

北京邮电大学

Beijing University of Posts and Telecommunications



Pascal-S: A Compiler Based on LLVM

编译原理课程设计汇报

第三组

汇报人：张成敏



北京邮电大学

Beijing University of Posts and Telecommunications



Outlines

一 技术路线

二 工程架构设计

三 遇到的问题及解决

四 亮点和创新点

五 反思和总结



一 技术路线

本组的技术路线中，词法分析与语法分析采用FLEX和BISON工具进行完成，语义分析过程主要体现在遍历AST抽象语法树各节点将节点信息转换为LLVM上下文信息的过程之中，属于人工编写方式，之后用LLVM上下文结构生成中间代码LLVM IR，并利用opt、clang等工具链对中间代码进行优化和转化为目标平台汇编代码与可执行二进制文件等操作。

下图是本组采用的技术路线一览：

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
FLEX	BISON	C++	LLVM-IR	LLVM-OPT	LLVM



二 工程架构设计

1 技术路线

根据本组技术路线的选择，我们将工程主要分为了如下部分：

- 1 主控程序：对整个程序流程进行控制和衔接
- 2 词法分析模块：将输入的源程序流转化为关键词流（token流）
- 3 语法分析模块：匹配token流与语法产生式规则，并执行翻译动作构建AST
- 4 语义分析模块：遍历AST中节点进行语义检查并将信息转化为LLVM Context信息存储
- 5 LLVM上下文模块：定义 `llvm::Module` 等信息，负责存储语义分析后的产物
- 6 输入输出控制：提供整个程序统一的输入接口以及格式化输出



二 工程架构设计

2 调用流程

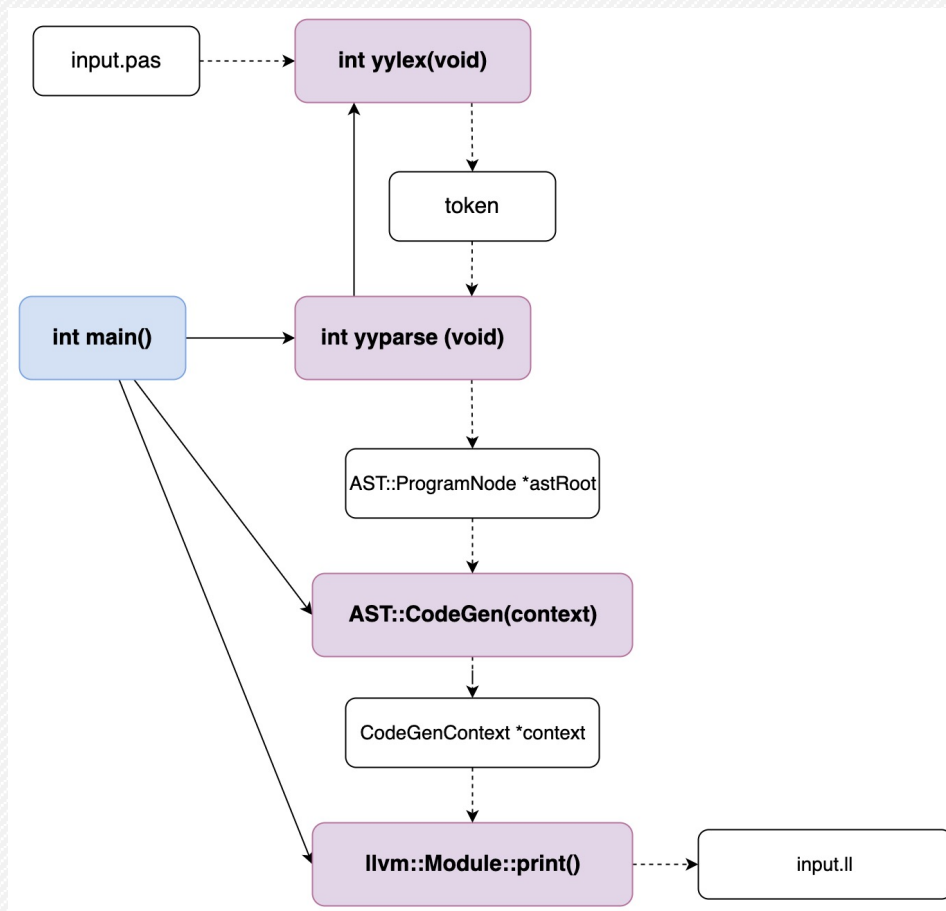
从调用流程上来看，主要的调用逻辑如下：

(1) main() 函数对用户输入命令进行判别，打开用户输入文件流

(2) main()调用语法分析程序运行，语法分析调用词法分析返回的token作为自身输入进行一遍过的源程序扫描，并执行制导翻译动作，构建语法树

(3) 调用语法树根节点CodeGen() 函数，该过程会遍历整个语法树，并将节点中信息转化为CodeGenContext存储

(4) main()函数调用CodeGenContext中的Illvm::Module进行中间代码生成等后续操作

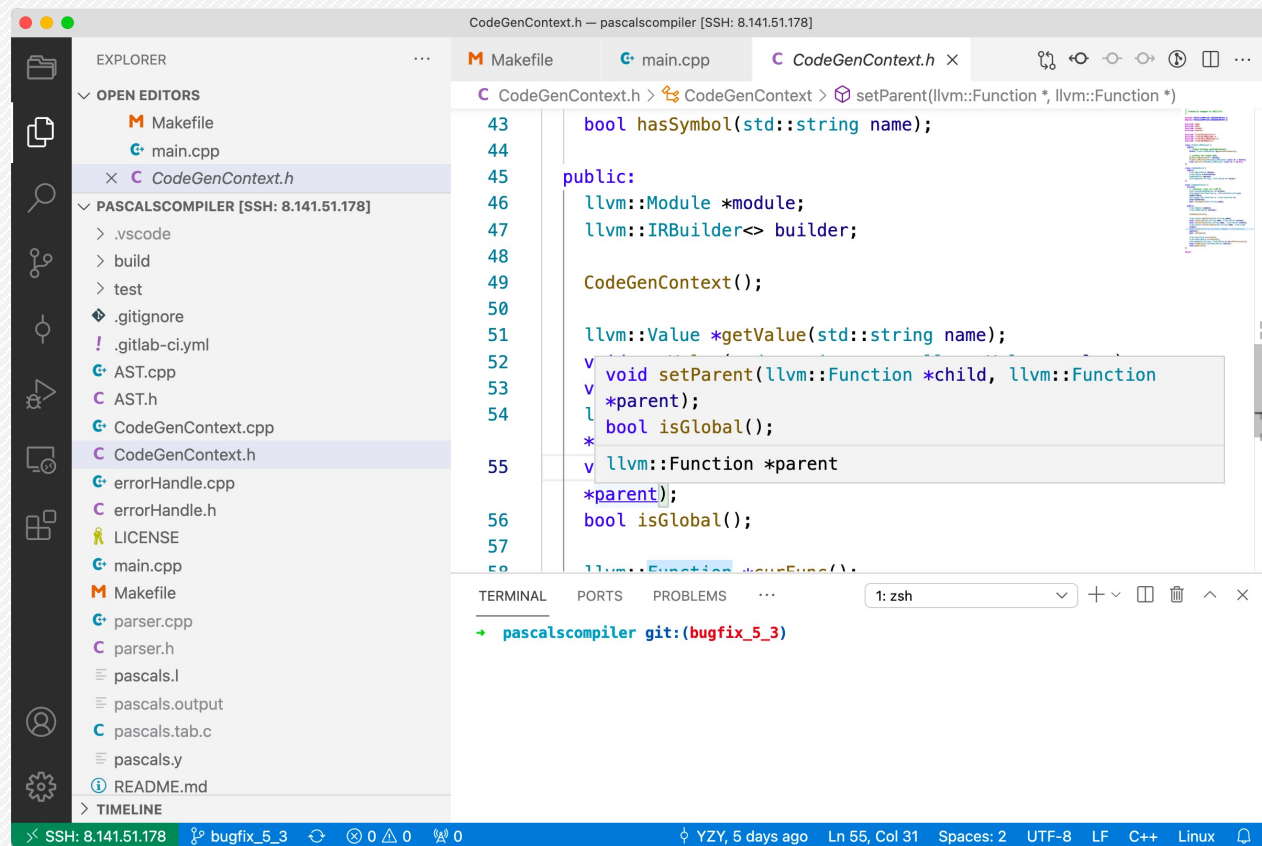




三 遇到的问题及解决

1 开发环境配置

由于本程序运行在工程建立在linux开发环境之上，且在技术路线的多个环节使用工具和库，对开发环境的要求较高，为了解决组内成员开发环境不统一等问题，我们租赁云主机作为开发平台，并对云主机进行了统一的环境配置，组员可以利用各自PC的VSCode编辑器远程连接主机，进行编写，其实现了代码高亮，代码一键跳转等功能。

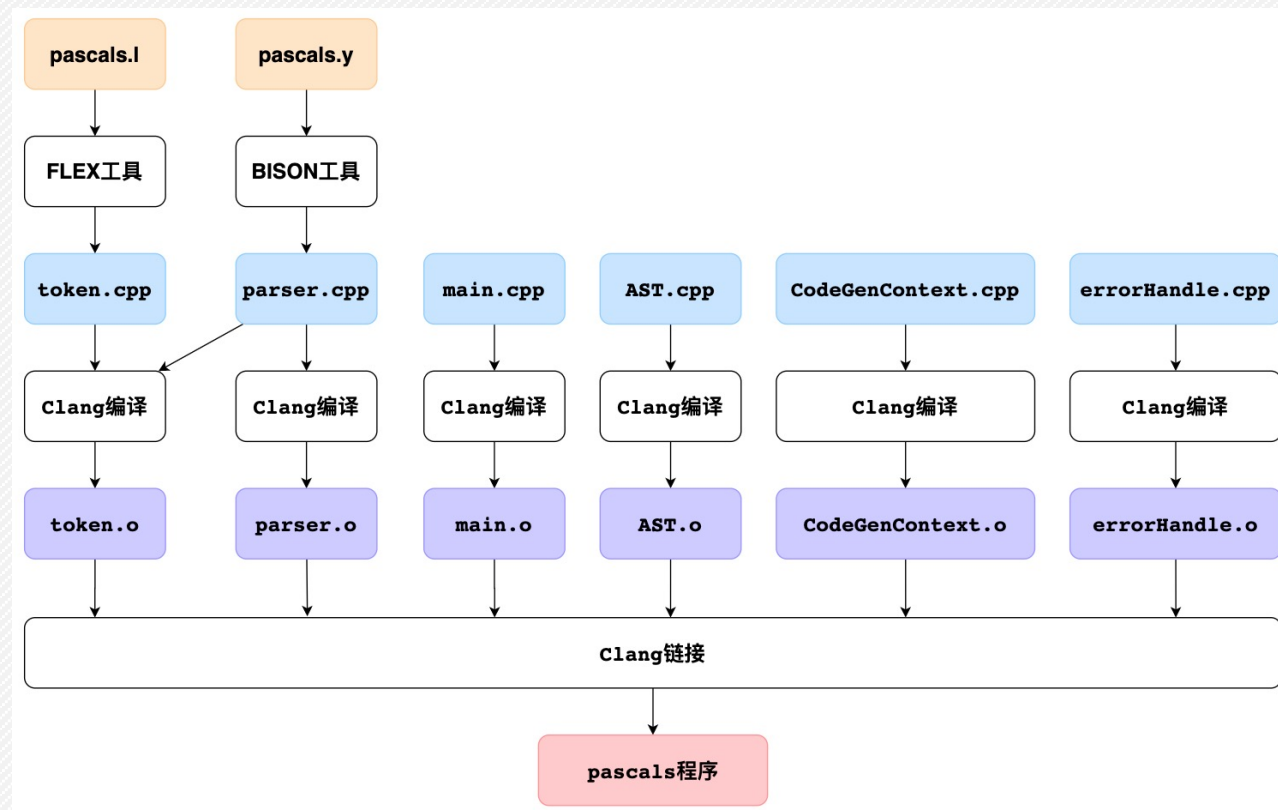




三 遇到的问题及解决 2 编译流程

本工程的编译流程较为复杂，既需要先调用flex和bison工具对编写的词法语法分析文件转换为C++代码，还需要对一系列源文件进行编译、链接等操作，且需要g++的llvm编译选项控制。

针对这些问题，我们编写了Makefile文件进行编译流程上的工具，只需make命令便可对工程进行编译和检查。右图是工程文件的编译流程图示。



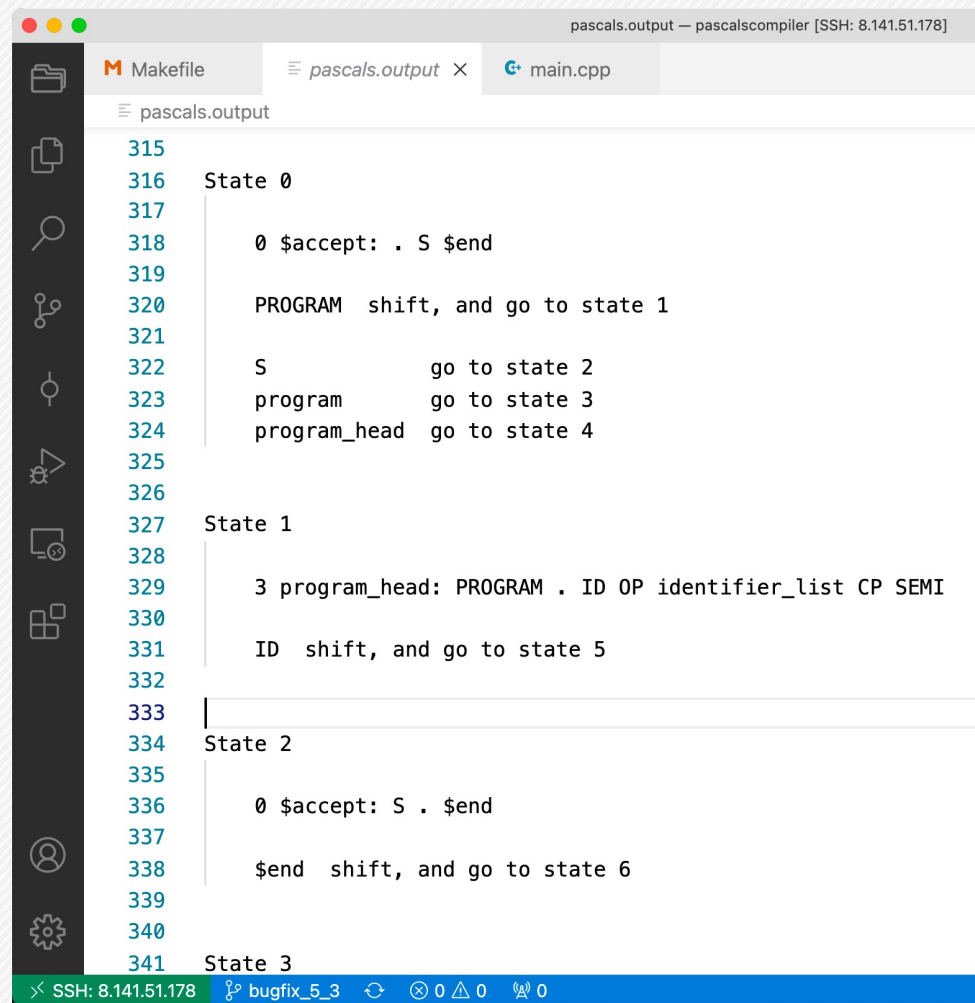


三 遇到的问题及解决

3 移进规约冲突

问题：写出的规则存在移进规约冲突，规约规约冲突。
其实真正在debug的时候解决冲突并不困难，但寻找冲突是很艰难的。

解决：一开始尝试人脑解决，成功解决了1个移进规约冲突，后续因始终无法解决部分冲突，查阅官方文档，编译bison时使用-v可以产生一个.output文件，会显示语法规则中所有可能的状态以及向前看一步的状态。如图，会显示语法规则中的未使用非终结符，终结符，以及移进规约冲突个数以及对应的State和每个State对应可能产生的动作。



```
pascals.output — pascalscompiler [SSH: 8.141.51.178]
Makefile  pascals.output x  main.cpp
pascals.output
315
316 State 0
317
318 0 $accept: . S $end
319
320 PROGRAM shift, and go to state 1
321
322 S go to state 2
323 program go to state 3
324 program_head go to state 4
325
326
327 State 1
328
329 3 program_head: PROGRAM . ID OP identifier_list CP SEMI
330
331 ID shift, and go to state 5
332
333
334 State 2
335
336 0 $accept: S . $end
337
338 $end shift, and go to state 6
339
340
341 State 3
SSH: 8.141.51.178  bugfix_5_3 0 0 0 0
```




三 遇到的问题及解决

4 语法分析左递归问题

问题：产生式含左递归时的AST构建，在Pascal-S的语法产生式中，有很多地方都含有左递归，如下图：这会对AST的构建造成一些困难，因为构建操作只能在bison进行规约完成时进行，而const_declarations的节点需要先被创建，然后再创建ConstDeclNode插入到ConstDeclsNode的list中。

解决：经过对bison语法分析的过程进行研究，确定以上产生之中的第二条会首先进行规约，因此在第二条产生式的代码段中加入创建新的ConstDeclsNode的语句，并将指针赋给const_declarations，再创建ConstDeclNode插入到ConstDeclsNode的list中。随后的递归过程中，被规约的永远是第一条，故仅做创建ConstDeclNode和insert的操作。

PowerShell ▼

自动换行 ☒

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



三 遇到的问题及解决

5 中间代码生成段错误

问题：编译程序运行过程中出现段错误。

解决：单步调试发现，构造AST过程中对于产生式推空的const_declarations、type_declarations、var_declarations和subprogram_declarations，未构造AST节点，也未对AST中指向这些节点的指针进行初始化（赋值为nullptr），导致中间代码生成阶段对未指向有效地址空间的指针进行解引用。经过小组讨论，我们一致决定对于推空的上述节点，构造一个成员变量List为空的节点，即产生式推空不是不构造节点，而是构造一个“空”节点（成员vector类型变量size为0）。

C++ ▼

自动换行 ☒

```
1  class VarDeclsNode: public BaseNode {
2      public:
3          std::vector<VarDeclNode *> varDeclList;
4          VarDeclsNode();
5          void insertVarDecl(VarDeclNode *varDel);
6          virtual llvm::Value *CodeGen(CodeGenContext &context);
7          virtual std::string description();
8  };
```



三 遇到的问题及解决

6 中间代码生成function/procedure形参赋值

问题：Pascal-S程序的function/procedure中，无法对形参进行赋值

解决：最开始对于传参的实现是直接将LLVM函数对象的参数名修改为形参名。但单步调试发现，LLVM的参数传递机制导致函数对象的参数的值无法被修改。为支持对形参的赋值，最终将实现方式改为不修改LLVM函数对象的参数名，而是在function/procedure开头分配名称为形参名的局部变量，并在函数调用时将实参的值赋给该局部变量。



三 遇到的问题及解决

7 中间代码生成对变量节点的处理

问题：Variable节点 Array类型左右值的区别

解决：在实现variableNode的中间代码的时候，arraytype类型与其它类型的变量稍有不同。由于AST设计的原因，当arraytype出现在左边时，返回值是当前值的指针；出现在右边时，返回值是当前值，此时需要将创建的指针进行load操作。因此，我们在构造这个节点时为其添加了一个属性isLeft，其可以用来判别该变量处于产生式的左部还是右部这样在生成代码时就可以通过该逻辑来生成不同的代码。

C++ ▼

自动换行 ☒

```
1  llvm::ArrayType *arrayType = (llvm::ArrayType *)vartype->getContainedType(0);
2  llvm::Value *Idxs[] = {llvm::ConstantInt::get(context.builder.getInt32Ty(), 0,
    true), exprValue};
3  llvm::Value *gep = context.builder.CreateGEP(value, Idxs);
4  if (this->isLeft)
5      return gep;
6  return context.builder.CreateLoad(gep, "load");
```




























四 亮点与创新点

1 软件工程管理

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，进行自动编译和运行测试，不断向主分支进行开发、测试和集成流程。

自动化测试保证了代码从编译角度上的正确性，为后期人工对代码的语义错误debug消除了代码语法上的错误，节约了时间，使职责更为明确，无法经过测试的代码无法进行进一步合入操作。

	#295812755		 master → 47856d7b Merge branch 'lexer' i...		🕒 00:00:56 📅 1 week ago	
	#295811991 latest		 lexer → 405c0833 debug .y		🕒 00:00:58 📅 1 week ago	
	#295576158		 master → ff7633cc Merge branch 'AST_1' ...		🕒 00:01:11 📅 2 weeks ago	
	#295566952 latest		 AST_1 → e05c2b55 Merge branch 'AST_1' ...		🕒 00:03:08 📅 2 weeks ago	
	#295482044		 AST_1 → 57025ec3 finish while for repeat		🕒 00:00:59 📅 2 weeks ago	



四 亮点与创新点

2 架构设计

本工程通过对语法树的节点设计，有效去除了词法分析语法分析与语义分析中间代码生成间的代码耦合关系，使得构建语法树和中间代码生成成为了两个相对独立的部分。在开发调试过程中，可以先完成AST声明后，在对 词法分析/语法分析 和 中间代码生成分别进行开发。

并在各自通过测试后统一合入master分支进行联调，在前半部分测试的重点是AST的构建以及节点内信息是否正确，对后半部分的测试重点则是保证手工代码和产生的IR代码间对应关系正确。如下图说明：



四 亮点与创新点

2 架构设计

其中节点的构造函数供parser()语法分析程序翻译制导动作调用，而CodeGen(CodeGenContext &context)用于语义分析中间代码生成。

C++ ▼

自动换行 ☒

```
1  class ConstVarNode : public BaseNode {
2      public:
3          enum ConstSignType {
4              MINUS,
5              POSITIVE,
6              NONE
7          };
8          ConstSignType constSignType;
9          UnsignConstVarNode *unsignConstVar;
10         ConstVarNode(ConstSignType constSignType, UnsignConstVarNode *unsignConstV
ar);|
11         virtual llvm::Value *CodeGen(CodeGenContext &context);
12         virtual std::string description();
13     };
```



五 反思和总结

在团队管理上，工程在部分进度安排上存在问题，尤其是轻视了CodeGen()函数编写的复杂性。并且在测试环节过于重视AST构造过程，没有及时测试生成的中间代码LLVM IR的准确性，导致后期工程量过大，部分功能没能完成。这个问题，主要是组长对llvm技术的调研不够深入，分工不均造成的，在以后的工程实践应当注意这个问题。

北京邮电大学

Beijing University of Posts and Telecommunications



感谢您的收看！

第三组
汇报人：张成敏

编译原理课程设计汇报