课程名称	编译原理
成员	陈 锰 (3170105197)
成员	席吉华 (3170105032)
	杨建伟 (3170101209)
学院	计算机科学与技术学院
系	软件工程专业
专业	软件工程专业
指导教师	王强

2020 年 5 月 31 日

序言

- 3.1 概述
- 3.2 开发环境
- 3.3 文件说明
- 3.4 组员分工

第二章 词法分析

- 2.1 Lex
- 2.2 正则表达式
- 2.3 具体实现
 - 2.3.1 定义区
 - 2.3.2 规则区

第三章 语法分析

- 3.1 Yacc
- 3.2 抽象语法树
 - 3.2.1 Node类
 - 3.2.2 Expression类和Statement类
 - 3.2.3 Program类
 - 3.2.4 Routine类
 - 3.2.5 Identifier类
 - 3.2.6 ConstValue类
 - 3.2.7 AstType类
 - 3.2.8 BinaryExpression类
 - 3.2.9 常量声明
 - 3.2.10 变量声明
 - 3.2.11 类型声明
 - 3.2.12 过程声明
 - 3.2.13 复合语句
 - 3.2.14 其他
- 3.3 语法分析的具体实现
- 3.4 抽象语法树可视化

第四章 语义分析

- 3.1 LLVM概述
- 3.2 LLVM IR
 - 3.2.1 IR布局
 - 3.2.2 IR上下文环境
 - 3.2.3 IR核心类
- 3.3 IR生成
 - 3.3.1 运行环境设计
 - 3.3.2 类型系统
 - 3.3.3 常量获取
 - 3.3.4 变量创建和存取
 - 3.3.5 标识符/数组引用
 - 3.3.6 二元操作
 - 3.3.7 赋值语句
 - 3.3.8 Program
 - 3.3.9 Routine
 - 3.3.10 常量/变量声明
 - 3.3.11 函数/过程声明
 - 3.3.12 函数/过程调用
 - 3.3.13 系统函数/过程
 - 3.3.14 分支语句
 - 3.3.14.1 if语句
 - 3.3.14.2 case语句
 - 3.3.15 循环语句
 - 3.3.15.1 for语句
 - 3.3.15.2 while语句
 - 3.3.15.3 repeat语句

3.3.16 goto语句

第四章 优化考虑

常量折叠

第五章 代码生成

5.1 选择目标机器

5.2 配置 Module

5.3 生成目标代码

第六章 测试案例

6.1 数据类型测试

6.1.1 内置类型测试

6.1.2 数组类型测试

6.2 运算测试

6.3 控制流测试

6.3.1 分支测试

6.3.2 循环测试

6.3.3 Goto测试

6.4 函数测试

6.4.1 简单函数测试

6.4.2 递归函数测试

6.4.3 引用传递测试

6.5 综合测试

6.5.1 测试用例1

6.5.2 测试用例2

6.5.3 测试用例3

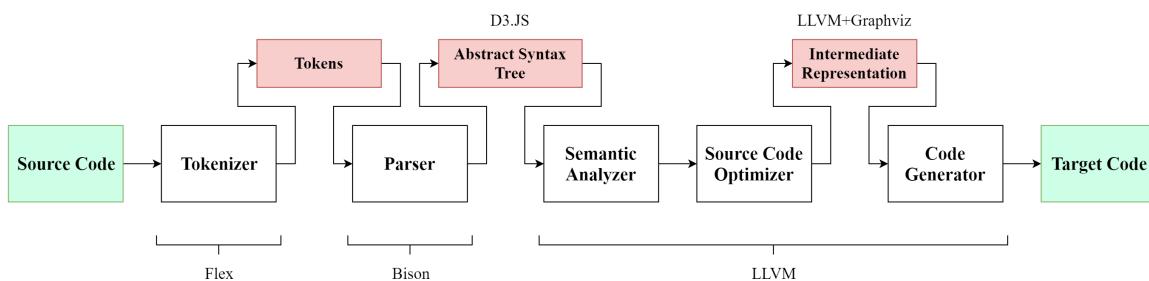
第七章 总结

序言

3.1 概述

本次实验小组基于C++语言设计并实现了一个SPL语言的编译系统，该系统以符合SPL语言规范的代码文本输入，输出为指定机器的目标代码。该SPL编译器的设计实现涵盖词法分析、语法分析、语义分析、优化考虑、代码生成等阶段和环节，所使用的技术包括但不限于：

- Flex实现词法分析
- Bison实现语法分析
- LLVM实现代码优化、中间代码生成、目标代码生成
- D3.js实现AST可视化
- LLVM+Graphviz实现CFG可视化



3.2 开发环境

- 操作系统：MacOS（推荐使用）或Linux
- 编译环境：
 - Flex 2.5.35 Apple(flex-32)
 - Bison 2.3 (GNU Bison)
 - LLVM 9.0.0
- 编辑器：XCode, Vim

3.3 文件说明

本次实验提交的文件及其说明如下：

- src: 源代码文件夹
 - spl.l: Flex源代码，主要实现词法分析，生成Token
 - spl.y: Yacc源代码，主要实现语法分析，生成抽象语法树
 - tokenizer.cpp: Flex根据spl.l生成的词法分析器
 - parser.hpp: Yacc根据spl.y生成的语法分析器头文件
 - parser.cpp: Yacc根据spl.y生成的语法分析器C++文件
 - ast.h: 抽象语法树头文件，定义所有AST节点类
 - ast.cpp: 抽象语法树实现文件，主要包含 codeGen 和 getJson 方法的实现
 - CodeGenerator.h: 中间代码生成器头文件，定义生成器环境
 - CodeGenerator.cpp: 中间代码生成器实现文件
 - main.cpp: 主函数所在文件，主要负责调用词法分析器、语法分析器、代码生成器
 - util.h: 项目的工具函数文件
 - Makefile: 定义编译链接规则
 - tree.json: 基于AST生成的JSON文件
 - tree.html: 可视化AST的网页文件

- spl: 编译器可执行程序
- doc: 报告文档文件夹
 - report.pdf: 报告文档
 - Slides.pdf: 展示文档
- test: 测试用例文件夹
 - testX.pas: SPL源程序测试用例

3.4 组员分工

组员	具体分工
杨建伟	词法分析, 语法分析, AST可视化
陈锰	语义分析, 中间代码生成
席吉华	运行环境设计, 目标代码生成

第壹章 词法分析

词法分析是计算机科学中将字符串序列转换为标记 (token) 序列的过程。在词法分析阶段，编译器读入源程序字符串流，将字符流转换为标记序列，同时将所需要的信息存储，然后将结果交给语法分析器。

2.1 Lex

SPL编译器的词法分析使用Lex (Flex) 完成，Lex是一个产生词法分析器的程序，是大多数UNIX系统的词法分析器产生程序。

Lex读入lex文件中定义的词法分析规则，输出C语言词法分析器源码。

标准lex文件由三部分组成，分别是定义区、规则区和用户子过程区。在定义区，用户可以编写C语言中的声明语句，导入需要的头文件或声明变量。在规则区，用户需要编写以正则表达式和对应的动作的形式的代码。在用户子过程区，用户可以定义函数。

```

1 {definitions}
2 %%
3 {rules}
4 %%
5 {user subroutines}
```

2.2 正则表达式

正则表达式是通过单个字符串描述，匹配一系列符合某个句法规则的字符串。在实际应用中，常用到的语法规则如下（摘录自[维基百科](#)）

字符	描述
\	将下一个字符标记为一个特殊字符 (File Format Escape, 清单见本表) 、或一个原义字符 (Identity Escape, 有 <code>^\$()*+?.[{} </code> 共计12个)、或一个向后引用 (backreferences) 、或一个八进制转义符。例如，“ <code>n</code> ”匹配字符“ <code>n</code> ”。“ <code>\n</code> ”匹配一个换行符。序列“ <code>\\"</code> ”匹配“ <code>\</code> ”而“ <code>\C</code> ”则匹配“ <code>C</code> ”。
^	匹配输入字符串的开始位置。如果设置了RegExp对象的multiline属性，^也匹配“ <code>\n</code> ”或“ <code>\r</code> ”之后的位置。
\$	匹配输入字符串的结束位置。如果设置了RegExp对象的multiline属性，\$也匹配“ <code>\n</code> ”或“ <code>\r</code> ”之前的位置。
*	匹配前面的子表达式零次或多次。例如， <code>zo</code> 能匹配“ <code>z</code> ”、“ <code>zo</code> ”以及“ <code>zoo</code> ”。等价于 <code>{0,}</code> 。
+	匹配前面的子表达式一次或多次。例如，“ <code>zo+</code> ”能匹配“ <code>zo</code> ”以及“ <code>zoo</code> ”，但不能匹配“ <code>z</code> ”。+等价于 <code>{1,}</code> 。
?	匹配前面的子表达式零次或一次。例如，“ <code>do(es)?</code> ”可以匹配“ <code>does</code> ”中的“ <code>do</code> ”和“ <code>does</code> ”。?等价于 <code>{0,1}</code> 。
{n}	n 是一个非负整数。匹配确定的 n 次。例如，“ <code>o{2}</code> ”不能匹配“ <code>Bob</code> ”中的“ <code>o</code> ”，但是能匹配“ <code>food</code> ”中的两个 <code>o</code> 。
{n,}	n 是一个非负整数。至少匹配 n 次。例如，“ <code>o{2,}</code> ”不能匹配“ <code>Bob</code> ”中的“ <code>o</code> ”，但能匹配“ <code>foooooood</code> ”中的所有 <code>o</code> 。“ <code>o{1,}</code> ”等价于“ <code>o+</code> ”。“ <code>o{0,}</code> ”则等价于“ <code>o*</code> ”。
{n,m}	m 和 n 均为非负整数，其中 $n \leq m$ 。最少匹配 n 次且最多匹配 m 次。例如，“ <code>o{1,3}</code> ”将匹配“ <code>foooooood</code> ”中的前三个 <code>o</code> 。“ <code>o{0,1}</code> ”等价于“ <code>o?</code> ”。请注意在逗号和两个数之间不能有空格。
?	非贪心量化 (Non-greedy quantifiers)：当该字符紧跟在任何一个其他重复修饰符 <code>(,+?, {n}, {n,}, {n,m*})</code> 后面时，匹配模式是 非贪婪的 。非贪婪模式尽可能少的匹配所搜索的字符串，而默认的贪婪模式则尽可能多的匹配所搜索的字符串。例如，对于字符串“ <code>oooo</code> ”，“ <code>o+?</code> ”将匹配单个“ <code>o</code> ”，而“ <code>o+?</code> ”将匹配所有“ <code>o</code> ”。
.	匹配除“ <code>\r</code> ”“ <code>\n</code> ”之外的任何单个字符。要匹配包括“ <code>\r</code> ”“ <code>\n</code> ”在内的任何字符，请使用像“ <code>(. \\r \\n)</code> ”的模式。
x y	没有包围在()里，其范围是整个正则表达式。例如，“ <code>z food</code> ”能匹配“ <code>z</code> ”或“ <code>food</code> ”。“ <code>(?:z f)oold</code> ”则匹配“ <code>zood</code> ”或“ <code>food</code> ”。

字符	描述
[xyz]	字符集合 (character class)。匹配所包含的任意一个字符。例如，“[abc]”可以匹配“plain”中的“a”。特殊字符仅有反斜线\保持特殊含义，用于转义字符。其它特殊字符如星号、加号、各种括号等均作为普通字符。脱字符^如果出现在首位则表示负值字符集合；如果出现在字符串中间就仅作为普通字符。连字符 - 如果出现在字符串中间表示字符范围描述；如果如果出现在首位（或末尾）则仅作为普通字符。右方括号应转义出现，也可以作为首位字符出现。
[^xyz]	排除型字符集合 (negated character classes)。匹配未列出的任意字符。例如，“[^abc]”可以匹配“plain”中的“plin”。
[a-z]	字符范围。匹配指定范围内的任意字符。例如，“[a-z]”可以匹配“a”到“z”范围内的任意小写字母字符。
[^a-z]	排除型的字符范围。匹配任何不在指定范围内的任意字符。例如，“[^a-z]”可以匹配任何不在“a”到“z”范围内的任意字符。
\d	匹配一个数字字符。等价于[0-9]。注意Unicode正则表达式会匹配全角数字字符。
\D	匹配一个非数字字符。等价于[^0-9]。
\n	匹配一个换行符。等价于\x0a和\cJ。
\r	匹配一个回车符。等价于\x0d和\cM。
\s	匹配任何空白字符，包括空格、制表符、换页符等等。等价于[\f\n\r\t\v]。注意Unicode正则表达式会匹配全角空格符。
\S	匹配任何非空白字符。等价于[^ \f\n\r\t\v]。
\w	匹配包括下划线的任何单词字符。等价于“[A-Za-z0-9_]”。注意Unicode正则表达式会匹配中文字符。
\W	匹配任何非单词字符。等价于“[^A-Za-z0-9_]”。

2.3 具体实现

2.3.1 定义区

SPL的Lex源程序在定义区导入了需要的头文件，包括ast.h（抽象语法树头文件）、parser.hpp（yacc生成的词法分析器头文件）、stdio.h（C语言标准输入输出头文件）、string（C++ std::string头文件），然后声明了lex需要的yywrap函数。

```

1  %{
2  #include "ast.h"
3  #include "parser.hpp"
4  #include <stdio.h>
5  #include <string>
6  extern "C" int yywrap() { }
7  %}

```

2.3.2 规则区

首先，需要排除空格、换行和回车的干扰，方法是把他们解析为；

1 | [\t\n]

{ ; }

然后解析关键字、运算符和界符，由于运算符与界符是固定的，所以正则表达式只需要也是固定字符。

```

1  "("                      { return LP; }
2  ")"                      { return RP; }
3  "["                      { return LB; }
4  "]"                      { return RB; }
5  "...                     { return DOTDOT; }
6  "."                      { return DOT; }
7  ","                      { return COMMA; }
8  ":"                      { return COLON; }
9  "*"                      { return MUL; }
10 "/*"                     { return DIV; }
11 "<>"                   { return UNEQUAL; }
12 "not"                    { return NOT; }
13 "+"                      { return PLUS; }
14 "-_"                     { return MINUS; }
15 ">="                     { return GE; }
16 ">"                      { return GT; }
17 "<="                     { return LE; }
18 "<"                      { return LT; }
19 "="                      { return EQUAL; }
20 ":="                     { return ASSIGN; }
21 "mod"                    { return MOD; }
22 ";"                      { return SEMI; }
23 "and"                    { return AND; }
24 "array"                  { return ARRAY; }
25 "begin"                  { return TOKEN_BEGIN; }

26 "case"                   { return CASE; }
27 "const"                  { return CONST; }
28 "div"                    { return DIV; }
29 "do"                     { return DO; }
30 "downto"                 { return DOWNTO; }
31 "else"                   { return ELSE; }
32 "end"                    { return END; }
33 "for"                    { return FOR; }
34 "function"               { return FUNCTION; }
35 "goto"                   { return GOTO; }
36 "if"                     { return IF; }
37 "of"                     { return OF; }
38 "or"                     { return OR; }
39 "procedure"               { return PROCEDURE; }
40 "program"                { return PROGRAM; }
41 "record"                 { return RECORD; }
42 "repeat"                 { return REPEAT; }
43 "then"                   { return THEN; }
44 "to"                     { return TO; }
45 "type"                   { return TYPE; }
46 "until"                  { return UNTIL; }
47 "var"                    { return VAR; }
48 "while"                  { return WHILE; }

```

标识符是由字母或下划线开头，由字母、数字和下划线组成的字符串，并且不能是关键字、SYS_CON、SYS_FUNCT、SYS_PROC、SYS_TYPE之外的ID。SPL编译器在词法分析阶段只校验是否符合标识符规则，而不校验是否存在。不同于运算符，标识符需要额外保存字符串值。

```
1 [a-zA-Z_][a-zA-Z0-9_]* { yyval.sval = new
2 std::string(yytext, yylen); return
3 IDENTIFIER; }
```

然后其他需要额外保存值的单词：

```
1 "boolean"|"char"|"integer"|"real" { yyval.sval =
2 new std::string(yytext, yylen); return
3 SYS_TYPE; }
4 "abs"|"chr"|"odd"|"ord"|"pred"|"sqr"|"sqrt"|"succ"
5 { yyval.sval =
6 new std::string(yytext, yylen); return
7 SYS_FUNCT; }
8 "false"|"maxint"|"true"
9 { yyval.sval =
10 new std::string(yytext, yylen); return SYS_CON;
11 }
12 "write"|"writeln"
13 { yyval.sval =
14 new std::string(yytext, yylen); return
15 SYS_PROC; }
16 "read"
17 { yyval.sval =
18 new std::string(yytext, yylen); return READ;
19 }
20 "[0-9]+\. [0-9]+ { double dtmp;
21 sscanf(yytext,
22 "%lf", &dtmp);
23 yyval.dval =
24 dtmp; return REAL;
25 }
26 "[0-9]+ { int itmp;
27 double tmp;
28 sscanf(yytext,
29 "%d", &itmp); }
```

```

30     itmp;
31
32     }
33     \\
34     yytext[1];
35
36     yyval.ival =
37         return INTEGER;
38     }
39     {
40         yyval.cval =
41             return CHAR;
42     }

```

在系统函数、类型等系统单词（以SYS开头）中，词法分析器需要记录字符串串值以使得语法分析器能区分是哪个函数或类型。

对于整型、浮点型，在词法分析阶段使用C语言转换为对应类型存储。

字符型使用以'开头结尾，中间为任意字符的正则表达式识别，将中间字符存储。

第3章 语法分析

在计算机科学和语言学中，语法分析是根据某种给定的形式文法对由单词序列（如英语单词序列）构成的输入文本进行分析并确定其语法结构的一种过程。在词法分析阶段，编译器接收词法分析器发送的标记序列，最终输出抽象语法树数据结构。

3.1 Yacc

SPL编译器的语法分析使用Yacc（Bison）完成。Yacc是Unix/Linux上一个用来生成编译器的编译器（编译器代码生成器）。Yacc生成的编译器主要是用C语言写成的语法解析器（Parser），需要与词法解析器Lex一起使用，再把两部分产生出来的C程序一并编译。

与Lex相似，Yacc的输入文件由以%%分割的三部分组成，分别是声明区、规则区和程序区。三部分的功能与Lex相似，不同的是规则区的正则表达式替换为CFG，在声明区要提前声明好使用到的终结符以及非终结符的类型。

```

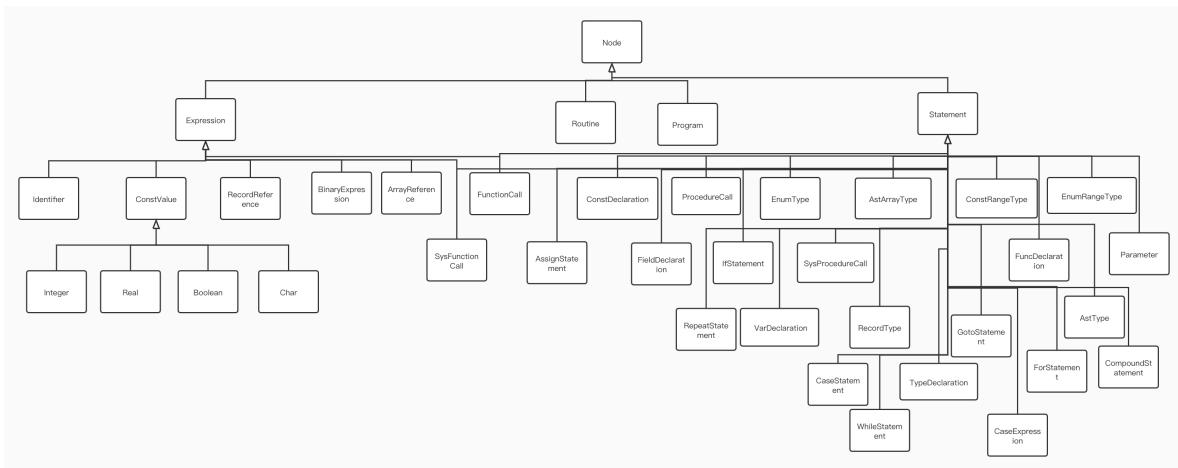
1 declarations
2 %%
3 rules
4 %%
5 programs

```

3.2 抽象语法树

语法分析器的输出是抽象语法树。在计算机科学中，抽象语法树是源代码语法结构的一种抽象表示。它以树状的形式表现编程语言的语法结构，树上的每个节点都表示源代码中的一种结构。之所以说语法是“抽象”的，是因为这里的语法并不会表示出真实语法中出现的每个细节。比如，嵌套括号被隐含在树的结构中，并没有以节点的形式呈现；而类似于if-condition-then这样的条件跳转语句，可以使用带有三个分支的节点来表示。

SPL编译器的抽象语法树利用面向对象的设计思想进行抽象，下面是语法树的继承体系：



3.2.1 Node类

Node类是一个抽象类，其意义为"抽象语法树的节点"，这是抽象语法树所有节点（在下文简称AST）的共同祖先。该类拥有两个纯虚函数，分别是codeGen和getJson，分别用于生成中间代码和生成AST可视化需要的Json数据。

```

1 class Node
2 {
3     public:
4         virtual llvm::value *codeGen(CodeGenerator & generator) = 0;
5         virtual string toJson(){return "";};
6 };
  
```

3.2.2 Expression类和Statement类

Expression和Statement是大部分实体类的父类。Expression的语义是表达式类，它的子类的特征是可获得值或可更改值，也就是左值或者右值，比如二元表达式或变量。Statement类的语义是语句类，它的子类的特征是该类会进行操作，比如赋值、比较、条件控制等。

另外，Statement类具有一个label属性，其意义是Goto语句的跳转标志。

```

1 // 表达式，特征是能返回值或能存储值
2 class Expression : public Node
3 {
4 };
5
6 // 语句，特征是能完成某些操作
7 class Statement : public Node
8 {
9     public:
10        // 用于Goto语句设置标号
11        void setLabel(int label)
12        // 用于得到Goto语句需要的标号，若不存在则返回-1
13        int getLabel()
14     private:
15        int label = -1;
16 };
  
```

3.2.3 Program类

Program类的意义是程序，该类是最顶层的实体类，包括程序名和Routine类对象

```

1 class Program : public Node {
2 public:
3     Program(string *programID, Routine *routine) : programID(programID),
4             routine(routine) { }
5     virtual llvm::Value *CodeGen(CodeGenerator & generator) override;
6     virtual string getJson() override;
7 private:
8     string *programID;
9     Routine *routine;
10 };

```

3.2.4 Routine类

Routine类的意义是一个过程，该类用在Program和Function类中。Routine类包括常量声明、类型声明、变量声明、函数声明和语句部分。

```

1 class Routine : public Node {
2 public:
3     Routine(ConstDeclList *cd, TypeDeclList *tp, VarDeclList *vd,
4             RoutineList *rl)
5         void setRoutineBody(CompoundStatement *routineBody) { this->routineBody
= routineBody; }
6         virtual llvm::Value *CodeGen(CodeGenerator & generator) override;
7         virtual string getJson() override;
8         void setGlobal()
9 private:
10     ConstDeclList *constDeclList;
11     VarDeclList *varDeclList;
12     TypeDeclList *typeDeclList;
13     RoutineList *routineList;
14     CompoundStatement *routineBody;
15 };

```

3.2.5 Identifier类

Identifier的意义是标识符，包括一个name字段。该类是实体类，实现了代码生成和json获取函数。

```

1 class Identifier : public Expression
2 {
3 public:
4     Identifier(string *name) : name(name)
5     string getName()
6     virtual string getJson() override;
7     virtual llvm::Value *CodeGen(CodeGenerator & generator) override;
8 private:
9     string *name;
10 };

```

3.2.6 ConstValue类

该类的意义是常量节点，由于常量的类型很多，所以ConstValue是一个抽象类，具体由Integer、Real、Boolean、Char四个子类完成，通过Union和getType函数获得真实的值。

```

1 class ConstValue : public Expression
2 {

```

```

3 public:
4     union value {
5         int i;
6         double r;
7         bool b;
8         char c;
9     };
10    virtual BuildInType getType() = 0;
11    virtual ConstValue::value getValue() = 0;
12    virtual ConstValue *operator-() = 0;
13    virtual bool isValidConstRangeType()
14    {
15        BuildInType t = getType();
16        return t == SPL_INTEGER || t == SPL_CHAR;
17    }
18 };

```

3.2.7 AstType类

该类的意义是SPL支持的类型，包括数组、记录、枚举、常量范围、枚举范围、内置类型和用户自定义类型。实现方法是该类集成以上各个类对象，通过TypeOfTypeEnum确定AstType的真实类型。

```

1 // 类型
2 class AstType : public Statement {
3 public:
4     enum TypeOfTypeEnum {
5         SPL_ARRAY,
6         SPL_RECORD,
7         SPL_ENUM,
8         SPL_CONST_RANGE,
9         SPL_ENUM_RANGE,
10        SPL_BUILD_IN,
11        SPL_USER_DEFINE,
12        SPL_VOID
13    };
14    AstArrayType *arrayType;
15    RecordType *recordType;
16    EnumType *enumType;
17    ConstRangeType *constRangeType;
18    EnumRangeType *enumRangeType;
19    BuildInType buildInType;
20    Identifier *userDefineType;
21    TypeOfTypeEnum type;
22 };

```

3.2.8 BinaryExpression类

该类的意义是二元表达式，节点存储有左表达式、右表达式和操作符

```

1 class BinaryExpression : public Expression {
2 public:
3     enum BinaryOperator {
4         SPL_PLUS,
5         SPL_MINUS,
6         SPL_MUL,
7         SPL_DIV,

```

```

8     SPL_GE,
9     SPL_GT,
10    SPL_LT,
11    SPL_LE,
12    SPL_EQUAL,
13    SPL_UNEQUAL,
14    SPL_OR,
15    SPL_MOD,
16    SPL_AND,
17    SPL_XOR,
18  };
19  BinaryExpression(Expression *lhs, BinaryOperator op, Expression *rhs) :
20  lhs(lhs), op(op), rhs(rhs) { }
21 private:
22   vector<string> opString{"+", "-", "*", "/", ">=", ">", "<", "<=", "==" ,
23   "!=" , "or" , "mod" , "and" , "xor"};
24   Expression *lhs;
25   Expression *rhs;
26   BinaryOperator op;
27 };

```

3.2.9 常量声明

常量声明的顶层类是ConstDeclaration，由变量名(Identifier)、常量值(ConstValue)、变量类型(AstType)三部分构成。

```

1 // 常量声明语句
2 class ConstDeclaration : public Statement
3 {
4 public:
5   ConstDeclaration(Identifier *ip, ConstValue *cp) : name(ip), value(cp),
6   globalFlag(false)
7   {
8   }
9   virtual llvm::Value *CodeGen(CodeGenerator & generator) override;
10  virtual string getJson() override;
11  void setGlobal()
12  bool isGlobal()
13 private:
14   Identifier *name;
15   ConstValue *value;
16   AstType *type;
17   bool globalFlag;
18 };

```

3.2.10 变量声明

变量声明的顶层类是VarDeclaration，由变量名列表(Vector<Identifier>)、变量类型(AstType)构成。

```

1 class VarDeclaration : public Statement {
2 public:
3     VarDeclaration(NameList *nl, AstType *td) : nameList(nl), type(td),
4     globalFlag(false) {}
5     virtual llvm::Value *CodeGen(CodeGenerator & generator) override;
6     virtual string getJson() override;
7     void setGlobal()
8     bool isGlobal()
9 private:
10    NameList *nameList;
11    AstType *type;
12    bool globalFlag;
13 };

```

3.2.11 类型声明

类型声明的顶层类是TypeDeclaration，由类型名(Identifier)、变量类型(AstType)三部分构成。

```

1 class TypeDeclaration : public Statement {
2 public:
3     virtual llvm::Value *CodeGen(CodeGenerator & generator) override;
4     TypeDeclaration(Identifier *name, AstType *type) : name(name), type(type)
5     virtual string getJson() override;
6 private:
7     Identifier *name;
8     AstType *type;
9 };

```

3.2.12 过程声明

过程声明的顶层类是FuncDeclaration，由类型名(Identifier)、参数类型列表(vector<Parameter>)、返回值类型(AstType)、子过程(Routine)构成。过程声明在SPL的语义中会区分是函数 (Function) 还是过程 (Procedure) 。在AST中，通过返回值类型指针是否为空来区分过程与函数。

```

1 class FuncDeclaration : public Statement {
2 public:
3     FuncDeclaration(Identifier *name, ParaList *paraList, AstType
4     *returnType);
5     FuncDeclaration(Identifier *name, ParaList *paraList);
6     virtual llvm::Value *CodeGen(CodeGenerator & generator) override;
7     void setRoutine(Routine *routine)
8     virtual string getJson() override;
9 private:
10    Identifier *name;
11    ParaList *paraList;
12    AstType *returnType;
13    Routine *subRoutine;
14 };

```

3.2.13 复合语句

复合语句的意义是语句列表，由一系列语句组成。该类由StatementList组成。

```

1 class Compoundstatement : public Statement {
2 public:
3     CompoundStatement(StatementList *stmtList) : stmtList(stmtList) { }
4     virtual llvm::Value *CodeGen(CodeGenerator & generator) override;
5
6     virtual string getJson() override;
7 private:
8     StatementList *stmtList;
9 };

```

3.2.14 其他

上面介绍顶层类和重要的底层类，其他的类大同小异，都是将语法树中需要的东西保存起来。能求值的继承自Expression类，例如BinaryExpression、ArrayReferenc等；能独立成句的继承自Statement类，如IfStatement、WhileStatement等；又能求值又独立成句的同时继承自Expression类和Statement类，如FunctionCall等。

3.3 语法分析的具体实现

首先在声明区声明好终结符和非终结符类型

```

1 %token LP RP LB RB DOT COMMA COLON
2           MUL UNEQUAL NOT PLUS MINUS
3           GE GT LE LT EQUAL ASSIGN MOD DOTDOT
4           SEMI
5           AND ARRAY TOKEN_BEGIN CASE CONST
6           DIV DO DOWNTO ELSE END
7           FOR FUNCTION GOTO
8           IF OF OR
9           PROCEDURE PROGRAM RECORD REPEAT
10          THEN TO TYPE UNTIL VAR WHILE
11 %token<iVal> INTEGER
12 %token<sVal> IDENTIFIER SYS_CON SYS_FUNCT SYS_PROC SYS_TYPE READ
13 %token<dVal> REAL
14 %token<cVal> CHAR
15
16 %type<identifier> name
17 %type<program> program
18 %type<sVal> program_head
19 %type<routine> routine routine_head sub_routine
20 %type<constDeclList> const_part const_expr_list
21 %type<typeDeclList> type_part type_decl_list
22 %type<typeDeclaration> type_definition
23 %type<varDeclList> var_part var_decl_list
24 %type<varDeclaration> var_decl
25 %type<routineList> routine_part
26 %type<constValue> const_value
27 %type<type> type_decl simple_type_decl
28 %type<nameList> array_type_decl record_type_decl
29 %type<fieldList> name_list
30 %type<fieldDeclaration> field_decl_list
31 %type<funcDeclaration> field_decl
32 %type<paraList> function_decl procedure_decl
33 %type<parameters> parameters para_decl_list

```

```

33 %type<parameter>           para_type_list var_para_list
34 val_para_list
35 %type<statement>          stmt non_label_stmt else_clause
36 %type<assignStatement>    assign_stmt
37 %type<statement>          proc_stmt
38 %type<expressionList>     expression_list
39 %type<expression>          expression expr term factor
40 %type<argsList>            args_list
41 %type<ifStatement>         if_stmt
42 %type<repeatStatement>    repeat_stmt
43 %type<whileStatement>     while_stmt
44 %type<forStatement>        for_stmt
45 %type<bval>               direction
46 %type<caseStatement>      case_stmt
47 %type<caseExprList>       case_expr_list
48 %type<caseExpression>     case_expr
49 %type<gotoStatement>      goto_stmt
50 %type<statementList>      stmt_list
51 %type<compoundStatement>  routine_body compound_stmt

```

接着按从下往上的顺序构造语法树，部分文法如下：

```

1 %start program
2 %%
3 //
4 name: IDENTIFIER           { $$ = new
5 Identifier($1); }
6 ;
7
7 program: program_head routine DOT           { $$ = new
8 Program($1, $2); root = $$; }
9 ;
10
10 program_head: PROGRAM IDENTIFIER SEMI      { $$ = $2; }
11 ;
12
13 routine: routine_head routine_body         { $$ = $1; $$-
14 >setRoutineBody($2); }
15 ;
16
16 routine_head: const_part type_part var_part routine_part { $$ = new
17 Routine($1, $2, $3, $4); }
18 ;
19
19 const_part :
20   CONST const_expr_list           { $$ = $2; }
21   |
22 ConstDeclList(); }
23 ;
23 const_expr_list :
24   const_expr_list name EQUAL const_value SEMI      { $$ = $1; $$-
25 >push_back(new ConstDeclaration($2, $4)); }
26   | name EQUAL const_value SEMI           { $$ = new
27 ConstDeclList(); $$->push_back(new ConstDeclaration($1, $3)); }
28 ;
28 const_value :

```

```

28     INTEGER
29     | REAL
30     | CHAR
31     | SYS_CON
32
33     "true")
34
35     new Boolean(true);
36
37     == "false")
38
39     new Boolean(false);
40
41     type_part :
42         TYPE type_decl_list
43         |
44         TypeDeclList();
45
46     type_decl_list :
47         type_decl_list type_definition
48         >push_back($2);
49         |
50         type_definition
51         TypeDeclList(); $$->push_back($1);
52
53     type_definition :
54         name EQUAL type_decl SEMI
55         TypeDeclaration($1, $3);
56
57     type_decl :
58         simple_type_decl
59         | array_type_decl
60         | record_type_decl
61
62     simple_type_decl :
63         SYS_TYPE
64
65         "integer")
66
67         new AstType(SPL_INTEGER);
68
69         == "boolean")
70
71         new AstType(SPL_BOOLEAN);
72
73         == "real")
74
75         new AstType(SPL_REAL);
76
77     { $$ = new
78     | $$ = new
79     | $$ = new
80     {
81         if(*$1 ==
82             $$ =
83
84         else if(*$1
85             $$ =
86
87         else
88             $$ =
89
90     }
91
92     ;
93
94
95     type_part :
96         TYPE type_decl_list
97         |
98         TypeDeclList();
99
100    type_decl_list :
101        type_decl_list type_definition
102        >push_back($2);
103        |
104        type_definition
105        TypeDeclList(); $$->push_back($1);
106
107    type_definition :
108        name EQUAL type_decl SEMI
109        TypeDeclaration($1, $3);
110
111    type_decl :
112        simple_type_decl
113        | array_type_decl
114        | record_type_decl
115
116    simple_type_decl :
117        SYS_TYPE
118
119        "integer")
120
121        new AstType(SPL_INTEGER);
122
123        == "boolean")
124
125        new AstType(SPL_BOOLEAN);
126
127        == "real")
128
129        new AstType(SPL_REAL);
130
131    { $$ = new
132    | $$ = new
133    | $$ = new
134    {
135        if(*$1 ==
136            $$ =
137
138        else if(*$1
139            $$ =
140
141        else
142            $$ =
143
144    }
145
146    ;
147
148
149    type_part :
150        TYPE type_decl_list
151        |
152        TypeDeclList();
153
154    type_decl_list :
155        type_decl_list type_definition
156        >push_back($2);
157        |
158        type_definition
159        TypeDeclList(); $$->push_back($1);
160
161    type_definition :
162        name EQUAL type_decl SEMI
163        TypeDeclaration($1, $3);
164
165    type_decl :
166        simple_type_decl
167        | array_type_decl
168        | record_type_decl
169
170    simple_type_decl :
171        SYS_TYPE
172
173        "integer")
174
175        new AstType(SPL_INTEGER);
176
177        == "boolean")
178
179        new AstType(SPL_BOOLEAN);
180
181        == "real")
182
183        new AstType(SPL_REAL);
184
185    { $$ = new
186    | $$ = new
187    | $$ = new
188    {
189        if(*$1 ==
190            $$ =
191
192        else if(*$1
193            $$ =
194
195        else
196            $$ =
197
198    }
199
200    ;
201
202
203    type_part :
204        TYPE type_decl_list
205        |
206        TypeDeclList();
207
208    type_decl_list :
209        type_decl_list type_definition
210        >push_back($2);
211        |
212        type_definition
213        TypeDeclList(); $$->push_back($1);
214
215    type_definition :
216        name EQUAL type_decl SEMI
217        TypeDeclaration($1, $3);
218
219    type_decl :
220        simple_type_decl
221        | array_type_decl
222        | record_type_decl
223
224    simple_type_decl :
225        SYS_TYPE
226
227        "integer")
228
229        new AstType(SPL_INTEGER);
230
231        == "boolean")
232
233        new AstType(SPL_BOOLEAN);
234
235        == "real")
236
237        new AstType(SPL_REAL);
238
239    { $$ = new
240    | $$ = new
241    | $$ = new
242    {
243        if(*$1 ==
244            $$ =
245
246        else if(*$1
247            $$ =
248
249        else
250            $$ =
251
252    }
253
254    ;
255
256
257    type_part :
258        TYPE type_decl_list
259        |
260        TypeDeclList();
261
262    type_decl_list :
263        type_decl_list type_definition
264        >push_back($2);
265        |
266        type_definition
267        TypeDeclList(); $$->push_back($1);
268
269    type_definition :
270        name EQUAL type_decl SEMI
271        TypeDeclaration($1, $3);
272
273    type_decl :
274        simple_type_decl
275        | array_type_decl
276        | record_type_decl
277
278    simple_type_decl :
279        SYS_TYPE
280
281        "integer")
282
283        new AstType(SPL_INTEGER);
284
285        == "boolean")
286
287        new AstType(SPL_BOOLEAN);
288
289        == "real")
290
291        new AstType(SPL_REAL);
292
293    { $$ = new
294    | $$ = new
295    | $$ = new
296    {
297        if(*$1 ==
298            $$ =
299
300        else if(*$1
301            $$ =
302
303        else
304            $$ =
305
306    }
307
308    ;
309
310
311    type_part :
312        TYPE type_decl_list
313        |
314        TypeDeclList();
315
316    type_decl_list :
317        type_decl_list type_definition
318        >push_back($2);
319        |
320        type_definition
321        TypeDeclList(); $$->push_back($1);
322
323    type_definition :
324        name EQUAL type_decl SEMI
325        TypeDeclaration($1, $3);
326
327    type_decl :
328        simple_type_decl
329        | array_type_decl
330        | record_type_decl
331
332    simple_type_decl :
333        SYS_TYPE
334
335        "integer")
336
337        new AstType(SPL_INTEGER);
338
339        == "boolean")
340
341        new AstType(SPL_BOOLEAN);
342
343        == "real")
344
345        new AstType(SPL_REAL);
346
347    { $$ = new
348    | $$ = new
349    | $$ = new
350    {
351        if(*$1 ==
352            $$ =
353
354        else if(*$1
355            $$ =
356
357        else
358            $$ =
359
360    }
361
362    ;
363
364
365    type_part :
366        TYPE type_decl_list
367        |
368        TypeDeclList();
369
370    type_decl_list :
371        type_decl_list type_definition
372        >push_back($2);
373        |
374        type_definition
375        TypeDeclList(); $$->push_back($1);
376
377    type_definition :
378        name EQUAL type_decl SEMI
379        TypeDeclaration($1, $3);
380
381    type_decl :
382        simple_type_decl
383        | array_type_decl
384        | record_type_decl
385
386    simple_type_decl :
387        SYS_TYPE
388
389        "integer")
390
391        new AstType(SPL_INTEGER);
392
393        == "boolean")
394
395        new AstType(SPL_BOOLEAN);
396
397        == "real")
398
399        new AstType(SPL_REAL);
400
401    { $$ = new
402    | $$ = new
403    | $$ = new
404    {
405        if(*$1 ==
406            $$ =
407
408        else if(*$1
409            $$ =
410
411        else
412            $$ =
413
414    }
415
416    ;
417
418
419    type_part :
420        TYPE type_decl_list
421        |
422        TypeDeclList();
423
424    type_decl_list :
425        type_decl_list type_definition
426        >push_back($2);
427        |
428        type_definition
429        TypeDeclList(); $$->push_back($1);
430
431    type_definition :
432        name EQUAL type_decl SEMI
433        TypeDeclaration($1, $3);
434
435    type_decl :
436        simple_type_decl
437        | array_type_decl
438        | record_type_decl
439
440    simple_type_decl :
441        SYS_TYPE
442
443        "integer")
444
445        new AstType(SPL_INTEGER);
446
447        == "boolean")
448
449        new AstType(SPL_BOOLEAN);
450
451        == "real")
452
453        new AstType(SPL_REAL);
454
455    { $$ = new
456    | $$ = new
457    | $$ = new
458    {
459        if(*$1 ==
460            $$ =
461
462        else if(*$1
463            $$ =
464
465        else
466            $$ =
467
468    }
469
470    ;
471
472
473    type_part :
474        TYPE type_decl_list
475        |
476        TypeDeclList();
477
478    type_decl_list :
479        type_decl_list type_definition
480        >push_back($2);
481        |
482        type_definition
483        TypeDeclList(); $$->push_back($1);
484
485    type_definition :
486        name EQUAL type_decl SEMI
487        TypeDeclaration($1, $3);
488
489    type_decl :
490        simple_type_decl
491        | array_type_decl
492        | record_type_decl
493
494    simple_type_decl :
495        SYS_TYPE
496
497        "integer")
498
499        new AstType(SPL_INTEGER);
500
501        == "boolean")
502
503        new AstType(SPL_BOOLEAN);
504
505        == "real")
506
507        new AstType(SPL_REAL);
508
509    { $$ = new
510    | $$ = new
511    | $$ = new
512    {
513        if(*$1 ==
514            $$ =
515
516        else if(*$1
517            $$ =
518
519        else
520            $$ =
521
522    }
523
524    ;
525
526
527    type_part :
528        TYPE type_decl_list
529        |
530        TypeDeclList();
531
532    type_decl_list :
533        type_decl_list type_definition
534        >push_back($2);
535        |
536        type_definition
537        TypeDeclList(); $$->push_back($1);
538
539    type_definition :
540        name EQUAL type_decl SEMI
541        TypeDeclaration($1, $3);
542
543    type_decl :
544        simple_type_decl
545        | array_type_decl
546        | record_type_decl
547
548    simple_type_decl :
549        SYS_TYPE
550
551        "integer")
552
553        new AstType(SPL_INTEGER);
554
555        == "boolean")
556
557        new AstType(SPL_BOOLEAN);
558
559        == "real")
560
561        new AstType(SPL_REAL);
562
563    { $$ = new
564    | $$ = new
565    | $$ = new
566    {
567        if(*$1 ==
568            $$ =
569
570        else if(*$1
571            $$ =
572
573        else
574            $$ =
575
576    }
577
578    ;
579
580
581    type_part :
582        TYPE type_decl_list
583        |
584        TypeDeclList();
585
586    type_decl_list :
587        type_decl_list type_definition
588        >push_back($2);
589        |
590        type_definition
591        TypeDeclList(); $$->push_back($1);
592
593    type_definition :
594        name EQUAL type_decl SEMI
595        TypeDeclaration($1, $3);
596
597    type_decl :
598        simple_type_decl
599        | array_type_decl
600        | record_type_decl
601
602    simple_type_decl :
603        SYS_TYPE
604
605        "integer")
606
607        new AstType(SPL_INTEGER);
608
609        == "boolean")
610
611        new AstType(SPL_BOOLEAN);
612
613        == "real")
614
615        new AstType(SPL_REAL);
616
617    { $$ = new
618    | $$ = new
619    | $$ = new
620    {
621        if(*$1 ==
622            $$ =
623
624        else if(*$1
625            $$ =
626
627        else
628            $$ =
629
630    }
631
632    ;
633
634
635    type_part :
636        TYPE type_decl_list
637        |
638        TypeDeclList();
639
640    type_decl_list :
641        type_decl_list type_definition
642        >push_back($2);
643        |
644        type_definition
645        TypeDeclList(); $$->push_back($1);
646
647    type_definition :
648        name EQUAL type_decl SEMI
649        TypeDeclaration($1, $3);
650
651    type_decl :
652        simple_type_decl
653        | array_type_decl
654        | record_type_decl
655
656    simple_type_decl :
657        SYS_TYPE
658
659        "integer")
660
661        new AstType(SPL_INTEGER);
662
663        == "boolean")
664
665        new AstType(SPL_BOOLEAN);
666
667        == "real")
668
669        new AstType(SPL_REAL);
670
671    { $$ = new
672    | $$ = new
673    | $$ = new
674    {
675        if(*$1 ==
676            $$ =
677
678        else if(*$1
679            $$ =
680
681        else
682            $$ =
683
684    }
685
686    ;
687
688
689    type_part :
690        TYPE type_decl_list
691        |
692        TypeDeclList();
693
694    type_decl_list :
695        type_decl_list type_definition
696        >push_back($2);
697        |
698        type_definition
699        TypeDeclList(); $$->push_back($1);
700
701    type_definition :
702        name EQUAL type_decl SEMI
703        TypeDeclaration($1, $3);
704
705    type_decl :
706        simple_type_decl
707        | array_type_decl
708        | record_type_decl
709
710    simple_type_decl :
711        SYS_TYPE
712
713        "integer")
714
715        new AstType(SPL_INTEGER);
716
717        == "boolean")
718
719        new AstType(SPL_BOOLEAN);
720
721        == "real")
722
723        new AstType(SPL_REAL);
724
725    { $$ = new
726    | $$ = new
727    | $$ = new
728    {
729        if(*$1 ==
730            $$ =
731
732        else if(*$1
733            $$ =
734
735        else
736            $$ =
737
738    }
739
740    ;
741
742
743    type_part :
744        TYPE type_decl_list
745        |
746        TypeDeclList();
747
748    type_decl_list :
749        type_decl_list type_definition
750        >push_back($2);
751        |
752        type_definition
753        TypeDeclList(); $$->push_back($1);
754
755    type_definition :
756        name EQUAL type_decl SEMI
757        TypeDeclaration($1, $3);
758
759    type_decl :
760        simple_type_decl
761        | array_type_decl
762        | record_type_decl
763
764    simple_type_decl :
765        SYS_TYPE
766
767        "integer")
768
769        new AstType(SPL_INTEGER);
770
771        == "boolean")
772
773        new AstType(SPL_BOOLEAN);
774
775        == "real")
776
777        new AstType(SPL_REAL);
778
779    { $$ = new
780    | $$ = new
781    | $$ = new
782    {
783        if(*$1 ==
784            $$ =
785
786        else if(*$1
787            $$ =
788
789        else
790            $$ =
791
792    }
793
794    ;
795
796
797    type_part :
798        TYPE type_decl_list
799        |
800        TypeDeclList();
801
802    type_decl_list :
803        type_decl_list type_definition
804        >push_back($2);
805        |
806        type_definition
807        TypeDeclList(); $$->push_back($1);
808
809    type_definition :
810        name EQUAL type_decl SEMI
811        TypeDeclaration($1, $3);
812
813    type_decl :
814        simple_type_decl
815        | array_type_decl
816        | record_type_decl
817
818    simple_type_decl :
819        SYS_TYPE
820
821        "integer")
822
823        new AstType(SPL_INTEGER);
824
825        == "boolean")
826
827        new AstType(SPL_BOOLEAN);
828
829        == "real")
830
831        new AstType(SPL_REAL);
832
833    { $$ = new
834    | $$ = new
835    | $$ = new
836    {
837        if(*$1 ==
838            $$ =
839
840        else if(*$1
841            $$ =
842
843        else
844            $$ =
845
846    }
847
848    ;
849
850
851    type_part :
852        TYPE type_decl_list
853        |
854        TypeDeclList();
855
856    type_decl_list :
857        type_decl_list type_definition
858        >push_back($2);
859        |
860        type_definition
861        TypeDeclList(); $$->push_back($1);
862
863    type_definition :
864        name EQUAL type_decl SEMI
865        TypeDeclaration($1, $3);
866
867    type_decl :
868        simple_type_decl
869        | array_type_decl
870        | record_type_decl
871
872    simple_type_decl :
873        SYS_TYPE
874
875        "integer")
876
877        new AstType(SPL_INTEGER);
878
879        == "boolean")
880
881        new AstType(SPL_BOOLEAN);
882
883        == "real")
884
885        new AstType(SPL_REAL);
886
887    { $$ = new
888    | $$ = new
889    | $$ = new
890    {
891        if(*$1 ==
892            $$ =
893
894        else if(*$1
895            $$ =
896
897        else
898            $$ =
899
900    }
901
902    ;
903
904
905    type_part :
906        TYPE type_decl_list
907        |
908        TypeDeclList();
909
910    type_decl_list :
911        type_decl_list type_definition
912        >push_back($2);
913        |
914        type_definition
915        TypeDeclList(); $$->push_back($1);
916
917    type_definition :
918        name EQUAL type_decl SEMI
919        TypeDeclaration($1, $3);
920
921    type_decl :
922        simple_type_decl
923        | array_type_decl
924        | record_type_decl
925
926    simple_type_decl :
927        SYS_TYPE
928
929        "integer")
930
931        new AstType(SPL_INTEGER);
932
933        == "boolean")
934
935        new AstType(SPL_BOOLEAN);
936
937        == "real")
938
939        new AstType(SPL_REAL);
940
941    { $$ = new
942    | $$ = new
943    | $$ = new
944    {
945        if(*$1 ==
946            $$ =
947
948        else if(*$1
949            $$ =
950
951        else
952            $$ =
953
954    }
955
956    ;
957
958
959    type_part :
960        TYPE type_decl_list
961        |
962        TypeDeclList();
963
964    type_decl_list :
965        type_decl_list type_definition
966        >push_back($2);
967        |
968        type_definition
969        TypeDeclList(); $$->push_back($1);
970
971    type_definition :
972        name EQUAL type_decl SEMI
973        TypeDeclaration($1, $3);
974
975    type_decl :
976        simple_type_decl
977        | array_type_decl
978        | record_type_decl
979
980    simple_type_decl :
981        SYS_TYPE
982
983        "integer")
984
985        new AstType(SPL_INTEGER);
986
987        == "boolean")
988
989        new AstType(SPL_BOOLEAN);
990
991        == "real")
992
993        new AstType(SPL_REAL);
994
995    { $$ = new
996    | $$ = new
997    | $$ = new
998    {
999        if(*$1 ==
1000            $$ =
1001
1002        else if(*$1
1003            $$ =
1004
1005        else
1006            $$ =
1007
1008    }
1009
1010    ;
1011
1012
1013    type_part :
1014        TYPE type_decl_list
1015        |
1016        TypeDeclList();
1017
1018    type_decl_list :
1019        type_decl_list type_definition
1020        >push_back($2);
1021        |
1022        type_definition
1023        TypeDeclList(); $$->push_back($1);
1024
1025    type_definition :
1026        name EQUAL type_decl SEMI
1027        TypeDeclaration($1, $3);
1028
1029    type_decl :
1030        simple_type_decl
1031        | array_type_decl
1032        | record_type_decl
1033
1034    simple_type_decl :
1035        SYS_TYPE
1036
1037        "integer")
1038
1039        new AstType(SPL_INTEGER);
1040
1041        == "boolean")
1042
1043        new AstType(SPL_BOOLEAN);
1044
1045        == "real")
1046
1047        new AstType(SPL_REAL);
1048
1049    { $$ = new
1050    | $$ = new
1051    | $$ = new
1052    {
1053        if(*$1 ==
1054            $$ =
1055
1056        else if(*$1
1057            $$ =
1058
1059        else
1060            $$ =
1061
1062    }
1063
1064    ;
1065
1066
1067    type_part :
1068        TYPE type_decl_list
1069        |
1070        TypeDeclList();
1071
1072    type_decl_list :
1073        type_decl_list type_definition
1074        >push_back($2);
1075        |
1076        type_definition
1077        TypeDeclList(); $$->push_back($1);
1078
1079    type_definition :
1080        name EQUAL type_decl SEMI
1081        TypeDeclaration($1, $3);
1082
1083    type_decl :
1084        simple_type_decl
1085        | array_type_decl
1086        | record_type_decl
1087
1088    simple_type_decl :
1089        SYS_TYPE
1090
1091        "integer")
1092
1093        new AstType(SPL_INTEGER);
1094
1095        == "boolean")
1096
1097        new AstType(SPL_BOOLEAN);
1098
1099        == "real")
1100
1101        new AstType(SPL_REAL);
1102
1103    { $$ = new
1104    | $$ = new
1105    | $$ = new
1106    {
1107        if(*$1 ==
1108            $$ =
1109
1110        else if(*$1
1111            $$ =
1112
1113        else
1114            $$ =
1115
1116    }
1117
1118    ;
1119
1120
1121    type_part :
1122        TYPE type_decl_list
1123        |
1124        TypeDeclList();
1125
1126    type_decl_list :
1127        type_decl_list type_definition
1128        >push_back($2);
1129        |
1130        type_definition
1131        TypeDeclList(); $$->push_back($1);
1132
1133    type_definition :
1134        name EQUAL type_decl SEMI
1135        TypeDeclaration($1, $3);
1136
1137    type_decl :
1138        simple_type_decl
1139        | array_type_decl
1140        | record_type_decl
1141
1142    simple_type_decl :
1143        SYS_TYPE
1144
1145        "integer")
1146
1147        new AstType(SPL_INTEGER);
1148
1149        == "boolean")
1150
1151        new AstType(SPL_BOOLEAN);
1152
1153        == "real")
1154
1155        new AstType(SPL_REAL);
1156
1157    { $$ = new
1158    | $$ = new
1159    | $$ = new
1160    {
1161        if(*$1 ==
1162            $$ =
1163
1164        else if(*$1
1165            $$ =
1166
1167        else
1168            $$ =
1169
1170    }
1171
1172    ;
1173
1174
1175    type_part :
1176        TYPE type_decl_list
1177        |
1178        TypeDeclList();
1179
1180    type_decl_list :
1181        type_decl_list type_definition
1182        >push_back($2);
1183        |
1184        type_definition
1185        TypeDeclList(); $$->push_back($1);
1186
1187    type_definition :
1188        name EQUAL type_decl SEMI
1189        TypeDeclaration($1, $3);
1190
1191    type_decl :
1192        simple_type_decl
1193        | array_type_decl
1194        | record_type_decl
1195
1196    simple_type_decl :
1197        SYS_TYPE
1198
1199        "integer")
1200
1201        new AstType(SPL_INTEGER);
1202
1203        == "boolean")
1204
1205        new AstType(SPL_BOOLEAN);
1206
1207        == "real")
1208
1209        new AstType(SPL_REAL);
1210
1211    { $$ = new
1212    | $$ = new
1213    | $$ = new
1214    {
1215        if(*$1 ==
1216            $$ =
1217
1218        else if(*$1
1219            $$ =
1220
1221        else
1222            $$ =
1223
1224    }
1225
1226    ;
1227
1228
1229    type_part :
1230        TYPE type_decl_list
1231        |
1232        TypeDeclList();
1233
1234    type_decl_list :
1235        type_decl_list type_definition
1236        >push_back($2);
1237        |
1238        type_definition
1239        TypeDeclList(); $$->push_back($1);
1240
1241    type_definition :
1242        name EQUAL type_decl SEMI
1243        TypeDeclaration($1, $3);
1244
1245    type_decl :
1246        simple_type_decl
1247        | array_type_decl
1248        | record_type_decl
1249
1250    simple_type_decl :
1251        SYS_TYPE
1252
1253        "integer")
1254
1255        new AstType(SPL_INTEGER);
1256
1257        == "boolean")
1258
1259        new AstType(SPL_BOOLEAN);
1260
1261        == "real")
1262
1263        new AstType(SPL_REAL);
1264
1265    { $$ = new
1266    | $$ = new
1267    | $$ = new
1268    {
1269        if(*$1 ==
1270            $$ =
1271
1272        else if(*$1
1273            $$ =
1274
1275        else
1276            $$ =
1277
1278    }
1279
1280    ;
1281
1282
1283    type_part :
1284        TYPE type_decl_list
1285        |
1286        TypeDeclList();
1287
1288    type_decl_list :
1289        type_decl_list type_definition
1290        >push_back($2);
1291        |
1292        type_definition
1293        TypeDeclList(); $$->push_back($1);
1294
1295    type_definition :
1296        name EQUAL type_decl SEMI
1297        TypeDeclaration($1, $3);
1298
1299    type_decl :
1300        simple_type_decl
1301        | array_type_decl
1302        | record_type_decl
1303
1304    simple_type_decl :
1305        SYS_TYPE
1306
1307        "integer")
1308
1309        new AstType(SPL_INTEGER);
1310
1311        == "boolean")
1312
1313        new AstType(SPL_BOOLEAN);
1314
1315        == "real")
1316
1317        new AstType(SPL_REAL);
1318
1319    { $$ = new
1320    | $$ = new
1321    | $$ = new
1322    {
1323        if(*$1 ==
1324            $$ =
1325
1326        else if(*$1
1327            $$ =
1328
1329        else
1330            $$ =
1331
1332    }
1333
1334    ;
1335
1336
1337    type_part :
1338        TYPE type_decl_list
1339        |
1340        TypeDeclList();
1341
1342    type_decl_list :
1343        type_decl_list type_definition
1344        >push_back($2);
1345        |
1346        type_definition
1347        TypeDeclList(); $$->push_back($1);
1348
1349    type_definition :
1350        name EQUAL type_decl SEMI
1351        TypeDeclaration($1, $3);
1352
1353    type_decl :
1354        simple_type_decl
1355        | array_type_decl
1356        | record_type_decl
1357
1358    simple_type_decl :
1359        SYS_TYPE
1360
1361        "integer")
1362
1363        new AstType(SPL_INTEGER);
1364
1365        == "boolean")
1366
1367        new AstType(SPL_BOOLEAN);
1368
1369        == "real")
1370
1371        new AstType(SPL_REAL);
1372
1373    { $$ = new
1374    | $$ = new
1375    | $$ = new
1376    {
1377        if(*$1 ==
1378            $$ =
1379
1380        else if(*$1
1381            $$ =
1382
1383        else
1384            $$ =
1385
1386    }
1387
1388    ;
1389
1390
1391    type_part :
1392        TYPE type_decl_list
1393        |
1394        TypeDeclList();
1395
1396    type_decl_list :
1397        type_decl_list type_definition
1398        >push_back($2);
1399        |
1400        type_definition
1401        TypeDeclList(); $$->push_back($1);
1402
1403    type_definition :
1404        name EQUAL type_decl SEMI
1405        TypeDeclaration($1, $3);
1406
1407    type_decl :
1408        simple_type_decl
1409        | array_type_decl
1410        | record_type_decl
1411
1412    simple_type_decl :
1413        SYS_TYPE
1414
1415        "integer")
1416
1417        new AstType(SPL_INTEGER);
1418
1419        == "boolean")
1420
1421        new AstType(SPL_BOOLEAN);
1422
1423        == "real")
1424
1425        new AstType(SPL_REAL);
1426
1427    { $$ = new
1428    | $$ = new
1429    | $$ = new
1430    {
1431        if(*$1 ==
1432            $$ =
1433
1434        else if(*$1
1435            $$ =
1436
1437        else
1438            $$ =
1439
1440    }
1441
1442    ;
1443
1444
1445    type_part :
1446        TYPE type_decl_list
1447        |
1448        TypeDeclList();
1449
1450    type_decl_list :
1451        type_decl_list type_definition
1452        >push_back($2);
1453        |
1454        type_definition
1455        TypeDeclList(); $$->push_back($1);
1456
1457    type_definition :
1458        name EQUAL type_decl SEMI
1459        TypeDeclaration($1, $3);
1460
1461    type_decl :
1462        simple_type_decl
1463        | array_type_decl
1464        | record_type_decl
1465
1466    simple_type_decl :
1467        SYS_TYPE
1468
1469        "integer")
1470
1471        new AstType(SPL_INTEGER);
1472
1473        == "boolean")
1474
1475        new AstType(SPL_BOOLEAN);
1476
1477        == "real")
1478
1479        new AstType(SPL_REAL);
1480
1481    { $$ = new
1482    | $$ = new
1483    | $$ = new
1484    {
1485        if(*$1 ==
1486            $$ =
1487
1488        else if(*$1
1489            $$ =
1490
1491        else
1492            $$ =
1493
1494    }
1495
1496    ;
1497
1498
1499    type_part :
1500        TYPE type_decl_list
1501        |
1502        TypeDeclList();
1503
1504    type_decl_list :
1505        type_decl_list type_definition
1506        >push_back($2);
1507        |
1508        type_definition
1509        TypeDeclList(); $$->push_back($1);
1510
1511    type_definition :
1512        name EQUAL type_decl SEMI
1513        TypeDeclaration($1, $3);
1514
1515    type_decl :
1516        simple_type_decl
1517        | array_type_decl
1518        | record_type_decl
1519
1520    simple_type_decl :
1521        SYS_TYPE
1522
1523        "integer")
1524
1525        new AstType(SPL_INTEGER);
1526
1527        == "boolean")
1528
1529        new AstType(SPL_BOOLEAN);
1530
1531        == "real")
1532
1533        new AstType(SPL_REAL);
1534
1535    { $$ = new
1536    | $$ = new
1537    | $$ = new
1538    {
1539        if(*$1 ==
1540            $$ =
1541
1542        else if(*$1
1543            $$ =
1544
1545        else
1546            $$ =
1547
1548    }
1549
1550    ;
1551
1552
1553    type_part :
1554        TYPE type_decl_list
1555        |
1556        TypeDeclList();
1557
1558    type_decl_list :
1559        type_decl_list type_definition
1560        >push_back($2);
1561        |
1562        type_definition
1563        TypeDeclList(); $$
```

```

68     == "char")
69
70     new AstType(SPL_CHAR);
71
72     "UNKNOWN_SYS_TYPE" << endl;
73
74     | name
75     AstType($1); }
76
77     | LP name_list RP
78     AstType(new EnumType($2)); }
79
80     | const_value DOTDOT const_value
81     AstType(new ConstRangeType($1, $3)); }
82
83     | MINUS const_value DOTDOT const_value
84     AstType(new ConstRangeType(-*$2, $4)); }
85
86     | MINUS const_value DOTDOT MINUS const_value
87     AstType(new ConstRangeType(-*$2, -$5)); }
88
89     | name DOTDOT name
90     AstType(new EnumRangeType($1, $3)); }
91
92     ;

```

3.4 抽象语法树可视化

语法树可视化使用d3.js完成，d3.js可以通过Json数据绘制出树图html，AST可以通过根节点(Program)的getJson方法获得Json数据。

语法树可视化工作流如下：

1. 生成AST
 2. 调用getJSON方法
 3. 在tree.html下构建一个服务器，推荐使用VSCode的liveServer，也可以自己搭建一个apache服务器，注意要把tree.html和tree.json放在同一路径下
 4. 打开浏览器，输入服务器地址，即可看到树图。

JSON数据获取函数：

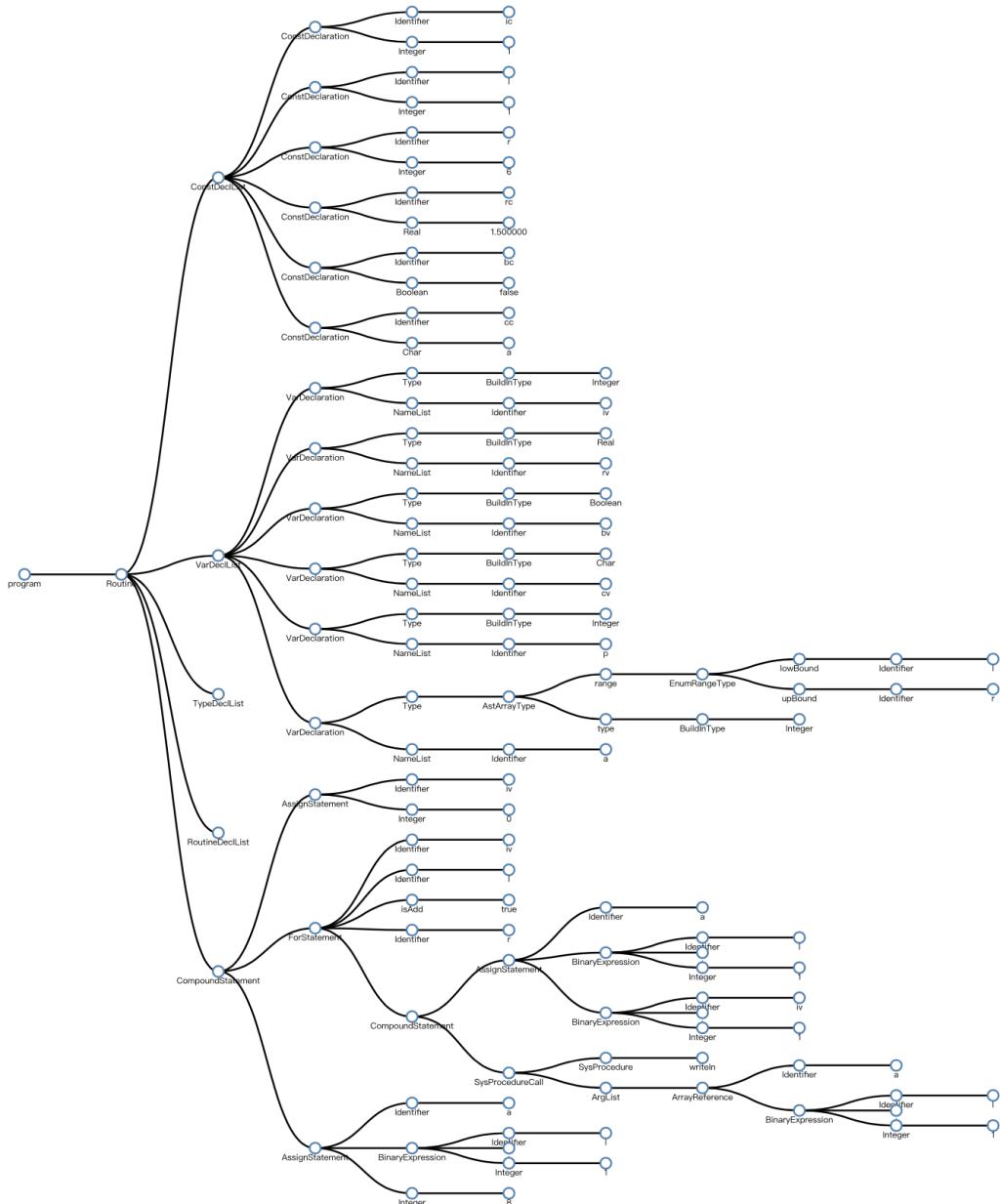
```
1 string getJsonString(string name) {
2     return "{ \"name\" : \"" + name + "\" }";
3 }
4
5 string getJsonString(string name, vector<string> children) {
6     string result = "{ \"name\" : \"" + name + "\", \"children\" : [ ";
7     int i = 0;
8     for(auto &child : children) {
9         if(i != children.size() - 1)
10            result += child + ", ";
11        else
12            result += child + " ";
13        i++;
14    }
15    return result + " ] }";
16 }
17
18 string getJsonString(string name, string value) {
19     return getJsonString(name, vector<string>{value});
20 }
```

```

21
22     string getJsonString(string name, string value, vector<string> children) {
23         string result = "{ \"name\" : \"" + name + "\", \"value\" : \"" + value
24 + "\", \"children\" : [ ";
25         int i = 0;
26         for(auto &child : children) {
27             if(i != children.size() - 1)
28                 result += child + ", ";
29             else
30                 result += child + " ";
31             i++;
32         }
33         return result + " ] }";
34     }

```

具体效果如下：



第参章 语义分析

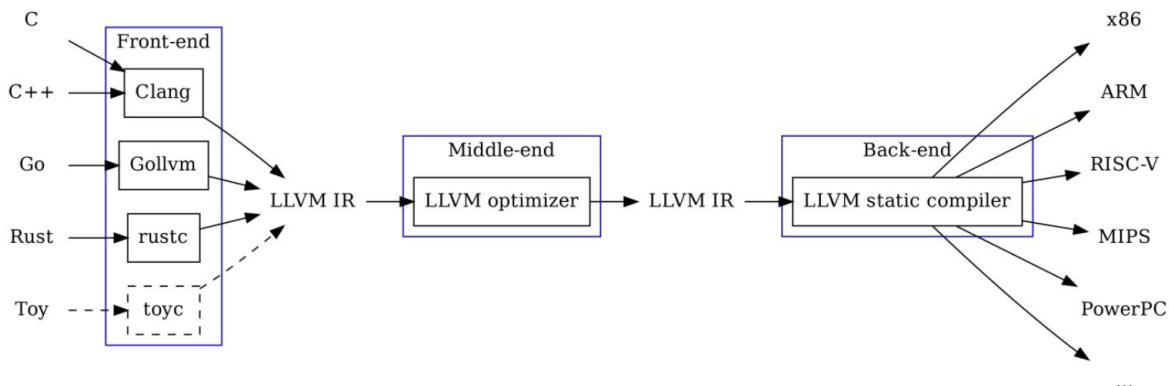
3.1 LLVM概述

LLVM(Low Level Virtual Machine)是以C++编写的编译器基础设施，包含一系列模块化的编译器组件和工具教练用俩开发编译器前端和后端。LLVM起源于2000年伊利诺伊大学Vikram Adve和Chris Lattner的研究，它是为了任意一种编程语言而写成的程序，利用虚拟技术创造出编译阶段、链接阶段、运行阶段以及闲置阶段的优化，目前支持Ada、D语言、Fortran、GLSL、Java字节码、Swift、Python、Ruby等十多种语言。

- 前端： LLVM最初被用来取代现有于GCC堆栈的代码产生器，许多GCC的前端已经可以与其运行，其中Clang是一个新的编译器，同时支持C、Objective-C以及C++。
- 中间端： LLVM IR是一种类似汇编的底层语言，一种强类型的精简指令集，并对目标指令集进行了抽象。 LLVM支持C++中对象形式、序列化bitcode形式和汇编形式。
- 后端： LLVM支持ARM、Qualcomm Hexagon、MPIS、Nvidia并行指令集等多种后端指令集。

3.2 LLVM IR

LLVM IR是LLVM的核心所在，通过将不同高级语言的前端转换成LLVM IR进行优化、链接后再传给不同的后端转换成为二进制代码，前端、优化、后端三个阶段互相解耦，这种模块化的设计使得LLVM优化不依赖于任何源码和目标机器。

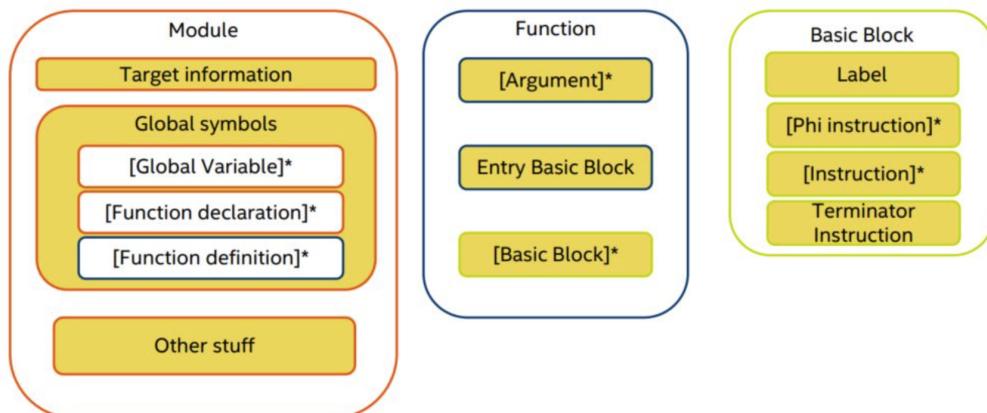


3.2.1 IR布局

每个IR文件称为一个Module，它是其他所有IR对象的顶级容器，包含了目标信息、全局符号和所依赖的其他模块和符号表等对象的列表，其中全局符号又包括了全局变量、函数声明和函数定义。

函数由参数和多个基本块组成，其中第一个基本块称为entry基本块，这是函数开始执行的起点，另外LLVM的函数拥有独立的符号表，可以对标识符进行查询和搜索。

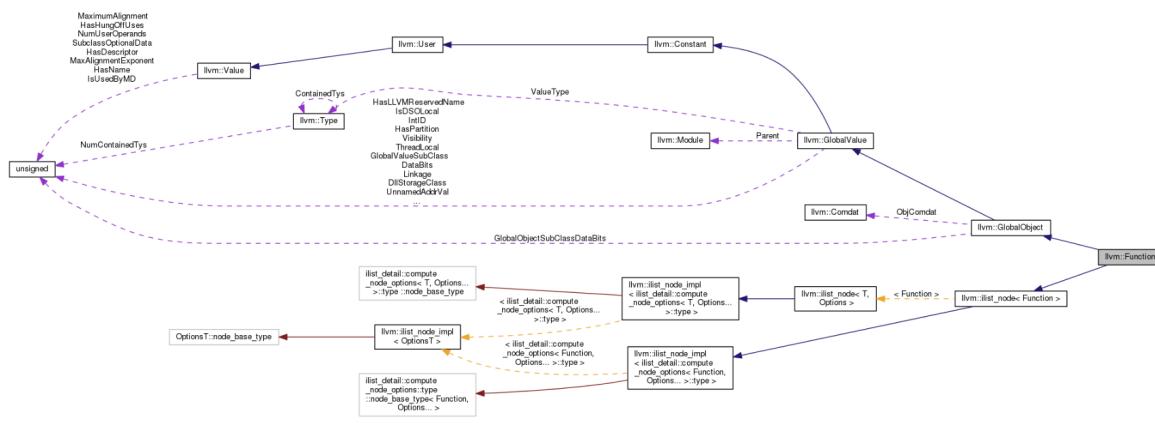
每一个基本块包含了标签和各种指令的集合，标签作为指令的索引用于实现指令间的跳转，指令包含Phi指令、一般指令以及终止指令等。



3.2.2 IR上下文环境

- LLVM::Context: 提供用户创建变量等对象的上下文环境，尤其在多线程环境下至关重要
 - LLVM::IRBuilder: 提供创建LLVM指令并将其插入基础块的API

3.2.3 IR核心类



- `llvm::Value`表示一个类型的值，具有一个`llvm::Type*`成员和一个use list，前者指向值的类型类，后者跟踪使用了该值的其他对象，可以通过迭代器进行访问。
 - 值的存取分别可以通过`llvm::LoadInst`和`llvm::StoreInst`实现，也可以借助`IRBuilder`的`CreateLoad`和`CreateStore`实现。
 - `llvm::Type`表示类型类，LLVM支持17种数据类型，可以通过`Type ID`判断类型：

```

1 enumTypeID {
2     // PrimitiveTypes - make sure LastPrimitiveTyID stays up to date.
3     VoidTyID = 0,      ///< 0: type with no size
4     HalfTyID,         ///< 1: 16-bit floating point type
5     FloatTyID,        ///< 2: 32-bit floating point type
6     DoubleTyID,       ///< 3: 64-bit floating point type
7     X86_FP80TyID,    ///< 4: 80-bit floating point type (x87)
8     FP128TyID,        ///< 5: 128-bit floating point type (112-bit
9     mantissa)
10    PPC_FP128TyID,   ///< 6: 128-bit floating point type (two 64-bits,
11     PowerPC)
12     LabelTyID,       ///< 7: Labels
13     MetadataTyID,    ///< 8: Metadata
14     X86_MMXTyID,    ///< 9: MMX vectors (64 bits, x86 specific)
15     TokenTyID,        ///< 10: Tokens
16
17     // Derived types... see DerivedTypes.h file.
18     // Make sure FirstDerivedTyID stays up to date!
19     IntegerTyID,      ///< 11: Arbitrary bit width integers
20     FunctionTyID,     ///< 12: Functions
21     StructTyID,       ///< 13: Structures
22     ArrayTyID,        ///< 14: Arrays
23     PointerTyID,      ///< 15: Pointers
24     VectorTyID        ///< 16: SIMD 'packed' format, or other vector type
25 };

```

- `llvm::Constant`表示各种常量的基类，包括`ConstantInt`整形常量、`ConstantFP`浮点型常量、`ConstantArray`数组常量、`ConstantStruct`结构体常量等。

3.3 IR生成

3.3.1 运行环境设计

LLVM IR的生成依赖上下文环境，我们构造了CodeGenerator类来保存环境，在递归遍历AST节点的时候传递CodeGenerator的实例进行每个节点的IR生成。CodeGenerator包括的环境配置：

- 静态全局的上下文变量和构造器变量

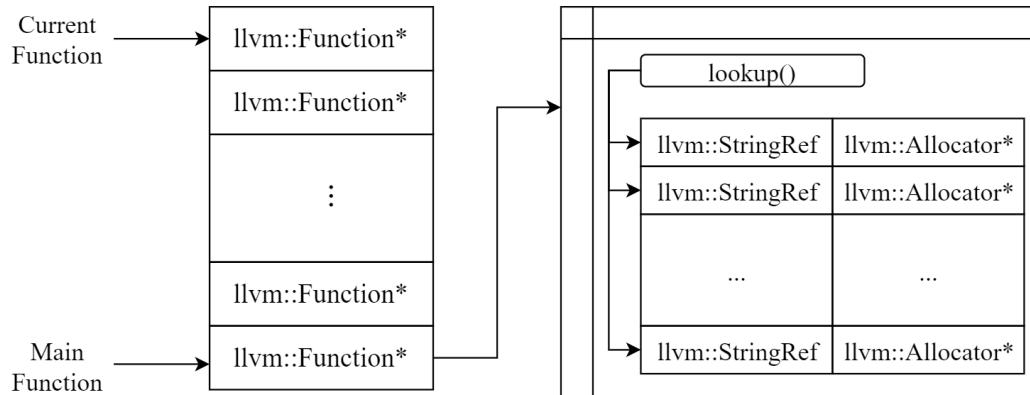
```
1 static llvm::LLVMContext TheContext;
2 static llvm::IRBuilder<> TheBuilder(TheContext);
```

- 公有的模块实例、地址空间、函数栈、标签基础块表

- 模块实例是中间代码顶级容器，用于包含所有变量、函数和指令
- 地址空间是LLVM版本升级后引入的地址空间变量
- 函数栈用于存储函数指针的栈，用于实现静态链（函数递归调用）和动态链（变量访问）
- 标签基础块表记录了带标签语句的基础块用于goto跳转实现，最大支持10000条带标签基础块

```
1 std::unique_ptr<llvm::Module> TheModule;
2 unsigned int TheAddrSpace;
3 std::vector<llvm::Function*> funcStack;
4 llvm::BasicBlock* labelBlock[10000];
```

- 符号表：结合函数指针栈和LLVM函数自带的符号表来实现，创建变量时可以自动插入函数的符号表，也可以通过 `getValueSymbolTable()->lookup(name)` 来查询和取出符号表中指定符号名字的值：



另外模块的全局变量可以从`llvm::Module`的`getGlobalVariable()`查询和获取，在`CodeGenerator.h`中实现了查询符号的函数：

```
1 llvm::Value* findValue(const std::string & name)
2 {
3     llvm::Value * result = nullptr;
4     for (auto it = funcStack.rbegin(); it != funcStack.rend(); it++)
5     {
6         if ((result = (*it)->getValueSymbolTable()->lookup(name)) !=
7             nullptr)
8         {
9             //std::cout << "Find " << name << " in " << std::string((*it)-
10             >getName()) << std::endl;
11             return result;
12         }
13     }
14 }
```

```

12     {
13         //std::cout << "Not Find " << name << " in " <<
14         std::string((*it)->getName()) << std::endl;
15     }
16     if ((result = TheModule->getGlobalVariable(name)) == nullptr)
17     {
18         throw std::logic_error("[ERROR]Undeclared variable: " + name);
19     }
20     //std::cout << "Find " << name << " in global" << std::endl;
21     return result;
22 }

```

3.3.2 类型系统

从AST节点的类型映射到LLVM IR类型可以直接利用LLVM的Type类实现，但IRBuilder提供了基于上下文的更为方便的创建方式，我们使用IRBuilder来构造变量类型：

```

1 llvm::Type* AstType::toLLVMType()
2 {
3     switch (this->type)
4     {
5         case SPL_ARRAY:
6             if (this->arrayType->range->type == SPL_CONST_RANGE)
7             {
8                 return llvm::ArrayType::get(this->arrayType->type-
9 >toLLVMType(), this->arrayType->range->constRangeType->size());
10            }
11            else
12            {
13                return llvm::ArrayType::get(this->arrayType->type-
14 >toLLVMType(), this->arrayType->range->enumRangeType->size());
15            }
16        case SPL_CONST_RANGE: return TheBuilder.getInt32Ty();
17        case SPL_ENUM_RANGE: return TheBuilder.getInt32Ty();
18        case SPL_BUILD_IN:
19            switch (buildInType)
20            {
21                case SPL_INTEGER: return TheBuilder.getInt32Ty();
22                case SPL_REAL: return TheBuilder.getDoubleTy();
23                case SPL_CHAR: return TheBuilder.getInt8Ty();
24                case SPL_BOOLEAN: return TheBuilder.getInt1Ty();
25            }
26            break;
27        case SPL_ENUM:
28        case SPL_RECORD:
29        case SPL_USER_DEFINE:
30        case SPL_VOID: return TheBuilder.getVoidTy();
31    }
32 }

```

其中：

- 内置类型Int, Char, Bool分别使用32、8、1位的Int类型
- Real使用Double类型
- Range使用32位Int类型
- Void对应Void类型

- 数组类型需要先构造数组元素类型并通过Range类型的上下限确定数组大小，分为常量范围数组和变量范围数组。

为了实现引用传递，引入指针类型（仅支持内置类型指针）：

```

1  llvm::Type* toLLVMPtrType(const BuildInType & type)
2  {
3      switch (type)
4      {
5          case SPL_INTEGER: return llvm::Type::getInt32PtrTy(TheContext);
6          case SPL_REAL:   return llvm::Type::getDoublePtrTy(TheContext);
7          case SPL_CHAR:   return llvm::Type::getInt8PtrTy(TheContext);
8          case SPL_BOOLEAN: return llvm::Type::getInt1PtrTy(TheContext);
9          default: throw logic_error("Not supported pointer type.");
10     }
11 }
```

3.3.3 常量获取

可以直接通过IRBuilder获取llvm::Constant：

```

1  llvm::Value *Integer::CodeGen(CodeGenerator & generator) {
2 //    LOG_I("Integer");
3     return TheBuilder.getInt32(this->value);
4 }
5
6  llvm::Value *Char::CodeGen(CodeGenerator & generator) {
7 //    LOG_I("Char");
8     return TheBuilder.getInt8(this->value);
9 }
10
11 llvm::Value *Real::CodeGen(CodeGenerator & generator) {
12 //LOG_I("Real");
13 //    return llvm::ConstantFP::get(TheContext, llvm::APFloat(this->value));
14     return llvm::ConstantFP::get(TheBuilder.getDoubleTy(), this->value);
15 }
16
17 llvm::Value *Boolean::CodeGen(CodeGenerator & generator) {
18 //LOG_I("Boolean");
19     return TheBuilder.getInt1(this->value);
20 }
```

3.3.4 变量创建和存取

LLVM中可以依赖函数和基础块上下文创建局部变量并通过传递一个llvm::StringRef值指定变量名称：

```

1  llvm::AllocaInst *CreateEntryBlockAlloca(llvm::Function *TheFunction,
2                                         llvm::StringRef VarName, llvm::Type* type)
3  {
4      llvm::IRBuilder<> TmpB(&TheFunction->getEntryBlock(), TheFunction-
5                               >getEntryBlock().begin());
6      return TmpB.CreateAlloca(type, nullptr, VarName);
7 }
```

- 访问变量值： llvm::LoadInst或者IRBuilder的CreateLoad实现
- 存储变量值： llvm::StoreInst或者IRBuilder的StoreLoad实现

3.3.5 标识符/数组引用

基于变量取值、函数栈以及LLVM函数的符号表可以实现标识符到`llvm::Value*`的映射从而返回标识符的值：

```
1 | llvm::Value *Identifier::CodeGen(CodeGenerator & generator) {
2 | //    LOG_I("Idnetifier");
3 |     return new llvm::LoadInst(generator.findValue(*this->name)), "tmp",
4 |     false, TheBuilder.GetInsertBlock());
5 | }
```

基于数组元素下标的引用过程：

- 根据数组标识符查询符号表得到数组地址
- 根据数组标识符查询得到数组的范围类型，分为常量范围类型和变量范围类型
- 根据范围类型的下限和索引下标的值计算地址偏移量
- 根据数组地址和地址偏移量获取数组元素地址
- 根据元素地址加载元素值

以上过程通过`getReference`函数实现获取元素引用：

```
1 | llvm::Value *ArrayReference::getReference(CodeGenerator & generator)
2 | {
3 |     string name = this->array->getName();
4 |     llvm::Value* arrayValue = generator.findValue(name), *indexValue;
5 |     if (generator.arrayMap[name]->range->type == AstType::SPL_CONST_RANGE)
6 |     {
7 |         indexValue = generator.arrayMap[name]->range->constRangeType-
8 | >mapIndex(this->index->CodeGen(generator), generator);
9 |     }
10 |     else
11 |     {
12 |         indexValue = generator.arrayMap[name]->range->enumRangeType-
13 | >mapIndex(this->index->CodeGen(generator), generator);
14 |     }
15 |     vector<llvm::Value*> indexList;
16 |     indexList.push_back(TheBuilder.getInt32(0));
17 |     indexList.push_back(indexValue);
18 |     return TheBuilder.CreateInBoundsGEP(arrayValue,
19 |     llvm::ArrayRef<llvm::Value*>(indexList));
20 | }
```

并在`ArrayReference`中加载元素的值：

```
1 | llvm::Value *ArrayReference::CodeGen(CodeGenerator & generator) {
2 | //LOG_I("Array Reference");
3 |     return TheBuilder.CreateLoad(this->getReference(generator), "arrRef");
4 | }
```

3.3.6 二元操作

LLVM的`IRBuilder`集成了丰富的二元操作接口，包括ADD, SUB, MUL, DIV, CMPGE, CMPLT, CMPEQ, CMPNE, AND, OR, SREM(MOD), XOR等，实验中设计了`BinaryOp`函数，根据操作符和两个操作数返回一个二元操作结果值（整型操作和浮点操作有区别）：

```

1  llvm::value *BinaryOp(llvm::Value *lvalue, BinaryExpression::BinaryOperator
2   op, llvm::value *rvalue)
3   {
4   //      printType(lvalue);
5   //      printType(rvalue);
6   bool flag = lvalue->getType()->isDoubleTy() || rvalue->getType()-
7   >isDoubleTy();
8   switch (op)
9   {
10    case BinaryExpression::SPL_PLUS: return flag ?
TheBuilder.CreateFAdd(lvalue, rvalue, "addtmpf") :
TheBuilder.CreateAdd(lvalue, rvalue, "addtmpi");
11
12    case BinaryExpression::SPL_MINUS: return flag ?
TheBuilder.CreateFSub(lvalue, rvalue, "subtmpf") :
TheBuilder.CreateSub(lvalue, rvalue, "subtmpi");
13
14    case BinaryExpression::SPL_MUL: return flag ?
TheBuilder.CreateFMul(lvalue, rvalue, "multmpf") :
TheBuilder.CreateMul(lvalue, rvalue, "multmpi");
15
16    case BinaryExpression::SPL_DIV: return
TheBuilder.CreateSDiv(lvalue, rvalue, "tmpDiv");
17
18    case BinaryExpression::SPL_GE: return
TheBuilder.CreateICmpSGE(lvalue, rvalue, "tmpSGE");
19
20    case BinaryExpression::SPL_GT: return
TheBuilder.CreateICmpSGT(lvalue, rvalue, "tmpSGT");
21
22    case BinaryExpression::SPL_LT: return
TheBuilder.CreateICmpSLT(lvalue, rvalue, "tmpSLT");
23
24    case BinaryExpression::SPL_LE: return
TheBuilder.CreateICmpSLE(lvalue, rvalue, "tmpSLE");
25
26    case BinaryExpression::SPL_EQUAL: return
TheBuilder.CreateICmpEQ(lvalue, rvalue, "tmpEQ");
27
28    case BinaryExpression::SPL_UNEQUAL: return
TheBuilder.CreateICmpNE(lvalue, rvalue, "tmpNE");
29
30    case BinaryExpression::SPL_OR: return TheBuilder.CreateOr(lvalue,
rvalue, "tmpOR");
31
32    case BinaryExpression::SPL_MOD: return
TheBuilder.CreateSRem(lvalue, rvalue, "tmpSREM");
33
34    case BinaryExpression::SPL_AND: return TheBuilder.CreateAnd(lvalue,
rvalue, "tmpAND");
35
36    case BinaryExpression::SPL_XOR: return TheBuilder.CreateXor(lvalue,
rvalue, "tmpXOR");
37  }
38 }

```

同时在BinaryExpression的代码生成中调用以上函数实现二进制表达式操作：

```

1  llvm::Value *BinaryExpression::CodeGen(CodeGen & generator) {
2      //LOG_I("Binary Expression");
3      llvm::Value* lvalue = this->lhs->CodeGen(generator);
4      llvm::Value* rvalue = this->rhs->CodeGen(generator);
5      return BinaryOp(lvalue, this->op, rvalue);
6  }

```

3.3.7 赋值语句

与标识符引用相对的是赋值语句，需要计算等式右表达式的值并赋予左表达式，分为标识符赋值和数组赋值。二者的区别就是标识符引用和数组元素引用，前者直接在符号表查询地址，后者需要根据下标和下限计算偏移量再获取元素的地址，这部分具体描述在[3.3.5](#)。

```

1  llvm::Value *AssignStatement::CodeGen(CodeGen & generator) {
2      //LOG_I("Assign Statement");
3      llvm::Value *res = nullptr;
4      this->forward(generator);
5      switch (this->type)
6      {
7          case ID_ASSIGN: res = TheBuilder.CreateStore(this->rhs-
8              >CodeGen(generator), generator.findValue(this->lhs->getName())); break;
9          case ARRAY_ASSIGN: res = TheBuilder.CreateStore(this->rhs-
10             >CodeGen(generator), (new ArrayReference(this->lhs, this->sub))->
11             getReference(generator)); break;
12          case RECORD_ASSIGN: res = nullptr; break;
13      }
14      this->backward();
15      return res;
16  }

```

3.3.8 Program

Program相当于程序的main函数，需要构建main函数的函数类型并创建函数实例，将main函数push进入函数栈，之后构造一个基础块作为指令的插入点，递归调用Routine的代码生成后函数出栈。同时由于main函数直接存在于模块中，需要把其Routine下的声明置为global，这点可以通过setGlobal()实现。

```

1  llvm::Value *Program::CodeGen(CodeGen & generator) {
2      //LOG_I("Program");
3      //Main function prototype
4      vector<llvm::Type*> argTypes;
5      llvm::FunctionType * funcType =
6          llvm::FunctionType::get(TheBuilder.getVoidTy(), makeArrayRef(argTypes),
7          false);
8      generator.mainFunction = llvm::Function::Create(funcType,
9          llvm::GlobalValue::ExternalLinkage, "main", generator.TheModule.get());
10     llvm::BasicBlock * basicBlock = llvm::BasicBlock::Create(TheContext,
11         "entrypoint", generator.mainFunction, 0);
12
13     generator.pushFunction(generator.mainFunction);
14     TheBuilder.SetInsertPoint(basicBlock);
15     //Create System functions
16     generator.printf = generator.createPrintf();
17     generator.scprintf = generator.createScprintf();
18     //Code generate
19
20 }

```

```

15     this->routine->setGlobal();
16     this->routine->CodeGen(generator);
17     TheBuilder.CreateRetVoid();
18     generator.popFunction();
19
20     return nullptr;
21 }
```

3.3.9 Routine

Routine包含了常量声明、变量声明、类型声明、子例程声明、函数体声明，该节点只需要依此调用子节点的CodeGen方法即可。

```

1 llvm::Value *Routine::CodeGen(CodeGenerator & generator) {
2     //LOG_I("Routine");
3     llvm::Value* res = nullptr;
4
5     //Const declaration part
6     for (auto & constDecl : *(this->constDeclList))
7     {
8         res = constDecl->CodeGen(generator);
9     }
10    //Variable declaration part
11    for (auto & varDecl : *(this->varDeclList))
12    {
13        res = varDecl->CodeGen(generator);
14    }
15    //Type declaration part
16    for (auto & typeDecl : *(this->typeDeclList))
17    {
18        res = typeDecl->CodeGen(generator);
19    }
20    //Routine declaration part
21    for (auto & routineDecl : *(this->routineList))
22    {
23        res = routineDecl->CodeGen(generator);
24    }
25
26    //Routine body
27    res = routineBody->CodeGen(generator);
28    return res;
29 }
```

3.3.10 常量/变量声明

常量声明和变量声明相似，只不过增加了初始化和常量属性声明。

- 全局常量/变量：调用llvm::GlobalVariable构造全局常量/变量，通过指定isConst参数来区分常量和变量。此时的初始化可以通过传递初始值作为llvm::GlobalVariable的参数实现。
- 局部常量/变量：调用3.3.4提到的CreateEntryBlockAlloca方法创建局部常量/变量。此时的初始化可以通过IRBuilder直接存入初始值。

```

1 llvm::Value *ConstDeclaration::CodeGen(CodeGenerator & generator) {
2     //LOG_I("Const Declaration");
3     string name = this->name->getName();
4     this->type = new AstType(this->value->getType());
```

```

5     if (this->isGlobal())
6     {
7         return new llvm::GlobalVariable(*generator.TheModule, this->type-
8             >toLLVMType(), true, llvm::GlobalValue::ExternalLinkage, this->type-
9             >initValue(this->value), name);
10    }
11    else
12    {
13        auto alloc = CreateEntryBlockAlloca(generator.getCurFunction(),
14 name, this->type->toLLVMType());
15        return TheBuilder.CreateStore(this->value->CodeGen(generator),
16 alloc);
17    }
18}

```

常量/变量创建之后将自动加入到当前函数的符号表或者整个模块的符号表中。

3.3.11 函数/过程声明

函数声明和过程声明类似，函数声明只需要基于过程声明增加返回值处理即可，这里把二者合在一起实现。声明包括：

- 函数类型声明
 - 参数类型考虑引用传递，所以需要处理非指针类型和指针类型
 - 函数声明的返回值类型为实际类型，过程声明的返回值类型为空
- 函数实例和基础块：根据函数类型创建函数实例并构建基础块作为代码插入点
- 函数入栈：将函数实例的指针推入函数栈
- 函数实参获取：可以通过`llvm::Function::arg_iterator`迭代遍历函数的实际参数并获取参数的值
 - 值传递：将获取的参数值直接存到局部变量中
 - 引用传递：通过`TheBuilder.CreateGEP`获取参数地址，并指定新的变量名，无需存储值，此
外为了在函数调用时可以区分引用传递参数和值传递参数，我们利用LLVM函数的参数属性
进行标识
- 函数返回值声明：函数声明需要创建函数返回值变量，同时以函数名称给其命名
- 函数体生成：调用函数体子节点生成代码
- 函数返回：创建返回实例，函数声明返回函数名的返回值，过程声明返回空值
- 函数出栈：将函数指针弹出栈顶，并将当前函数指针重新指向栈顶

```

1  llvm::Value *FuncDeclaration::CodeGen(CodeGen & generator) {
2      //LOG_I("Function Declaration");
3      //Prototype
4      vector<llvm::Type*> argTypes;
5      for (auto & argType : *(this->paraList))
6      {
7          if (argType->isVar)
8          {
9              argTypes.insert(argTypes.end(), argType->nameList->size(),
10                  toLLVMPtrType(argType->getType()->buildInType()));
11          }
12      }
13      argTypes.insert(argTypes.end(), argType->nameList->size(),
14          argType->getType()->toLLVMType());
15  }

```

```

15     }
16     llvm::FunctionType *funcType = llvm::FunctionType::get(this-
>returnType->toLLVMTy(), argTypes, false);
17     llvm::Function *function = llvm::Function::Create(funcType,
18     llvm::GlobalValue::InternalLinkage, this->name->getName(),
19     generator.TheModule.get());
20     generator.pushFunction(function);
21
22     //Block
23     llvm::BasicBlock *newBlock = llvm::BasicBlock::Create(TheContext,
24     "entrypoint", function, nullptr);
25     TheBuilder.SetInsertPoint(newBlock);
26
27     //Parameters
28     llvm::Function::arg_iterator argIt = function->arg_begin();
29     int index = 1;
30     for (auto & args : *(this->paraList))
31     {
32         for (auto & arg : *(args->nameList))
33         {
34             llvm::Value *alloc = nullptr;
35             if (args->isVar)
36             {
37                 //Check value
38                 alloc = generator.findValue(arg->getName());
39                 function->addAttribute(index, llvm::Attribute::NonNull);
40                 alloc = TheBuilder.CreateGEP(argIt++,
41                     TheBuilder.getInt32(0), arg->getName());
42                 TheBuilder.CreateStore(argIt++, alloc);
43             }
44             index++;
45         }
46     }
47
48     //Return
49     llvm::Value *res = nullptr;
50     if (this->returnType->type != AstType::SPL_VOID)
51     {
52         res = CreateEntryBlockAlloca(function, this->name->getName(), this-
>returnType->toLLVMTy());
53     }
54
55     //Sub routine
56     this->subRoutine->CodeGen(generator);
57
58     //Return value
59     if (this->returnType->type != AstType::SPL_VOID)
60     {
61         auto returnInst = this->name->CodeGen(generator);
62         TheBuilder.CreateRet(returnInst);
63     }
64     else
65     {

```

```

66     TheBuilder.CreateRetVoid();
67 }
68
69 //Pop back
70 generator.popFunction();
71 TheBuilder.SetInsertPoint(&(generator.getCurFunction()-
72 >getBasicBlockList().back());
73 return function;
74 }
```

3.3.12 函数/过程调用

函数调用和过程调用类似：

- 从Module查找函数名称从而获取函数指针
- 创建函数参数向量，逐个计算参数表达式的值，为了区分值传递和引用传递，需要通过Function::arg_iterator遍历函数的参数，并判断是否具有之前标记的属性：
 - 值传递：调用代码生成计算并加载值
 - 引用传递：直接查询符号表传递地址
- 通过IRBuilder的CreateCall构造函数调用

```

1  llvm::Value *FunctionCall::CodeGen(CodeGenerator & generator) {
2      //LOG_I("Function Call");
3      this->forward(generator);
4      llvm::Function *function = generator.TheModule->getFunction(this-
5      >function->getName());
6      if (function == nullptr)
7      {
8          throw domain_error("[ERROR] Function not defined: " + this-
9          >function->getName());
10     }
11     vector<llvm::Value*> args;
12     llvm::Function::arg_iterator argIt = function->arg_begin();
13     for (auto & arg : *(this->args))
14     {
15         if (argIt->hasNonNullAttr())
16         {
17             cout << "Pass a pointer" << endl;
18             llvm::Value * addr =
19             generator.findValue(dynamic_cast<Identifier*>(arg)->getName());
20             args.push_back(addr);
21         }
22         else
23         {
24             cout << "Pass a value" << endl;
25             args.push_back(arg->CodeGen(generator));
26         }
27         argIt++;
28     }
29     llvm::Value *res = TheBuilder.CreateCall(function, args, "calltmp");
30     this->backward();
31     return res;
32 }
```

3.3.13 系统函数/过程

本次实验基于C语言的printf和scanf设计了读写系统函数：read和write(writeln)。

在CodeGenerator.h中定义了printf和scanf的原型，包括参数类型、返回值和调用环境等：

```
1  llvm::Function* createPrintf()
2  {
3      std::vector<llvm::Type*> arg_types;
4      arg_types.push_back(TheBuilder.getInt8PtrTy());
5      auto printf_type = llvm::FunctionType::get(TheBuilder.getInt32Ty(),
6          llvm::makeArrayRef(arg_types), true);
7      auto func = llvm::Function::Create(printf_type,
8          llvm::Function::ExternalLinkage, llvm::Twine("printf"), TheModule.get());
9      func->setCallingConv(llvm::CallingConv::C);
10     return func;
11 }
12
13 llvm::Function* createScanf()
14 {
15     auto scanf_type = llvm::FunctionType::get(TheBuilder.getInt32Ty(),
16         true);
17     auto func = llvm::Function::Create(scanf_type,
18         llvm::Function::ExternalLinkage, llvm::Twine("scanf"), TheModule.get());
19     func->setCallingConv(llvm::CallingConv::C);
20     return func;
21 }
```

在Program节点生成代码时在Module下创建了以上函数实例，之后即可在SysProcedureCall节点中获取函数原型并生成函数代码：

- 构造函数实参，根据参数类型指定输出的格式化字符，同时生成参数值到参数向量

类型	格式化字符	备注
integer	%d	
char	%c	
bool	%d	false对应0, true对应1
real	%lf	不能为%f, 因为llvm::Double类型无法用%f输入

- 判断是否换行，writeln函数在格式字符串末尾增加换行符
- 拼接格式化字符串和参数向量
- 通过IRBuilder调用函数

```
1  llvm::Value *SysProcedurecall::sysProcWrite(CodeGen & generator, bool
2  isLineBreak)
3  {
4      string formatStr = "";
5      vector<llvm::Value*> params;
6      for (auto & arg : *(this->args))
7      {
8          llvm::Value* argValue = arg->codeGen(generator);
9          if (argValue->getType() == TheBuilder.getInt32Ty())
10         {
11             formatStr += "%d";
12         }
13     }
14 }
```

```

12     else if (argValue->getType() == TheBuilder.getInt8Ty())
13     {
14         formatStr += "%c";
15     }
16     else if (argValue->getType() == TheBuilder.getInt1Ty())
17     {
18         formatStr += "%d";
19     }
20     else if (argValue->getType()->isDoubleTy())
21     {
22         formatStr += "%lf";
23     }
24     else
25     {
26         throw logic_error("[ERROR]Invalid type to write.");
27     }
28     params.push_back(argValue);
29 }
30 if (isLineBreak)
31 {
32     formatStr += "\n";
33 }
34 auto formatConst = llvm::ConstantDataArray::getString(TheContext,
formatStr.c_str());
35 auto formatStrVar = new llvm::GlobalVariable(*generator.TheModule),
llvm::ArrayType::get(TheBuilder.getInt8Ty(), formatStr.size() + 1), true,
llvm::GlobalValue::ExternalLinkage, formatConst, ".str");
36 auto zero = llvm::Constant::getNullValue(TheBuilder.getInt32Ty());
37 llvm::Constant* indices[] = {zero, zero};
38 auto varRef = llvm::ConstantExpr::get getElementPtr(formatStrVar-
>getType()->getElementType(), formatStrVar, indices);
//    auto varRef
39 params.insert(params.begin(), varRef);
40 return TheBuilder.CreateCall(generator.printf,
41 llvm::makeArrayRef(params), "printf");
42 }
```

```

1 llvm::Value *SysProcedureCall::sysProcRead(CodeGenerator & generator)
2 {
3     string formatStr = "";
4     vector<llvm::Value*> params;
5     auto arg = this->args->front();
6     llvm::Value *argAddr, *argValue;
7     //Just common variable
8     argAddr = generator.findValue(dynamic_cast<Identifier*>(arg)-
>getName());
9     argValue = arg->CodeGen(generator);
10    if (argValue->getType() == TheBuilder.getInt32Ty())
11    {
12        formatStr += "%d";
13    }
14    else if (argValue->getType() == TheBuilder.getInt8Ty())
15    {
16        formatStr += "%c";
17    }
18    else if (argValue->getType() == TheBuilder.getInt1Ty())
19    {
```

```

20         formatStr += "%d";
21     }
22     else if (argValue->getType()->isDoubleTy())
23     {
24         formatStr += "%lf";
25     }
26     else
27     {
28         throw logic_error("[ERROR]Invalid type to read.");
29     }
30     params.push_back(argAddr);
31     params.insert(params.begin(),
32 TheBuilder.CreateGlobalStringPtr(formatStr));
33     return TheBuilder.CreateCall(generator.scnf, params, "scanf");
34 }
```

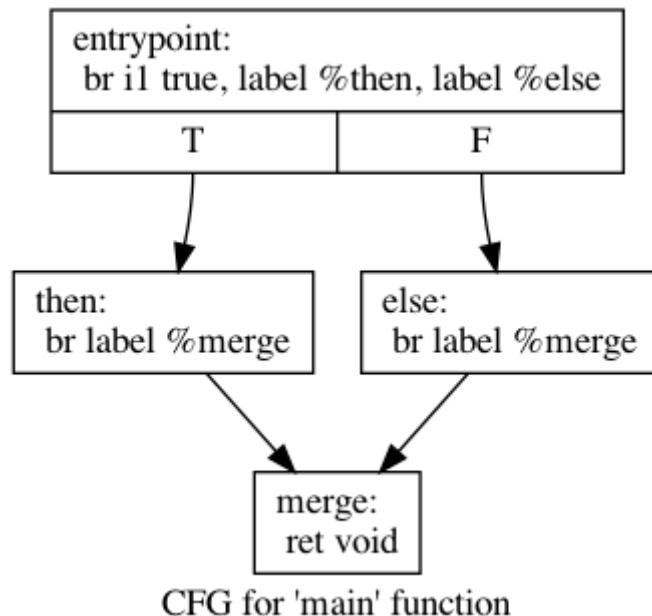
3.3.14 分支语句

3.3.14.1 if语句

if语句可以被抽象为四个基础块：

- ifCond: 条件判断基础块，计算条件表达式的值并创建跳转分支
- then: 条件为真执行的基础块，结束之后跳转到merge基础块
- else: 条件为假执行的基础块，结束之后跳转到merge基础块，对于没有else部分的分支语句我们创建一个空的else基础块来达到简化代码的目的
- merge: if语句结束之后的基础块，作为后面代码的插入点

以上基础快跳转的DAG图：



```

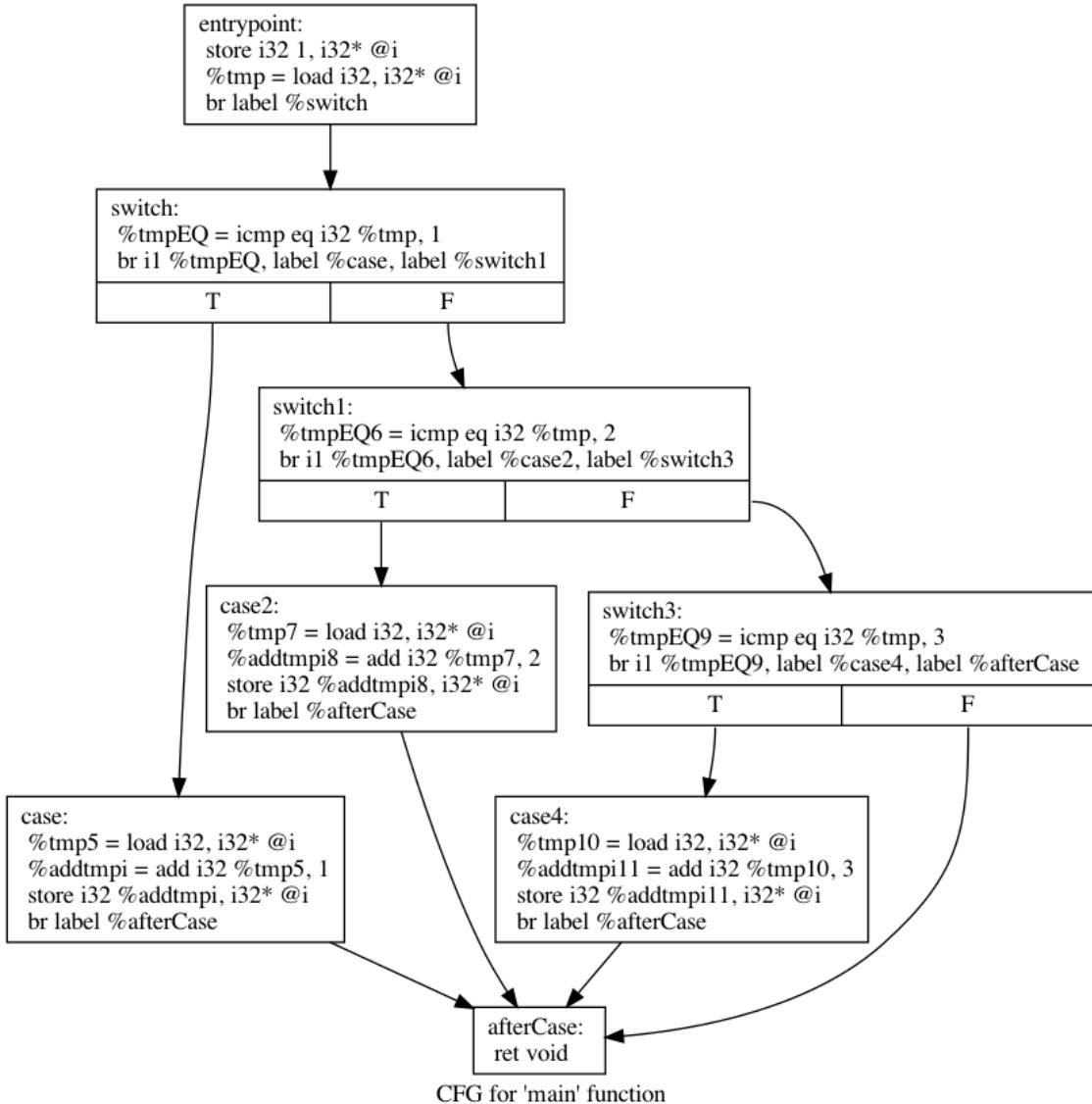
1 llvm::Value *IfStatement::CodeGen(CodeGen &generator) {
2     LOG_I("If Statement");
3     this->forward(generator);
4
5     llvm::Value *condValue = this->condition->CodeGen(generator),
6     *thenValue = nullptr, *elseValue = nullptr;
7     condValue = TheBuilder.CreateICmpNE(condValue,
8     llvm::ConstantInt::get(llvm::Type::getInt1Ty(TheContext), 0, true),
9     "ifCond");
```

```

7     llvm::Function *TheFunction = generator.getCurFunction();
8     llvm::BasicBlock *thenBB = llvm::BasicBlock::Create(TheContext, "then",
9     TheFunction);
10    llvm::BasicBlock *elseBB = llvm::BasicBlock::Create(TheContext, "else",
11    TheFunction);
12    llvm::BasicBlock *mergeBB = llvm::BasicBlock::Create(TheContext,
13    "merge", TheFunction);
14
15    //Then
16    auto branch = TheBuilder.CreateCondBr(condValue, thenBB, elseBB);
17    TheBuilder.SetInsertPoint(thenBB);
18    thenValue = this->thenStatement->CodeGen(generator);
19    TheBuilder.CreateBr(mergeBB);
20    thenBB = TheBuilder.GetInsertBlock();
21
22    //Else
23    TheBuilder.SetInsertPoint(elseBB);
24    if (this->elseStatement != nullptr)
25    {
26        elseValue = this->elseStatement->CodeGen(generator);
27    }
28    TheBuilder.CreateBr(mergeBB);
29    elseBB = TheBuilder.GetInsertBlock();
30
31    //Merge
32    TheBuilder.SetInsertPoint(mergeBB);
33
34    this->backward();
35    return branch;
36 }
```

3.3.14.2 case语句

case语句可以看作多个if语句，本实验将其抽象为多个基础块对和一个after基础块，事先计算好表达式的值之后，进入第一个基础块对，每个基础块对的第一块判断条件值是否与当前值相等，相等则跳转到第二块，第二块执行完毕后跳转到after基础块执行case语句之后的代码，否则跳转到下一对基础块对，下一个基础块对同样如此，直到最后一个基础块对时，不管是否相等都需要跳转到after基础块，以下是具有三个分支语句的case语句DAG：



```

1  llvm::Value *CaseStatement::CodeGen(CodeGen & generator) {
2      LOG_I("Case Statement");
3      this->forward(generator);
4
5      llvm::Value *cmpValue = this->value->CodeGen(generator), *condValue =
6      nullptr;
7      llvm::Function *TheFunction = generator.getCurFunction();
8      llvm::BasicBlock *switchBB = llvm::BasicBlock::Create(TheContext,
9      "switch", TheFunction);
10     llvm::BasicBlock *afterBB = llvm::BasicBlock::Create(TheContext,
11      "afterCase", TheFunction);
12     vector<llvm::BasicBlock*> caseBBS;
13     for (int i = 1; i <= this->caseExprList->size(); i++)
14     {
15         caseBBS.push_back(llvm::BasicBlock::Create(TheContext, "case",
16         TheFunction));
17     }
18
19     auto branch = TheBuilder.CreateBr(switchBB);
20     for (int i = 0; i < this->caseExprList->size(); i++)
21     {
22         //Switch
23         TheBuilder.SetInsertPoint(switchBB);
24     }
25 }
```

```

20     condValue = BinaryOp(cmpValue, BinaryExpression::SPL_EQUAL,
21 (*caseExprList)[i]->value->codeGen(generator));
22     if (i < this->caseExprList->size() - 1)
23     {
24         TheBuilder.CreateCondBr(condValue, caseBBs[i], caseBBs[i + 1]);
25     }
26     else
27     {
28         TheBuilder.CreateCondBr(condValue, caseBBs[i], afterBB);
29     }
30     switchBB = TheBuilder.GetInsertBlock();
31
32     //Case
33     TheBuilder.SetInsertPoint(caseBBs[i]);
34     (*caseExprList)[i]->codeGen(generator);
35     TheBuilder.CreateBr(afterBB);
36
37     //Default
38     TheBuilder.SetInsertPoint(switchBB);
39     TheBuilder.CreateBr(afterBB);
40
41     //After
42     TheBuilder.SetInsertPoint(afterBB);
43     this->backword();
44
45     return branch;

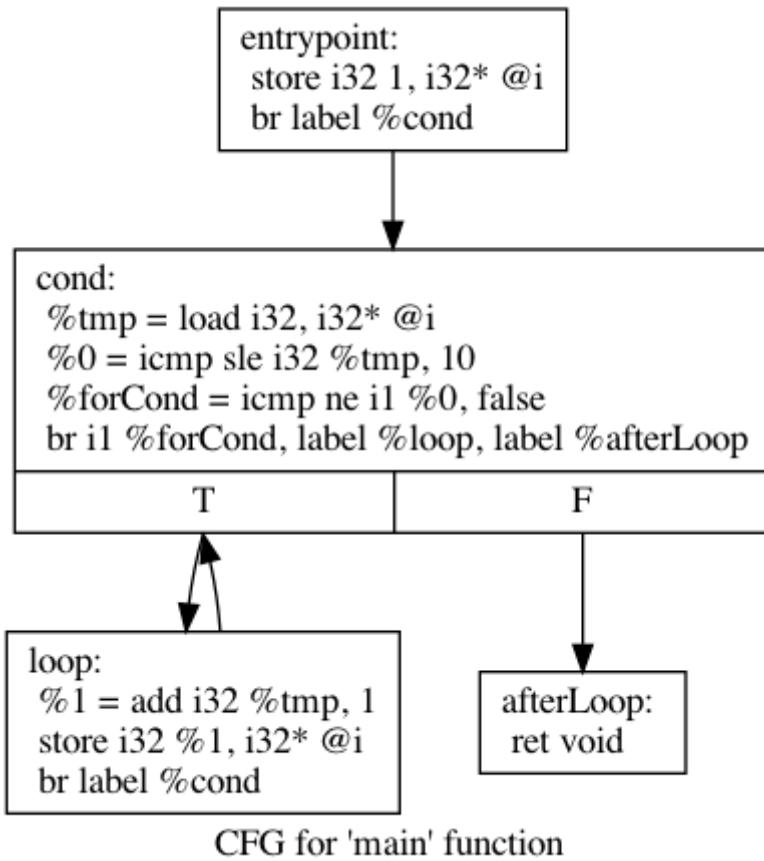
```

3.3.15 循环语句

3.3.15.1 for语句

for语句可以被抽象为三个基础块：

- forCond：计算条件表达式的值，判断循环条件，为真进入loop块，否则进入afterLoop块
- loop：条件为真时执行的循环体基础块，完成循环变量的递增或递减，之后跳转到forCond基础块
- afterLoop：跳出循环之后执行的基础块，作为之后代码的插入点



CFG for 'main' function

```

1  llvm::Value *ForStatement::CodeGen(CodeGenerator & generator) {
2      LOG_I("For Statement");
3      this->forward(generator);
4      //Init
5      llvm::Function *TheFunction = generator.getCurFunction();
6      llvm::Value* startValue = this->value->CodeGen(generator);
7      llvm::Value* endValue = this->step->CodeGen(generator);
8      llvm::Value *condValue = nullptr, *curValue = nullptr, *varValue =
9      generator.findValue(this->var->getName());
10     TheBuilder.CreateStore(startValue, varValue);
11
12     llvm::BasicBlock *condBB = llvm::BasicBlock::Create(TheContext, "cond",
13     TheFunction);
14     llvm::BasicBlock *loopBB = llvm::BasicBlock::Create(TheContext, "loop",
15     TheFunction);
16     llvm::BasicBlock *afterBB = llvm::BasicBlock::Create(TheContext,
17     "afterLoop", TheFunction);
18
19     // curValue = TheBuilder.CreateLoad(varValue, this->var->getName());
20     curValue = this->var->CodeGen(generator);
21     if (this->isAdd)
22     {
23         condValue = TheBuilder.CreateICmpSLE(curValue, endValue);
24     }
25     else
26     {
27         condValue = TheBuilder.CreateICmpSGE(curValue, endValue);
28     }

```

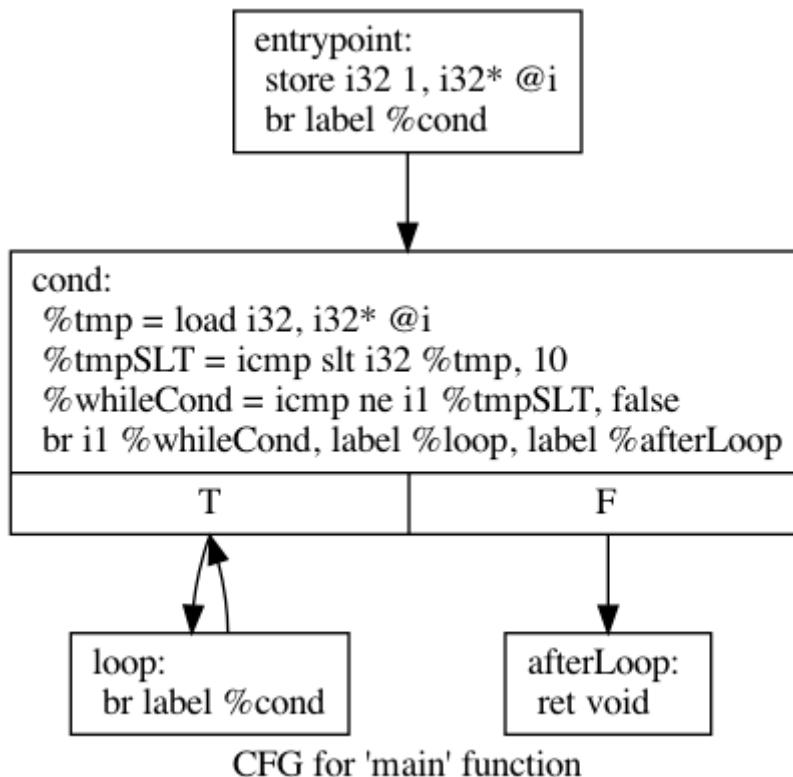
```

28     condValue = TheBuilder.CreateICmpNE(condValue,
29     llvm::ConstantInt::get(llvm::Type::getInt1Ty(TheContext), 0, true),
30     "forCond");
31
32     //Loop
33     TheBuilder.SetInsertPoint(loopBB);
34     this->stmt->codeGen(generator);
35     llvm::Value *tmpValue = TheBuilder.CreateAdd(curValue,
36     TheBuilder.getInt32(this->isAdd ? 1 : -1));
37     TheBuilder.CreateStore(tmpValue, varValue);
38     TheBuilder.CreateBr(condBB);
39     loopBB = TheBuilder.GetInsertBlock();
40
41     //After
42     TheBuilder.SetInsertPoint(afterBB);
43     this->backward();
44     return branch;
45 }

```

3.3.15.2 while语句

while语句类似for语句，区别在于loop基础块中没有循环变量的递增、递减操作：



```

1  llvm::Value *WhileStatement::codeGen(CodeGenerator & generator) {
2      LOG_I("While Statement");
3      this->forward(generator);
4      llvm::Function *TheFunction = generator.getCurFunction();
5      llvm::BasicBlock *condBB = llvm::BasicBlock::create(TheContext, "cond",
6      TheFunction);
6      llvm::BasicBlock *loopBB = llvm::BasicBlock::create(TheContext, "loop",
TheFunction);

```

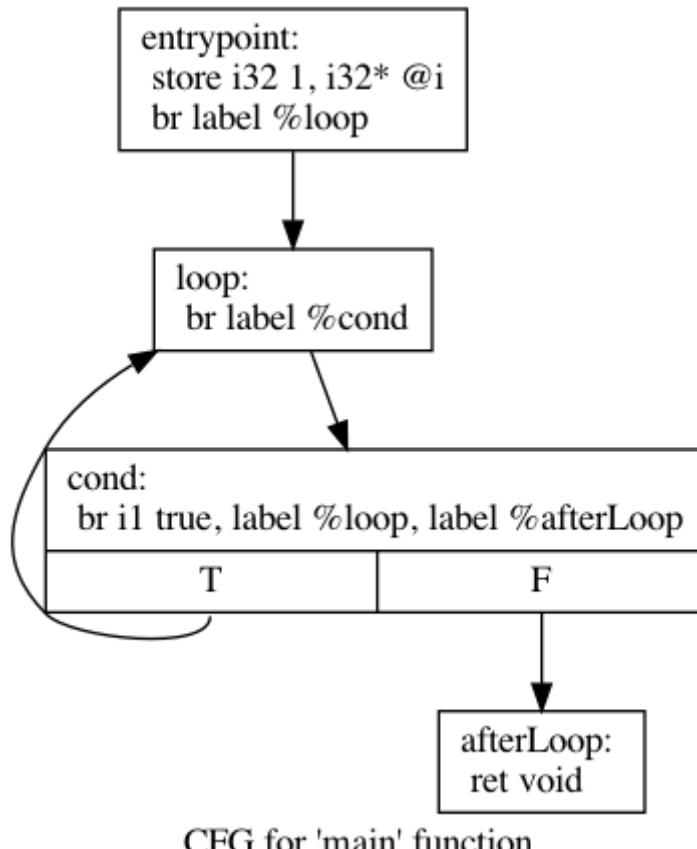
```

7     llvm::BasicBlock *afterBB = llvm::BasicBlock::Create(TheContext,
8         "afterLoop", TheFunction);
9
10    //Cond
11    TheBuilder.CreateBr(condBB);
12    TheBuilder.SetInsertPoint(condBB);
13    llvm::Value *condValue = this->condition->CodeGen(generator);
14    condValue = TheBuilder.CreateICmpNE(condValue,
15        llvm::ConstantInt::get(llvm::Type::getInt1Ty(TheContext), 0, true),
16        "whileCond");
17    auto branch = TheBuilder.CreateCondBr(condValue, loopBB, afterBB);
18    condBB = TheBuilder.GetInsertBlock();
19
20    //Loop
21    TheBuilder.SetInsertPoint(loopBB);
22    this->stmt->CodeGen(generator);
23    TheBuilder.CreateBr(condBB);
24
25    //After
26    TheBuilder.SetInsertPoint(afterBB);
27    this->backward();
28    return branch;
29 }

```

3.3.15.3 repeat语句

repeat语句与while语句类似，区别在于先进入loop基础块，因此loop块一定会执行：



CFG for 'main' function

```

1  llvm::Value *RepeatStatement::CodeGen(CodeGen & generator) {
2      LOG_I("Repeate Statement");
3      this->forward(generator);
4

```

```

5     llvm::Function *TheFunction = generator.getCurFunction();
6     llvm::BasicBlock *condBB = llvm::BasicBlock::create(TheContext, "cond",
7     TheFunction);
8     llvm::BasicBlock *loopBB = llvm::BasicBlock::create(TheContext, "loop",
9     TheFunction);
10    llvm::BasicBlock *afterBB = llvm::BasicBlock::create(TheContext,
11        "afterLoop", TheFunction);

12
13 //Loop
14 TheBuilder.CreateBr(loopBB);
15 TheBuilder.SetInsertPoint(loopBB);
16 for (auto & stmt : *(this->repeatStatement))
17 {
18     stmt->CodeGen(generator);
19 }
20 TheBuilder.CreateBr(condBB);
21 loopBB = TheBuilder.GetInsertBlock();

22 //Cond
23 TheBuilder.SetInsertPoint(condBB);
24 llvm::Value *condValue = this->condition->CodeGen(generator);
25 condValue = TheBuilder.CreateICmpNE(condValue,
26     llvm::ConstantInt::get(llvm::Type::getInt1Ty(TheContext), 0, true),
27     "repeateCond");
28     auto branch = TheBuilder.CreateCondBr(condValue, loopBB, afterBB);

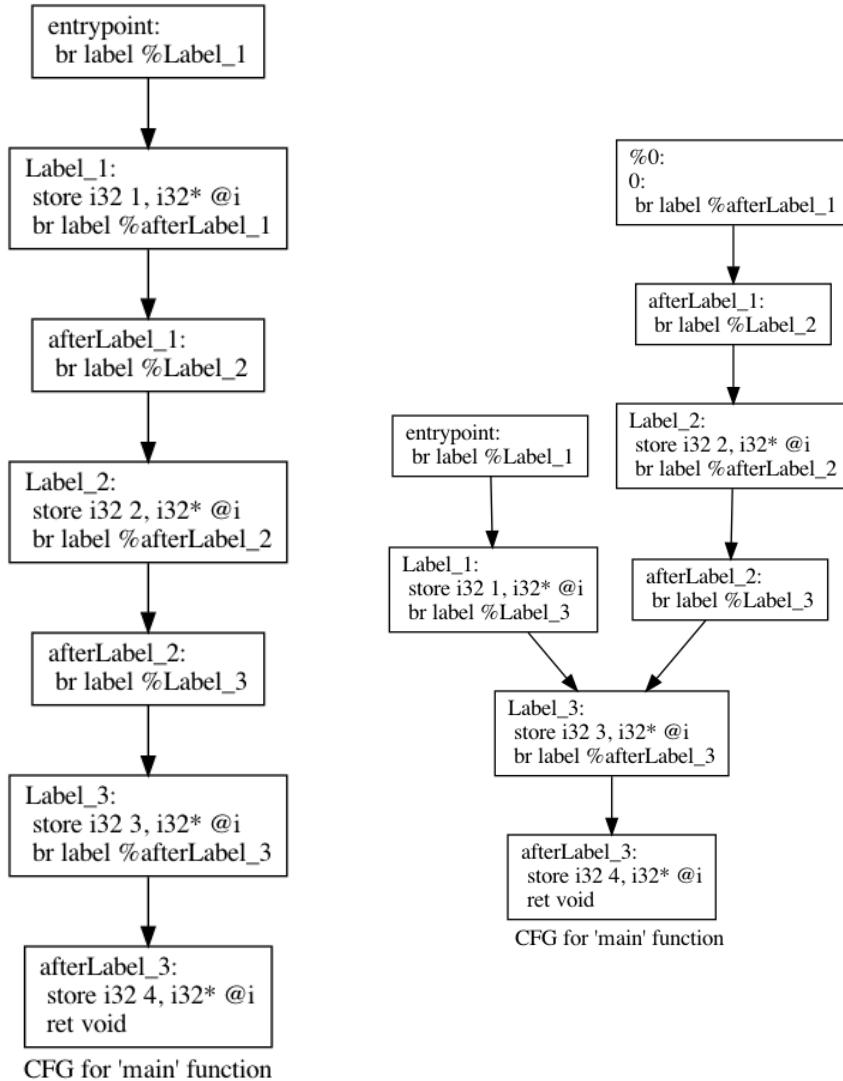
29 //After
30 TheBuilder.SetInsertPoint(afterBB);
31 this->backword();
32 return branch;
33 }
```

3.3.16 goto语句

goto语句要求记录语句的标签，所以在CodeGenerotor中我们使用一个基础块指针向量来记录所有带标签块的地址，向量的下标就作为标签进行索引，在语法分析部分会将带标签语句的标签置为相应值，无标签语句的标签默认为-1。

所有继承自Statement类的语句均可携带标签，因此在这些语句代码生成的过程中需要判断是否携带标签，是则需要构建新的基础块作为代码插入点并将基础块记录在基础块向量中，在代码生成完毕需要从当前基础块跳转到正常的基础块进行代码生成，本实验直接提供一个after基础块作为新的插入点。

以下是三个带标签的语句在标签1语句使用 `goto 3;` 代码前后的DAG变化：



为了实现插入基础块和恢复新的插入点，本实验定义了forward和backward两个辅助函数：

- forward：带标签语句调用forward，负责生成新的标签基础块和新的after基础块，跳转到新的基础块并设置其为插入点

```

1 void Statement::forward(CodeGenerator & generator)
2 {
3     LLVM::Function *TheFunction = generator.getCurFunction();
4     if (this->label >= 0)
5     {
6         if (generator.labelBlock[label] == nullptr)
7         {
8             generator.labelBlock[label] =
9                 LLVM::BasicBlock::Create(TheContext, "Label_" + to_string(label),
10                TheFunction);
11         }
12         if (this->afterBB == nullptr)
13         {
14             this->afterBB = LLVM::BasicBlock::Create(TheContext,
15                "afterLabel_" + to_string(this->label), TheFunction);
16         }
17         TheBuilder.CreateBr(generator.labelBlock[label]);
18         TheBuilder.SetInsertPoint(generator.labelBlock[label]);
19     }
20 }

```

- backward: 带标签语句调用backward, 负责跳转到after基础块并将其设置为代码插入点

```

1 void Statement::backward()
2 {
3     if (this->label >= 0 && afterBB != nullptr)
4     {
5         TheBuilder.CreateBr(this->afterBB);
6         TheBuilder.SetInsertPoint(this->afterBB);
7     }
8 }
```

现在goto语句代码生成时, 先判断标签是否有效, 是则跳转到该标签对应的基础块, 需要注意的是:

- 跳转到的标签在此标签语句之后: 先判断该标签对应基础块是否为空, 是则提前创建一块作为dummy基础块, 等真正的标签语句生成时直接使用该块即可
- goto语句本身也可携带标签: 给goto语句套上forward和backward函数即可

```

1 llvm::Value *GotoStatement::CodeGen(CodeGen &generator) {
2     LOG_I("Goto Statement");
3     this->forward(generator);
4     llvm::Value *res = nullptr;
5     if (generator.labelBlock[this->toLabel] == nullptr)
6     {
7         generator.labelBlock[this->toLabel] =
8             llvm::BasicBlock::Create(TheContext, "Label_" + to_string(this->toLabel),
9             generator.getCurFunction());
10    res = TheBuilder.CreateBr(generator.labelBlock[this->toLabel]);
11    this->backward();
12    return res;
13 }
```

第四章 优化考虑

常量折叠

常量折叠是一种非常常见和重要的优化, 在LLVM生成IR的时候支持通过IRBuilder进行常量折叠优化, 而不需要通过AST中的任何额外操作来提供支持。在调用IRBuilder时它会自动检查是否存在常量折叠的机会, 如果有则直接返回常量而不是创建计算指令。以下是一个常量折叠的例子:

- 优化前

```

1 define double @test(double %x) {
2 entry:
3     %addtmp = fadd double 2.000000e+00, 1.000000e+00
4     %addtmp1 = fadd double %addtmp, %x
5     ret double %addtmp1
6 }
```

- 优化后

```
1 define double @test(double %x) {
2     entry:
3         %addtmp = fadd double 3.000000e+00, %x
4         ret double %addtmp
5 }
```

第五章 代码生成

5.1 选择目标机器

LLVM 支持本地交叉编译。我们可以将代码编译为当前计算机的体系结构，也可以像针对其他体系结构一样轻松地进行编译。LLVM 提供了 `sys::getDefaultTargetTriple`，它返回当前计算机的目标三元组：

```
1 auto TargetTriple = sys::getDefaultTargetTriple();
```

在获取Target前，初始化所有目标以发出目标代码：

```
1 InitializeAllTargetInfos();
2 InitializeAllTargets();
3 InitializeAllTargetMCs();
4 InitializeAllAsmParsers();
5 InitializeAllAsmPrinters();
```

使用目标三元组获得 Target：

```
1 std::string Error;
2 auto Target = TargetRegistry::lookupTarget(TargetTriple, Error);
3
4 // Print an error and exit if we couldn't find the requested target.
5 // This generally occurs if we've forgotten to initialise the
6 // TargetRegistry or we have a bogus target triple.
7 if (!Target) {
8     errs() << Error;
9     return 1;
10 }
```

`TargetMachine` 类提供了我们要定位的机器的完整机器描述：

```
1 auto CPU = "generic";
2 auto Features = "";
3
4 TargetOptions opt;
5 auto RM = Optional<Reloc::Model>();
6 auto TargetMachine = Target->createTargetMachine(TargetTriple, CPU, Features,
opt, RM);
```

5.2 配置 Module

配置模块，以指定目标和数据布局，可以方便了解目标和数据布局。

```
1 generator.TheModule->setDataLayout(TargetMachine->createDataLayout());
2 generator.TheModule->setTargetTriple(TargetTriple);
```

5.3 生成目标代码

- 先定义要将文件写入的位置

```
1 auto Filename = "output.o";
2 std::error_code EC;
3 raw_fd_ostream dest(Filename, EC, sys::fs::OF_None);
4
5 if (EC) {
6     errs() << "Could not open file: " << EC.message();
7     return 1;
8 }
```

- 定义一个发出目标代码的过程，然后运行该 pass

```
1 legacy::PassManager pass;
2 auto FileType = TargetMachine::CGFT_ObjectFile;
3
4 if (TargetMachine->addPassesToEmitFile(pass, dest, nullptr, FileType)) {
5     errs() << "TargetMachine can't emit a file of this type";
6     return 1;
7 }
8
9 pass.run(*generator.TheModule);
10 dest.flush();
```

更简便的方法是使用LLVM自带的工具套件：llvm-as和llc

- llvm-as可以将生成的IR文件编译为以.bc为后缀的字节码
- llc可以将字节码编译成以.s为后缀的汇编代码

```
1 llvm-as spl.ll -o spl.bc
2 llc -march=x86_64 spl.bc -o spl.s
```

第六章 测试案例

6.1 数据类型测试

6.1.1 内置类型测试

- 测试代码

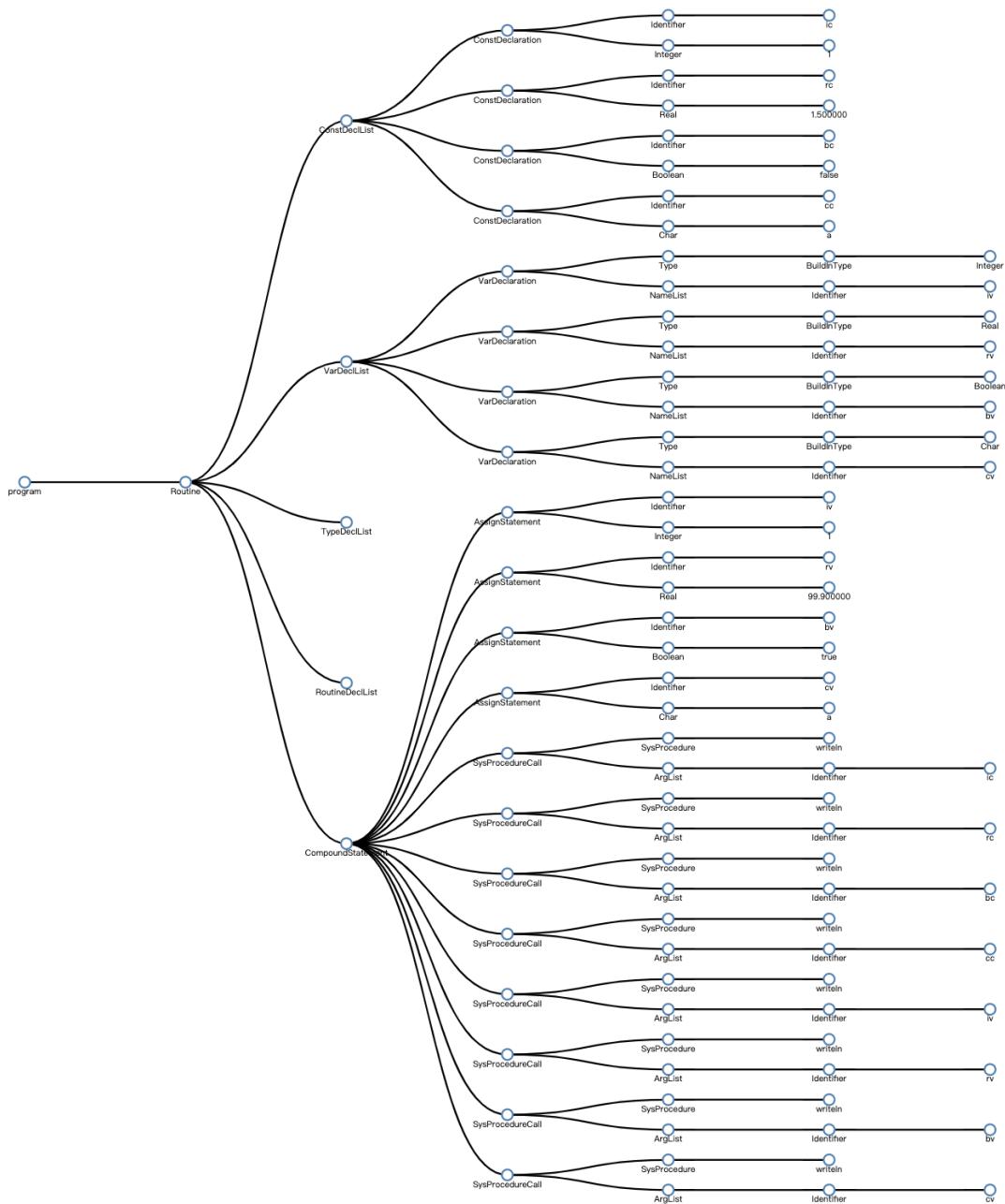
```
1 program buildInTypeTest;
2 const
3     ic = 1;
4     rc = 1.5;
5     bc = false;
6     cc = 'a';
7 var
8     iv : integer;
9     rv : real;
```

```

10     bv : boolean;
11     cv : char;
12
13 begin
14     iv := 1;
15     rv := 99.9;
16     bv := true;
17     cv := 'a';
18     writeln(ic);
19     writeln(rc);
20     writeln(bc);
21     writeln(cc);
22     writeln(iv);
23     writeln(rv);
24     writeln(bv);
25     writeln(cv);
26 end
27 .

```

- AST



- IR

```

1 ; ModuleID = 'main'
2 source_filename = "main"
3
4 @ic = constant i32 1
5 @rc = constant double 1.500000e+00
6 @bc = constant i1 false
7 @cc = constant i8 97
8 @iv = global i32 0
9 @rv = global double 0.000000e+00
10 @bv = global i1 false
11 @cv = global i8 0
12 @.str = constant [4 x i8] c"%d\0A\00"
13 @.str.1 = constant [5 x i8] c"%lf\0A\00"
14 @.str.2 = constant [4 x i8] c"%d\0A\00"
15 @.str.3 = constant [4 x i8] c"%c\0A\00"
16 @.str.4 = constant [4 x i8] c"%d\0A\00"
17 @.str.5 = constant [5 x i8] c"%lf\0A\00"
18 @.str.6 = constant [4 x i8] c"%d\0A\00"
19 @.str.7 = constant [4 x i8] c"%c\0A\00"
20
21 define internal void @main() {
22 entrypoint:
23     store i32 1, i32* @iv
24     store double 9.990000e+01, double* @rv
25     store i1 true, i1* @bv
26     store i8 97, i8* @cv
27     %tmp = load i32, i32* @ic
28     %printf = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([4 x
29 i8], [4 x i8]* @.str, i32 0, i32 0), i32 %tmp)
30     %tmp1 = load double, double* @rc
31     %printf2 = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([5 x
32 i8], [5 x i8]* @.str.1, i32 0, i32 0), double %tmp1)
33     %tmp3 = load i1, i1* @bc
34     %printf4 = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([4 x
35 i8], [4 x i8]* @.str.2, i32 0, i32 0), i1 %tmp3)
36     %tmp5 = load i8, i8* @cc
37     %printf6 = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([4 x
38 i8], [4 x i8]* @.str.3, i32 0, i32 0), i8 %tmp5)
39     %tmp7 = load i32, i32* @iv
40     %printf8 = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([4 x
41 i8], [4 x i8]* @.str.4, i32 0, i32 0), i32 %tmp7)
42     %tmp9 = load double, double* @rv
43     %printf10 = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([5 x
44 i8], [5 x i8]* @.str.5, i32 0, i32 0), double %tmp9)
45     %tmp11 = load i1, i1* @bv
46     %printf12 = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([4 x
47 i8], [4 x i8]* @.str.6, i32 0, i32 0), i1 %tmp11)
48     %tmp13 = load i8, i8* @cv
49     %printf14 = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([4 x
50 i8], [4 x i8]* @.str.7, i32 0, i32 0), i8 %tmp13)
51     ret void
52 }
53
54 declare i32 @printf(i8*, ...)

```

- 汇编指令

```

1      .section    __TEXT,__text,regular,pure_instructions
2      .macosx_version_min 10, 15
3      .p2align    4, 0x90          ## -- Begin function main
4      _main:                      ## @main
5          .cfi_startproc
6      ## %bb.0:                      ## %entrypoint
7          pushq    %rax
8          .cfi_def_cfa_offset 16
9          movl    $1, _iv(%rip)
10         movabsq $4636730254480218522, %rax ## imm = 0x4058F999999999A
11         movq    %rax, _rv(%rip)
12         movb    $1, _bv(%rip)
13         movb    $97, _cv(%rip)
14         movl    _ic(%rip), %esi
15         leaq    _.str(%rip), %rdi
16         xorl    %eax, %eax
17         callq   _printf
18         movsd   _rc(%rip), %xmm0      ## xmm0 = mem[0],zero
19         leaq    _.str.1(%rip), %rdi
20         movb    $1, %al
21         callq   _printf
22         leaq    _.str.2(%rip), %rdi
23         movzb1 _bc(%rip), %esi
24         xorl    %eax, %eax
25         callq   _printf
26         leaq    _.str.3(%rip), %rdi
27         movzb1 _cc(%rip), %esi
28         xorl    %eax, %eax
29         callq   _printf
30         movl    _iv(%rip), %esi
31         leaq    _.str.4(%rip), %rdi
32         xorl    %eax, %eax
33         callq   _printf
34         movsd   _rv(%rip), %xmm0      ## xmm0 = mem[0],zero
35         leaq    _.str.5(%rip), %rdi
36         movb    $1, %al
37         callq   _printf
38         movzb1 _bv(%rip), %esi
39         leaq    _.str.6(%rip), %rdi
40         xorl    %eax, %eax
41         callq   _printf
42         movzb1 _cv(%rip), %esi
43         leaq    _.str.7(%rip), %rdi
44         xorl    %eax, %eax
45         callq   _printf
46         popq    %rax
47         retq
48         .cfi_endproc
49                                     ## -- End function
50         .section    __TEXT,__const
51         .globl    _ic                  ## @ic
52         .p2align   2
53         _ic:

```

```
54     .long    1                         ## 0x1
55
56     .globl   _rc                      ## @rc
57     .p2align 3
58 _rc:
59     .quad   4609434218613702656      ## double 1.5
60
61     .globl   _bc                      ## @bc
62 _bc:
63     .byte   0                         ## 0x0
64
65     .globl   _cc                      ## @cc
66 _cc:
67     .byte   97                        ## 0x61
68
69     .globl   _iv                      ## @iv
70 .zerofill __DATA,__common,_iv,4,2
71     .globl   _rv                      ## @rv
72 .zerofill __DATA,__common,_rv,8,3
73     .globl   _bv                      ## @bv
74 .zerofill __DATA,__common,_bv,1,0
75     .globl   _cv                      ## @cv
76 .zerofill __DATA,__common,_cv,1,0
77     .globl   _.str                    ## @.str
78 _.str:
79     .asciz  "%d"
80
81     .globl   _.str.1                  ## @.str.1
82 _.str.1:
83     .asciz  "%lf"
84
85     .globl   _.str.2                  ## @.str.2
86 _.str.2:
87     .asciz  "%d"
88
89     .globl   _.str.3                  ## @.str.3
90 _.str.3:
91     .asciz  "%c"
92
93     .globl   _.str.4                  ## @.str.4
94 _.str.4:
95     .asciz  "%d"
96
97     .globl   _.str.5                  ## @.str.5
98 _.str.5:
99     .asciz  "%lf"
100
101    .globl   _.str.6                  ## @.str.6
102 _.str.6:
103    .asciz  "%d"
104
105    .globl   _.str.7                  ## @.str.7
106 _.str.7:
107    .asciz  "%c"
108
109
110 .subsections_via_symbols
```

- 运行结果

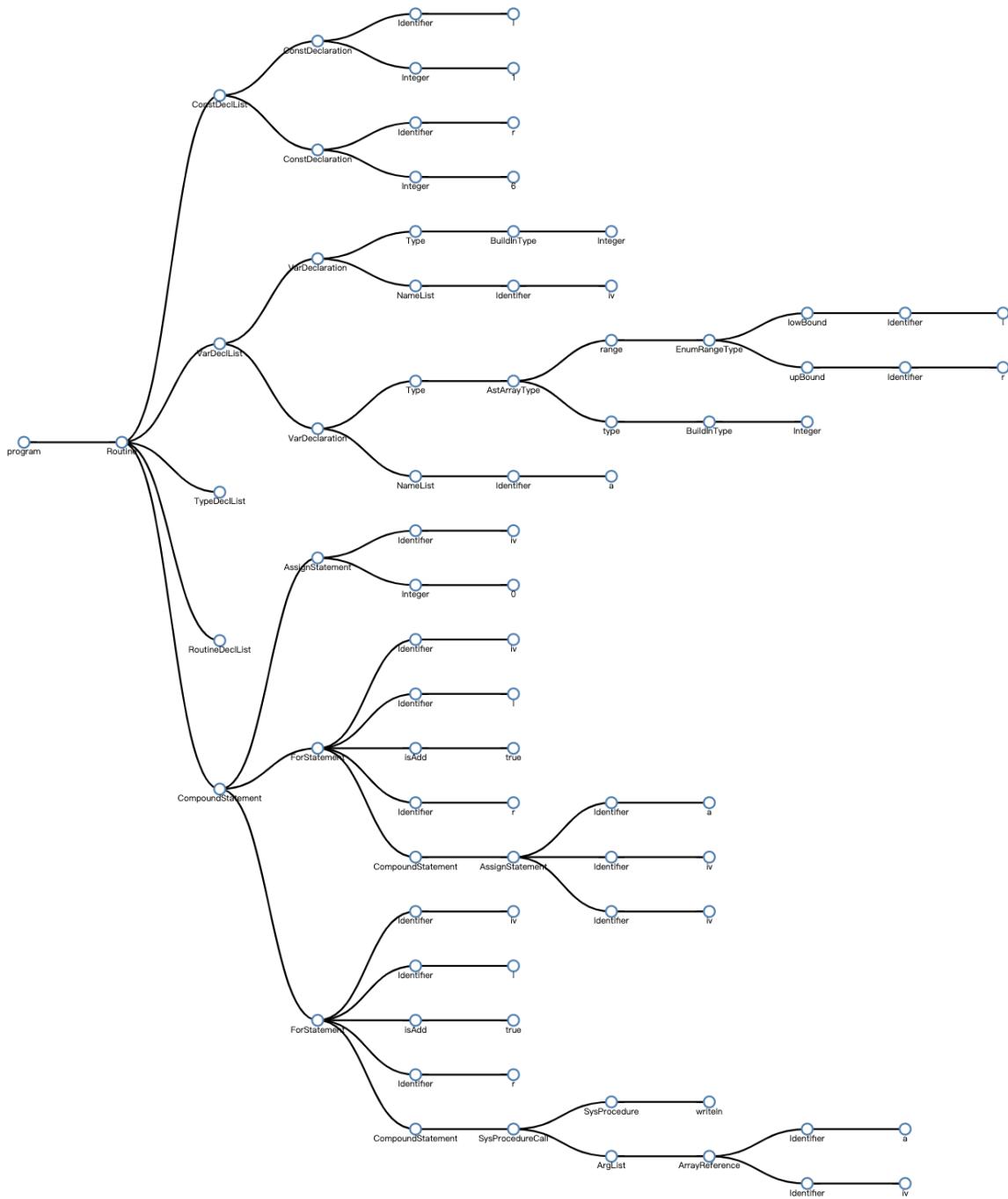
```
→ SPL-Compiler git:(master) ✘ lli spl.ll
1
1.500000
0
a
1
99.900000
1
a
```

6.1.2 数组类型测试

- 测试代码

```
1 program ArrayTest;
2 const
3     l = 1;
4     r = 6;
5 var
6     iv : integer;
7     a : array[1..r] of integer;
8
9 begin
10    iv := 0;
11    for iv := l to r do
12    begin
13        a[iv] := iv;
14    end;
15    for iv := l to r do
16    begin
17        writeln(a[iv]);
18    end;
19 end
20 .
```

- AST



- IR

```

1 ; ModuleID = 'main'
2 source_filename = "main"
3
4 @l = constant i32 1
5 @r = constant i32 6
6 @iv = global i32 0
7 @a = global [6 x i32] zeroinitializer
8 @_str = constant [4 x i8] c"%d\0A\00"
9
10 define internal void @main() {
entrypoint:
11     %tmp = load i32, i32* @r
12     %tmp1 = load i32, i32* @l
13     store i32 0, i32* @iv
14     %tmp2 = load i32, i32* @l
15 }
```

```

16 %tmp3 = load i32, i32* @r
17 store i32 %tmp2, i32* @iv
18 br label %cond
19
20 cond: ; preds = %loop,
21 %entrypoint
22 %tmp4 = load i32, i32* @iv
23 %0 = icmp sle i32 %tmp4, %tmp3
24 %forCond = icmp ne i1 %0, false
25 br i1 %forCond, label %loop, label %afterLoop
26
27 loop: ; preds = %cond
28 %tmp5 = load i32, i32* @iv
29 %tmp6 = load i32, i32* @iv
30 %subtmpi = sub i32 %tmp6, %tmp1
31 %1 = getelementptr inbounds [6 x i32], [6 x i32]* @a, i32 0, i32 %subtmpi
32 store i32 %tmp5, i32* %1
33 %2 = add i32 %tmp4, 1
34 store i32 %2, i32* @iv
35 br label %cond
36
37 afterLoop: ; preds = %cond
38 %tmp7 = load i32, i32* @1
39 %tmp8 = load i32, i32* @r
40 store i32 %tmp7, i32* @iv
41 br label %cond9
42
43 cond9: ; preds = %loop10,
44 %afterLoop
45 %tmp12 = load i32, i32* @iv
46 %3 = icmp sle i32 %tmp12, %tmp8
47 %forCond13 = icmp ne i1 %3, false
48 br i1 %forCond13, label %loop10, label %afterLoop11
49
50 loop10: ; preds = %cond9
51 %tmp14 = load i32, i32* @iv
52 %subtmpi15 = sub i32 %tmp14, %tmp1
53 %4 = getelementptr inbounds [6 x i32], [6 x i32]* @a, i32 0, i32 %subtmpi15
54 %arrRef = load i32, i32* %4
55 %printf = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([4 x i8], [4 x i8]* @.str, i32 0, i32 0), i32 %arrRef)
56 %5 = add i32 %tmp12, 1
57 store i32 %5, i32* @iv
58 br label %cond9
59
60 afterLoop11: ; preds = %cond9
61 ret void
62
63 declare i32 @printf(i8*, ...)
64 declare i32 @scanf(...)
65

```

- 汇编指令

```

1   .section    __TEXT,__text,regular,pure_instructions
2   .macosx_version_min 10, 15
3   .p2align    4, 0x90          ## -- Begin function main
4 _main:                      ## @main
5   .cfi_startproc
6   ## %bb.0:                      ## %entrypoint
7   pushq    %rbp
8   .cfi_def_cfa_offset 16
9   pushq    %r15
10  .cfi_def_cfa_offset 24
11  pushq    %r14
12  .cfi_def_cfa_offset 32
13  pushq    %r12
14  .cfi_def_cfa_offset 40
15  pushq    %rbx
16  .cfi_def_cfa_offset 48
17  .cfi_offset %rbx, -48
18  .cfi_offset %r12, -40
19  .cfi_offset %r14, -32
20  .cfi_offset %r15, -24
21  .cfi_offset %rbp, -16
22  movl    _r(%rip), %r15d
23  movl    _l(%rip), %ebx
24  movl    %ebx, _iv(%rip)
25  leaq    _a(%rip), %r12
26  .p2align    4, 0x90
27 LBB0_1:                      ## %cond
28                                     ## =>This Inner Loop Header:
29 Depth=1
30   movl    _iv(%rip), %eax
31   cmpl    %r15d, %eax
32   jg    LBB0_3
33   ## %bb.2:                      ## %loop
34                                     ## in Loop: Header=BB0_1 Depth=1
35   movl    _iv(%rip), %ecx
36   movl    %ecx, %edx
37   subl    %ebx, %edx
38   movslq %edx, %rdx
39   movl    %ecx, (%r12,%rdx,4)
40   incl    %eax
41   movl    %eax, _iv(%rip)
42   jmp    LBB0_1
43 LBB0_3:                      ## %afterLoop
44   movl    %ebx, _iv(%rip)
45   leaq    _str(%rip), %r14
46   .p2align    4, 0x90
47 LBB0_4:                      ## %cond9
48                                     ## =>This Inner Loop Header:
49 Depth=1
50   movl    _iv(%rip), %ebp
51   cmpl    %r15d, %ebp
52   jg    LBB0_6
53   ## %bb.5:                      ## %loop10
54                                     ## in Loop: Header=BB0_4 Depth=1
55   movl    _iv(%rip), %eax
56   subl    %ebx, %eax
      cltq
      movl    (%r12,%rax,4), %esi

```

```

57    movq    %r14, %rdi
58    xorl    %eax, %eax
59    callq   _printf
60    incl    %ebp
61    movl    %ebp, _iv(%rip)
62    jmp    LBB0_4
63 LBB0_6:                                ## %afterLoop11
64    popq    %rbx
65    popq    %r12
66    popq    %r14
67    popq    %r15
68    popq    %rbp
69    retq
70    .cfi_endproc
71                                     ## -- End function
72    .section   __TEXT,__const
73    .globl   _1                         ## @1
74    .p2align  2
75 _1:
76    .long    1                         ## 0x1
77
78    .globl   _r                         ## @r
79    .p2align 2
80 _r:
81    .long    6                         ## 0x6
82
83    .globl   _iv                        ## @iv
84    .zerofill __DATA,__common,_iv,4,2
85    .globl   _a                         ## @a
86    .zerofill __DATA,__common,_a,24,4
87    .globl   _.str                      ## @.str
88 _.str:
89    .asciz  "%d\n"
90
91
92    .subsections_via_symbols
93

```

- 运行结果

```

→ SPL-Complier git:(master) ✘ lli spl.ll
1
2
3
4
5
6

```

6.2 运算测试

- 测试代码

```

1 program ComputeTest;
2 const
3     ic = 1;

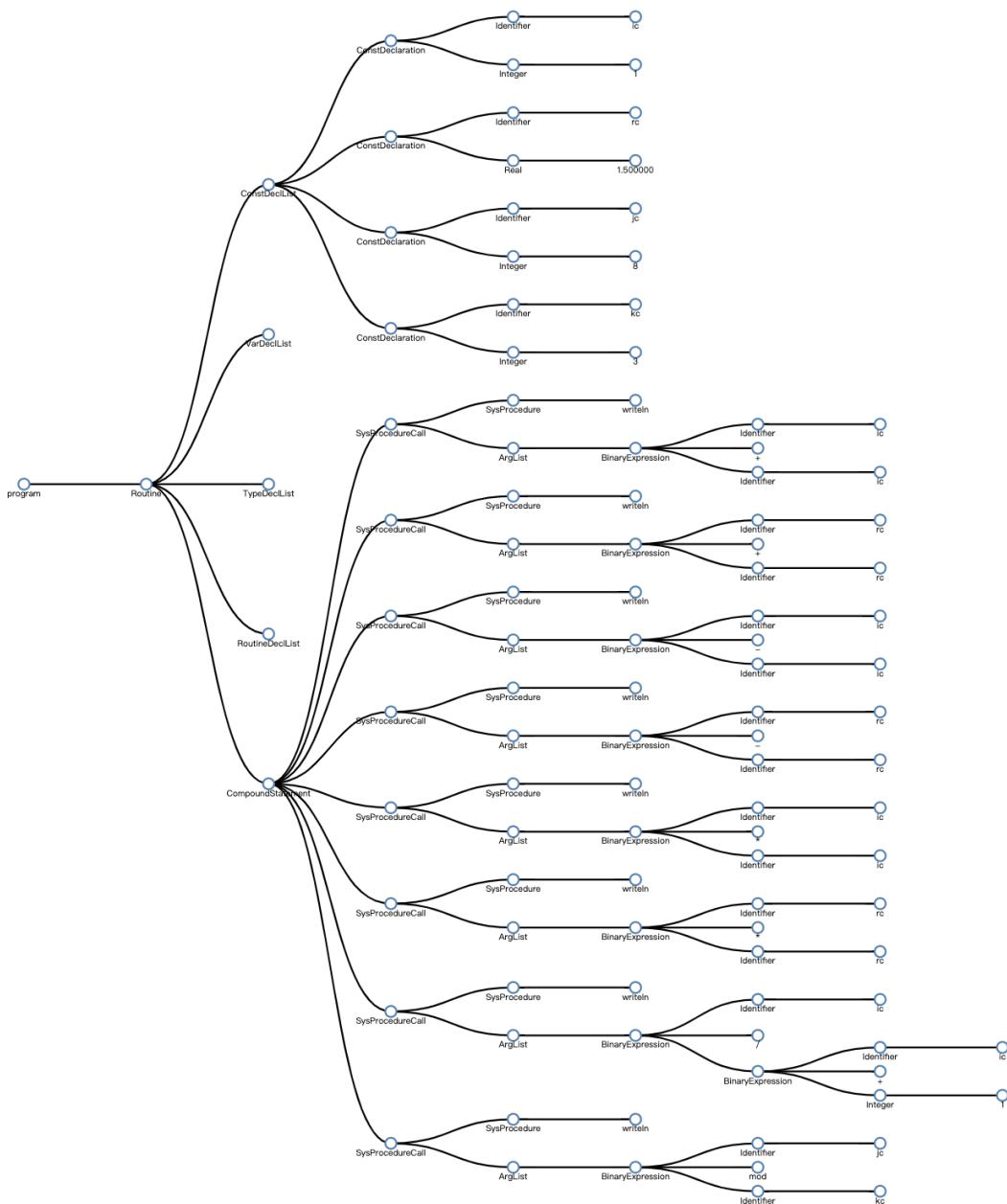
```

```

4   rc = 1.5;
5   jc = 8;
6   kc = 3;
7 begin
8   writeln(ic + ic);
9   writeln(rc + rc);
10  writeln(ic - ic);
11  writeln(rc - rc);
12  writeln(ic * ic);
13  writeln(rc * rc);
14  writeln(ic / (ic + 1));
15  writeln(jc mod kc);
16 end
17 .
18

```

- AST



- IR

```

1 ; ModuleID = 'main'
2 source_filename = "main"

```

```

3 @ic = constant i32 1
4 @rc = constant double 1.500000e+00
5 @jc = constant i32 8
6 @kc = constant i32 3
7 @.str = constant [4 x i8] c"%d\0A\00"
8 @.str.1 = constant [5 x i8] c"%1f\0A\00"
9 @.str.2 = constant [4 x i8] c"%d\0A\00"
10 @.str.3 = constant [5 x i8] c"%1f\0A\00"
11 @.str.4 = constant [4 x i8] c"%d\0A\00"
12 @.str.5 = constant [5 x i8] c"%1f\0A\00"
13 @.str.6 = constant [4 x i8] c"%d\0A\00"
14 @.str.7 = constant [4 x i8] c"%d\0A\00"
15
16
17 define internal void @main() {
18 entrypoint:
19     %tmp = load i32, i32* @ic
20     %tmp1 = load i32, i32* @ic
21     %addtmpi = add i32 %tmp, %tmp1
22     %printf = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([4 x
23         i8], [4 x i8]* @.str, i32 0, i32 0), i32 %addtmpi)
24     %tmp2 = load double, double* @rc
25     %tmp3 = load double, double* @rc
26     %addtmpf = fadd double %tmp2, %tmp3
27     %printf4 = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([5 x
28         i8], [5 x i8]* @.str.1, i32 0, i32 0), double %addtmpf)
29     %tmp5 = load i32, i32* @ic
30     %tmp6 = load i32, i32* @ic
31     %subtmpi = sub i32 %tmp5, %tmp6
32     %printf7 = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([4 x
33         i8], [4 x i8]* @.str.2, i32 0, i32 0), i32 %subtmpi)
34     %tmp8 = load double, double* @rc
35     %tmp9 = load double, double* @rc
36     %subtmpf = fsub double %tmp8, %tmp9
37     %printf10 = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([5 x
38         i8], [5 x i8]* @.str.3, i32 0, i32 0), double %subtmpf)
39     %tmp11 = load i32, i32* @ic
40     %tmp12 = load i32, i32* @ic
41     %multtmpi = mul i32 %tmp11, %tmp12
42     %printf13 = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([4 x
43         i8], [4 x i8]* @.str.4, i32 0, i32 0), i32 %multtmpi)
44     %tmp14 = load double, double* @rc
45     %tmp15 = load double, double* @rc
46     %multtmpf = fmul double %tmp14, %tmp15
47     %printf16 = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([5 x
48         i8], [5 x i8]* @.str.5, i32 0, i32 0), double %multtmpf)
49     %tmp17 = load i32, i32* @ic
50     %tmp18 = load i32, i32* @ic
51     %addtmpi19 = add i32 %tmp18, 1
52     %tmpDiv = sdiv i32 %tmp17, %addtmpi19
53     %printf20 = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([4 x
54         i8], [4 x i8]* @.str.6, i32 0, i32 0), i32 %tmpDiv)
55     %tmp21 = load i32, i32* @jc
56     %tmp22 = load i32, i32* @kc
57     %tmpSREM = srem i32 %tmp21, %tmp22
58     %printf23 = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([4 x
59         i8], [4 x i8]* @.str.7, i32 0, i32 0), i32 %tmpSREM)
60
61     ret void

```

```
53 }
54
55 declare i32 @printf(i8*, ...)
56
57 declare i32 @scanf(...)
```

- 汇编指令

```
1 .section    __TEXT,__text,regular,pure_instructions
2 .macosx_version_min 10, 15
3 .p2align    4, 0x90          ## -- Begin function main
4 _main:                                ## @main
5     .cfi_startproc
6 ## %bb.0:                                ## %entrypoint
7     pushq    %rbx
8     .cfi_def_cfa_offset 16
9     .cfi_offset %rbx, -16
10    movl    _ic(%rip), %ebx
11    leal    (%rbx,%rbx), %esi
12    leaq    _.str(%rip), %rdi
13    xorl    %eax, %eax
14    callq   _printf
15    movsd   _rc(%rip), %xmm0      ## xmm0 = mem[0],zero
16    addsd   %xmm0, %xmm0
17    leaq    _.str.1(%rip), %rdi
18    movb    $1, %al
19    callq   _printf
20    leaq    _.str.2(%rip), %rdi
21    xorl    %esi, %esi
22    xorl    %eax, %eax
23    callq   _printf
24    movsd   _rc(%rip), %xmm0      ## xmm0 = mem[0],zero
25    subsd   %xmm0, %xmm0
26    leaq    _.str.3(%rip), %rdi
27    movb    $1, %al
28    callq   _printf
29    movl    %ebx, %esi
30    imull   %ebx, %esi
31    leaq    _.str.4(%rip), %rdi
32    xorl    %eax, %eax
33    callq   _printf
34    movsd   _rc(%rip), %xmm0      ## xmm0 = mem[0],zero
35    mulsd   %xmm0, %xmm0
36    leaq    _.str.5(%rip), %rdi
37    movb    $1, %al
38    callq   _printf
39    leal    1(%rbx), %ecx
40    movl    %ebx, %eax
41    cltd
42    idivl  %ecx
43    leaq    _.str.6(%rip), %rdi
44    movl    %eax, %esi
45    xorl    %eax, %eax
46    callq   _printf
47    movl    _jc(%rip), %eax
48    cltd
49    idivl  _kc(%rip)
```

```
50    leaq    _._str.7(%rip), %rdi
51    movl    %edx, %esi
52    xorl    %eax, %eax
53    callq   _printf
54    popq    %rbx
55    retq
56    .cfi_endproc
57                                ## -- End function
58    .section   __TEXT,__const
59    .globl   _ic
60    .p2align   2
61 _ic:
62    .long    1
63                                ## 0x1
64    .globl   _rc
65    .p2align   3
66 _rc:
67    .quad    4609434218613702656
68                                ## double 1.5
69    .globl   _jc
70    .p2align   2
71 _jc:
72    .long    8
73                                ## 0x8
74    .globl   _kc
75    .p2align   2
76 _kc:
77    .long    3
78                                ## 0x3
79    .globl   _str
80    .str:
81    .asciz  "%d\n"
82
83    .globl   _str.1
84    .str.1:
85    .asciz  "%lf\n"
86
87    .globl   _str.2
88    .str.2:
89    .asciz  "%d\n"
90
91    .globl   _str.3
92    .str.3:
93    .asciz  "%lf\n"
94
95    .globl   _str.4
96    .str.4:
97    .asciz  "%d\n"
98
99    .globl   _str.5
100   .str.5:
101   .asciz  "%lf\n"
102
103   .globl   _str.6
104   .str.6:
105   .asciz  "%d\n"
106
107   .globl   _str.7
108   .str.7:
109   .asciz  "%d\n"
```

```
108 |     .str.7:  
109 |         .asciz    "%d\n"  
110 |  
111 |  
112 |     .subsections_via_symbols
```

- 运行结果

```
→ SPL-Complier git:(master) ✘ lli spl.ll  
2  
3.000000  
0  
0.000000  
1  
2.250000  
0  
2
```

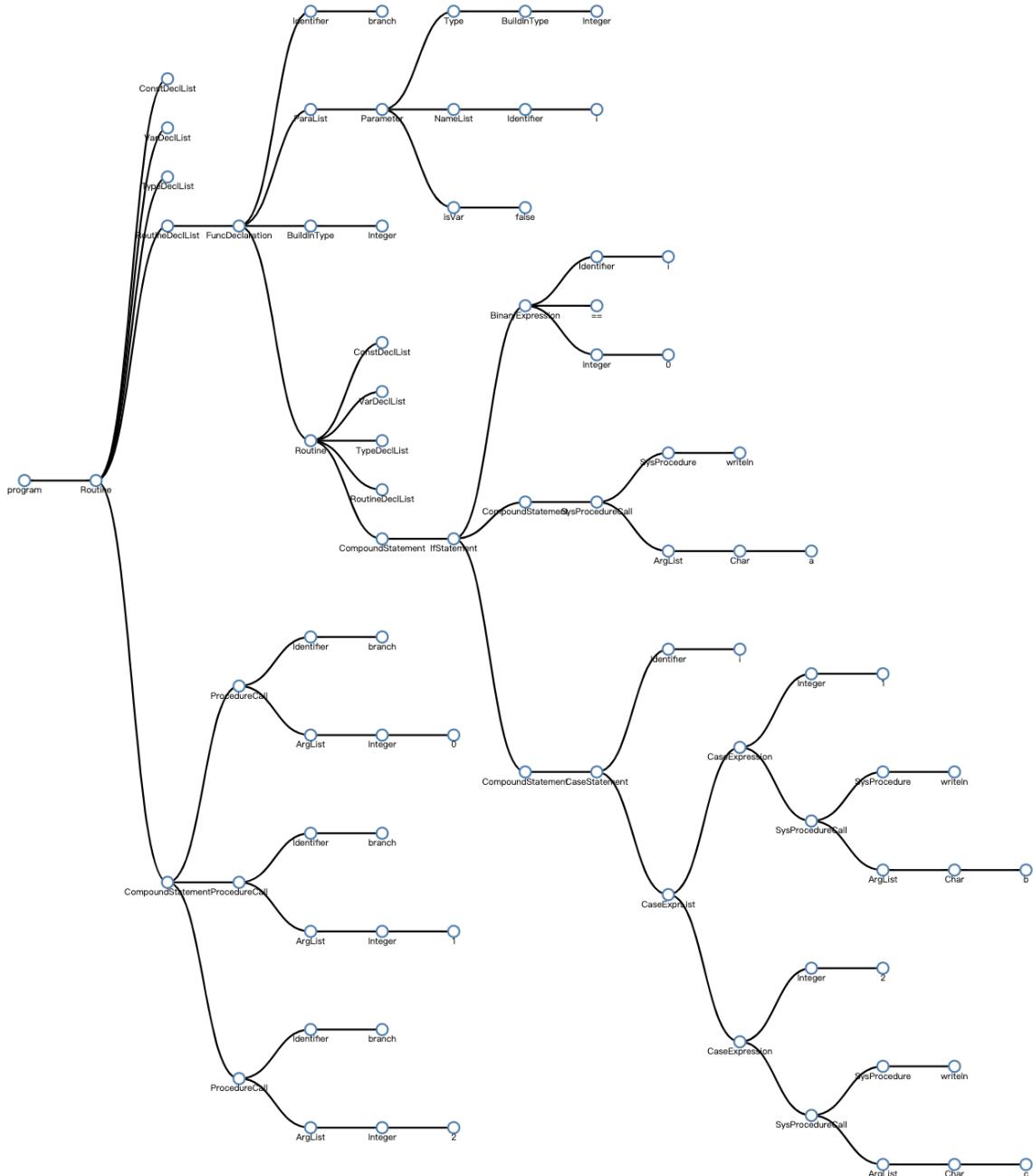
6.3 控制流测试

6.3.1 分支测试

- 测试代码

```
1 program BranchTest;  
2  
3 function branch(i : integer) : integer;  
4 begin  
5     if i = 0 then  
6         begin  
7             writeln('a');  
8         end  
9     else  
10        begin  
11            case i of  
12                1: writeln('b');  
13                2: writeln('c');  
14            end  
15            ;  
16        end  
17        ;  
18    end  
19    ;  
20  
21 begin  
22     branch(0);  
23     branch(1);  
24     branch(2);  
25 end  
26 .
```

- AST



- IR

```

1 ; ModuleID = 'main'
2 source_filename = "main"
3
4 @.str = constant [4 x i8] c"%c\0A\00"
5 @.str.1 = constant [4 x i8] c"%c\0A\00"
6 @.str.2 = constant [4 x i8] c"%c\0A\00"
7
8 define internal void @main() {
9 entrypoint:
10 %calltmp = call i32 @branch(i32 0)
11 %calltmp1 = call i32 @branch(i32 1)
12 %calltmp2 = call i32 @branch(i32 2)
13 ret void
14 }
15
16 declare i32 @printf(i8*, ...)
17
18 declare i32 @scanf(...

```

```

19
20 define internal i32 @branch(i32) {
21 entrypoint:
22     %branch = alloca i32
23     %i = alloca i32
24     store i32 %0, i32* %i
25     %tmp = load i32, i32* %i
26     %tmpEQ = icmp eq i32 %tmp, 0
27     %ifCond = icmp ne i1 %tmpEQ, false
28     br i1 %ifCond, label %then, label %else
29
30 then:                                ; preds = %entrypoint
31     %printf = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([4 x
32     i8], [4 x i8]* @.str, i32 0, i32 0), i8 97)
33     br label %merge
34
35 else:                                 ; preds = %entrypoint
36     %tmp1 = load i32, i32* %i
37     br label %switch
38
39 merge:                                ; preds = %afterCase,
40 %then
41     %tmp7 = load i32, i32* %branch
42     ret i32 %tmp7
43
44 switch:                               ; preds = %else
45     %tmpEQ3 = icmp eq i32 %tmp1, 1
46     br i1 %tmpEQ3, label %case, label %case2
47     %tmpEQ5 = icmp eq i32 %tmp1, 2
48     br i1 %tmpEQ5, label %case2, label %afterCase
49     br label %afterCase
50
51 afterCase:                            ; preds = %switch,
52 %case, %switch, %case
53     br label %merge
54
55 case:                                  ; preds = %switch
56     %printf4 = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([4 x
57     i8], [4 x i8]* @.str.1, i32 0, i32 0), i8 98)
58     br label %afterCase
59
60 case2:                                 ; preds = %switch,
61 %switch
62     %printf6 = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([4 x
63     i8], [4 x i8]* @.str.2, i32 0, i32 0), i8 99)
64     br label %afterCase
65
66 }

```

- ### • 汇编指令

```

8     .cfi_def_cfa_offset 16
9     xorl    %edi, %edi
10    callq   _branch
11    movl    $1, %edi
12    callq   _branch
13    movl    $2, %edi
14    callq   _branch
15    popq    %rax
16    retq
17    .cfi_endproc
18                                     ## -- End function
19    .p2align 4, 0x90           ## -- Begin function branch
20    _branch:                  ## @branch
21    .cfi_startproc
22    ## %bb.0:                  ## %entrypoint
23    pushq   %rax
24    .cfi_def_cfa_offset 16
25    movl    %edi, (%rsp)
26    testl   %edi, %edi
27    jne LBB1_3
28    ## %bb.1:                  ## %then
29    leaq    _._str(%rip), %rdi
30    movl    $97, %esi
31    jmp LBB1_2
32    LBB1_3:                  ## %else
33    cmpl    $1, (%rsp)
34    jne LBB1_5
35    ## %bb.4:                  ## %case
36    leaq    _._str.1(%rip), %rdi
37    movl    $98, %esi
38    jmp LBB1_2
39    LBB1_5:                  ## %case2
40    leaq    _._str.2(%rip), %rdi
41    movl    $99, %esi
42    LBB1_2:                  ## %merge
43    xorl   %eax, %eax
44    callq   _printf
45    movl    4(%rsp), %eax
46    popq    %rcx
47    retq
48    .cfi_endproc
49                                     ## -- End function
50    .section __TEXT,__const
51    .globl  _._str             ## @._str
52    _._str:
53    .asciz "%c\n"
54
55    .globl  _._str.1           ## @._str.1
56    _._str.1:
57    .asciz "%c\n"
58
59    .globl  _._str.2           ## @._str.2
60    _._str.2:
61    .asciz "%c\n"
62
63
64 .subsections_via_symbols

```

- 运行结果

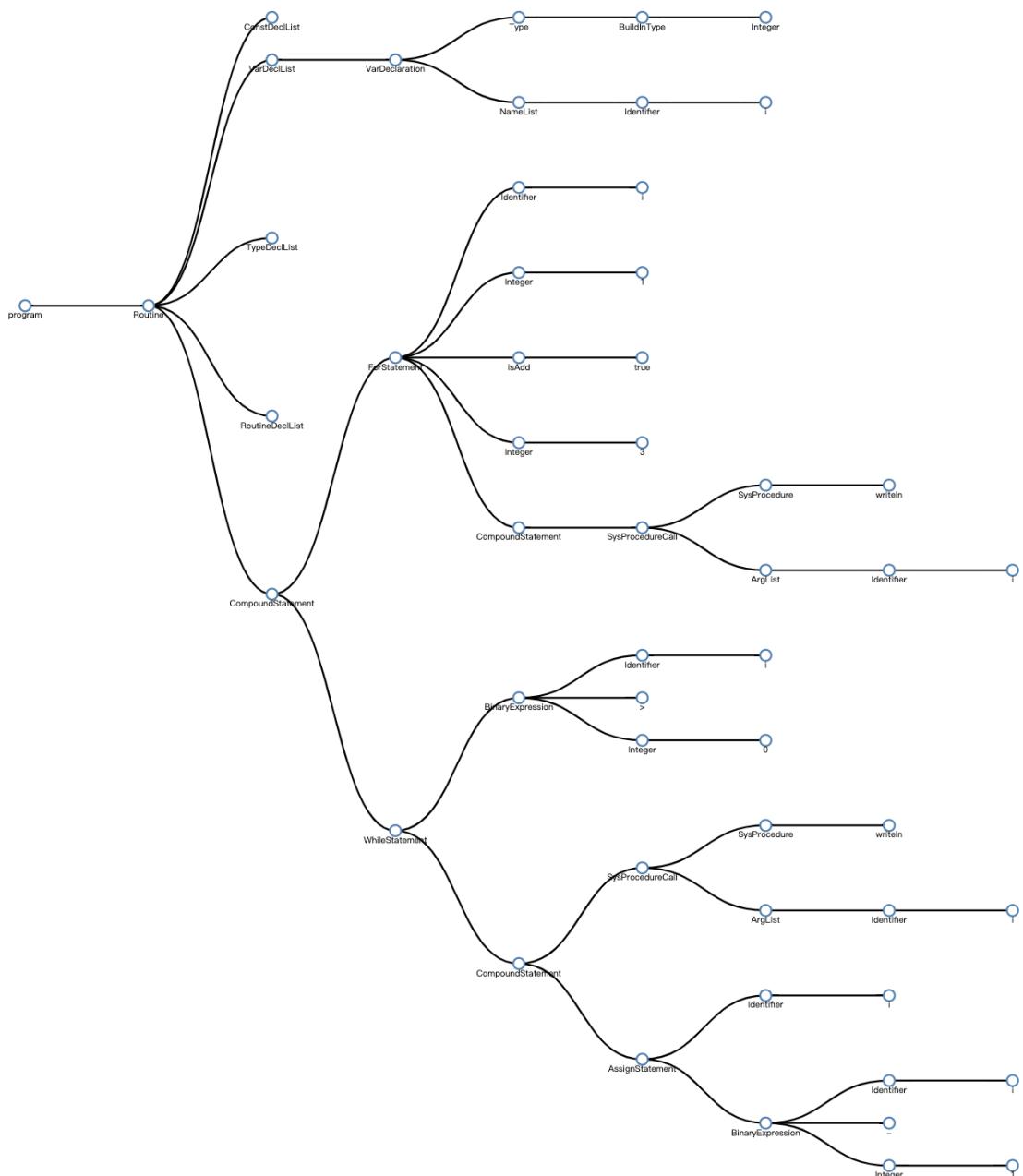
```
→ SPL-Complier git:(master) ✘ lli spl.ll
a
b
c
```

6.3.2 循环测试

- 测试代码

```
1 |
```

- AST



- IR

```
1 ; ModuleID = 'main'
2 source_filename = "main"
```

```

3
4 @i = global i32 0
5 @.str = constant [4 x i8] c"%d\0A\00"
6 @.str.1 = constant [4 x i8] c"%d\0A\00"
7
8 define internal void @main() {
9   entrypoint:
10    store i32 1, i32* @i
11    br label %cond
12
13  cond:                                ; preds = %loop,
14  %entrypoint
15    %tmp = load i32, i32* @i
16    %0 = icmp sle i32 %tmp, 3
17    %forCond = icmp ne i1 %0, false
18    br i1 %forCond, label %loop, label %afterLoop
19
20  loop:                                ; preds = %cond
21    %tmp1 = load i32, i32* @i
22    %printf = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([4 x
23      i8], [4 x i8]* @.str, i32 0, i32 0), i32 %tmp1)
24    %1 = add i32 %tmp, 1
25    store i32 %1, i32* @i
26    br label %cond
27
28
29  afterLoop:                            ; preds = %cond
30    br label %cond2
31
32  cond2:                                ; preds = %loop3,
33  %afterLoop
34    %tmp5 = load i32, i32* @i
35    %tmpSGT = icmp sgt i32 %tmp5, 0
36    %whileCond = icmp ne i1 %tmpSGT, false
37    br i1 %whileCond, label %loop3, label %afterLoop4
38
39  loop3:                                ; preds = %cond2
40    %tmp6 = load i32, i32* @i
41    %printf7 = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([4 x
42      i8], [4 x i8]* @.str.1, i32 0, i32 0), i32 %tmp6)
43    %tmp8 = load i32, i32* @i
44    %subtmpi = sub i32 %tmp8, 1
45    store i32 %subtmpi, i32* @i
46    br label %cond2
47
48  afterLoop4:                            ; preds = %cond2
49    ret void
50

```

- 汇编指令

```

1 .section  __TEXT,__text,regular,pure_instructions
2 .macosx_version_min 10, 15

```

```

3     .p2align    4, 0x90          ## -- Begin function main
4 _main:
5     .cfi_startproc
6 ## %bb.0:                      ## %entrypoint
7     pushq    %rbp
8     .cfi_def_cfa_offset 16
9     pushq    %rbx
10    .cfi_def_cfa_offset 24
11    pushq    %rax
12    .cfi_def_cfa_offset 32
13    .cfi_offset %rbx, -24
14    .cfi_offset %rbp, -16
15    movl    $1, _i(%rip)
16    leaq    _._str(%rip), %rbx
17    .p2align    4, 0x90
18 LBB0_1:                         ## %cond
19                                         ## =>This Inner Loop Header:
20                                         Depth=1
21     movl    _i(%rip), %ebp
22     cmpl    $3, %ebp
23     jg    LBB0_3
24 ## %bb.2:                         ## %loop
25                                         ##   in Loop: Header=B0_1 Depth=1
26     movl    _i(%rip), %esi
27     movq    %rbx, %rdi
28     xorl    %eax, %eax
29     callq   _printf
30     incl    %ebp
31     movl    %ebp, _i(%rip)
32     jmp    LBB0_1
33 LBB0_3:                           ## %afterLoop
34     leaq    _._str.1(%rip), %rbx
35     cmpl    $0, _i(%rip)
36     jle    LBB0_6
37     .p2align    4, 0x90
38 LBB0_5:                           ## %loop3
39                                         ## =>This Inner Loop Header:
40                                         Depth=1
41     movl    _i(%rip), %esi
42     movq    %rbx, %rdi
43     xorl    %eax, %eax
44     callq   _printf
45     decl    _i(%rip)
46     cmpl    $0, _i(%rip)
47     jg    LBB0_5
48 LBB0_6:                           ## %afterLoop4
49     addq    $8, %rsp
50     popq    %rbx
51     popq    %rbp
52     retq
53     .cfi_endproc
54                                         ## -- End function
55     .globl   _i
56     .zerofill __DATA,__common,_i,4,2
57     .section  __TEXT,__const
58     .globl   _._str                     ## @._str
59 _._str:
60     .asciz  "%d\n"

```

```
59      .globl  __str.1          ## @.str.1
60  __str.1:
61      .asciz  "%d\n"
62
63
64
65 .subsections_via_symbols
66
```

- 运行结果

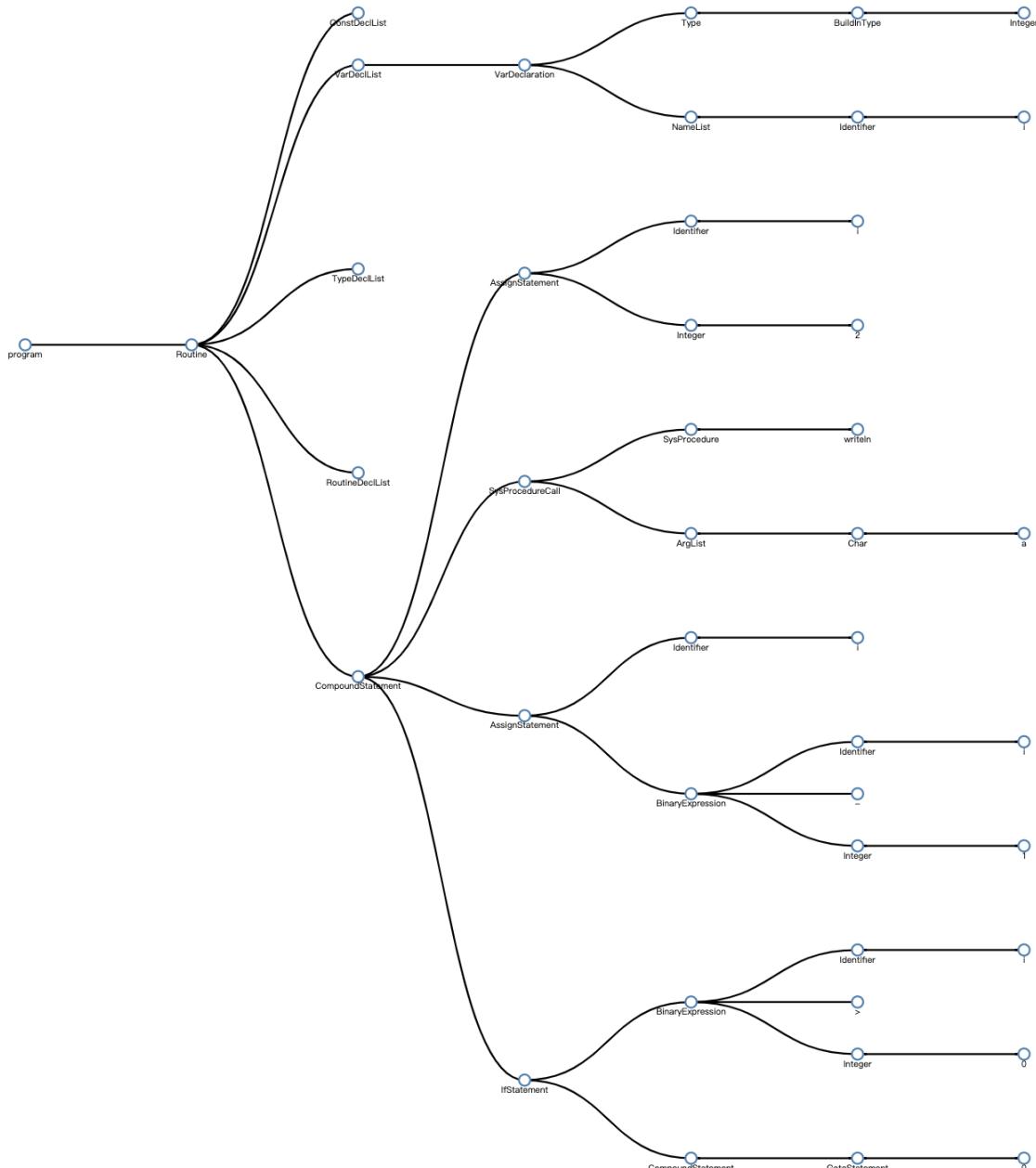
```
→ SPL-Compiler git:(master) ✘ lli spl.ll
1
2
3
4
3
2
1
```

6.3.3 Goto测试

- 测试代码

```
1 program GotoTest;
2
3 var
4     i : integer;
5
6 begin
7     i := 2;
8 0:
9     writeln('a');
10    i := i - 1;
11    if i > 0 then
12        begin
13            goto 0;
14        end
15    ;
16
17 end
18 .
```

- AST



- IR

```

1 ; ModuleID = 'main'
2 source_filename = "main"
3
4 @i = global i32 0
5 @_str = constant [4 x i8] c"%c\0A\00"
6
7 define internal void @main() {
8   entrypoint:
9     store i32 2, i32* @i
10    br label %Label_0
11
12  Label_0:                                     ; preds = %entrypoint
13  %entrypoint
14    %printf = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([4 x
15      i8], [4 x i8]* @_str, i32 0, i32 0), i8 97)
16    br label %afterLabel_0
17
18  afterLabel_0:                                ; preds = %Label_0
19    %tmp = load i32, i32* @i

```

```

18 %subtmpi = sub i32 %tmp, 1
19 store i32 %subtmpi, i32* @i
20 %tmp1 = load i32, i32* @i
21 %tmpSGT = icmp sgt i32 %tmp1, 0
22 %ifCond = icmp ne i1 %tmpSGT, false
23 br i1 %ifCond, label %then, label %else
24
25 then: ; preds = %afterLabel_0
26 br label %Label_0
27 br label %merge
28
29 else: ; preds = %afterLabel_0
30 br label %merge
31
32 merge: ; preds = %else, %then
33 ret void
34 }
35
36 declare i32 @printf(i8*, ...)
37
38 declare i32 @scanf(...)

```

- 汇编指令

```

1 .section __TEXT,__text,regular,pure_instructions
2 .macosx_version_min 10, 15
3 .p2align 4, 0x90      ## -- Begin function main
4 _main:           ## @main
5   .cfi_startproc
6 ## %bb.0:          ## %entrypoint
7   pushq %rbx
8   .cfi_def_cfa_offset 16
9   .cfi_offset %rbx, -16
10  movl $2, _i(%rip)
11  leaq _._str(%rip), %rbx
12  .p2align 4, 0x90
13 LBB0_1:          ## %Label_0
14                      ## =>This Inner Loop Header:
Depth=1
15  movq %rbx, %rdi
16  movl $97, %esi
17  xorl %eax, %eax
18  callq _printf
19  movl _i(%rip), %eax
20  decl %eax
21  movl %eax, _i(%rip)
22  testl %eax, %eax
23  jg LBB0_1
24 ## %bb.2:          ## %else
25  popq %rbx
26  retq
27  .cfi_endproc
28                      ## -- End function
29  .globl _i           ## @i
30  .zerofill __DATA,__common,_i,4,2
31  .section __TEXT,__const
32  .globl _._str         ## @_str

```

```
33 .str:
34     .asciz "%c\n"
35
36
37 .subsections_via_symbols
```

- 运行结果

```
→ SPL-Compiler git:(master) ✘ lli spl.ll
a
a
```

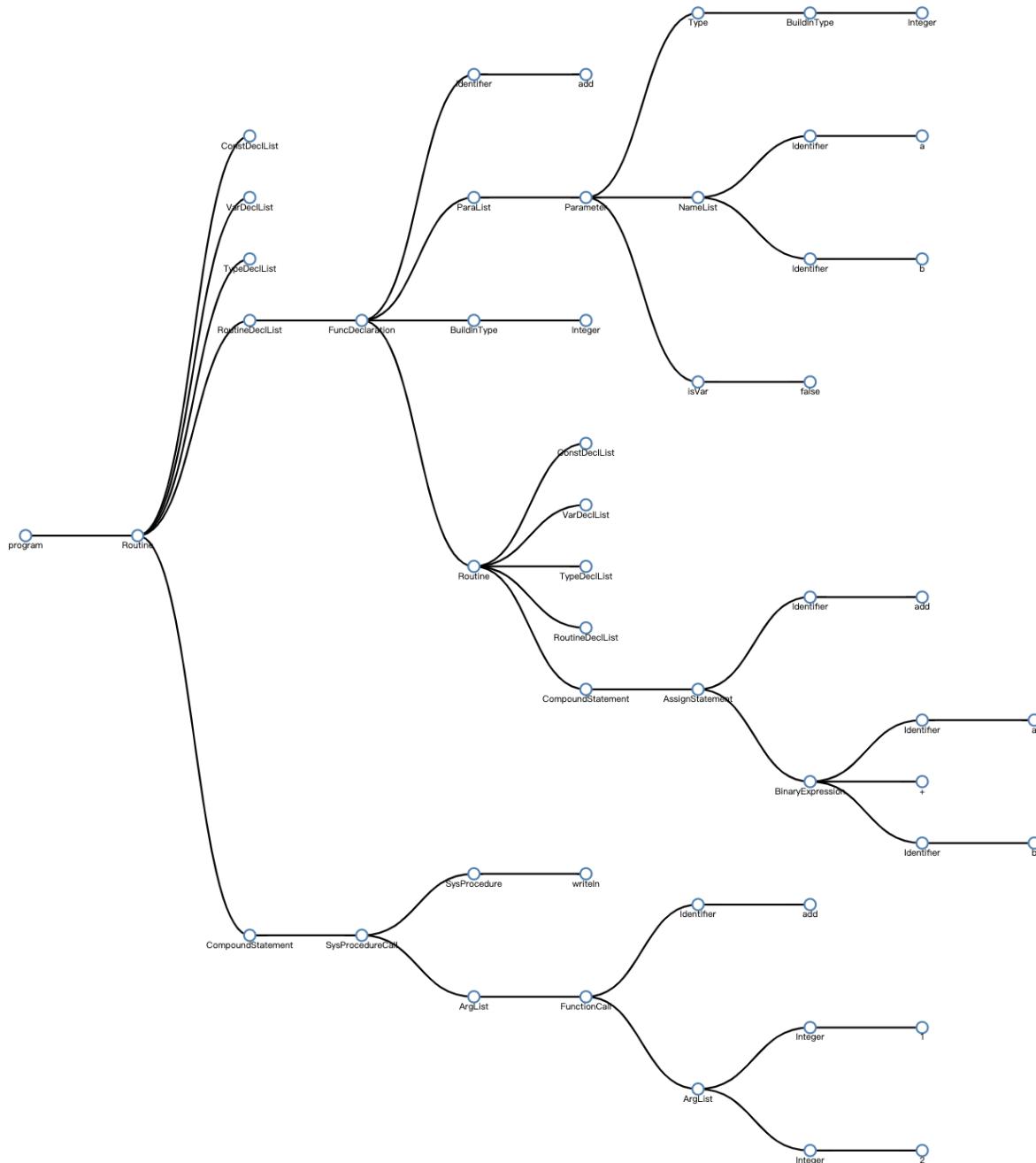
6.4 函数测试

6.4.1 简单函数测试

- 测试代码

```
1 program hello;
2
3 function add(a, b : integer) : integer;
4 begin
5     add := a + b;
6 end
7 ;
8
9 begin
10    writeln(add(1, 2));
11 end
12 .
```

- AST



- IR

```

1 ; ModuleID = 'main'
2 source_filename = "main"
3
4 @.str = constant [4 x i8] c"%d\0A\00"
5
6 define internal void @main() {
7 entrypoint:
8   %calltmp = call i32 @add(i32 1, i32 2)
9   %printf = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([4 x
i8], [4 x i8]* @.str, i32 0, i32 0), i32 %calltmp)
10  ret void
11 }
12
13 declare i32 @printf(i8*, ...)
14
15 declare i32 @scanf(...)
16
17 define internal i32 @add(i32, i32) {

```

```

18 entrypoint:
19   %add = alloca i32
20   %b = alloca i32
21   %a = alloca i32
22   store i32 %0, i32* %a
23   store i32 %1, i32* %b
24   %tmp = load i32, i32* %a
25   %tmp1 = load i32, i32* %b
26   %addtmpi = add i32 %tmp, %tmp1
27   store i32 %addtmpi, i32* %add
28   %tmp2 = load i32, i32* %add
29   ret i32 %tmp2
30 }
```

- 汇编指令

```

1    .section    __TEXT,__text,regular,pure_instructions
2    .macosx_version_min 10, 15
3    .p2align    4, 0x90          ## -- Begin function main
4    _main:                                ## @main
5      .cfi_startproc
6    ## %bb.0:                                ## %entrypoint
7      pushq    %rax
8      .cfi_def_cfa_offset 16
9      movl    $1, %edi
10     movl    $2, %esi
11     callq    _add
12     leaq    _._str(%rip), %rdi
13     movl    %eax, %esi
14     xorl    %eax, %eax
15     callq    _printf
16     popq    %rax
17     retq
18     .cfi_endproc
19                               ## -- End function
20     .p2align    4, 0x90          ## -- Begin function add
21     _add:                                ## @add
22     .cfi_startproc
23    ## %bb.0:                                ## %entrypoint
24     movl    %edi, %eax
25     movl    %edi, -12(%rsp)
26     movl    %esi, -8(%rsp)
27     addl    %esi, %eax
28     movl    %eax, -4(%rsp)
29     retq
30     .cfi_endproc
31                               ## -- End function
32     .section    __TEXT,__const
33     .globl    _._str                ## @_str
34     _._str:
35     .asciz    "%d\n"
36
37
38 .subsections_via_symbols
```

- 运行结果

→ SPL-Compiler git:(master) ✘ lli spl.ll

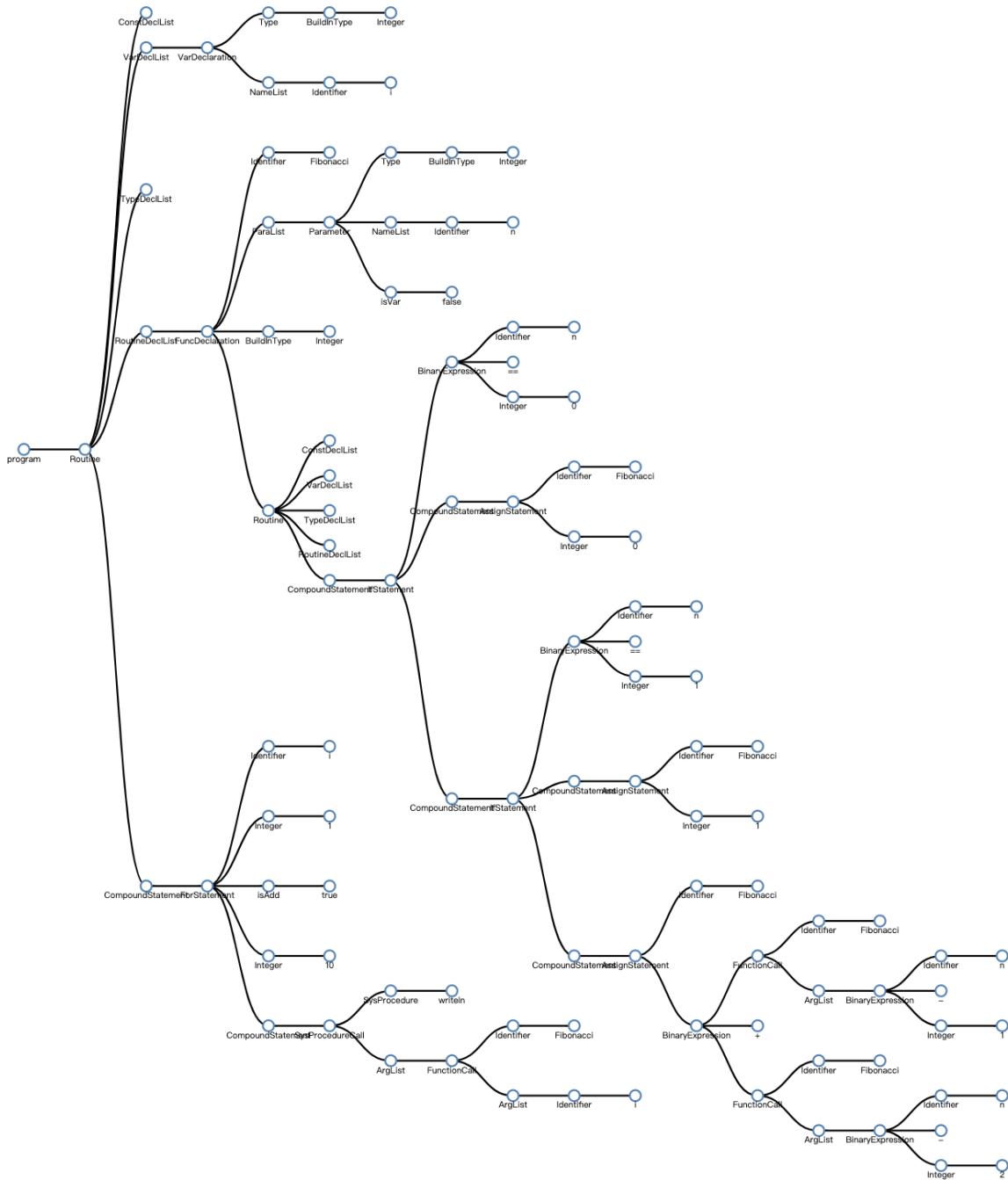
3

6.4.2 递归函数测试

- 测试代码

```
1 program RecursiveFunctionTest;
2
3 var
4     i : integer;
5
6 function Fibonacci(n : integer) : integer;
7 begin
8     if n = 0 then
9         begin
10            Fibonacci := 0;
11        end
12     else
13         begin
14             if n = 1 then
15                 begin
16                     Fibonacci := 1;
17                 end
18             else
19                 begin
20                     Fibonacci := Fibonacci(n - 1) + Fibonacci(n - 2);
21                 end
22                 ;
23         end
24     ;
25 end
26 ;
27
28 begin
29     for i := 1 to 10 do
30         begin
31             writeln(Fibonacci(i));
32         end
33     ;
34 end
35 .
```

- AST



- IR

```

1 ; ModuleID = 'main'
2 source_filename = "main"
3
4 @i = global i32 0
5 @_str = constant [4 x i8] c"%d\0A\00"
6
7 define internal void @main() {
8   entrypoint:
9     store i32 1, i32* @i
10    br label %cond
11
12  cond:                                         ; preds = %loop,
13  %entrypoint
14    %tmp = load i32, i32* @i
15    %0 = icmp sle i32 %tmp, 10
16    %forCond = icmp ne i1 %0, false
17    br i1 %forCond, label %loop, label %afterLoop

```

```

17
18 loop: ; preds = %cond
19   %tmp1 = load i32, i32* @i
20   %calltmp = call i32 @Fibonacci(i32 %tmp1)
21   %printf = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([4 x
22     i8], [4 x i8]* @.str, i32 0, i32 0), i32 %calltmp)
23   %1 = add i32 %tmp, 1
24   store i32 %1, i32* @i
25   br label %cond
26
27 afterLoop: ; preds = %cond
28   ret void
29 }
30
31 declare i32 @printf(i8*, ...)
32
33 declare i32 @scanf(...)
34
35 define internal i32 @Fibonacci(i32) {
36   entrypoint:
37   %Fibonacci = alloca i32
38   %n = alloca i32
39   store i32 %0, i32* %n
40   %tmp = load i32, i32* %n
41   %tmpEQ = icmp eq i32 %tmp, 0
42   %ifCond = icmp ne i1 %tmpEQ, false
43   br i1 %ifCond, label %then, label %else
44
45 then: ; preds = %entrypoint
46   store i32 0, i32* %Fibonacci
47   br label %merge
48
49 else: ; preds = %entrypoint
50   %tmp1 = load i32, i32* %n
51   %tmpEQ2 = icmp eq i32 %tmp1, 1
52   %ifCond3 = icmp ne i1 %tmpEQ2, false
53   br i1 %ifCond3, label %then4, label %else5
54
55 merge: ; preds = %merge6, %then
56   %tmp11 = load i32, i32* %Fibonacci
57   ret i32 %tmp11
58
59 then4: ; preds = %else
60   store i32 1, i32* %Fibonacci
61   br label %merge6
62
63 else5: ; preds = %else
64   %tmp7 = load i32, i32* %n
65   %subtmpi = sub i32 %tmp7, 1
66   %calltmp = call i32 @Fibonacci(i32 %subtmpi)
67   %tmp8 = load i32, i32* %n
68   %subtmpi9 = sub i32 %tmp8, 2
69   %calltmp10 = call i32 @Fibonacci(i32 %subtmpi9)
70   %addtmpi = add i32 %calltmp, %calltmp10
71   store i32 %addtmpi, i32* %Fibonacci
72   br label %merge6
73
74 merge6: ; preds = %else5, %then4

```

```
74     br label %merge  
75 }  
76
```

- 汇编指令

```
1      .section    __TEXT,__text,regular,pure_instructions  
2      .macosx_version_min 10, 15  
3      .p2align   4, 0x90          ## -- Begin function main  
4      _main:                      ## @main  
5      .cfi_startproc  
6      ## %bb.0:                  ## %entrypoint  
7      pushq   %rbp  
8      .cfi_def_cfa_offset 16  
9      pushq   %rbx  
10     .cfi_def_cfa_offset 24  
11     pushq   %rax  
12     .cfi_def_cfa_offset 32  
13     .cfi_offset %rbx, -24  
14     .cfi_offset %rbp, -16  
15     movl   $1, _i(%rip)  
16     leaq   _._str(%rip), %rbx  
17     .p2align   4, 0x90  
18     LBB0_1:                   ## %cond  
19                                     ## =>This Inner Loop Header:  
20     Depth=1  
21     movl   _i(%rip), %ebp  
22     cmpl   $10, %ebp  
23     jg    LBB0_3  
24     ## %bb.2:                  ## %loop  
25                                     ## in Loop: Header=BB0_1 Depth=1  
26     movl   _i(%rip), %edi  
27     callq  _Fibonacci  
28     movq   %rbx, %rdi  
29     movl   %eax, %esi  
30     xorl   %eax, %eax  
31     callq  _printf  
32     incl   %ebp  
33     movl   %ebp, _i(%rip)  
34     jmp   LBB0_1  
35     LBB0_3:                   ## %afterLoop  
36     addq   $8, %rsp  
37     popq   %rbx  
38     popq   %rbp  
39     retq  
40     .cfi_endproc  
41                                     ## -- End function  
42     .p2align   4, 0x90          ## -- Begin function Fibonacci  
43     _Fibonacci:                ## @Fibonacci  
44     .cfi_startproc  
45     ## %bb.0:                  ## %entrypoint  
46     pushq   %rbx  
47     .cfi_def_cfa_offset 16  
48     subq   $16, %rsp  
49     .cfi_def_cfa_offset 32  
50     .cfi_offset %rbx, -16  
51     movl   %edi, 8(%rsp)
```

```

51    testl  %edi, %edi
52    jne LBB1_3
53 ## %bb.1:                                ## %then
54    movl   $0, 12(%rsp)
55    jmp LBB1_2
56 LBB1_3:                                ## %else
57    cmpl   $1, 8(%rsp)
58    jne LBB1_5
59 ## %bb.4:                                ## %then4
60    movl   $1, 12(%rsp)
61    jmp LBB1_2
62 LBB1_5:                                ## %else5
63    movl   8(%rsp), %edi
64    decl   %edi
65    callq  _Fibonacci
66    movl   %eax, %ebx
67    movl   8(%rsp), %edi
68    addl   $-2, %edi
69    callq  _Fibonacci
70    addl   %ebx, %eax
71    movl   %eax, 12(%rsp)
72 LBB1_2:                                ## %merge
73    movl   12(%rsp), %eax
74    addq   $16, %rsp
75    popq   %rbx
76    retq
77 .cfi_endproc
78                                     ## -- End function
79 .globl  _i                                ## @i
80 .zerofill __DATA,__common,__i,4,2
81 .section __TEXT,__const
82 .globl  _.str                               ## @.str
83 _.str:
84     .asciz "%d\n"
85
86
87 .subsections_via_symbols
88

```

- 运行结果

```

→ SPL-Complier git:(master) ✘ lli spl.ll
1
1
2
3
5
8
13
21
34
55

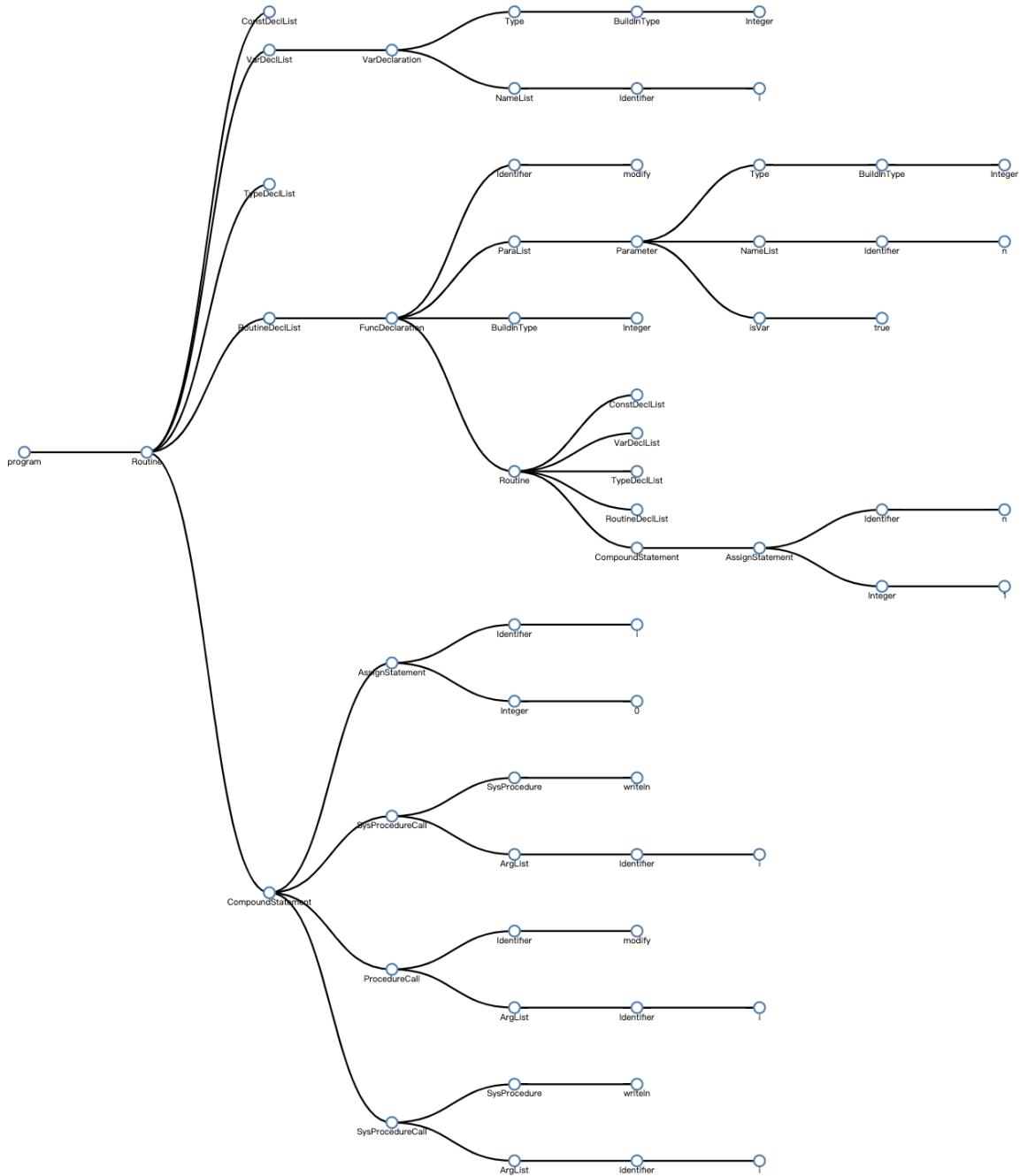
```

6.4.3 引用传递测试

- 测试代码

```
1 program ReferenceParameter;
2
3 var
4     i : integer;
5
6 function modify(var n : integer) : integer;
7 begin
8     n := 1;
9 end
10 ;
11
12 begin
13     i := 0;
14     writeln(i);
15     modify(i);
16     writeln(i);
17 end
18 .
```

- AST



• IR

```
1 ; ModuleID = 'main'
2 source_filename = "main"
3
4 @i = global i32 0
5 @_str = constant [4 x i8] c"%d\0A\00"
6 @_str.1 = constant [4 x i8] c"%d\0A\00"
7
8 define internal void @main() {
9 entrypoint:
10   store i32 0, i32* @i
11   %tmp = load i32, i32* @i
12   %printf = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([4 x i8], [4 x i8]* @_str, i32 0, i32 0), i32 %tmp)
13   %calltmp = call i32 @modify(i32* @i)
14   %tmp1 = load i32, i32* @i
```

```

15    %printf2 = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([4 x
16        i8], [4 x i8]* @.str.1, i32 0, i32 0), i32 %tmp1)
17    ret void
18 }
19
20 declare i32 @printf(i8*, ...)
21
22 declare i32 @scanf(...)
23
24 define internal i32 @modify(i32* nonnull) {
25     entrypoint:
26         %modify = alloca i32
27         %n = getelementptr i32, i32* %0, i32 0
28         store i32 1, i32* %n
29         %tmp = load i32, i32* %modify
30         ret i32 %tmp
31 }
```

- 汇编指令

```

1      .section      __TEXT,__text,regular,pure_instructions
2      .macosx_version_min 10, 15
3      .p2align     4, 0x90          ## -- Begin function main
4      _main:                      ## @main
5          .cfi_startproc
6      ## %bb.0:                  ## %entrypoint
7          pushq   %rbx
8          .cfi_def_cfa_offset 16
9          .cfi_offset %rbx, -16
10         movl    $0, _i(%rip)
11         leaq    _i(%rip), %rbx
12         leaq    _._str(%rip), %rdi
13         xorl    %esi, %esi
14         xorl    %eax, %eax
15         callq   _printf
16         movq    %rbx, %rdi
17         callq   _modify
18         movl    _i(%rip), %esi
19         leaq    _._str.1(%rip), %rdi
20         xorl    %eax, %eax
21         callq   _printf
22         popq    %rbx
23         retq
24     .cfi_endproc
25
26     ## -- End function
27     .p2align     4, 0x90          ## -- Begin function modify
28     _modify:                   ## @modify
29     .cfi_startproc
30     ## %bb.0:                  ## %entrypoint
31     movl    $1, (%rdi)
32     movl    -4(%rsp), %eax
33     retq
34     .cfi_endproc
35
36     .globl  _i                  ## @i
37     .zerofill __DATA,__common,_i,4,2
```

```

37     .section    __TEXT,__const
38     .globl    _._str                      ## @._str
39     _._str:
40         .asciz    "%d\n"
41
42     .globl    _._str.1                   ## @_str.1
43     _._str.1:
44         .asciz    "%d\n"
45
46
47     .subsections_via_symbols
48

```

- 运行结果

```

→ SPL-Compiler git:(master) x lli spl.ll
0
1

```

6.5 综合测试

6.5.1 测试用例1

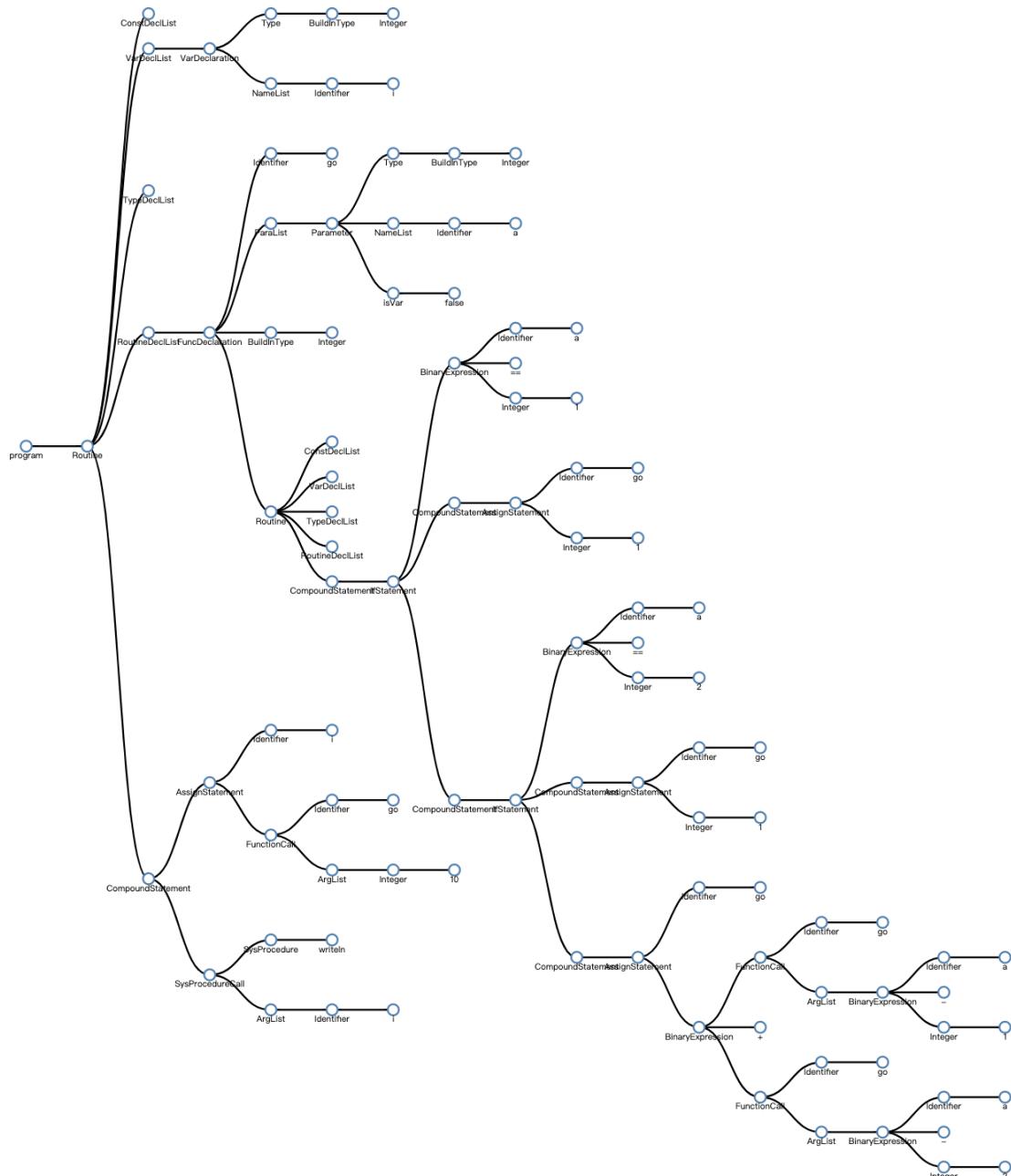
- 测试代码

```

1 program hello;
2 var
3     i : integer;
4
5 function go(a : integer): integer;
6 begin
7     if a = 1 then
8         begin
9             go := 1;
10        end
11     else
12         begin
13             if a = 2 then
14                 begin
15                     go := 1;
16                 end
17             else
18                 begin
19                     go := go(a - 1) + go(a - 2);
20                 end
21             ;
22         end
23     ;
24 end
25 ;
26
27 begin
28     i := go(10);
29     writeln(i);
30 end
31 .

```

- AST



• |R

```
1 ; ModuleID = 'main'
2 source_filename = "main"
3
4 @i = global i32 0
5 @_str = constant [4 x i8] c"%d\0A\00"
6
7 define internal void @main() {
8 entrypoint:
9     %calltmp = call i32 @go(i32 10)
10    store i32 %calltmp, i32* @i
11    %tmp = load i32, i32* @i
12    %printf = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([4 x
13 i8], [4 x i8]* @_str, i32 0, i32 0), i32 %tmp)
14    ret void
15 }
```

```

15 declare i32 @printf(i8*, ...)
16
17 define internal i32 @go(i32) {
18 entrypoint:
19     %go = alloca i32
20     %a = alloca i32
21     store i32 %0, i32* %a
22     %tmp = load i32, i32* %a
23     %tmpEQ = icmp eq i32 %tmp, 1
24     %ifCond = icmp ne i1 %tmpEQ, false
25     br i1 %ifCond, label %then, label %else
26
27
28 then:                                ; preds = %entrypoint
29     store i32 1, i32* %go
30     br label %merge
31
32 else:                                ; preds = %entrypoint
33     %tmp1 = load i32, i32* %a
34     %tmpEQ2 = icmp eq i32 %tmp1, 2
35     %ifCond3 = icmp ne i1 %tmpEQ2, false
36     br i1 %ifCond3, label %then4, label %else5
37
38 merge:                                ; preds = %merge6, %then
39     %tmp11 = load i32, i32* %go
40     ret i32 %tmp11
41
42 then4:                                ; preds = %else
43     store i32 1, i32* %go
44     br label %merge6
45
46 else5:                                ; preds = %else
47     %tmp7 = load i32, i32* %a
48     %subtmpi = sub i32 %tmp7, 1
49     %calltmp = call i32 @go(i32 %subtmpi)
50     %tmp8 = load i32, i32* %a
51     %subtmpi9 = sub i32 %tmp8, 2
52     %calltmp10 = call i32 @go(i32 %subtmpi9)
53     %addtmpi = add i32 %calltmp, %calltmp10
54     store i32 %addtmpi, i32* %go
55     br label %merge6
56
57 merge6:                                ; preds = %else5, %then4
58     br label %merge
59 }

```

- 汇编指令

```

1 .section    __TEXT,__text,regular,pure_instructions
2 .macosx_version_min 10, 15
3 .p2align    4, 0x90          ## -- Begin function main
4 _main:                      ## @main
5     .cfi_startproc
6     ## %bb.0:                      ## %entrypoint
7     pushq    %rax
8     .cfi_def_cfa_offset 16
9     movl    $10, %edi

```

```

10    callq  _go
11    movl    %eax, _i(%rip)
12    leaq    _.str(%rip), %rdi
13    movl    %eax, %esi
14    xorl    %eax, %eax
15    callq  _printf
16    popq    %rax
17    retq
18    .cfi_endproc
19                                ## -- End function
20    .p2align   4, 0x90      ## -- Begin function go
21 _go:                           ## @go
22    .cfi_startproc
23 ## %bb.0:                      ## %entrypoint
24    pushq   %rbx
25    .cfi_def_cfa_offset 16
26    subq    $16, %rsp
27    .cfi_def_cfa_offset 32
28    .cfi_offset %rbx, -16
29    movl    %edi, 8(%rsp)
30    cmpl    $1, %edi
31    je LBB1_1
32 ## %bb.3:                      ## %else
33    cmpl    $2, 8(%rsp)
34    jne LBB1_4
35 LBB1_1:                         ## %then
36    movl    $1, 12(%rsp)
37 LBB1_2:                           ## %merge
38    movl    12(%rsp), %eax
39    addq    $16, %rsp
40    popq    %rbx
41    retq
42 LBB1_4:                           ## %else5
43    movl    8(%rsp), %edi
44    decl    %edi
45    callq  _go
46    movl    %eax, %ebx
47    movl    8(%rsp), %edi
48    addl    $-2, %edi
49    callq  _go
50    addl    %ebx, %eax
51    movl    %eax, 12(%rsp)
52    jmp LBB1_2
53    .cfi_endproc
54                                ## -- End function
55    .globl  _i                  ## @i
56 .zerofill __DATA,__common,_i,4,2
57    .section   __TEXT,__const
58    .globl  _.str                 ## @.str
59 _.str:
60    .asciz  "%d\n"
61
62
63 .subsections_via_symbols

```

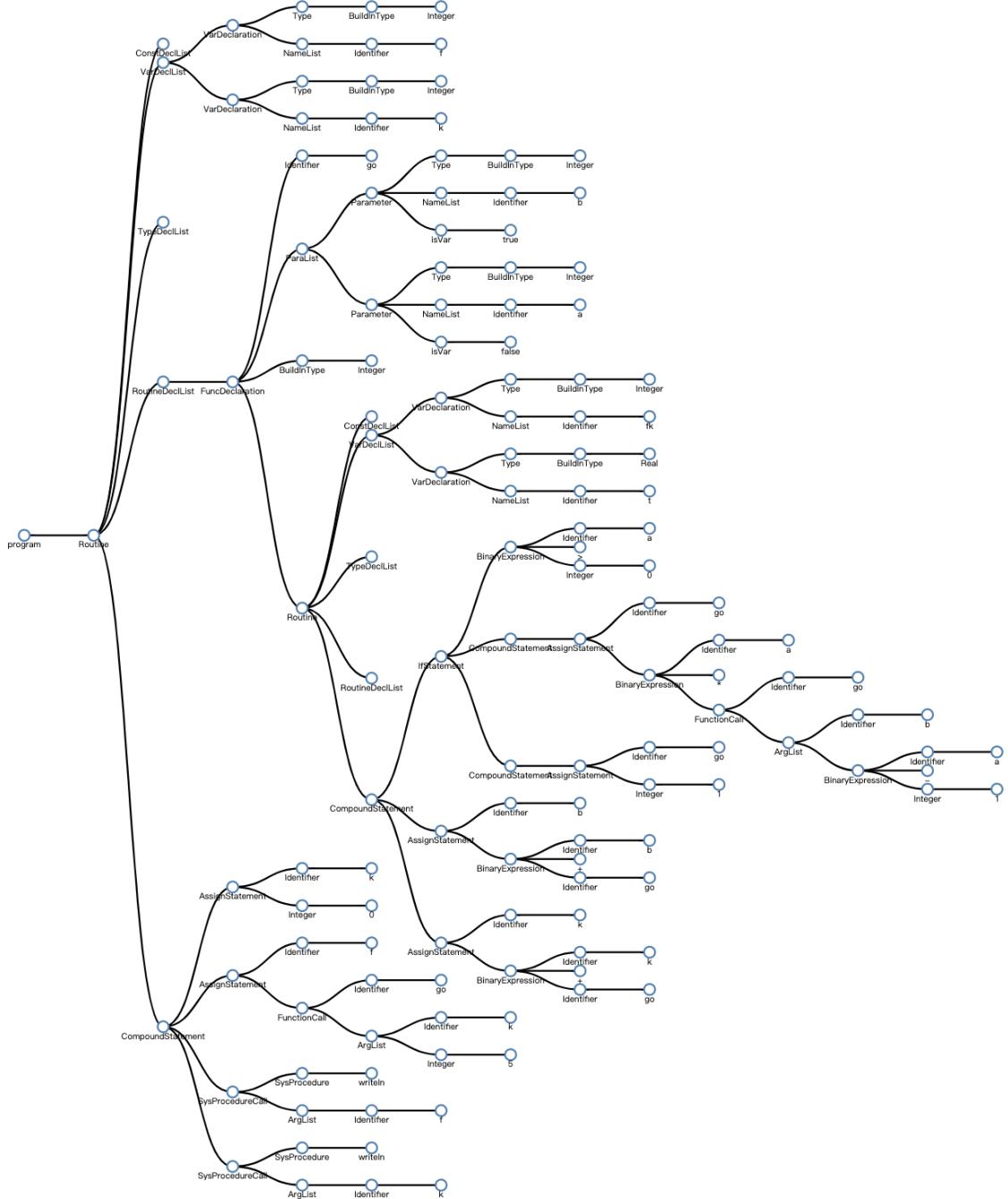
- 运行结果

6.5.2 测试用例2

- 测试代码

```
1 program hello;
2 var
3     f : integer;
4     k : integer;
5 function go(var b : integer; a : integer): integer;
6 var
7     fk : integer;
8     t : real;
9
10 begin
11     if a > 0 then
12         begin
13             go := a * go(b , a - 1);
14         end
15     else
16         begin
17             go := 1;
18         end
19     ;
20     b := b + go;
21     k := k + go;
22 end
23 ;
24
25 begin
26     k := 0;
27     f := go(k , 5);
28     writeln(f);
29     writeln(k);
30 end
31 .
```

- AST



- IR

```

1 ; ModuleID = 'main'
2 source_filename = "main"
3
4 @f = global i32 0
5 @k = global i32 0
6 @_str = constant [4 x i8] c"%d\0A\00"
7 @_str.1 = constant [4 x i8] c"%d\0A\00"
8
9 define internal void @main() {
10 entrypoint:
11   store i32 0, i32* @k
12   %calltmp = call i32 @go(i32* @k, i32 5)
13   store i32 %calltmp, i32* @f
14   %tmp = load i32, i32* @f
15   %printf = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([4 x i8], [4 x i8]* @_str, i32 0, i32 0), i32 %tmp)

```

```

16    %tmp1 = load i32, i32* @k
17    %printf2 = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([4 x
18    i8], [4 x i8]* @_str.1, i32 0, i32 0), i32 %tmp1)
19    ret void
20 }
21
22 declare i32 @printf(i8*, ...)
23
24 define internal i32 @go(i32* nonnull, i32) {
entrypoint:
25    %t = alloca double
26    %fk = alloca i32
27    %go = alloca i32
28    %a = alloca i32
29    %b = getelementptr i32, i32* %0, i32 0
30    store i32 %1, i32* %a
31    %tmp = load i32, i32* %a
32    %tmpSGT = icmp sgt i32 %tmp, 0
33    %ifCond = icmp ne i1 %tmpSGT, false
34    br i1 %ifCond, label %then, label %else
35
36 then:                                     ; preds = %entrypoint
37    %tmp1 = load i32, i32* %a
38    %tmp2 = load i32, i32* %a
39    %subtmpi = sub i32 %tmp2, 1
40    %calltmp = call i32 @go(i32* %b, i32 %subtmpi)
41    %multtmpi = mul i32 %tmp1, %calltmp
42    store i32 %multtmpi, i32* %go
43    br label %merge
44
45 else:                                     ; preds = %entrypoint
46    store i32 1, i32* %go
47    br label %merge
48
49 merge:                                    ; preds = %else, %then
50    %tmp3 = load i32, i32* %b
51    %tmp4 = load i32, i32* %go
52    %addtmpi = add i32 %tmp3, %tmp4
53    store i32 %addtmpi, i32* %b
54    %tmp5 = load i32, i32* @k
55    %tmp6 = load i32, i32* %go
56    %addtmpi7 = add i32 %tmp5, %tmp6
57    store i32 %addtmpi7, i32* @k
58    %tmp8 = load i32, i32* %go
59    ret i32 %tmp8
60 }

```

- 汇编指令

```

1   .section    __TEXT,__text,regular,pure_instructions
2   .macosx_version_min 10, 15
3   .p2align    4, 0x90          ## -- Begin function main
4   _main:           ## @main
5   .cfi_startproc
6   ## %bb.0:           ## %entrypoint
7   pushq    %rax
8   .cfi_def_cfa_offset 16

```

```

 9    movl    $0, _k(%rip)
10   leaq    _k(%rip), %rdi
11   movl    $5, %esi
12   callq   _go
13   movl    %eax, _f(%rip)
14   leaq    _._str(%rip), %rdi
15   movl    %eax, %esi
16   xorl    %eax, %eax
17   callq   _printf
18   movl    _k(%rip), %esi
19   leaq    _._str.1(%rip), %rdi
20   xorl    %eax, %eax
21   callq   _printf
22   popq    %rax
23   retq
24   .cfi_endproc
25                                     ## -- End function
26   .p2align 4, 0x90          ## -- Begin function go
27 _go:                           ## @go
28   .cfi_startproc
29 ## %bb.0:                      ## %entrypoint
30   pushq   %r14
31   .cfi_def_cfa_offset 16
32   pushq   %rbx
33   .cfi_def_cfa_offset 24
34   subq    $24, %rsp
35   .cfi_def_cfa_offset 48
36   .cfi_offset %rbx, -24
37   .cfi_offset %r14, -16
38   movq    %rdi, %rbx
39   movl    %esi, 4(%rsp)
40   testl   %esi, %esi
41   jle LBB1_2
42 ## %bb.1:                      ## %then
43   movl    4(%rsp), %r14d
44   leal    -1(%r14), %esi
45   movq    %rbx, %rdi
46   callq   _go
47   imull   %r14d, %eax
48   movl    %eax, (%rsp)
49   jmp LBB1_3
50 LBB1_2:                         ## %else
51   movl    $1, (%rsp)
52 LBB1_3:                           ## %merge
53   movl    (%rsp), %eax
54   addl    %eax, (%rbx)
55   movl    (%rsp), %eax
56   addl    %eax, _k(%rip)
57   addq    $24, %rsp
58   popq    %rbx
59   popq    %r14
60   retq
61   .cfi_endproc
62                                     ## -- End function
63   .globl   _f                   ## @f
64   .zerofill __DATA,__common,_f,4,2
65   .globl   _k                   ## @k
66   .zerofill __DATA,__common,_k,4,2

```

```

67     .section    __TEXT,__const
68     .globl    _._str                      ## @._str
69     _._str:
70         .asciz   "%d\n"
71
72     .globl    _._str.1                   ## @_str.1
73     _._str.1:
74         .asciz   "%d\n"
75
76
77     .subsections_via_symbols

```

- 运行结果

```

→ SPL-Compiler git:(master) ✘ lli spl.ll
120
308

```

6.5.3 测试用例3

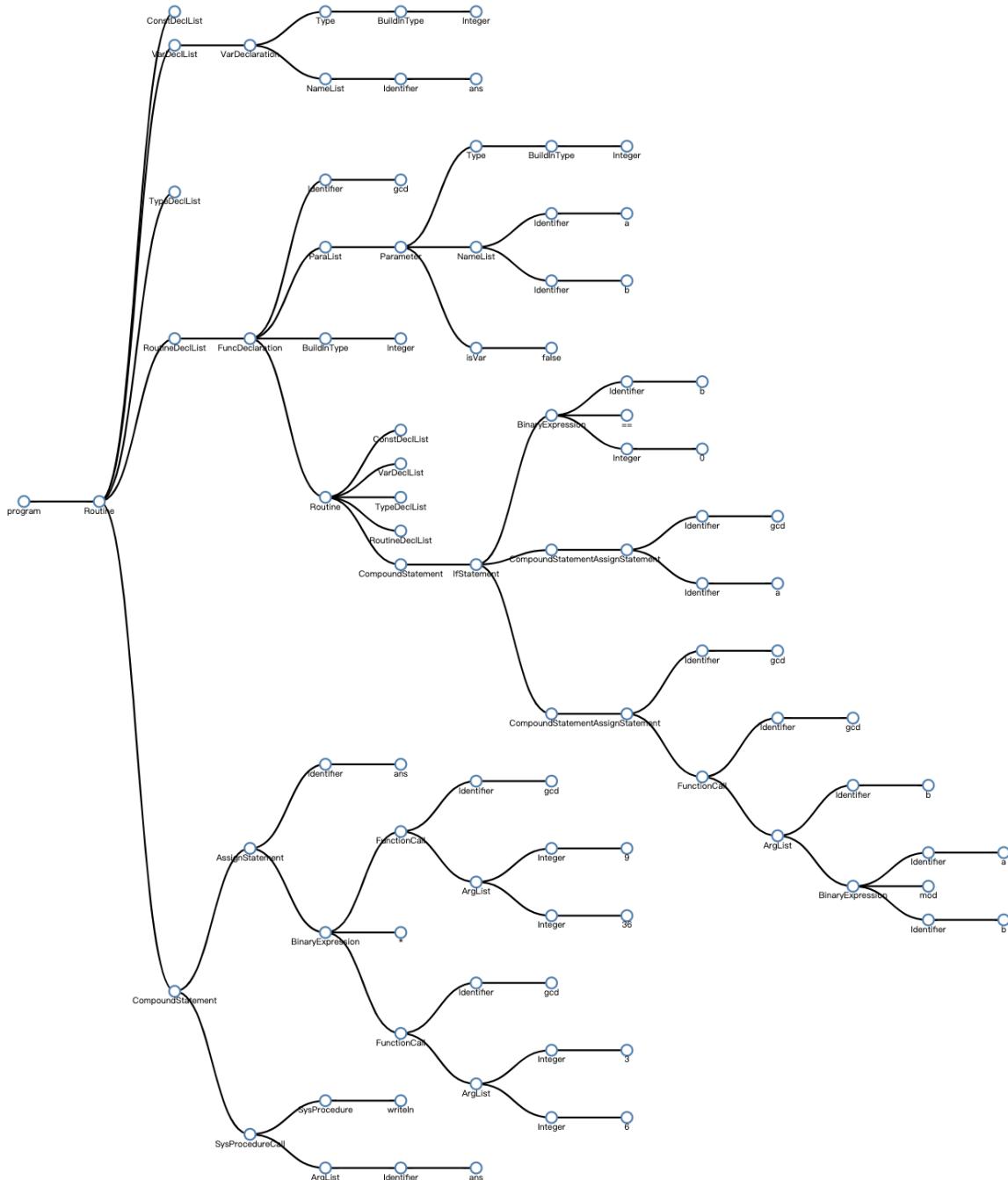
- 测试代码

```

1 program hello;
2 var
3     ans : integer;
4
5 function gcd(a, b : integer) : integer;
6 begin
7     if b = 0 then begin
8         gcd := a;
9     end
10    else begin
11        gcd := gcd(b, a mod b);
12    end
13    ;
14 end
15 ;
16
17 begin
18     ans := gcd(9, 36) * gcd(3, 6);
19     writeln(ans);
20 end
21 .

```

- AST



• IR

```
1 ; ModuleID = 'main'
2 source_filename = "main"
3
4 @ans = global i32 0
5 @_str = constant [4 x i8] c"%d\\0A\\00"
6
7 define internal void @main() {
8 entrypoint:
9     %calltmp = call i32 @gcd(i32 9, i32 36)
10    %calltmp1 = call i32 @gcd(i32 3, i32 6)
11    %multmpi = mul i32 %calltmp, %calltmp1
12    store i32 %multmpi, i32* @ans
13    %tmp = load i32, i32* @ans
14    %printf = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([4 x
15 i8], [4 x i8]* @_str, i32 0, i32 0), i32 %tmp)
16    ret void
}
```

```

17 declare i32 @printf(i8*, ...)
18
19
20 define internal i32 @gcd(i32, i32) {
21 entrypoint:
22     %gcd = alloca i32
23     %b = alloca i32
24     %a = alloca i32
25     store i32 %0, i32* %a
26     store i32 %1, i32* %b
27     %tmp = load i32, i32* %b
28     %tmpEQ = icmp eq i32 %tmp, 0
29     %ifCond = icmp ne i1 %tmpEQ, false
30     br i1 %ifCond, label %then, label %else
31
32 then:                                     ; preds = %entrypoint
33     %tmp1 = load i32, i32* %a
34     store i32 %tmp1, i32* %gcd
35     br label %merge
36
37 else:                                     ; preds = %entrypoint
38     %tmp2 = load i32, i32* %b
39     %tmp3 = load i32, i32* %a
40     %tmp4 = load i32, i32* %b
41     %tmpSREM = srem i32 %tmp3, %tmp4
42     %calltmp = call i32 @gcd(i32 %tmp2, i32 %tmpSREM)
43     store i32 %calltmp, i32* %gcd
44     br label %merge
45
46 merge:                                    ; preds = %else, %then
47     %tmp5 = load i32, i32* %gcd
48     ret i32 %tmp5
49 }
```

- 汇编指令

```

1 .section    __TEXT,__text,regular,pure_instructions
2 .macosx_version_min 10, 15
3 .p2align    4, 0x90          ## -- Begin function main
4 _main:                                ## @main
5     .cfi_startproc
6 ## %bb.0:                                ## %entrypoint
7     pushq    %rbx
8     .cfi_def_cfa_offset 16
9     .cfi_offset %rbx, -16
10    movl    $9, %edi
11    movl    $36, %esi
12    callq   _gcd
13    movl    %eax, %ebx
14    movl    $3, %edi
15    movl    $6, %esi
16    callq   _gcd
17    imull   %ebx, %eax
18    movl    %eax, _ans(%rip)
19    leaq    _.str(%rip), %rdi
20    movl    %eax, %esi
21    xorl    %eax, %eax
```

```

22    callq  _printf
23    popq    %rbx
24    retq
25    .cfi_endproc
26                                ## -- End function
27    .p2align   4, 0x90      ## -- Begin function gcd
28 _gcd:                           ## @gcd
29    .cfi_startproc
30 ## %bb.0:                      ## %entrypoint
31    subq    $24, %rsp
32    .cfi_def_cfa_offset 32
33    movl    %edi, 12(%rsp)
34    movl    %esi, 20(%rsp)
35    testl   %esi, %esi
36    jne LBB1_2
37 ## %bb.1:                      ## %then
38    movl    12(%rsp), %eax
39    jmp LBB1_3
40 LBB1_2:                         ## %else
41    movl    20(%rsp), %edi
42    movl    12(%rsp), %eax
43    cltd
44    idivl  %edi
45    movl    %edx, %esi
46    callq  _gcd
47 LBB1_3:                         ## %merge
48    movl    %eax, 16(%rsp)
49    movl    16(%rsp), %eax
50    addq    $24, %rsp
51    retq
52    .cfi_endproc
53                                ## -- End function
54    .globl  _ans                  ## @ans
55 .zerofill __DATA,__common,_ans,4,2
56    .section   __TEXT,__const
57    .globl  __.str                ## @.str
58 __.str:
59    .asciz  "%d\n"
60
61
62 .subsections_via_symbols

```

- 运行结果

→ SPL-Compiler git:(master) x lli spl.ll
27

第柒章 总结

本次实验我们小组三人设计完成了一个SPL编译器，目前支持：

- 数据类型
 - 内置类型int, char, bool, real
 - 范围类型
 - 常量范围类型

- 变量范围类型
 - 数组类型
 - 引用类型
- 表达式
 - 一元操作: MINUS
 - 二元操作: ADD, SUB, MUL, DIV, CMPGE, CMPL, CMPGT, CMPLT, CMPEQ, CMPNE, AND, OR, SREM(MOD), XOR
 - 变量引用
 - 数组引用
- 语句
 - 赋值语句
 - 标签语句
 - 复合语句
 - 嵌套语句
 - 流程控制语句
 - 分支语句: if-then-else, case
 - 循环语句: for, while, repeat
 - 跳转语句: goto
- 函数/过程
 - 函数/过程的声明、定义、调用
 - 系统IO函数: read, write, writeln
 - 值传递和引用传递
- 优化: 常量折叠
- 可视化
 - AST可视化
 - DAG可视化