一、需求分析

1 Pascal-S语言说明 （源语言）

1.1 基本简介

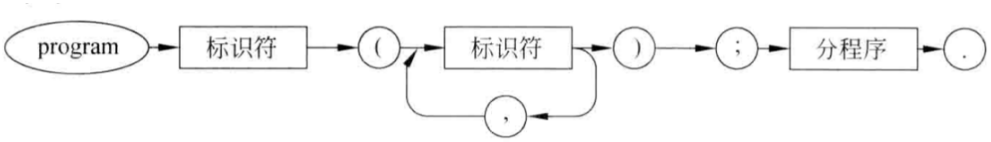
Pascal-S语言中包含整数类型、实数类型、布尔类型、字符类型4种基本数据类型，还有数组和记录两种构造类型，另外，用户还可以定义自己的类型。

在程序结构和语句结构上，与Pascal语言没有太大的区别，只是没有标号说明，当然也不存在goto语句。

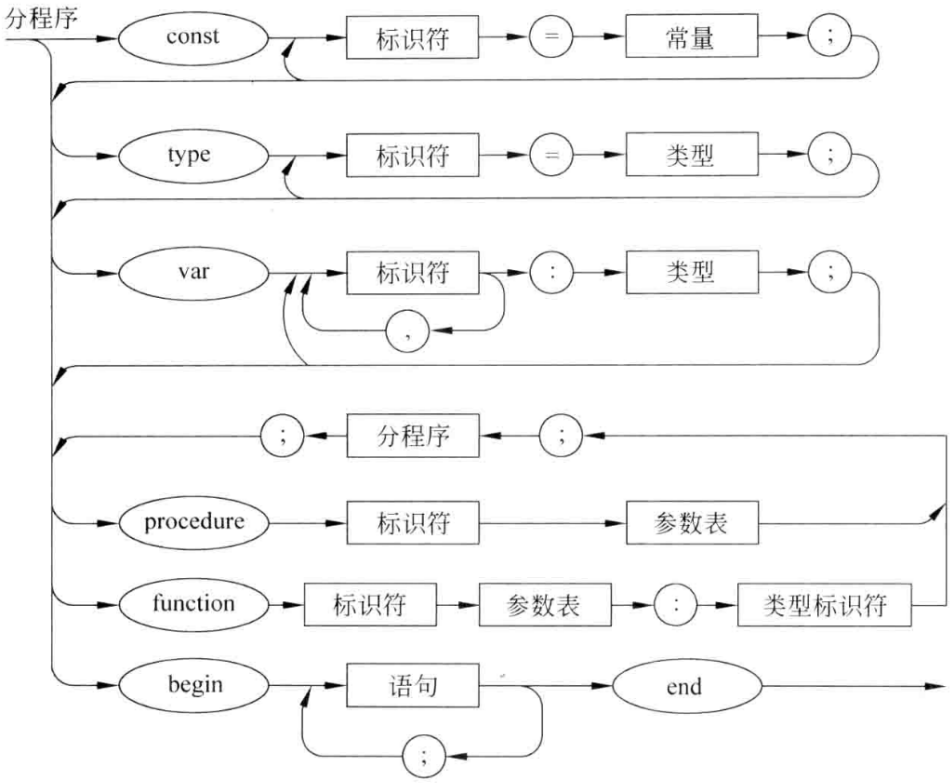
Pascal-S语言允许过程或函数带有参数，支持传值调用，不允许过程作为参数传递（没有返回值），允许函数作为结果返回。

1.2 语法图（核心部分）

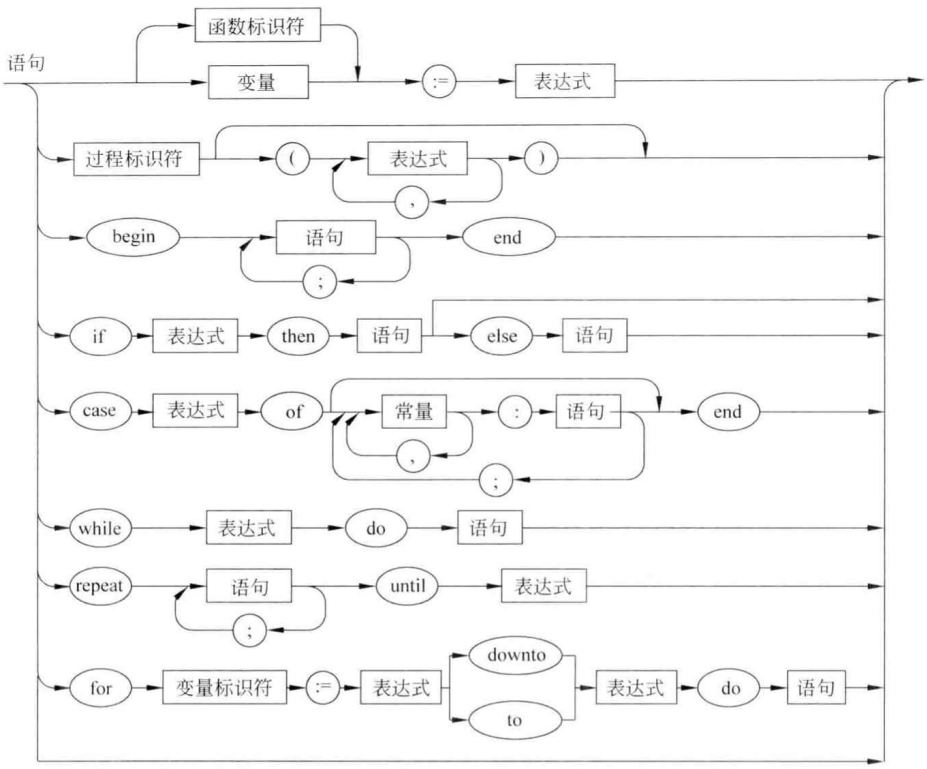
- 程序



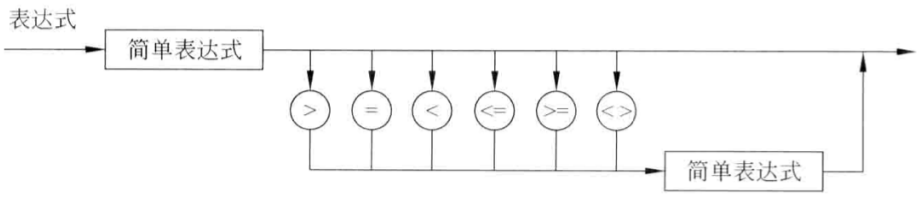
- 分程序


分程序
const
标识符
=
常量
;
type
标识符
=
类型
;
var
标识符
:
类型
;
,
;
分程序
;
procedure
标识符
参数表
function
标识符
参数表
:
类型标识符
begin
语句
end
;

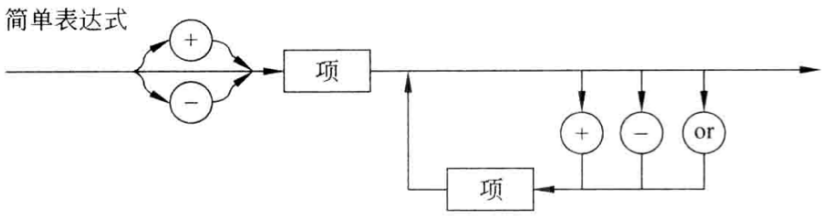
- 语句


语句
函数标识符
变量
:=
表达式
过程标识符
(
表达式
)
,
begin
语句
end
;
if
表达式
then
语句
else
语句
case
表达式
of
常量
:
语句
end
,
;
while
表达式
do
语句
repeat
语句
until
表达式
;
for
变量标识符
:=
表达式
downto
to
表达式
do
语句

- 表达式


表达式
简单表达式
>
=
<
<=
>=
<>
简单表达式

- 简单表达式



1.3 文法描述

1.3.1 关键字

Pascal-S语言中涉及的关键字有 and、array、begin、boolean、case、const、div、do、downto、else、and、for、function、if、integer、mod、not、of、or、procedure、program、real、record、repeat、then、to、type、until、var、while

1.3.2 专用符号

算术算符	+、-、*、/、mod、div
逻辑算符	<、<=、>、>=、=、<>、and、or、not
赋值号	:=
子界符	..
分界符	,、;、.、(、)、[、]、{、}
注释起止符	(*、*)、{、}

1.3.3 词法说明

- 程序中的注释可以出现在任何单词符号之后，用分界符号 “{” 和 “}” 或 “(*” 和 “*)” 括起来
- 程序中的关键字（除开头的program和末尾处的end之外）的前后必须有空格符或换行符，其他单词符号间的空格符是可选的
- 关键字作为保留字
- 标识符记号id匹配以字母开头的字母数字串，其最大长度规定为8个字符。正则表达式描述为

$$\mathbf{letter \rightarrow [a\text{-}zA\text{-}Z]}$$

$$\mathbf{digit \rightarrow [0\text{-}9]}$$

$$\mathbf{Id \rightarrow letter(letter|digit)^*}$$

- “数”的记号num匹配无符号整型常数或无符号实型常数。正则表达式描述为

$$\mathbf{digits \rightarrow digit(digit)^*}$$

$$\mathbf{optional\_fraction \rightarrow .digits \mid \varepsilon}$$

$$\mathbf{optional\_exponent \rightarrow E(+|-|\ \varepsilon)digits \mid \varepsilon}$$

num → digits optional_fraction optional_exponent

- 关系运算符relop代表 “=” 、 “<>” 、 “<” 、 “<=” 、 “>” 、 “>=”
- **addop**代表运算符 “+” 、 “-” 、 “or”
- **mulop**代表运算符 “*” 、 “/” 、 “div” 、 “mod” 、 “and”
- **assignop**代表赋值号 “:=”

1.3.4 产生式合集

PowerShell

```
program → program_head program_body.

program_head → program id (identifier_list);

program_body → const_declarations
               type_declarations
               var_declarations
               subprogram_declarations
               compound_statement

identifier_list → identifier_list,id | id

const_declarations → const const_declaration; | ε

const_declaration → const_declaration; id = const_variable
                  | id = const_variable

const_variable → +id | -id | id |
                 +num | -num | num |
                 'letter'

type_declarations → type type_declaration; | ε

type_declaration → type_declaration; id = type | id = type

type → standard_type
             | record record_body end
             | array [periods] of type

standard_type → integer | real | boolean | char

record_body → var_declaration
            | ε

periods → periods,period | period

period → const_variable .. const_variable
```

```
38
39  var_declarations → var var_declaration;
40                   | ε
41
42  var_declaration → var_declaration; identifier_list: type
43                  | identifier_list: type
44
45  subprogram_declarations → subprogram_declarations subprogram_declaration;
46                          |  ε
47
48  subprogram_declaration → subprogram_head program_body
49
50  subprogram_head → function id formal_parameter: standard_type;
51                  | procedure id formal_parameter;
52
53  formal_parameter → ( parameter_lists )
54                   | ε
55
56  parameter_lists → parameter_lists; parameter_list | parameter_list
57
58  parameter_list → var_parameter | value_parameter
59
60  var_parameter → var value_parameter
61
62  value_parameter → identifier_list : standard_type
63
64  compound_statement → begin statement_list end
65
66  statement_list → statement_list; statement
67                 | statement
68
69  statement → variable assignop expression
70              | call_procedure_statement
71              | compound_statement
72              | if expression then statement else_part
73              | case expression of case_body end
74              | while expression do statement
75              | repeat statement_list until expression
76              | for id assignop expression updwon expression do statement
77              | ε
78
79  variable → id id_varparts
80
81  id_varparts → id_varparts id_varpart | ε
82
83  id_varpart → [ expression_list ] | .id
84
85  else_part → else statement | ε
```

```
86
87  case_body → branch_list | ε
88
89  branch_list → branch_list;branch | branch
90
91  branch → const_list : statement
92
93  const_list → const_list,const_variable | const_variable
94
95  updown → to | downto
96
97  call_procedure_statement → id | id(expression_list)
98
99  expression_list → expression_list, expression | expression
100
101 expression → simple_expression relop simple_expression | simple_expression
102
103 simple_expression → term | +term | -term | simple_expression addop term
104
105 term → term mulop factor | factor
106
107 factor → unsign_const_variable
108         | variable
109         | id (expression_list)
110         | (expression)
111         | not factor
112
113 unsign_const_variable → id | num | 'letter'
```

注：实际实现会对以上小部分产生式进行修改以解决移进规约/规约规约冲突。

2 汇编语言语法说明 （目标语言）

由于汇编指令系统庞大，因而需构建指令系统体系，其指令数量庞大，格式复杂，可记忆性差等。指令中最难的是指令所支持的寻址方式，其实质就是指令中操作数如何获取。对于处理器而言，就是如何找到他所需的数据。但对于计算机底层的汇编语言而言，这种寻址方式将涉及大量的计算存储格式，与复杂的存储管理方式紧密相关，因而难以理解。最后，汇编指令还关系到如何影响标志位，但处理器标志位非常复杂，因而对其机制掌握就比较困难。

- **传送指令**

包括通用数据传送指令MOV、条件传送指令CMOV*cc*、堆栈操作指令PUSH/PUSHA/PUSHAD/POP/POPA/POPAD、交换指令XCHG/XLAT/BSWAP、地址或段描述符选择子传送指令LEA/LDS/LES/LFS/LGS/LSS等。

- **逻辑运算**

这部分指令用于执行算术和逻辑运算，包括加法指令ADD/ADC、减法指令SUB/SBB、加一指令INC、减一指令DEC、比较操作指令CMP、乘法指令MUL/IMUL、除法指令DIV/IDIV、符号扩展指令CBW/CWDE/CDQE、十进制调整指令DAA/DAS/AAA/AAS、逻辑运算指令NOT/AND/OR/XOR/TEST等。

- **移位指令**

这部分指令用于将寄存器或内存操作数移动指定的次数。包括逻辑左移指令SHL、逻辑右移指令SHR、算术左移指令SAL、算术右移指令SAR、循环左移指令ROL、循环右移指令ROR等。

- **位操作**

这部分指令包括位测试指令BT、位测试并置位指令BTS、位测试并复位指令BTR、位测试并取反指令BTC、位向前扫描指令BSF、位向后扫描指令BSR等。

- **控制转移**

这部分包括无条件转移指令JMP、条件转移指令JCC/JCXZ、循环指令LOOP/LOOPE/LOOPNE、过程调用指令CALL、子过程返回指令RET、中断指令INTn、INT3、INTO、IRET等。

- **串操作**

这部分指令用于对数据串进行操作，包括串传送指令MOVS、串比较指令CMPS、串扫描指令SCANS、串加载指令LODS、串保存指令STOS，这些指令可以有选择地使用REP/REPE/REPZ/REPNE和REPNZ的前缀以连续操作。

- **输入输出**

这部分指令用于同外围设备交换数据，包括端口输入指令IN/INS、端口输出指令OUT/OUTS。

3 功能描述

设计实现的Pascal-S编译程序可以对Pascal-S源代码进行编译，依次经过词法分析、语法分析、语义分析、中间代码生成、中间代码优化、汇编代码生成、最终生成可执行程序。

我们的编译程序可以识别所有的Pascal-S单词，在词法出问题时会直接进行词法的报错。

- 程序支持const类型定义，type类型定义，var类型定义，子程序定义和主程序定义。
- const数据类型支持+id，-id，id，+num，-num以及字符。
- 程序支持的数据类型有standard type（integer，real，boolean，char）以及array
- array的定义仅支持一维数组且数组下标默认从0开始
- 子程序定义实现了function和procedure，只支持传值调用
- 主程序中实现了赋值语句，过程调用语句（function+procedure），if then，case of，while do，repeat until，for id assignop语句（Pascal-S中定义的均支持）

4 工具说明

1. Lex

> http://dinosaur.compilertools.net/

Lex帮助编写程序，这些程序的控制流是由输入流中的正则表达式实例指导的。它非常适合编辑器-脚本类型转换和分段输入，以便为解析例程做准备。

Lex的源(source)是一个正则表达式表和相应的程序片段。这个表被转换成一个程序，该程序读取输入流，将其复制到输出流，并将输入划分为与给定表达式匹配的字符串。当每个这样的字符串被识别时，相应的程序片段就会被执行。对这些表达式的识别是由Lex生成的确定性有限自动机完成的。用户编写的程序片段将按照相应正则表达式在输入流中出现的顺序执行。

2. Yacc

> http://dinosaur.compilertools.net/

Yacc提供了一种通用工具来描述计算机程序的输入。Yacc用户指定输入的结构，以及在识别每个这样的结构时要调用的代码。Yacc将这种规范转换为处理输入过程的子例程;通常，让这个子例程处理用户应用程序中的大部分控制流是方便和合适的。

3. Llvm

> https://releases.llvm.org/6.0.1/docs/index.html

LLVM是构架编译器(compiler)的框架系统，以C++编写而成，用于优化以任意程序语言编写的程序的编译时间(compile-time)、链接时间(link-time)、运行时间(run-time)以及空闲时间(idle-time)，对开发者保持开放，并兼容已有脚本。

4. Make

> https://www.gnu.org/software/make/manual/make.html

在软件开发中，make是一个工具程序（Utility software），经由读取叫做“makefile”的文件，自动化建构软件。它是一种转化文件形式的工具，转换的目标称为“target”；与此同时，它也检查文件的依赖关系，如果需要的话，它会调用一些外部软件来完成任务。它被用来编译源代码，生成结果代码，然后把结果代码连接起来生成可执行文件或者库文件。

5. GitLab 仓库与CI/CD

> https://about.gitlab.com/

本工程使用git进行版本控制，使用多个分支防止前后端开发的耦合和冲突，并且使用gitlab进行远程仓库的管理。

GitLab CI 是GitLab内置的进行持续集成的工具，只需要在仓库根目录下创建.gitlab-ci.yml 文件，并配置GitLab Runner；每次提交的时候，gitlab将自动识别到.gitlab-ci.yml文件，并且使用Gitlab Runner执行该脚本。在本工程中它与Make工具配合使用，使得小组成员上传的代码可以被自动编译与检查。

二、总体设计

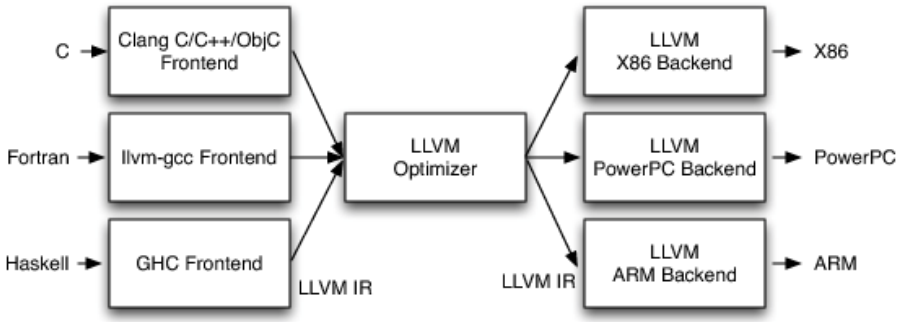
1 技术路线

整体采用的技术路线以使用工具为主，具体路线如下：

词法分析	语法分析	语义分析	中间代码生成	优化	目标代码生成
Lex	Yacc	Yacc	采用LLVM-IR进行中间代码表示 并且利用LLVM工具对其进行优化		采用LLVM工具生成自定平台的汇编代码

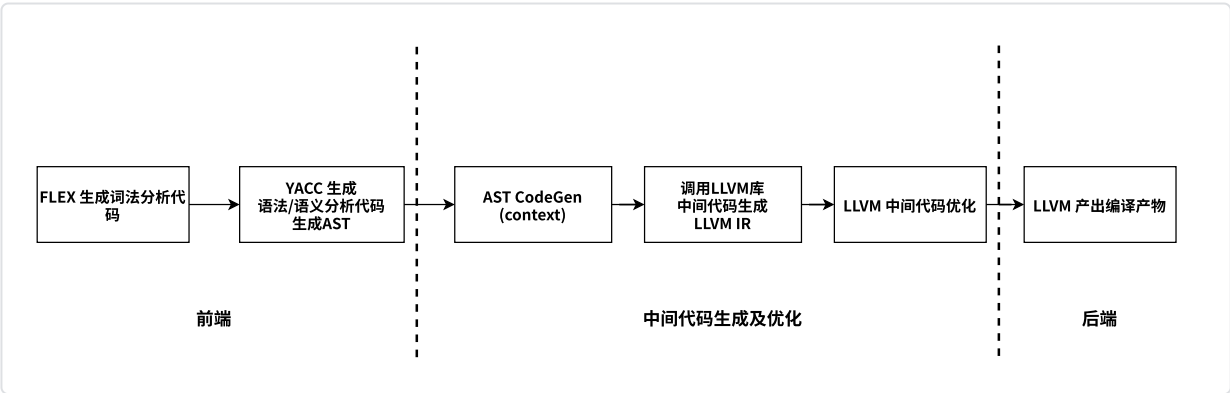
2 实现方法说明

由于LLVM的设计思想采用三段化设计，将前端后端进行分离，并利用LLVM-IR中间代码进行统一的中间代码优化。



根据这一基本思想，得到关键思路分为如下步骤：

1. 将源程序字符流进行词法分析产生Token关键字流
2. 利用Token关键字流进行语法分析
3. 语法分析翻译制导构建 AST 抽象语法树
4. 遍历 AST 并将 AST 节点执行语义分析将保存的信息转换为 LLVM 上下文信息
5. LLVM 上下文信息转换为LLVM-IR
6. LLVM-IR 代码优化
7. 将LLVM-IR利用LLVM和g++工具转化为汇编代码和可执行文件



3 工程整体设计

3.1 工程模块划分与关系

依据上文中对整体的实现方法说明，可以得出我们工程建构的基本划分应当以AST作为中间的链接关键，并通过AST将工程部件分为四部分：

- 第一部分：词法分析、语法分析、语义分析（利用AST节点定义构建AST）
- 第二部分：AST节点的结构设计
- 第三部分：生成AST后遍历AST并进行IR生成
- 第四部分：对产生的IR代码进行优化、生成汇编与可执行文件等操作

对于上述的四个部分，我们主要是利用如下设计实现的：

3.1.1 主控程序

- 实现文件：main.cpp
- 功能：对整个程序的流程进行控制
- 输入：源程序字符流
- 输出：IR 代码文件、汇编文件、bitcode文件、可执行二进制
- 调用者：无
- 供外部调用的函数：int main()

3.1.2 词法分析

- 实现文件：pascals.l（利用flex工具生成 token.cpp）
- 功能：词法分析
- 输入：源程序字符流
- 输出：token流
- 调用者：定义在parser.cpp 中的 parser()
- 供外部调用的函数：int yylex(void) 返回token

3.1.3 语法分析&语义分析

- 实现文件：pascals.y（利用bison工具生成 parser.cpp）
- 功能：语法分析 语义分析
- 输入：token流
- 输出：AST （利用全局变量 AST::ProgramNode *astRoot进行构建）
- 调用者：int main()
- 供外部调用的函数：int yyparse (void)

3.1.4 AST 定义

- 实现文件：AST.cpp AST.h
- 功能：AST节点设计定义 与 CodeGen(CodeGenContext &context) 实现
- 输入：AST::ProgramNode *astRoot（根节点）
- 输出：隐式将信息输出至CodeGenContext

- 调用者：int main() int yyparse (void)
- 供外部调用的函数：int yyparse (void)

3.1.5 CodeGenContext定义

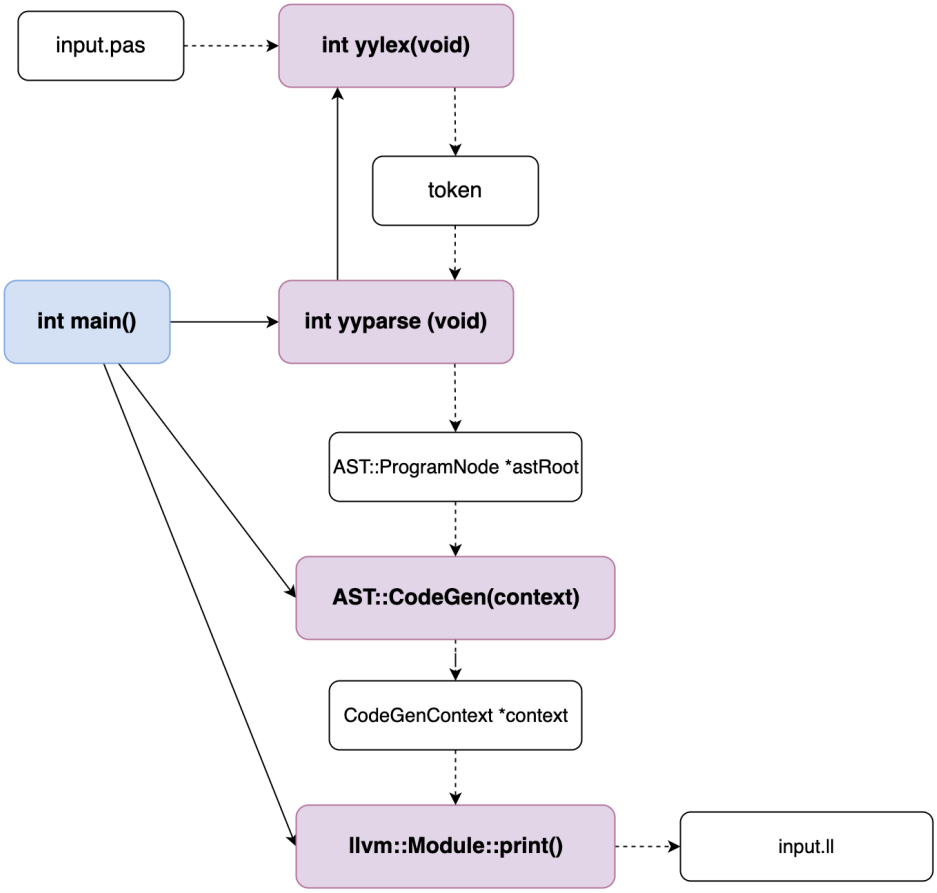
- 实现文件：CodeGenContext.h CodeGenContext.cpp
- 功能：将AST通过调用 CodeGen(CodeGenContext &context) 将信息转化为 CodeGenContext 存储
- 输入：AST CodeGen() 隐式输入信息 包括代码块 符号表插入
- 输出：.ll 文件中 的 IR 代码
- 调用者：CodeGen(CodeGenContext &context)
- 供外部调用的函数：llvm::Value *getValue(std::string name), llvm::Function *curFunc(), llvm::BasicBlock *curBlock(), void pushBlock(llvm::BasicBlock *block), void llvm::Module::print(raw_ostream &OS, AssemblyAnnotationWriter *AAW) 等
- 重要的成员变量：llvm::Module *module, llvm::IRBuilder<> builder

3.1.6 IR代码优化与目标代码生成

- 实现文件：main.cpp
- 功能：通过工具链调用方式将IR代码进行优化转汇编等操作
- 输入：.ll 文件
- 输出：优化后的 IR 代码文件、汇编文件、bitcode文件、可执行二进制
- 调用者：int main()

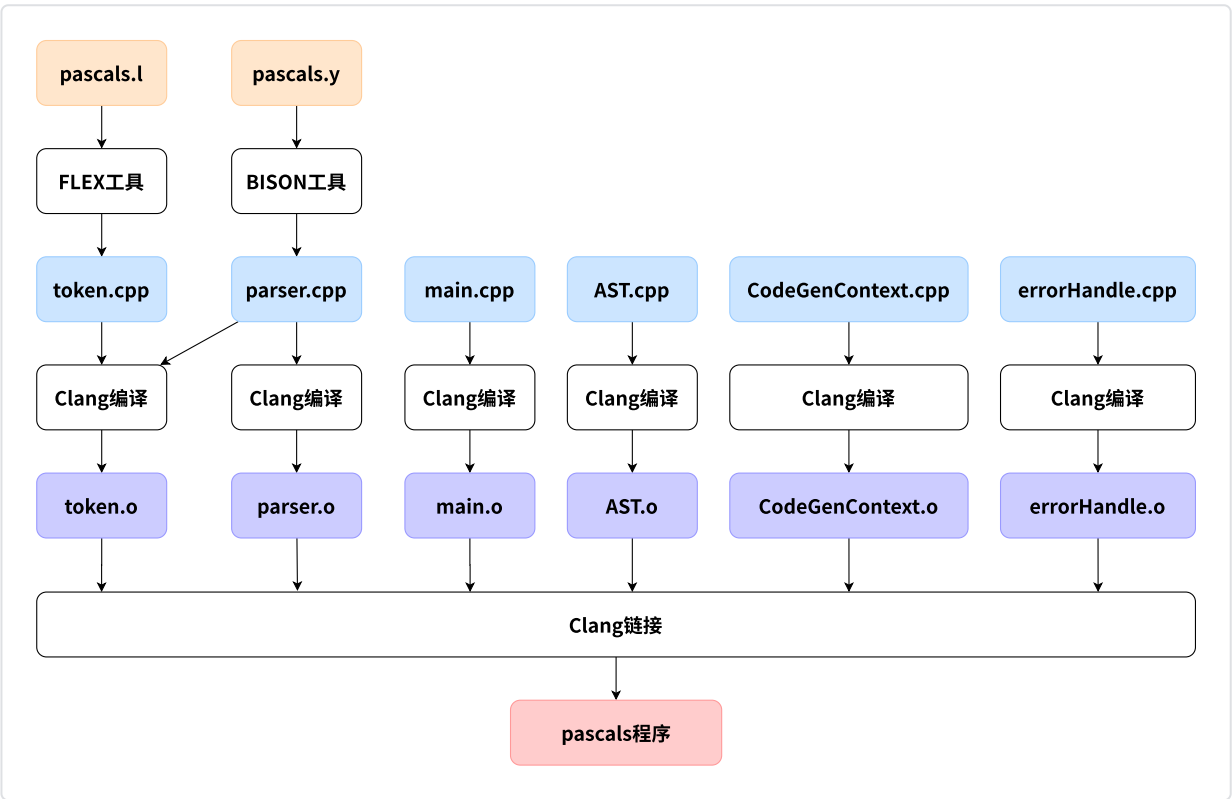
3.2 模块调用关系图

> 其中白色块代表信息，有可能是文件，有可能是对象或流
>
> 紫色和蓝色代表处理过程
>
> 实线箭头指向的是被调用的过程



3.3 工程文件编译流程

工程程序的编译流程如下，其中白色方框表示运行编译、生成、链接等过程



4 软件工程方法实践

本工程采用先完成一个最小化的需求后，逐步进行需求迭代的开发方式，配合gitlab进行代码的分支控制与版本控制，利用gitlab中的CI/CD编写自动化脚本，进行编译和运行测试，不断向主分支进行开发、测试和集成流程。

此外，由于本工程中运用的工具链较多，环境较为复杂，为了解决成员各自设备环境不一致的问题，我们统一在云服务器上分别建立用户进行代码开发，提高了开发调试的效率。

最后在程序安装方面我们还编写了.sh脚本帮助用户一键安装我们的编译程序，自动检查系统中的依赖关系，并自动化解决依赖。可以从下面的链接查看安装教程。

> https://gitlab.com/chengmin_zhang/pascals_compiler_installer

三、详细设计

1 词法分析

根据源语言Pascal-S的词法，编写flex代码，实现Pascal-S语言的词法分析。

Pascal-S语言的结构可见需求分析中对Pascal-S语言的说明，对语言的词法分析原理比较简单，对应输入的文件识别出对应的词法。并返回相应的值给bison做语法分析。

词法分析程序的主要目标是完整，无错误的识别出源语言代码中的所有单词。

根据Pascal-S语言的词法，设计如下的flex识别代码。

- 正则表达式部分：

```Fortran
DIGIT [0-9]
LETTER [a-zA-Z]
ID {LETTER}({LETTER}|{DIGIT})*
DIGITS {DIGIT}+
FRACTION "."{DIGITS}
EXPONENT [Ee][+-]?{DIGITS}
INTNUM {DIGITS}
REALNUM {DIGITS}({FRACTION})?({EXPONENT})?
WRAP [\n]
DELIM [" "\t\f\v]
WS {DELIM}+
OTHERS .
```

- 识别符号部分（部分示例）：

```Kotlin
"+"         { beginToken(TOKENLENGTH); return PLUS; }
"<>"        { beginToken(TOKENLENGTH); return NE; }
```

- 识别关键字部分：

```
Kotlin
```

```
begin       { beginToken(TOKENLENGTH); return BEGINN; }
integer     { beginToken(TOKENLENGTH); return INTEGER; }
real        { beginToken(TOKENLENGTH); return REAL; }
char        { beginToken(TOKENLENGTH); return CHAR; }
false       { beginToken(TOKENLENGTH); return FALSE; }
true        { beginToken(TOKENLENGTH); return TRUE; }
```

- 其他部分（使用正则表达式）：

下列识别的对应内容可以对应正则表达式部分的描述进行理解。

```
PowerShell
```

```
"'"{OTHERS}"'"    { beginToken(TOKENLENGTH); yylval.c = yytext[1]; return LETTE
R; }
{ID}        { beginToken(TOKENLENGTH); yylval.s = strdup(yytext); return ID; }
{INTNUM}    { beginToken(TOKENLENGTH); yylval.t = atoi(yytext); return INTNUM; }
{REALNUM}   { beginToken(TOKENLENGTH); yylval.f = atof(yytext); return REALNUM;
 }
{WRAP}      { beginToken(TOKENLENGTH); }
{DELIM}     { beginToken(TOKENLENGTH); }
{WS}        { beginToken(TOKENLENGTH); }
{OTHERS}    { beginToken(TOKENLENGTH); parseError = 1; printError("Invalid Inpu
t"); }
```

注：beginToken是对识别的单词进行统计，对单词长度，行数等内容进行记录，在后端使用。

2 语法分析

根据Pascal-S源语言的产生式，可以编写对应的bison（yacc）代码进行语法识别与匹配。

bison中使用了多种数据类型，具体定义如下：

```
C++
```

```
%union {
    bool b;
    int t;
    double f;
    char c;
    char* s;
}
```

可以看到，union中支持int，double，bool，char，char*五种基本数据类型。

注：其实union中还需设计一些用于构建AST的指针类型，因与语法分析无直接关联，故在此并不写出

bison（yacc）对说明文件中的 %token NUMBER 声明的对应。bison（yacc）坚持定义所有的符号记号本身，而不是从别的地方引入一个定义。但是却有可能通过在记号声明中的记号名之后书写一个值来指定将赋给记号的数字值。

如下是我们bison代码的定义部分。

Visual Basic

```
/*四种数据类型*/
%token <t> NE GE GT LE LT AS
%token <f> REALNUM
%token <t> INTNUM
%token <c> CHAR EQ OP CP PLUS MINUS MULTIPLY DIVIDE OB CB DOT COMMA COLON SEMI
LETTER
%token <b> TRUE FALSE
%token BEGINN DO DOWNTO REPEAT END THEN TO FOR UNTIL WHILE
%token FUNCTION PROCEDURE PROGRAM VAR RECORD TYPE ARRAY CONST
%token IF ELSE CASE OF AND OR NOT BOOLEAN DIV MOD INTEGER REAL
%token <s> ID
```

根据flex识别出并返回给bison的终结符，定义为%token。如INTNUM，REALNUM这类记号，会给其数字值进行赋值，以获得int类型数字，realnum类型数字本身的值（通过yylval实现）

bison（yacc）采用LALR(1)语法分析方法。输入的是巴科斯范式（BNF）表达的语法规则以及语法规约的处理代码，输出的是基于表驱动的编译器，包含输入的语法规约的处理代码部分。

bison（yacc）与flex（lex）一起使用，bison利用flex识别并返回给它出的词法，去匹配对应bison代码编写好的语法。并进行相应的移进 / 规约操作。

可见，对Pascal-S代码的语法分析过程就是将Pascal-S的产生式合集转化为bison代码的过程。

但由于产生式中内容的部分被我们进行了修改，以消除一些移进规约，规约规约冲突。

如下是bison代码的规则部分（部分示例）：

```
Ada
```

```
S : program

program : program_head program_body DOT {}

program_head : PROGRAM ID OP identifier_list CP SEMI {}

program_body : const_parts type_parts var_parts subprogram_declarations compound
_statement
               {}
```

可以看到编写的代码和产生式有一定的区别，在编写的过程中也产生了不少的问题，如移进规约冲突，规约规约冲突等。

关于这部分的调试的相关过程和信息会在课设设计总结中进行描述和讨论。

3 语义分析

3.1 AST 节点定义

在AST的节点定义方面，其主要工作是供构建AST使用，在节点的含义上，是语法产生式的一种形式化表示，其中包含了成员变量封装、继承等结构化设计，以下举一些有特点的处理方式的例子：

- **公共父类 BaseNode**

```
C++
```

```
class BaseNode {
  public:
    enum NodeType type = BASE;

    virtual ~BaseNode() {}
    virtual llvm::Value *CodeGen(CodeGenContext &context) = 0;
    virtual std::string description() = 0;
};
```

> enum NodeType type 记录节点的 type 值用作标记
>
> llvm::Value *CodeGen(CodeGenContext &context) 函数用作该节点的信息转入context中
>
> std::string description() 用作返回节点的描述信息 帮助调试

- **根节点 ProgramNode**

C++

```cpp
class ProgramNode : public BaseNode {
  public:
    ProgramHeadNode *programHead;
    ProgramBodyNode *programBody;

    ProgramNode(ProgramHeadNode *programHead, ProgramBodyNode *programBody);

    virtual llvm::Value *CodeGen(CodeGenContext &context);
    virtual std::string description();
};
```

> 对应产生式 program → program_head program_body
>
> ProgramNode 包含 programHead 和 programBody 两个成员变量

- **StatementListNode**

C++

```cpp
class StatementListNode : public BaseNode {
  public:
    std::vector<StatementNode *> statementList;

    StatementListNode();

    void insertStatement(StatementNode *statementNode);

    virtual llvm::Value *CodeGen(CodeGenContext &context);
    virtual std::string description();
};
```

> 对应产生式 statement_list → statement_list; statement
>
> | statement
>
> 由于存在左递归这里设计本节点包含std::vector<StatementNode *> statementList成员变量
>
> 通过insertStatement(StatementNode *statementNode) 方式来插入StatementNode 表示左递归

- **ExpressionNode**

C++

```cpp
class ExpressionNode : public FactorNode {
  public:
    virtual llvm::Value *CodeGen(CodeGenContext &context) = 0;
    virtual std::string description() = 0;
};
```

> 对应产生式 expression → simple_expression **relop** simple_expression | simple_expression
>
> 由于expression必定会产生为simple_expression **relop** simple_expression或simple_expression所以设计ExpressionNode为纯虚的Class类型供可能的两种情况节点继承，如下：

- **RelopExprNode**

C++

```cpp
class RelopExprNode : public ExpressionNode {
  public:
    enum RelopType {
      EQ,NE,LT,LE,GT,GE
    };

    SimpleExprNode *leftExpression;
    RelopType relopType;
    SimpleExprNode *rightExpression;

    RelopExprNode(SimpleExprNode *leftExpression, RelopType relopType, SimpleExprNode *rightExpression);

    virtual llvm::Value *CodeGen(CodeGenContext &context);
    virtual std::string description();
};
```

- **UnsignedTermNode**

C++

```cpp
class UnsignedTermNode : public SimpleExprNode {
      public:
        enum MulopType {
          MULT,DIV,INTDIV,MOD,AND,NONEMUL
        };

        MulopType mulopType;
        UnsignedTermNode *leftTerm;
        FactorNode *rightFactor;

        UnsignedTermNode(MulopType mulopType, UnsignedTermNode *leftTerm, Factor Node *rightFactor);
        UnsignedTermNode(FactorNode *rightFactor);

        virtual llvm::Value *CodeGen(CodeGenContext &context);
        virtual std::string description();
    };
```

> 对应产生式 term → term **mulop** factor | factor
>
> 由于存在左递归且属于链式的递归形式但递归的结束肯定是term → factor 所以对于UnsignedTermNode设计了两种构造函数并且利用MulopType:: NONEMUL来进行递归终止标记

3.2 翻译制导构建 AST

3.2.1 AST构建思路

在定义了AST的结构后，需要在bison中填入相应的节点构造函数，才能实现程序AST的构造。

AST的构建不是直接进行的，而是在语法分析时，借助bison的规约动作代码段才能实现。想要正确构建一棵AST，则首先需要了解bison语法分析的原理。

由于bison使用的是LALR(1)分析法，在语法分析过程中，对程序的扫描是从左往右进行的。根据这一特点，对PASCAL-S的产生式进行分析，来实现AST的构造。

3.2.2 AST构建过程分析

本部分对AST的实际构建过程进行分析。

一段PASCAL-S程序一定是从program开始的：

PowerShell

```
program : program_head program_body DOT { $$ = new AST::ProgramNode($1,$2); astRoot=$$; }
        ;
```

在语法分析的过程中，program将在program_head和program_body都被规约之后才被规约，也就是分析过程的最后。此时，规约动作代码段会创建一个ProgramNode，将ProgramHeadNode和ProgramBodyNode作为其构造函数的参数传入，并将AST根结点指向ProgramNode，完成AST的构建。

由于整体代码量较大并且重复性比较高，下面仅选取几个具有代表性的例子，说明其中AST构建操作的含义；bison代码中，$$所代表的是产生式左部符号的变量，$1、$2等则依次代表产生式右部从左开始的符号。

- 一般情况

```Ruby
program_head : PROGRAM ID OP identifier_list CP SEMI { $$ = new AST::ProgramHeadNode(new AST::IdNode($2), $4); }
             ;
```

ProgramHeadNode的构造比较直观，规约动作直接创建一个ProgramHeadNode并将IdNode和IdListNode作为其构造函数的参数传入，再将创建的节点赋给$$即program_head所指向的变量。文法中大部分AST的构造都遵循这个思路。

- 产生式可推空

```PowerShell
const_parts : CONST const_declarations { $$=$2; }
            | { $$ = new AST::ConstDeclsNode(); }
            ;
```

遇到产生式可推空的情况，在推空时也仍然需要构造一个空节点而不插入任何元素，从而方便代码生成时进行判断。文法中还有很多类似此处的可推空产生式，遵循同样思路。

- 产生式具有左递归

```PowerShell
const_declarations : const_declarations ID EQ const_variable SEMI { $$ = $1;
                     $$->insertConstDecl(new AST::ConstDeclNode(new AST::IdNode($2),$4)); }
                   | ID EQ const_variable SEMI { $$=new AST::ConstDeclsNode();
                                                 $$->insertConstDecl(new AST::ConstDeclNode(new AST::IdNode($1),$3)); }
                   ;
```

文法中有很多产生式具有左递归，如const_declarations。在这种情况中，规约时会先规约产生式第二条（此时只扫描到一个ID EQ const_variable SEMI），故此时应先创建一个ConstDeclsNode，再

创建一个ConstDeclNode插入其中；之后的递归过程中则反复规约第一条产生式，向ConstDeclsNode中插入新的ConstDeclNode。

- VariableNode

```PHP
variable : ID OB expression_list CB {
    AST::IdVarPartsNode *id_varparts = new AST::IdVarPartsNode();
    AST::IdVarPartNode *id_varpart = new AST::IdVarPartNode($3);
    id_varparts->insertIdVarPart(id_varpart);
    $$ = new AST::VariableNode(new AST::IdNode($1), id_varparts);
};
```

由于id_varparts可推空会导致出现移进规约冲突，所以在产生式中删除了id_varparts，直接用OB expression_list CB来实现同样功能。但AST的构建中仍需要IdVarPartsNode和IdVarPartNode，所以在代码段中仍需要单独创建这两个节点并存在变量中。

- NotFactorNode

```PowerShell
factor : unsign_const_variable { $$ = $1; }
       | variable { $$ = $1; }
       | ID OP expression_list CP { $$ = new AST::CallProceStatNode(new AST::IdNode($1), $3); }
       | OP expression CP { $$ = $2; }
       | NOT factor { if($2->type == AST::NOT_FACTOR) { $$ = $2; ((AST::NotFactorNode *)$$)->increaseNotNums(); }
                      else { $$ = new AST::NotFactorNode($2); } }
       ;
```

NotFactorNode的特殊之处在于，一个NotFactorNode是在FactorNode的基础上构建的。如果factor本身是单纯的factor，则需要创建一个NotFactorNode，FactorNode作为其构造函数的参数传入；如果factor已经是一个NotFactorNode了，则直接调用它的increaseNotNums成员函数，将not个数增一。

3 中间代码生成

中间代码生成部分的主要工作是编写AST各节点类的CodeGen方法，使编译程序能够通过遍历执行构建的AST各节点的CodeGen方法，将AST翻译为LLVM-IR。

重要AST节点类的中间代码生成实现如下：

- ProgramNode：依次执行成员变量programHead和programBody的CodeGen方法，并创建空返回值。

C++

```cpp
llvm::Value *ProgramNode::CodeGen(CodeGenContext &context) {
    this->programHead->CodeGen(context);
    this->programBody->CodeGen(context);
    context.builder.CreateRetVoid();
    return 0;
}
```

- ProgramHeadNode：将program创建为一个llvm::Function对象，然后在其中创建一个llvm::BasicBlock对象。

C++

```cpp
llvm::Value *ProgramHeadNode::CodeGen(CodeGenContext &context) {
    std::vector<llvm::Type *> paramTypes;
    llvm::FunctionType *funcType = llvm::FunctionType::get(context.builder.getVoidTy(), paramTypes, false);
    llvm::Function *func = llvm::Function::Create(funcType, llvm::GlobalValue::LinkageTypes::ExternalLinkage, "main", context.module);
    context.setParent(func, nullptr);
    llvm::BasicBlock *block = llvm::BasicBlock::Create(globalInstance, "", func);
    context.pushBlock(block);
    context.builder.SetInsertPoint(block);
    for (std::string id : this->idList->idList) {
        context.builder.CreateGlobalStringPtr("", id);
    }
    return 0;
}
```

- ProgramBodyNode：依次执行成员变量constDecls、typeDecls、varDecls、subProgDecls和compoundStat的CodeGen方法。

C++

```cpp
llvm::Value *ProgramBodyNode::CodeGen(CodeGenContext &context) {
    this->constDecls->CodeGen(context);
    this->typeDecls->CodeGen(context);
    this->varDecls->CodeGen(context);
    this->subProgDecls->CodeGen(context);
    this->compoundStat->CodeGen(context);
    return 0;
}
```

- IdNode：根据标识符获取常量或变量地址，加载并返回地址处的数值。

C++

```cpp
llvm::Value *IdNode::CodeGen(CodeGenContext &context) {
    llvm::Value *val = context.getValue(this->idName);
    return context.builder.CreateLoad(val);
}
```

- VarDeclNode：对变量列表中的每个变量，根据上下文判断作用域。若变量为全局变量（program中声明），则在静态存储区为其分配内存；若为局部变量（function/procedure中声明），则在当前函数栈中为其分配内存。

C++

```cpp
llvm::Value *VarDeclNode::CodeGen(CodeGenContext &context) {
    for (std::string id : this->idList->idList) {
        llvm::Type *vartype = this->typeNode->convertToLLVMType(context);
        if (context.isGlobal()) {
            context.setVariable(id, vartype);
        } else {
            context.builder.CreateAlloca(vartype, nullptr, id);
        }
    }
    return 0;
}
```

- AssignExprStatNode：首先判断赋值操作的左值是否为常量，若是常量，则输出“常量无法被赋值”的错误信息，并终止编译；否则，比较左值和右值的类型，按需对右值进行强制类型转换，最后将右值存入左值。

C++

```cpp
llvm::Value *AssignExprStatNode::CodeGen(CodeGenContext &context) {
    llvm::Value *L = this->variable->CodeGen(context);
    llvm::GlobalVariable *g = llvm::dyn_cast<llvm::GlobalVariable>(L);
    if (g != nullptr && g->isConstant()) {
        throw std::logic_error("Constant " + this->variable->id->idName + "cannot be assigned!");
    }
    llvm::Value *R = this->expression->CodeGen(context);
    llvm::Type *LTy = context.builder.CreateLoad(L)->getType();
    llvm::Type *RTy = R->getType();

    if (LTy->isDoubleTy() && RTy->isIntegerTy()) {
        R = context.builder.CreateSIToFP(R, LTy);
    } else if (LTy->isIntegerTy() && RTy->isDoubleTy()) {
        R = context.builder.CreateFPToSI(R, LTy);
    }
    return context.builder.CreateStore(R, L);
}
```

- IntegerNode、CharNode、BooleanNode、RealNode：根据各基本数据类型创建llvm相应类型的数值节点。

C++

```cpp
llvm::Value *IntegerNode::CodeGen(CodeGenContext &context) {
    return llvm::ConstantInt::get(context.builder.getInt32Ty(), this->value, true);
}
llvm::Value *CharNode::CodeGen(CodeGenContext &context) {
    return llvm::ConstantInt::get(context.builder.getInt8Ty(), this->value, true);
}
llvm::Value *BooleanNode::CodeGen(CodeGenContext &context) {
    return llvm::ConstantInt::get(context.builder.getInt1Ty(), this->value, true);
}
llvm::Value *RealNode::CodeGen(CodeGenContext &context) {
    return llvm::ConstantFP::get(context.builder.getDoubleTy(), this->value);
}
```

- CallProceStatNode：首先根据function/procedure的标识符获取调用的llvm::Function对象的指针，若指针为空，则输出“未声明的函数或过程”的错误信息，并终止编译；否则，依次调用各实参表达式的CodeGen方法，最后通过CreateCall将实参传递给调用的llvm::Function对象。

C++

```cpp
llvm::Value *CallProceStatNode::CodeGen(CodeGenContext &context) {
    llvm::Function *callee = context.module->getFunction(this->id->idName);
    if (callee == nullptr) {
        throw std::domain_error("Undeclared function or procedure: " + this->id->idName + "!");
    }
    std::vector<llvm::Value *> args;
    if (callee->arg_size()) {
        for (ExpressionNode *expr : this->exprList->expressions) {
            args.push_back(expr->CodeGen(context));
        }
    }
    return context.builder.CreateCall(callee, args);
}
```

- ArrayType：ArrayType分为两个部分，一个部分是数组类型定义节点。数组定义与其它类型稍有不同，所以我们虽然按照表达式构建了AST节点，但是对于periodslist和period节点我们则不需生成，因此我们直接访问了节点中的属性来构建ArrayType，代码如下：

PHP

```php
 if (this->periods->periodList.size() > 1) {
        std::cout << "Warning: Array doesn't support multi-dimension!" << std::endl;
    }

    llvm::Value *num = this->periods->periodList[0]->CodeGen(context);
    llvm::ConstantInt *CI = llvm::dyn_cast<llvm::ConstantInt>(num);
    int arrayLength = CI->getSExtValue();

    llvm::Type *arrayType = this->typeNode->convertToLLVMType(context);

    return llvm::ArrayType::get(arrayType, arrayLength);
```

还有一部分是放在VariableNode节点进行处理的。若确定variable的类型为arraytype之后，则应该生成GEP指令，之后，再确定其为左值还是右值，对于左值，返回该指针，否则则返回指针所load的值，代码如下：

PHP

```
    if (!vartype->getContainedType(0)->isArrayTy()) {
        return value;
    }

    if (!this->idVarParts->idVarParts[0]->varPartType == IdVarPartNode::EXPRESSION_LIST_PART) {
        throw std::logic_error("The variable varparts is illegal");
    }

    ExprListNode *exprList = this->idVarParts->idVarParts[0]->expressionList;
    ExpressionNode *expr = exprList->expressions[0];
    llvm::Value *exprValue = expr->CodeGen(context);
    llvm::Type *exprType = exprValue->getType();
    if (exprType != context.builder.getInt32Ty()) {
        throw std::logic_error("The array index type is illegal");
    }

    llvm::ArrayType *arrayType = (llvm::ArrayType *)vartype->getContainedType(0);

    llvm::Value *Idxs[] = {llvm::ConstantInt::get(context.builder.getInt32Ty(), 0, true), exprValue};

    llvm::Value *gep = context.builder.CreateGEP(value, Idxs);

    if (this->isLeft) {
        return gep;
    }
    return context.builder.CreateLoad(gep, "load");
```

4 中间代码优化

在中间代码的优化中，我们调用了llvm中的opt工具，来对产生的LLVM-IR执行多次的Pass。针对于Pascal-S的特点，我们选择了如下几个Pass来进行优化：

- -dce：去除无用指令：该Pass用于去除在程序中无法到达的指令以及在去除上述指令后无法到达的指令。
- -deadargelim：去除无用变量：该Pass可以去除程序中的无用变量，他们中的有些传递到函数中，但不会被使用，可以将这些变量进行去除。
- -reassociate： 优化计算式：该Pass会对计算式的计算顺序进行调整，使得常数传播，常数合并等pass得以执行。例如：4 + (x + 5) ⇒ x + (4 + 5)。