

编译原理实验报告

编译原理实验报告

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 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
 - 3.3.6
 - 3.3.7
 - 3.3.8
 - 3.3.9
 - 3.3.10

第四章 优化考虑

第五章 代码生成

第六章 测试案例

- 6.1 数据类型测试

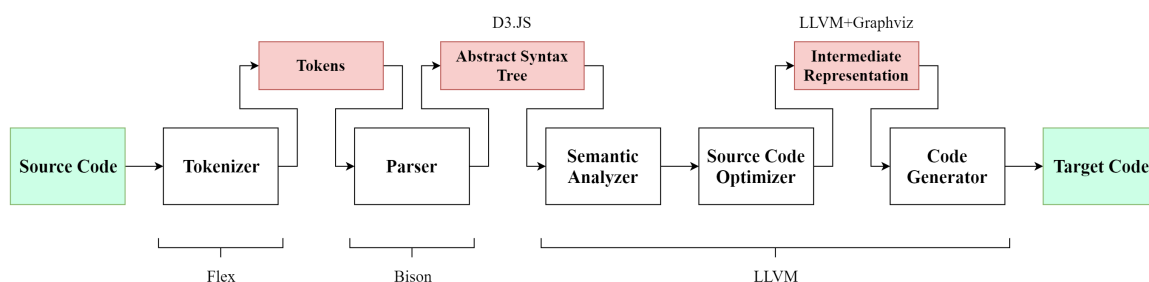
6.1.1 内置类型测试	
6.1.2 数组类型测试	
6.2 运算测试	
6.3 控制流测试	
6.3.1 分支测试	
6.3.2 循环测试	
6.3.3 Goto测试	
6.4 函数测试	
6.4.1 简单函数测试	
6.4.2 递归函数测试	
6.4.3 引用传递测试	
6.5 综合测试	
6.5.1 测试用例1	
6.5.2 测试用例2	
6.5.3 测试用例3	
第柒章 总结	

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为输入，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，所使用的具体技术包括但不限于：

- 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 `getJoson` 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
<code>\</code>	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>\(</code> ”则匹配“ <code>(</code> ”。
<code>^</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， <code>^</code> 也匹配“ <code>\n</code> ”或“ <code>\r</code> ”之后的位置。
<code>\$</code>	匹配输入字符串的结束位置。如果设置了RegExp对象的Multiline属性， <code>\$</code> 也匹配“ <code>\n</code> ”或“ <code>\r</code> ”之前的位置。
<code>*</code>	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> 。
<code>+</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> ”。 <code>+</code> 等价于 <code>{1,}</code> 。
<code>?</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> ”。 <code>?</code> 等价于 <code>{0,1}</code> 。
<code>{n}</code>	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> 。
<code>{n,}</code>	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>foooooo</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> ”。
<code>{n,m}</code>	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>foooooo</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。
<code>?</code>	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>+,?</code> ， <code>{n}</code> ， <code>{n,}</code> ， <code>{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> ”。
<code>.</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。
<code>x y</code>	没有包围在 <code>()</code> 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)ood</code> ”则匹配“ <code>zood</code> ”或“ <code>food</code> ”。
<code>[xyz]</code>	字符集合（character class）。匹配所包含的任意一个字符。例如，“ <code>[abc]</code> ”可以匹配“ <code>plain</code> ”中的“ <code>a</code> ”。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 <code>^</code> 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 <code>-</code> 如果出现在字符串中间表示字符范围描述；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。
<code>[^xyz]</code>	排除型字符集合（negated character classes）。匹配未列出的任意字符。例如，“ <code>[^abc]</code> ”可以匹配“ <code>plain</code> ”中的“ <code>plin</code> ”。
<code>[a-z]</code>	字符范围。匹配指定范围内的任意字符。例如，“ <code>[a-z]</code> ”可以匹配“ <code>a</code> ”到“ <code>z</code> ”范围内的任意小写字母字符。

字符	描述
<code>[^a-z]</code>	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，“ <code>[^a-z]</code> ”可以匹配任何不在“a”到“z”范围内的任意字符。
<code>\d</code>	匹配一个数字字符。等价于 <code>[0-9]</code> 。注意Unicode正则表达式会匹配全角数字字符。
<code>\D</code>	匹配一个非数字字符。等价于 <code>[^0-9]</code> 。
<code>\n</code>	匹配一个换行符。等价于 <code>\x0a</code> 和 <code>\cj</code> 。
<code>\r</code>	匹配一个回车符。等价于 <code>\x0d</code> 和 <code>\cM</code> 。
<code>\s</code>	匹配任何空白字符，包括空格、制表符、换页符等等。等价于 <code>[\f\n\r\t\v]</code> 。注意Unicode正则表达式会匹配全角空格符。
<code>\S</code>	匹配任何非空白字符。等价于 <code>[^\f\n\r\t\v]</code> 。
<code>\w</code>	匹配包括下划线的任何单词字符。等价于“ <code>[A-Za-z0-9_]</code> ”。注意Unicode正则表达式会匹配中文字符。
<code>\W</code>	匹配任何非单词字符。等价于“ <code>[^A-Za-z0-9_]</code> ”。

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_]*           {
2                                     yy1val.sval = new
   std::string(yytext, yyleng);
3                                     return
   IDENTIFIER;
4                                     }

```

然后其他需要额外保存值的单词：

```

1  "boolean"|"char"|"integer"|"real"          {
2                                          yylval.sval =
3  new std::string(yytext, yyleng);
4                                          return
5  SYS_TYPE;
6                                          }
7  "abs"|"chr"|"odd"|"ord"|"pred"|"sqr"|"sqrt"|"succ" {
8                                          yylval.sval =
9  new std::string(yytext, yyleng);
10                                         return
11  SYS_FUNCT;
12                                         }
13  "false"|"maxint"|"true"                    {
14                                          yylval.sval =
15  new std::string(yytext, yyleng);
16                                         return
17  SYS_PROC;
18                                         }
19  "read"                                       {
20                                          yylval.sval =
21  new
22  std::string(yytext, yyleng);
23                                         return  READ;
24                                         }
25  [0-9]+\.[0-9]+                             {
26                                          double dtmp;
27  sscanf(yytext,
28  "%lf", &dtmp);
29                                          yylval.dval =
30  dtmp;
31                                         return  REAL;
32                                         }
33  [0-9]+                                       {
34                                          int itmp;
35
36  double tmp;
37
38  sscanf(yytext,
39  "%d", &itmp);
40
41  itmp;
42
43                                         return  INTEGER;
44                                         }
45  \'.\'                                         {
46                                          yylval.cval =
47  yytext[1];
48                                         return  CHAR;
49                                         }

```

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。

第貳章 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。

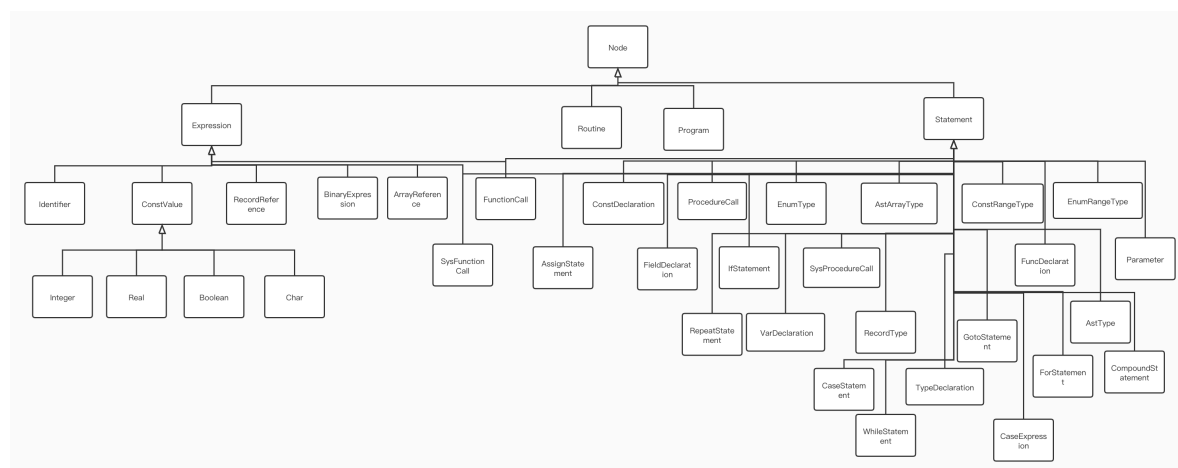
与Lex相似，Yacc的输入文件由以%%分割的三部分组成，分别是声明区、规则区和程序区。三部分的功能与Lex相似，不同的是规则区的正则表达式替换为CFG，在声明区要提前声明好使用到的终结符以及非终结符的类型。

```
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),
        routine(routine) { }
4      virtual llvm::Value *codegen(CodeGenerator & generator) override;
5      virtual string toJson() override;
6  private:
7      string *programID;
8      Routine *routine;
9  };

```

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,
RoutineList *rl)
4     void setRoutineBody(CompoundStatement *routineBody) { this->routineBody
= routineBody; }
5     virtual llvm::Value *codegen(CodeGenerator & generator) override;
6     virtual string toJson() override;
7     void setGlobal()
8 private:
9     ConstDeclList *constDeclList;
10    VarDeclList *varDeclList;
11    TypeDeclList *typeDeclList;
12    RoutineList *routineList;
13    CompoundStatement *routineBody;
14 };

```

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 toJson() 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TypeOf枚举确定AstType的真实类型。

```
1 // 类型
2 class AstType : public Statement {
3 public:
4     enum TypeOf {
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    TypeOf 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;
```

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),
        globalFlag(false)
6      {
7      }
8      virtual llvm::Value *codegen(CodeGenerator & generator) override;
9      virtual string toJson() override;
10     void setGlobal()
11     bool isGlobal()
12 private:
13     Identifier *name;
14     ConstValue *value;
15     AstType *type;
16     bool globalFlag;
17 };

```

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),
        globalFlag(false) {}
4      virtual llvm::Value *codegen(CodeGenerator & generator) override;
5      virtual string toJson() override;
6      void setGlobal()
7      bool isGlobal()
8  private:
9      NameList *nameList;
10     AstType *type;
11     bool globalFlag;
12 };

```

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 toJson() 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 toJson() 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 toJson() 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 DOWNT0 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                             array_type_decl record_type_decl
29  %type<nameList>           name_list
30  %type<fieldList>          field_decl_list
31  %type<fieldDeclaration>   field_decl
32  %type<funcDeclaration>    function_decl procedure_decl
33                             function_head procedure_head
34  %type<paraList>           parameters para_decl_list
35  %type<parameter>          para_type_list var_para_list
36                             val_para_list
37  %type<statement>          stmt non_label_stmt else_clause
38  %type<assignStatement>    assign_stmt
39  %type<statement>          proc_stmt
40  %type<expressionList>     expression_list
41  %type<expression>         expression expr term factor
42  %type<argsList>           args_list
43  %type<ifStatement>        if_stmt
44  %type<repeatStatement>    repeat_stmt
45  %type<whileStatement>     while_stmt
46  %type<forStatement>       for_stmt
47  %type<bVal>                direction
48  %type<caseStatement>      case_stmt
49  %type<caseExprList>       case_expr_list
50  %type<caseExpression>     case_expr
51  %type<gotoStatement>      goto_stmt
52  %type<statementList>      stmt_list
53  %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 { $$ = new
   Identifier($1); }
5
6
7  program: program_head routine DOT { $$ = new
   Program($1, $2); root = $$; }
8
9
10 program_head: PROGRAM IDENTIFIER SEMI { $$ = $2; }
11
12
13 routine: routine_head routine_body { $$ = $1; $$-
   >setRoutineBody($2); }
14
15
16 routine_head: const_part type_part var_part routine_part { $$ = new
   Routine($1, $2, $3, $4); }
17
18
19 const_part :
20     CONST const_expr_list { $$ = $2; }
21     |
   ConstDeclList(); }
22
23
24 const_expr_list :
25     const_expr_list name EQUAL const_value SEMI { $$ = $1; $$-
   >push_back(new ConstDeclaration($2, $4)); }
26     | name EQUAL const_value SEMI { $$ = new
   ConstDeclList(); $$->push_back(new ConstDeclaration($1, $3)); }
27
28 const_value :
29     INTEGER { $$ = new
   Integer($1); }
30     | REAL { $$ = new
   Real($1); }
31     | CHAR { $$ = new
   Char($1); }
32     | SYS_CON {
33         if(*$1 ==
34             "true")
35             $$ =
36             new Boolean(true);
37         else if(*$1
38             == "false")
39             $$ =
40             new Boolean(false);
41         else
42             $$ =
43             new Integer(0x7FFFFFFF);
44     }
45
46 type_part :
47     TYPE type_decl_list { $$ = $2; }

```



```

43         |
TypeDeclList(); }
44
45
46 type_decl_list :
47     type_decl_list type_definition { $$ = $1; $$-
>push_back($2); }
48     | type_definition { $$ = new
TypeDeclList(); $$->push_back($1); }
49
50
51 type_definition :
52     name EQUAL type_decl SEMI { $$ = new
TypeDeclaration($1, $3); }
53
54
55 type_decl :
56     simple_type_decl { $$ = $1; }
57     | array_type_decl { $$ = $1; }
58     | record_type_decl { $$ = $1; }
59
60 simple_type_decl :
61     SYS_TYPE {
62         if(*$1 ==
"integer")
63             $$ =
new AstType(SPL_INTEGER);
64         else if(*$1
== "boolean")
65             $$ =
new AstType(SPL_BOOLEAN);
66         else if(*$1
== "real")
67             $$ =
new AstType(SPL_REAL);
68         else if(*$1
== "char")
69             $$ =
new AstType(SPL_CHAR);
70         else
71             cout <<
"UNKNOWN SYS_TYPE" << endl;
72     }
73     | name { $$ = new
AstType($1); }
74     | LP name_list RP { $$ = new
AstType(new EnumType($2)); }
75     | const_value DOTDOT const_value { $$ = new
AstType(new ConstRangeType($1, $3)); }
76     | MINUS const_value DOTDOT const_value { $$ = new
AstType(new ConstRangeType(-*$2, $4)); }
77     | MINUS const_value DOTDOT MINUS const_value { $$ = new
AstType(new ConstRangeType(-*$2, -*$5)); }
78     | name DOTDOT name { $$ = new
AstType(new EnumRangeType($1, $3)); }
79

```

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 workflow 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
+ "\", \"children\" : [ ";
24     int i = 0;
25     for(auto &child : children) {
26         if(i != children.size() - 1)
27             result += child + ", ";
28         else
29             result += child + " ";
30         i++;
31     }
32     return result + " ] }";
33 }
```

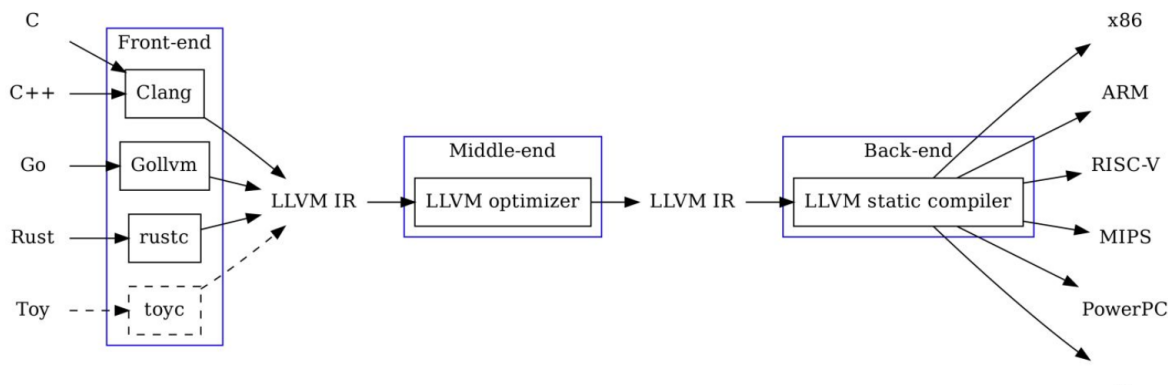
具体效果如下：

```

graph LR
    program((program)) --- Routine((Routine))
    Routine --- ConstDeclList((ConstDeclList))
    Routine --- TypeDeclList((TypeDeclList))
    Routine --- RoutineDeclList((RoutineDeclList))
    Routine --- CompoundStatement((CompoundStatement))
    Routine --- AssignStatement((AssignStatement))
    
    ConstDeclList --- ConstDeclaration((ConstDeclaration))
    ConstDeclaration --- Identifier1((Identifier))
    Identifier1 --- iC((iC))
    ConstDeclaration --- Integer1((Integer))
    Integer1 --- iI((iI))
    
    ConstDeclList --- ConstDeclaration2((ConstDeclaration))
    ConstDeclaration2 --- Identifier2((Identifier))
    Identifier2 --- r((r))
    ConstDeclaration2 --- Integer2((Integer))
    Integer2 --- b((b))
    
    ConstDeclList --- ConstDeclaration3((ConstDeclaration))
    ConstDeclaration3 --- Identifier3((Identifier))
    Identifier3 --- rc((rc))
    ConstDeclaration3 --- Real((Real))
    Real --- 1500000((1500000))
    
    ConstDeclList --- ConstDeclaration4((ConstDeclaration))
    ConstDeclaration4 --- Identifier4((Identifier))
    Identifier4 --- bc((bc))
    ConstDeclaration4 --- Boolean((Boolean))
    Boolean --- false((false))
    
    ConstDeclList --- ConstDeclaration5((ConstDeclaration))
    ConstDeclaration5 --- Identifier5((Identifier))
    Identifier5 --- cc((cc))
    ConstDeclaration5 --- Char((Char))
    Char --- a((a))
    
    TypeDeclList --- VarDeclaration1((VarDeclaration))
    VarDeclaration1 --- Type1((Type))
    Type1 --- BuildInType1((BuildInType))
    BuildInType1 --- Integer1_2((Integer))
    VarDeclaration1 --- NameList1((NameList))
    NameList1 --- Identifier6((Identifier))
    Identifier6 --- iv((iv))
    
    TypeDeclList --- VarDeclaration2((VarDeclaration))
    VarDeclaration2 --- Type2((Type))
    Type2 --- BuildInType2((BuildInType))
    BuildInType2 --- Real((Real))
    VarDeclaration2 --- NameList2((NameList))
    NameList2 --- Identifier7((Identifier))
    Identifier7 --- rv((rv))
    
    TypeDeclList --- VarDeclaration3((VarDeclaration))
    VarDeclaration3 --- Type3((Type))
    Type3 --- BuildInType3((BuildInType))
    BuildInType3 --- Boolean((Boolean))
    VarDeclaration3 --- NameList3((NameList))
    NameList3 --- Identifier8((Identifier))
    Identifier8 --- bv((bv))
    
    TypeDeclList --- VarDeclaration4((VarDeclaration))
    VarDeclaration4 --- Type4((Type))
    Type4 --- BuildInType4((BuildInType))
    BuildInType4 --- Char((Char))
    VarDeclaration4 --- NameList4((NameList))
    NameList4 --- Identifier9((Identifier))
    Identifier9 --- cv((cv))
    
    TypeDeclList --- VarDeclaration5((VarDeclaration))
    VarDeclaration5 --- Type5((Type))
    Type5 --- BuildInType5((BuildInType))
    BuildInType5 --- Integer((Integer))
    VarDeclaration5 --- NameList5((NameList))
    NameList5 --- Identifier10((Identifier))
    Identifier10 --- p((p))
    
    TypeDeclList --- VarDeclaration6((VarDeclaration))
    VarDeclaration6 --- Type6((Type))
    Type6 --- AstArrayType((AstArrayType))
    AstArrayType --- range((range))
    range --- EnumRangeType((EnumRangeType))
    EnumRangeType --- lowBound((lowBound))
    lowBound --- Identifier11((Identifier))
    Identifier11 --- lb((lb))
    EnumRangeType --- upBound((upBound))
    upBound --- Identifier12((Identifier))
    Identifier12 --- ub((ub))
    Type6 --- type((type))
    type --- BuildInType6((BuildInType))
    BuildInType6 --- integer((integer))
    
    RoutineDeclList --- NamesList((NamesList))
    NamesList --- Identifier13((Identifier))
    Identifier13 --- iv1((iv))
    
    CompoundStatement --- ForStatement((ForStatement))
    ForStatement --- Identifier14((Identifier))
    Identifier14 --- iv2((iv))
    ForStatement --- IsAdd((IsAdd))
    IsAdd --- true((true))
    ForStatement --- Identifier15((Identifier))
    Identifier15 --- r((r))
    ForStatement --- AssignStatement2((AssignStatement))
    AssignStatement2 --- BinaryExpression1((BinaryExpression))
    BinaryExpression1 --- Identifier16((Identifier))
    Identifier16 --- i((i))
    BinaryExpression1 --- Identifier17((Identifier))
    Identifier17 --- i1((i))
    BinaryExpression1 --- Integer13((Integer))
    Integer13 --- i13((i))
    BinaryExpression1 --- Identifier18((Identifier))
    Identifier18 --- iv3((iv))
    BinaryExpression1 --- Integer14((Integer))
    Integer14 --- i14((i))
    
    CompoundStatement --- CompoundStatement2((CompoundStatement))
    CompoundStatement2 --- SysProcedureCall((SysProcedureCall))
    SysProcedureCall --- SysProcedure((SysProcedure))
    SysProcedure --- writeIn((writeIn))
    writeIn --- Identifier19((Identifier))
    Identifier19 --- s((s))
    SysProcedureCall --- ArgList((ArgList))
    ArgList --- ArrayReference((ArrayReference))
    ArrayReference --- Identifier20((Identifier))
    Identifier20 --- i2((i))
    ArrayReference --- Identifier21((Identifier))
    Identifier21 --- i21((i))
    ArrayReference --- Integer15((Integer))
    Integer15 --- i15((i))
    
    CompoundStatement2 --- AssignStatement3((AssignStatement))
    AssignStatement3 --- Identifier22((Identifier))
    Identifier22 --- a((a))
    AssignStatement3 --- BinaryExpression2((BinaryExpression))
    BinaryExpression2 --- Identifier23((Identifier))
    Identifier23 --- i23((i))
    BinaryExpression2 --- Identifier24((Identifier))
    Identifier24 --- i24((i))
    BinaryExpression2 --- Integer16((Integer))
    Integer16 --- i16((i))
    
    AssignStatement --- Identifier25((Identifier))
    Identifier25 --- i25((i))
    AssignStatement --- BinaryExpression3((BinaryExpression))
    BinaryExpression3 --- Identifier26((Identifier))
    Identifier26 --- i26((i))
    BinaryExpression3 --- Integer17((Integer))
    Integer17 --- i17((i))

```

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、链接后再传给不同目标的后端转换成二进制代码，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。

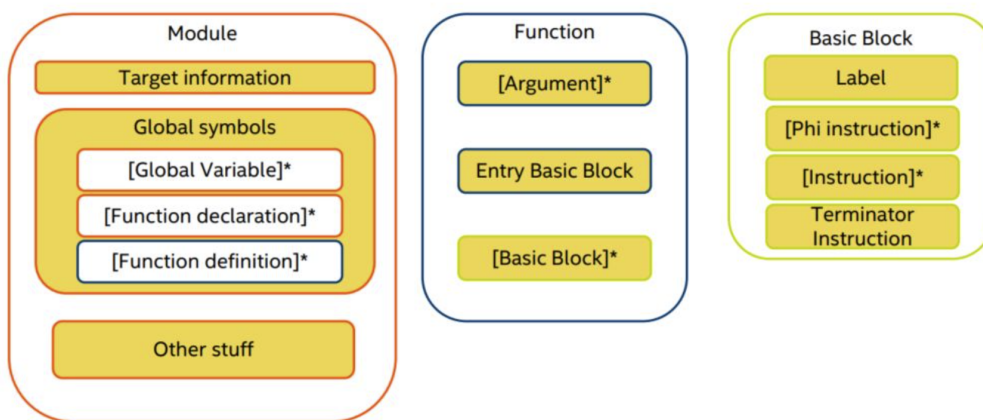


3.2.1 IR布局

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。

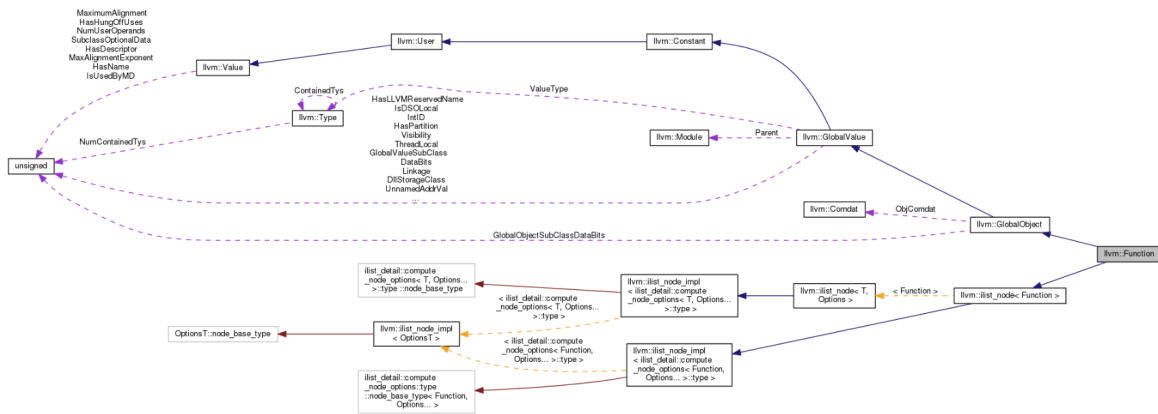
每一个基本块包含了标签和各种指令的集合，标签作为指令的索引用于实现指令间的跳转，指令包含Phi指令、一般指令以及终止指令等。



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



- `Irm::Value`表示一个类型的值，具有一个`Irm::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`Irm::LoadInst`和`Irm::StoreInst`实现，也可以借助IRBuilder的`CreateLoad`和`CreateStore`实现。
- `Irm::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  enum TypeID {
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
mantissa)
9      PPC_FP128TyID,    ///< 6: 128-bit floating point type (two 64-bits,
PowerPC)
10     LabelTyID,        ///< 7: Labels
11     MetadataTyID,     ///< 8: Metadata
12     X86_MMXTyID,      ///< 9: MMX vectors (64 bits, x86 specific)
13     TokenTyID,        ///< 10: Tokens
14
15     // Derived types... see DerivedTypes.h file.
16     // Make sure FirstDerivedTyID stays up to date!
17     IntegerTyID,      ///< 11: Arbitrary bit width integers
18     FunctionTyID,     ///< 12: Functions
19     StructTyID,       ///< 13: Structures
20     ArrayTyID,        ///< 14: Arrays
21     PointerTyID,      ///< 15: Pointers
22     VectorTyID        ///< 16: SIMD 'packed' format, or other vector
type
23 };

```

- `Irm::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 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];

```

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    }
33 }

```

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和确定数组大小，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

3.3.6

3.3.7

3.3.8

3.3.9

3.3.10

第四章 优化考虑

(每个阶段的优化考虑)

第五章 代码生成

(所有语句的代码生成的处理)

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 |
```

- IR
- 汇编指令
- 运行结果

6.1.2 数组类型测试

- 测试代码

```
1 |
```

- IR
- 汇编指令
- 运行结果

6.2 运算测试

- 测试代码

1 |

- IR
- 汇编指令
- 运行结果

6.3 控制流测试

6.3.1 分支测试

- 测试代码

1 |

- IR
- 汇编指令
- 运行结果

6.3.2 循环测试

- 测试代码

1 |

- IR
- 汇编指令
- 运行结果

6.3.3 Goto测试

- 测试代码

1 |

- IR
- 汇编指令
- 运行结果

6.4 函数测试

6.4.1 简单函数测试

- 测试代码

1 |

- IR
- 汇编指令
- 运行结果

6.4.2 递归函数测试

- 测试代码

1 |

- IR
- 汇编指令
- 运行结果

6.4.3 引用传递测试

- 测试代码

1 |

- IR
- 汇编指令
- 运行结果

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     end;
22 end
23 ;
24 end
25 ;
26
27 begin
28     i := go(10);
29     writeln(i);
30 end
```

- 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      %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     ret void
14 }
15
16 declare i32 @printf(i8*, ...)
17
18 define internal i32 @go(i32) {
19 entrypoint:
20     %go = alloca i32
21     %a = alloca i32
22     store i32 %0, i32* %a
23     %tmp = load i32, i32* %a
24     %tmpEQ = icmp eq i32 %tmp, 1
25     %ifCond = icmp ne i1 %tmpEQ, false
26     br i1 %ifCond, label %then, label %else
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
20     .p2align    4, 0x90                ## -- End function
21     .p2align    4, 0x90                ## -- Begin function go
22  _go:                                ## @go
23     .cfi_startproc
24  ## %bb.0:                             ## %entrypoint
25     pushq    %rbx
26     .cfi_def_cfa_offset 16
27     subq     $16, %rsp
28     .cfi_def_cfa_offset 32
29     .cfi_offset %rbx, -16
30     movl     %edi, 8(%rsp)
31     cmpl     $1, %edi
32     je      LBB1_1
33  ## %bb.3:                             ## %else
34     cmpl     $2, 8(%rsp)
35     jne     LBB1_4
36  LBB1_1:                                ## %then
37     movl     $1, 12(%rsp)
38  LBB1_2:                                ## %merge
39     movl     12(%rsp), %eax
40     addq     $16, %rsp
41     popq     %rbx
42     retq
43  LBB1_4:                                ## %else5
44     movl     8(%rsp), %edi
45     decl     %edi
46     callq    _go
47     movl     %eax, %ebx
48     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 .

```

- 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
16 i8], [4 x i8]* @.str, i32 0, i32 0), i32 %tmp)
17     %tmp1 = load i32, i32* @k
18     %printf2 = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([4 x
19 i8], [4 x i8]* @.str.1, i32 0, i32 0), i32 %tmp1)
20     ret void
21 }
22
23 declare i32 @printf(i8*, ...)
24
25 define internal i32 @go(i32* nonnull, i32) {
26 entrypoint:
27     %t = alloca double
28     %fk = alloca i32
29     %go = alloca i32
30     %a = alloca i32
31     %b = getelementptr i32, i32* %0, i32 0
32     store i32 %1, i32* %a
33     %tmp = load i32, i32* %a
34     %tmpSGT = icmp sgt i32 %tmp, 0
35     %ifCond = icmp ne i1 %tmpSGT, false
36     br i1 %ifCond, label %then, label %else
37
38 then:
39     ; preds = %entrypoint
40     %tmp1 = load i32, i32* %a
41     %tmp2 = load i32, i32* %a
42     %subtmpi = sub i32 %tmp2, 1
43     %calltmp = call i32 @go(i32* %b, i32 %subtmpi)
44     %multmpi = mul i32 %tmp1, %calltmp
45     store i32 %multmpi, i32* %go
46     br label %merge
47
48 else:
49     ; preds = %entrypoint
50     store i32 1, i32* %go
51     br label %merge
52
53 merge:
54     ; preds = %else, %then
55     %tmp3 = load i32, i32* %b
56     %tmp4 = load i32, i32* %go
57     %addtmpi = add i32 %tmp3, %tmp4
58     store i32 %addtmpi, i32* %b
59     %tmp5 = load i32, i32* @k
60     %tmp6 = load i32, i32* %go
61     %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
26    .p2align    4, 0x90                ## -- End function
27    _go:                                     ## -- Begin function go
28    ## @go
29    .cfi_startproc
30    ## %bb.0:                                ## %entrypoint
31    pushq    %r14
32    .cfi_def_cfa_offset 16
33    pushq    %rbx
34    .cfi_def_cfa_offset 24
35    subq     $24, %rsp
36    .cfi_def_cfa_offset 48
37    .cfi_offset %rbx, -24
38    .cfi_offset %r14, -16
39    movq     %rdi, %rbx
40    movl     %esi, 4(%rsp)
41    testl    %esi, %esi
42    jle LBB1_2
43    ## %bb.1:                                ## %then
44    movl     4(%rsp), %r14d
45    leal     -1(%r14), %esi
46    movq     %rbx, %rdi
47    callq    _go
48    imull    %r14d, %eax
49    movl     %eax, (%rsp)
50    jmp LBB1_3
51    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

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 .

```

- 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     ret void
16 }
17
18 declare i32 @printf(i8*, ...)
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
27     .p2align    4, 0x90                     ## -- End function
28     _gcd:                                     ## -- Begin function gcd
29     .cfi_startproc                         ## @gcd
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
54     .globl    _ans                         ## -- End function
55     .zerofill __DATA,__common,_ans,4,2    ## @ans
56     .section    __TEXT,__const
57     .globl    _.str                       ## @.str
58     _.str:
59     .asciz    "%d\n"
60
61

```

- 运行结果

第柒章 总结
