浙江大学

本科实验报告

课程名称：	编译原理
姓　　名：	黄梓淇、官泽隆、尹幽潭
学　　院：	计算机科学与技术
系：	计算机科学与技术
专　　业：	计算机科学与技术
学　　号：	3180106025、3180103008、3180105171
指导教师：	冯雁

序：

0.1 实验环境

Windows 环境下的编译和运行
（1）Visual Studio 2017
（2）YACC、LEX 开发环境（Lex 和 Yacc 的集成开发包）
（3）LLVM 开发环境
（4）GraphViz 软件及 python 包

0.2 分工明细

姓名	分工
官泽隆	lex 词法分析、解析生成 AST、中间代码生成 提供 I/O 库和 JIT 环境
尹幽潭	yacc 语法分析、语法生成树生成、语义分析+错误检测
黄梓淇	词法语法设计修订、可视化语法树代码、批处理代码、合并语义分析与中间代码生成的代码、测试、报告撰写、PPT 撰写、展示

第一章、词法分析

1.1 正规表达式

1. 定义

letter	[_a-zA-Z](注意：包括下划线)
digit	[0-9]
Hex	[a-fA-F0-9]

2. Token

Token	识别
NUMBER （包含十进制表示的 number 以及十六进制的 number） 注意：本编译器仅支持 int 类型计算 因此 NUMBER 的值是 int 类型	[1-9]{digit}*
	0[0-7]*
	0[xX]{Hex}+
CONST	“const”

INT	“int”
VOID	“void”
IF	“if”
ELSE	“else”
WHILE	“while”
BREAK	“break”
CONTINUE	“continue”
RETURN	“return”
IDENT	{letter}({letter}|{digit})*
‘+’	“+”
‘-’	“-”
‘*’	“*”
‘/’	“/”
‘%’	“%”
LE_OP	“<=”
GE_OP	“>=”
‘<’	“<”
‘>’	“>”
EQ_OP	“==”
NE_OP	“!=”
‘!’	“!”
‘=’	“=”
AND_OP	“&&”
OR_OP	“||”
‘;’	“;”
‘,’	“,”
‘[’	“[“
‘]’	“]”
‘(’	“(“

‘)’	“)”
‘{’	“{“
‘}’	“}”

共 34 个 token

3. 辅助：

词法需处理注释并需跳过空格和错误单词：

“//”[^\n]*	just consume
“/*”	comment();
[\t\n\v\f]	just consume（注意：包括空格）
.	error();错误处理

注释处理代码：

```
void comment(void){
        char c, prev = 0;
        while ((c = input()) != 0){      /* (EOF maps to 0) */
                if (c == '/' && prev == '*')
                        return;
                prev = c;
        }
        error("unterminated comment");
}
```

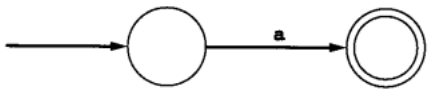
1.2 实现原理和方法

——实现原理参考教材总结而得

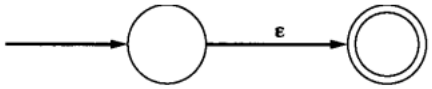
1.2.1 从正则表达式到 NFA

1. 基本正则表达式

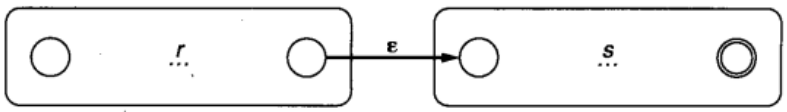
基本正则表达式格式 a、ε 或⊙，其中 a 表示字母表中单个字符的匹配，ε 表示空串的匹配，而⊙则表示根本不是串的匹配。与正则表达式 a 等同的 NFA（即在其语言中准确接受）的是：



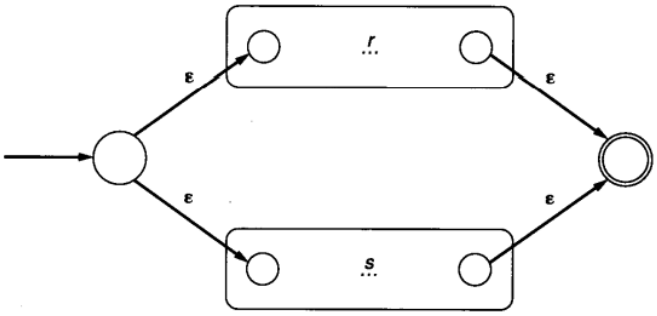
类似地，与 ε 等同的 NFA 是：



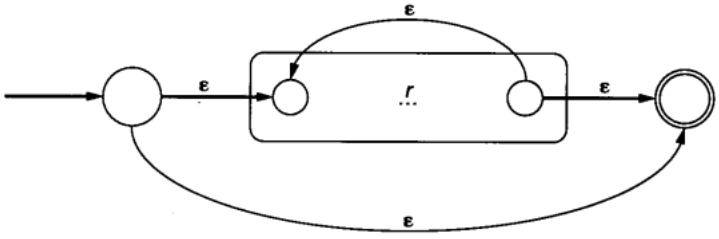
2. 并置



3. 在各选项中选择



4. 重复



1.2.2 从 NFA 到 DFA

1. 子集构造

首先计算 M 初始状态的大三 ε -闭包，它就变成~M 的初始状态。对于这个集合以及随后的每个集合，计算 a 字符之上的转换如下所示：假设有状态的 S 集和字母表中的字符 a，计算集合 S′a = {t | 对于 S 中的一些 s，在 a 上有从 s 到 t 的转换}。接着计算~S′a，它是 S′a 的闭包。这就定义了子集构造中的一个新状态和一个新的转换 $S \xrightarrow{a} \sim S'a$ ，继续这个过程直到不再产生新的状态或转换。当接受这些构造的状态时，按照包含了 M 的接受状态的方式作出记号。这就是 DFA 的~M。

2. 利用子集构造模拟 NFA

模拟 NFA 的一种方法是使用子集构造，但并非是构造与 DFA 相关的所有状态，而是在由下一个输入字符指出的每个点上只构造一个状态。因此，这样只构造了在给出的输入串上被取用的 D FA 的路径中真正发生的状态集合。

3. DFA 中的状态数最小化

以最乐观的假设开始：创建两个集合，其中之一包含了所有的接受状态，而另一个则由所有的非接受状态组成。假设这样来划分原始 D FA 的状态，还要考虑字母表中每个 a 上的转换。如果所有的接受状态在 a 上都有到接受状态的 转换，那么这样就定义了一个由新接受状态（所有旧接受状态的集合）到其自身的 a-转换。类似地，如果所有的接受状态在 a 上都有到非接受状态的转换，那么这也定义了由新接受状态到新的非接受状态（所有旧的非接受状态的集合）的 a-转换。另一方面，如果接受状态 s 和 t 在 a 上有转换且位于不同的集合，则这组状态不能定义任何 a-转换，此时就称作 a 区分了状态 s 和 t。在这种情况下必须根据考虑中状态集合（即所有接受状态的集合）的 a-转换的位置而将它们分隔开。当然状态的每个其他集合都有类似的语句，而且一旦要考虑字母表中的所有字符时，就必须移到它们的位置之上。当然如果还要分隔别的集合，就得返回到

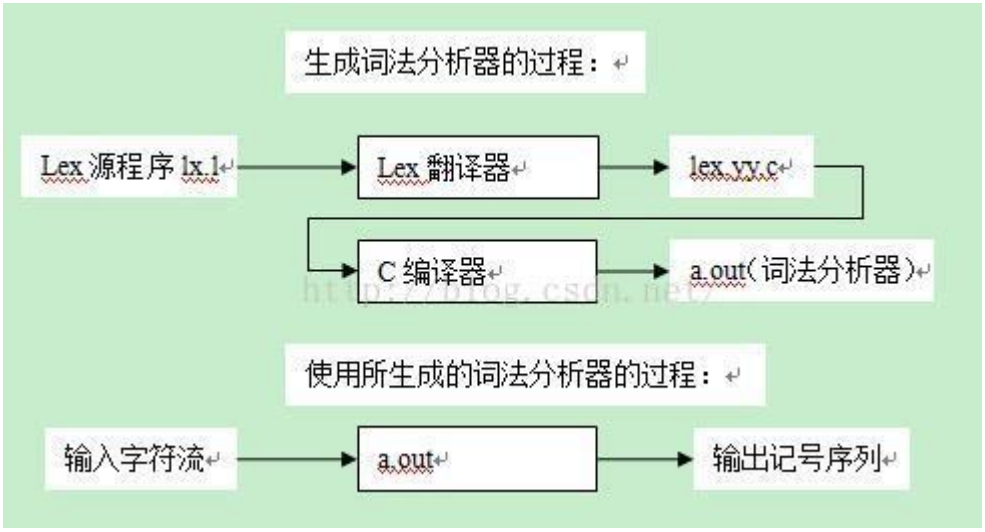
开头并重复 这一过程。我们继续将原始 D FA 的各部分状态集中到集合里，并一直持续到所有集合只有一个元素（在这种情况下，就显示原始 DFA 为最小）或一直是到再没有集合可以分隔了。

1.2.3 基于 Lex 的词法分析

在本实验中，我们采用工具 Lex 来辅助进行词法分析，相关代码在 cc.l 文件内存放。

1. Lex 基本原理

Lex 的基本工作原理为：由正规式生成 NFA，将 NFA 变换成 DFA，DFA 经化简后，模拟生成词法分析器。也即上述我们描述的词法分析原理。具体流程见下图：



2. lex 源程序的格式：

```
{definitions}
%%
{rules}
%%
{auxiliary routines}
```

第一部分是定义段，包含 2 个部分：以 C 语法写的一些定义和声明和一组正规定义和状态定义：

```
%{
C语法写的一些定义和声明
%}
一组正规定义和状态定义
```

第二部分是词法规则段，词法规则段列出的是词法分析器需要匹配的正规式，以及匹配该正规式后需要进行的相关动作，例如：

```
"/*"            { count(); formerComment(); }
"*/"            { count(); latterComment(); }
```

第三部分是辅助函数段，辅助函数段用 C 语言语法来写，辅助函数一般是在词法规则段中用到的函数。这一部分一般会被直接拷贝到 lex.yy.c 中。

3. Lex 中的元字符约定

格　式	含　义
a	字符*a*
"a"	即使*a* 是一个元字符，它仍是字符*a*
\a	当*a* 是一个元字符时，为字符*a*
a*	*a* 的零次或多次重复
a+	*a* 的一次或多次重复
a?	一个可选的*a*
a|b	*a* 或*b*
(a)	*a* 本身
[abc]	字符*a*、*b* 或*c* 中的任一个
[a-d]	字符*a*、*b*、*c* 或*d* 中的任一个
[^ab]	除了*a* 或*b* 外的任一个字符
.	除了新行之外的任一个字符
{xxx}	名字*xxx* 表示的正则表达式

4. 我们的 Lex 代码：
由于代码很长，请自行打开 code 目录下的 cc.l

第二章、语法分析

——实现原理参考教材总结而得，文法参考编译原理大赛的 Sysy 语言修改得到

2.1 上下文无关文法

类型	文法
编译单元	CompUnit→ CompUnit Decl | Decl | CompUnit FuncDef | FuncDef
声明	Decl→ConstDecl | VarDecl
常量声明	ConstDecl→ **CONST** BType ConstDef ConstDef_list **';'** ConstDef_list → ConstDef_ list **','** ConstDef | ε
基本类型	BType→**INT**
常数定义	ConstDef→**IDENT** **'='** ConstInitVal | **IDENT** **'['** ConstExp **']'** **'='** ConstInitVal

常量初值	ConstInitVal→ConstExp |'{' '}' |'{' ConstExp ConstExp_ list '}' ConstExp_ list → ConstExp_ list ',' ConstExp | ε
变量声明	VarDecl→BType VarDef VarDef_ list ';' VarDef_ list → VarDef_ list ',' VarDef | ε
变量定义	VarDef→ **IDENT** | **IDENT** '[' ConstExp ']' | **IDENT** '=' InitVal | **IDENT** '[' ConstExp ']' '=' InitVal
变量初值	InitVal→Exp | '{' '}' | '{'Exp Exp_ list '}' Exp_ list → Exp_ list ',' Exp | ε
函数定义	FuncDef→BType **IDENT** '(' ')' Block | BType **IDENT** '(' FuncFParams ')' Block
函数形参表	FuncFParams→ FuncFParam FuncFParam_ list FuncFParam_ list → FuncFParam_ list ',' FuncFParam | ε
函数形参	FuncFParam→BType **IDENT** | BType **IDENT** '[' ']'
语句块	Block→'{' BlockItem_list '}' BlockItem_list → BlockItem_list BlockItem | ε
语句块项	BlockItem→Decl | Stmt
语句	Stmt→LVal '=' Exp ';' | Exp ';' | ';' | Block | **IF** '(' Cond ')' Stmt | **IF** '(' Cond ')' Stmt **ELSE** Stmt | **WHILE** '(' Cond ')' Stmt | **BREAK** ';' | **CONTINUE** ';' | **RETURN** Exp ';'

表达式	EXP→AddExp
条件表达式	Cond→LOrExp
左值表达式	LVal→ **IDENT** | **IDENT** **'['** Exp **']'**
基本表达式	PrimaryExp→**'('** Exp **')'** | LVal| **NUMBER**
一元表达式	UnaryExp→PrimaryExp | **IDENT** **'('** **')'** | **IDENT** **'('** FuncRParams **')'** | UnaryOp UnaryExp
单目运算符	UnaryOp→'+'|'-'|'!' (！仅在条件表达式中出现，需要在 yacc 中解决)
函数实参表	FuncRParams→Exp Exp_list Exp_list → Exp_list **','** Exp | ε
加减表达式	AddExp→ MulExp | AddExp **'+'** MulExp | AddExp **'-'** MulExp
乘除模表达式	MulExp→UnaryExp | MulExp **'*'** UnaryExp | MulExp **'/'** UnaryExp | MulExp **'%'** UnaryExp
关系表达式	RelExp→AddExp | RelExp **'<'** AddExp | RelExp **'>'** AddExp | RelExp **LE_OP** AddExp | RelExp **GE_OP** AddExp
相等性表达式	EqExp→RelExp | EqExp **EQ_OP** RelExp | EqExp **NE_OP** RelExp
逻辑与表达式	LAndExp→EqExp | LAndExp **AND_OP** EqExp
逻辑或表达式	LOrExp→LAndExp | LOrExp **OR_OP** LAndExp

表达式	ConstExp→AddExp (使用的 Ident 必须是常量，需要在 yacc 中解决)

2.2 实现原理和方法

1.2.1 LALR(1)分析

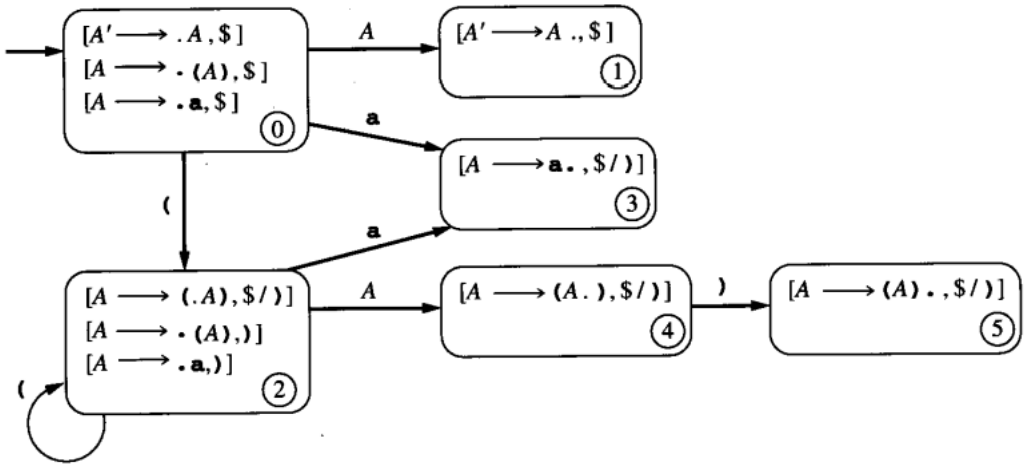
1. LALR(1)分析的第 1 个原则

LR(1)项目的 DFA 的状态核心是 LR(0)项目的 DFA 的一个状态。

2. LALR(1)分析的第 2 个原则

若有具有相同核心的 LR(1)项目的 DFA 的两个状态 s1 和 s2，假设在符号 X 上有一个从 s1 到状态 t1 的转换，那么在 X 上就还有一个从状态 s2 到一个状态 t2 的转换，且状态 t1 和 t2 具有相同的核心。

3. 这两个原则允许我们构造 LALR(1)项目的 DFA(DFA of LALR(1) items)，它是通过识别具有相同核心的所有状态以及为每个 LR(0)项目构造出先行符号的并，而从 LR(1)项目的 DFA 构造出来。因此，这个 D FA 的每个 LALR(1)项目都将一个 LR(0)项目作为它的第 1 个成分，并将一个先行记号的集合作为它的第 2 个成分。示例：（教材上例 5.17 的 LALR(1)项目集合的 DFA）



1.2.2 基于 YACC 的语法分析

1. YACC 原理

YACC 包含三个部分：

- 总控程序，也可以称为驱动程序。

  对所有的 LR 分析器总控程序都是相同的。

- 分析表或分析函数。

  不同的文法分析表将不同，同一个文法采用的 LR 分析器不同时，分析表也不同，分析表又可分为动作(ACTION)表和状态转换(GOTO)表两个部分，它们都可用二维数组表示。

- 分析栈，包括文法符号栈和相应的状态栈。

  它们均是先进后出栈。 分析器的动作由栈顶状态和当前输入符号所决定(LR(0)分析器不需要向前查看输入符号)。

YACC 的分析器工作过程如下:

其中 SP 为栈指针，S[i]为状态栈，X[i]为文法符号栈。状态转换表内容按关系 GOTO[Si，X] = Sj 确定，该关系式是指当栈顶状态为 Si 遇到当前文法符号为 X 时应转向状态 Sj。X 为终结符或非终结符。ACTION[Si，a]规定了栈顶状态为 Si 是遇到输入符号 a 应执行的动作。

2. YACC 源代码格式

```
{definitions}
%%
{rules}
%%
{auxiliary routines}
```

YACC 的总体格式于 Lex 大同小异。

第一部分是定义段，包含 2 个部分：以 C 语法写的一些定义和声明和 Yacc 需要用来建立分析程序的有关记号、数据类型以及文法规则的信息：

```
%{
C语法写的一些定义和声明
%}
Yacc需要用来建立分析程序的有关记号、数据类型以及文法规则的信息
```

第二部分是规则段，语法规则段列出的是语法分析器需要的文法规则，以及进行改文法规约后需要进行的相关动作，例如：

```
// 声明
Decl:
    ConstDecl{$$ = createGrammarTree("Decl", 1, $1);}
    | VarDecl{$$ = createGrammarTree("Decl", 1, $1);}
    ;
```

第三部分是辅助函数段，辅助函数段用 C 语言语法来写，辅助函数一般是在语法规则段中用到的函数或 yyerror 这类自动调用的函数。这一部分一般会被直接拷贝到 y.tab.c 中。

3. Yacc 关于 if-else 的解决

Yacc 分析器遇到 shift/reduce 冲突后，会自动优先 shift 而不是 reduce，因此这样就可以很轻松的解决 if-else 的 else 悬挂问题。

4. 语法分析树数据结构

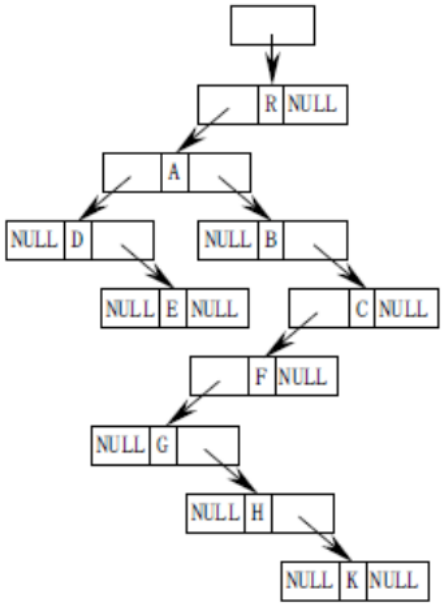
- 数据结构定义：

```
struct grammarTree
{
    string name;
    string content;
    int lineno;
    int id;
    struct grammarTree *left; // 1st child
    struct grammarTree *right; // next sibling
    // prep for tailor; classified on Nb_opr
```

```
    using Type_t = enum { BinExpr, List, Garbage, NA };
    Type_t type() const {   //返回节点的类型，如BinExpr}
    bool orphan() { return right == nullptr; }
    int nb_child() {//返回这个节点的孩子数目}
    grammarTree *fold_lchain() {//用于裁剪节点，删除本节点并将本节点左节点返回，
                                同时本节点的右兄弟接入左节点的右兄弟中}
    grammarTree *fold_rchain() {//用于裁剪节点，删除本节点并将本节点右节点返回}
    void tailor(); // to be called from root用于裁剪语法树节点
    grammarTree *tailor_inner();
    ~grammarTree() {// cascading deletion级联的删除节点}
};
```

- 数据结构分析：

(1) 语法树数据结构采用左孩子右兄弟的多叉树转二叉树的定义方式，类似下图所示：



(2) 参数说明：

- name 是语法树节点的名称，如 IDENT、NUMBER，与 Token 名称一致
- content 是语法树节点的内容，针对 IDENT、NUMBER，记录其具体值
- lineno 记录其出现行号，方便后续报错
- id 是赋予每一个语法树节点的唯一标识符
- left 指向左孩子
- right 指向右兄弟

(3) 内嵌函数：

- Type_t type()返回节点的类型，如 BinExpr
- bool orphan() 判断有没有右兄弟
- int nb_child()返回这个节点的孩子数目

- grammarTree *fold_lchain() 用于裁剪节点，删除本节点并将本节点左节点返回，同时本节点的右兄弟接入左节点的右兄弟中
- grammarTree *fold_rchain()用于裁剪节点，删除本节点并将本节点右节点返回
void tailor()用于裁剪语法树节点
grammarTree *tailor_inner();被 tailor 调用
~grammarTree()用于级联的释放空间(删除树)

5. 我们的 YACC 代码
由于代码很长，请自行打开 code 目录下的 cc.y

第三章、语义分析

3.1 实现的错误类型检测（包含词法语法语义分析阶段）

3.1.1 语言规则

我们对我们实现的编辑器/定义的语言做出以下假设：

文件假设：

假设 1：可以在命令行一次性读入多个文件编译，没有实质意义内容的空白文件将会直接跳过。

词法假设：

假设 2：仅支持八进制、十六进制和十进制 int 类型数据。

假设 3：不匹配错误格式的变量名。

文法假设：

假设 4：仅支持我们所定义的 token 和 rule。

语义假设：

定义假设

假设 5：只支持 int 数据类型

假设 6：const 型变量定义时必须初始化，且初始化形式只能是 const int a=NUMBER。

假设 7：const 数组定义时必须初始化，且初始化形式只能是 const int a[NUMBER] = {NUMBER, NUMBER, … , NUMBER},初始化大小与定义的大小一致。

假设 8：非 const 变量定义时可以不初始化，如果初始化，仅适用 const 的规则或适用非数组型变量。

假设 9：非 const 数组定义时可以不初始化，如果初始化，仅适用 const 的规则。

函数假设

假设 10：必须返回且只能返回 int 类型

假设 11：参数不能是 const

假设 12：没有声明且不能声明，首次出现即定义，不能在函数体中（嵌套）定义

假设 13：参数不能是其他函数

除了以上需要特别说明的以外，相同符号的使用或规则与 c 语言的使用或规则并无区别，可以视为 c 的子集。当然，很可能会出现其他能够在 c 里面出现的书写，但是在我们的编辑器里不支持的情形，也可能会出现我们未描述到的假设或规则，比如数组索引不能使用比较表达式。一般情况下，没有特别说明不可以，就是可以这样做，或者产生不那么精细的报错。

3.1.2 错误类型

依据上述假设，我们可以定义以下一些错误类型：

1. 未定义的数据类型：在词法中实现了对一些 c++中常用，但我们的语言中没有的数据类型的关键字的匹配，可以报错未定义的数据类型。
2. 错误的数据类型：在词法中实现了对一些 c++中常用，但我们的语言中没有的数据类型的匹配，可以报错不支持的数据类型；还实现了对错误的整型数字的匹配，报错错误的数据格式。
3. 注释错误：可以检测 c 语言风格的注释不匹配。
4. 未能匹配的符号：报错未知符号。
5. 错误定义的 const/var 变量：针对每种错误有细微不同的说明，具体定义错在哪里。
6. 符号重复定义/未定义：针对不同情形有略微不同的说明，指出具体什么符号。
7. 符号使用错误：int 变量作为函数或数组使用。
8. 表达式错误：对于我们的语言而言，表达式错误主要有三种：
   ①数组类型变量加入计算
   ②比较表达式两边类型不匹配
   ③赋值给数字或函数返回
9. 函数返回错误：没有返回或者返回数组。
10. 函数调用错误：参数个数不匹配或参数类型不匹配。
11. 其他未说明的错误：如果产生错误，则提示 syntax error。

一些错误将在运行的时候检测或是提出，他们不会引起语义错误。例如：作为数组索引的表达式其实非法访问数组等，我们只检查表达式的类型错误。当然，可能还会出现我们所预料之外的其他语义错误，导致语义分析的意外结束，进而导致我们编辑器运行的错误。

3.2 实现原理

3.2.1 数据结构

在语义分析中，我们实现一个符号表来记录程序中的符号。从最基础的开始，显然，变量应该有属于它自己的数据结构 Data_，数据结构定义：

```
struct Data_{
    myDataType data_type;
    bool is_r_value;
    union{
        myBasicType basic; // 基本类型
        struct{
            myData *elem;
            int size;
        } array; // 数组类型
```

```
    };
    int value;
};
```

is_r_value 记录变量是否是右值。数组通过 array->array.elem 形式的链表保存。

其中，myDataType 代表变量的结构类型，即，是数组还是单个变量：

```
typedef enum DataType_
{
    BASIC,
    ARRAY,
} myDataType;
```

myBasicType 代表变量的数据类型，我们只用一种数据类型 int：

```
typedef enum BasicType_
{
    INT,
} myBasicType;
```

除了变量，函数当然也应该有属于自己的数据结构 Func_：

```
struct Func_
{
    myData *ret_type;
    int param_num;
    myParam *param_list; // 函数参数列表
};
```

其中，根据我们设定的语言规则，函数返回类型只能是 int 且必须返回，用一个变量的数据结构记录函数的返回类型，myParam*定义函数的参数，param_num 定义函数的参数个数。myParam 的具体定义如下所示：

```
struct Param_
{
    myData *type;  // 参数的类型
    myParam *next; // 下一个参数
    string para_name;
};
```

使用 myData(即 Data_)定义每个参数,next 指针用于构建参数的名字,para_name 的目的是为了给错误信息输出提供方便。而且由于我们的语言没有声明，实参和形参的名字不能不同，所以定义一个名字是很有必要的。Data_没有单独定义名字主要出于以下的考虑：

①我们的符号表主要功能是创建符号和查表，不对符号数据进行维护，查表主要依赖于下文提出的更高层次的符号数据结构体的名。

②变量的名字可以很容易地从叶子节点得到，而且一般使用的地方能容易地访问到叶子节点。

③为数字建立 myData*变量的时候，也不太好决定用什么名字。

有了记录函数和变量的结构，我们就可以定义出我们的符号结构体：

```
typedef enum SymbolType_
```

```
{
    VAR,
    FUNC,
    CONST
} mySymbolType;
struct Symbol_
{
    string name;
    mySymbolType symbol_type;
    union
    {
        myData *type;
        myFunc *func;
    };
};
```

根据语言规则，我们将符号设置为 VAR，FUNC 和 CONST 三种类型。一个符号可能是函数也可能是变量。在符号的基础上，我们构建符号表的数据结构，使用哈希散列表，这里直接自己手搭一个散列表：申请一个大数组，计算一个散列函数的值，然后根据该值将对应的符号放到数组相应下标的位置即可。对于符号表来说，最简单的方法就是使用 hash 函数将符号名中的所有字符相加，然后对符号表的大小取模即可。这里我们使用了 github 上找到的一个散列函数：

```
unsigned int calHash(string name)
{
    unsigned int val = 0, tmp;
    for (int i = 0; i < name.length(); i++)
    {
        val = (val << 2) + name.at(i);
        if (tmp = val & ~(HASH_SIZE - 1))
            val = (val ^ (tmp >> 12)) & (HASH_SIZE - 1);
    }
    return val;
}
```

我们首先将单个符号连成链表：

```
struct SymbolList_
{
    mySymbol *symbol;
    mySymbolList *next;
};
```

然后使用哈希表，结构体 hashSet_就是我们的符号表。

```
#define HASH_SIZE 16384
typedef struct hashSet_ *myHashSet;
typedef struct Bucket_
{
```

```
    mySymbolList *symbol_list;
} myBucket;
struct hashSet_
{
    int size;
    myBucket *buckets;
};
```

如上就是我们使用的符号表的数据结构。在语义分析中，我们的想法是尽可能减少其他非必要复杂结构的使用。

3.2.2 分析过程

我们按照错误类型来展开说明语义分析具体是怎么实现的。

3.2.2.1 总体思路

我们采用递归下降的方式分析可能存在的语义错误。对于文法分析中建立的语法树，我们在语义分析中顺着每条规则进行分析，从根节点直到叶子结点。如果存在推导错误，给出报错

3.2.2.2 作用域

按照 c 语言的标准建立作用域，符号表中，每个符号的名字由它所在的作用域+它在程序中实际的名字构成。维护一个作用域栈，栈的成员就是符号，在文法的分析 block 的地方，新建一个作用域并入栈，block 分析完后，栈最顶上的作用域出栈。函数本身作为一个作用域，建立的作用域名字就是函数的名字，符号类型是 FUNC；while 语句的作用域将会涉及到 break 和 continue 和检查，以 VAR 为类型以示区分；其他作用域以 CONST 为类型。分析开始时加入一个 global 作用域。

3.2.2.2 定义时错误

错误定义：

在出现定义推导式时，置分析定义表达式为真，开始分析，在递归中碰到具体数据的时候开始检查，分析结束后置表达式标志为假。可以有很不同的检查标志，分别标志要检查数组定义的空间，初始化的格式等等。

重复定义：

对于函数，在符号表中寻找有无一样的名字，有就是重复定义了。（函数作为作用域，其符号名字就是程序中的名字，不会加作用域前缀）。对于变量，需要逆序遍历作用域栈，每次给当前要检查的名字加上作用域名字前缀，然后在符号表中寻找有无一样的名字，有就是重复定义了。通过插入符号表的函数，就可以检查符号表里有无重复。

没有定义：

对于函数，同样，在符号表中寻找有无一样的名字，没有就是没有定义。对于变量，首先在作用域下检查有无该变量，如果遍历到作用域是函数，还要检查函数参数中是否已经定义过该变量。

3.2.2.3 表达式与函数

在出现表达式推导时，置分析表达式标志为真，开始分析，在递归中碰到具体数据时开始检查，分析结束后置表达式标志为假。

对于函数返回，维护一个 myData 类型的变量，在函数定义推导式处，置检查函数返回标志，分析结束后（递归，函数结尾最后总会回到这里），检查有没有调用 checkFuncRet 函数即可知道函数有没有返回值，checkFuncRet 函数检查维护的 myData 和符号表里记录的定义能否匹配。

对于函数调用，维护一个成员为 myParam 的容器，在函数调用推导式处，置检查函数调用标志，递归结束后，用 checkFuncCall 函数检查维护的容器和符号表里的函数定义是否匹配。

3.2.2.4 控制语句：只要当前作用域栈中有对应 while 类型的，就没有错误；否则就有错误。

3.2.2.5 关键函数及其作用

checkArray 用于检查定义的数组有无错误

```
void checkArray(mySymbol *symb, int lineno)
```

checkRepeatFuncDef 检查函数有无重复定义

```
bool checkRepeatFuncDef(mySymbol *symb, int lineno)
```

checkRepeatVarDef 检查变量有无重复定义

```
void checkRepeatVarDef(mySymbol *symb, int lineno)
```

检查函数返回

```
void checkFuncRet(int lineno)
```

检查函数调用

```
void checkFuncCall(string func_name, int lineno)
```

检查没有定义

```
bool checkNotDef(string name, int lineno, string form)
```

在检查有无定义时，根据需要获取数据到维护的数据结构中。按照检查的数据是变量、数组、函数 3 种，和当前正在检查的是表达式、函数返回还是函数调用 3 种交叉，有 3*3=9 种情形，每种都要按照实际设计，并处理特殊情况。在叶子结点，即分析变量名字和数字的地方，也分别要按当前正在检查的是什么，分情况处理。由于检查的类型可能嵌套，比如函数调用或参与表达式计算等等，所以会产生很复杂的控制，需要十分谨慎小心。

第四章、优化考虑（每个阶段的优化考虑）

4.1 词法语法分析的优化考虑

1. **词法分析加入代码检测不支持的 C 关键字**

由于我们的语法是类 C 语言，因此很多 C 语言相关关键字会被识别为 IDENT 导致语法分析报错，二作为类 C 语言很可能会不小心输入不支持的 C 关键字，因此专门加入了代码对于不支持的 C 关键字进行报错。

2. **语法分析加入语法错误检测报错**

原本我们初版语法分析直接采用 YACC 自带的报错函数，因此只会报错 Syntax Error。这样的报错指向性不强，因此我们优化后能够将报错具体到行号，并且对于一些常见的错误类型能够较为精准的报出对应的错误。

3. **语法分析进行生成树的剪枝**

由于我们将优先级和结合性做进了文法当中，导致我们的语法分析树十分冗余不易理解，因此加入了裁剪生成树的代码，将一些关于递归、优先级和结合性的推到节点裁剪掉，获得一棵较为简洁易懂的语法生成树以供后续 AST 生成和 IR 生成使用。

4. **撰写了 Python 工程，借助工具 GraphViz 实现语法生成树的可视化**

在语法分析程序中我们输出指定格式的包含树结构的文件，然后使用 python 工程 viewTree 中 main.py 函数来实现生成树可视化，命令行执行格式为：

```
usage: main.py [-h] [-path PATH] [-path PATH] ... [-path PATH]
```

PATH 即为需可视化文件的相对路径，可同时进行多个文件的可视化。

5.2 语义分析的优化考虑

在使用 lex 和 yacc 完成了词法与文法分析以后，我们小组决定一名同学写中间代码，另一名同学写语义分析和 AST 生成。由于误会，写中间代码的同学误以为文法分析时建立的树就是 AST。其实该树是基于每条文法推导规则建立的，每条规则的左侧为树的父节点，右侧为子节点，“忠实”地记录了推导的详细过程。写中间代码的同学通过剪枝得到了可供后续开发的 AST，而写语义分析的同学也定义了自己的符号表。所幸误会发现得及时，为了降低开发的困难和提升开发速度，虽然会使程序风格的整体性和一致性遭到破坏，我们还是选择按照各自的思路继续深入。

在此背景下，语义分析中的符号表仅供 debug 使用，它只记录每个符号被定义时的初始状态，采取递归下降遍历语法树的方案，分析可能存在的语义错误。这样就不用在整个语义分析过程中精心维护符号表，只用创建该表，然后读取该表的条目即可，涉及到代码段中的符号使用和变化，结合 C++ 的 STL 模板，以语句为单位进行解析，创建临时的容器存储语句中的符号，分析完一条语句就刷新容器。这样除了在创建符号和遍历符号表的时候，可以避免使用链表，大大降低了开发的难度和出错的可能，同时也提高了语义分析的速度。而对于开发中间代码及后续过程的同学而言，他所使用的剪枝方法，极大地压缩了臃肿的语法树，也可以看作是语义分析的一种优化。

除了在技术方面选择容器而非链表，以尽可能减小符号表的规模之外，我们还根据我们语言的简单特性，对表达式的语义分析采用了简化的处理方案。由于我们做出了以下的假设：

①函数的返回类型只能是 int；

②函数的参数不能是 const；

③只支持 int 数据类型；

④不支持数组的指针表达部分；

⑤没有其他的数据类型或数据结构。

我们很容易得出以下结论：

*除了等于和不等于表达式，只要表达式里出现代表数组的变量名，就是一个错误的表达式。

而需要处理*中唯一的例外：合法地函数调用数组是表达式的一部分。

基于此结论检查表达式的语义错误的效率将会大大提高，无需再为表达式维护数据结构，只需要关注代码段中有没有出现数组变量名即可。当然，随之而来的缺点就是需要为等于和不等于表达式的类型检查设计单独的机制。

除了上述时间和空间方面的考虑，在语义分析检测错误环节，我们并没有设计额外的代码优化内容。一种可能的想法是在语义分析阶段就计算出一些表达式的值，这样的确可以更精细地分析错误，压缩符号表或 AST 的空间，提升效率，但是开发也更为困难。由于产生了前述的误会，我们是在比较“臃肿”的树上直接进行的语义分析，递归下降的时候有很多冗余的子节点，在语义分析中动态计算符号值还要求额外分析运算符的优先级和结合性，考虑到链表在函数之间的传递和其他多出来的内容，感觉得不偿失，还不如就直接剪枝后继续开发，不在语义阶段提供优化。

4.3 中间代码生成的优化考虑

借助 LLVM 库函数：

```
static std::unique_ptr<legacy::FunctionPassManager> TheFPM;
Function *TheFunction;
TheFPM->run(*TheFunction);
```

以函数为单位进行中间代码优化。

第五章、代码生成　（所有语句的代码生成的处理）

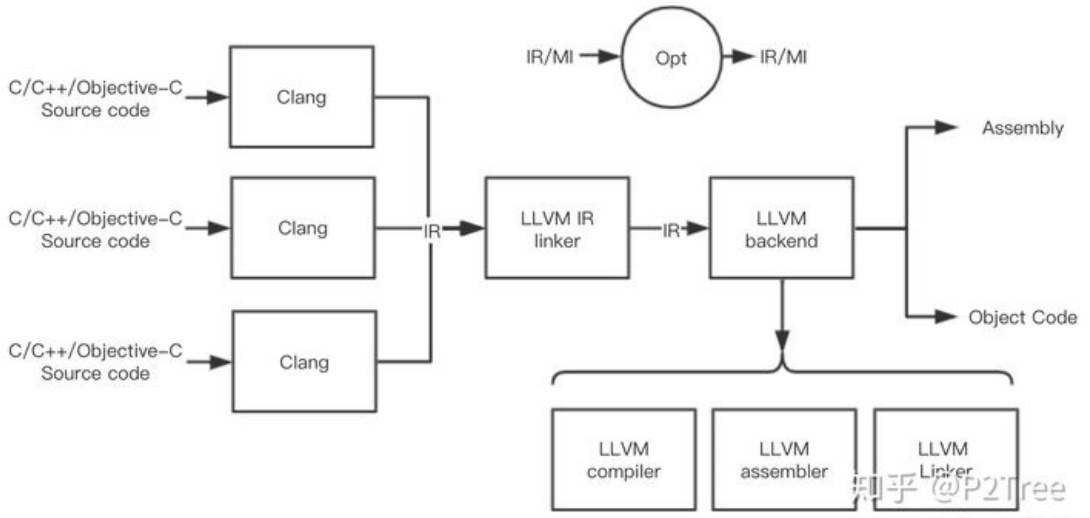
5.1 中间代码生成：生成 LLVM 的 IR

5.1.1 LLVM 简介

LLVM 最初设计时，目标是做优化方面的研究，所以只是想搭建一套虚拟机，当时的全称叫 Low Level Virtual machine，后来因为要变成编译器，官方就放弃了这个称呼，但 LLVM 的简称被保留下来。因为 LLVM 只是一个编译器框架，所以还需要一个前端来支撑整个系统，由此 Clang 被研发了出来。而我们的编译器后端采用了 LLVM，因此我们需要生成 LLVM IR 作为中间代码链接我们的词法语

法分析到 LLVM 后端。

LLVM 的简单架构如下图：



5.1.2 LLVM 中间代码格式说明(仅简单介绍我们使用的部分)

1. 注释

在 LLVM IR 中，注释以;开头，并一直延伸到行尾。

2. 在 LLVM IR 中，开头一定有：target datalayout，它注明了目标汇编代码的数据分布。这里我们是在代码中使用 LLVM 提供的函数自动获取的。

3. main 函数

define i32 @main(){}是定义主函数的方式，如最简单翻译：

```
int main() {
    return 0;
}
```

```
define i32 @main() {
    ret i32 0
}
```

4. LLVM IR 中数据表示及赋值

- 存储在数据区中的全局变量：

@global_variable = global i32 0

(定义了一个 i32 类型的全局变量@global_variable，并且将其初始化为 0)

- 只读的全局变量，即常量：

@global_constant = constant i32 0

(定义了一个 i32 类型的全局常量@global_ constant，并且将其初始化为 0)

- 虚拟寄存器

对于寄存器而言，我们只需要像普通的赋值语句一样操作，但需要注意名字必须以%开头。如：

%local_variable = add i32 1, 2

此时，%local_variable 这个变量就代表一个寄存器，它此时的值就是 1 和 2 相加的结果。

如果我们把所有没有被保留的寄存器都用光了，那么 LLVM IR 会帮我们把这些被保留的寄存器放在栈上，然后继续使用这些被保留寄存器。当函数退出时，会帮我们自动从栈上获取到相应的值放回寄存器内。由此简化了操作。

- 栈上的变量

在需要操作地址以及需要可变变量（之后会提到为什么）时，我们就需要使用栈。LLVM IR 对栈的使用十分简单，直接使用 alloca 指令即可。如：

%local_variable = alloca i32

表示声明一个在栈上的变量。

- 全局变量和栈上变量的获取与赋值操作

如果我们要将值存储到全局变量或栈上变量，需要 store 命令：

store i32 1, i32 @global_variable*

这个代表将 i32 类型的值 1 赋给 i32*类型的全局变量@global_variable 所指的内存区域中，栈上变量赋值也类似。

如果我们要获取@global_variable 的值，就需要

%1 = load i32, i32 @global_variable*

这个指令的意思是，把一个 i32*类型的指针@global_variable 的 i32 类型的值赋给虚拟寄存器%1

5. LLVM IR 中的类型

这里我们只用到了整型(整型是指 i1, i8, i16, i32, i64 这类的数据类型)也就是 int 类型。

6. LLVM IR 中的控制语句

- Label：标签与汇编语言的标签一致，也是以:结尾作标记
- 比较指令

LLVM IR 提供的比较指令为 icmp。其接受三个参数：比较方案以及两个比较参数，如：

%comparison_result = icmp uge i32 %a, %b

uge 是比较方案，%a 和%b 就是用来比较的两个数，而 icmp 则返回一个 i1 类型的值，也就是 C++中的 bool 值，用来表示结果是否为真。

uge 可以是：eq 与 ne，分别代表相等或不相等；无符号的比较 ugt, uge, ult, ule，分别代表大于、大于等于、小于、小于等于；有符号的比较 sgt, sge, slt, sle，功能类比无符号的比较符。

- 条件跳转：

LLVM IR 提供的条件跳转指令是 br，其接受三个参数，第一个参数是 i1 类型的值，用于作判断；第二和第三个参数分别是值为 true 和 false 时需要跳转到的标签。如：

br i1 %comparison_result, label %A, label %B

表示当%comparison_result 的值(上面的比较结果)，为 true 去 label A，为 false 去 label B

- 无条件跳转

在 LLVM IR 中，我们同样可以使用 br 进行条件跳转。如，如果要直接跳转到 start 标签处，则可以：

br label %start

- 接着借助 Label、比较、条件跳转和无条件跳转我们就可以构造出 while 和 if-else 语句出来。

7. LLVM IR 中的函数

- 函数定义

一个函数定义最基本的框架，就是返回值(i32)+函数名(@foo)+参数列表((i32 %a, i64 %b))+函数体({ ret i32 0 }),如：

```
define i32 @foo(i32 %a, i64 %b) {
    ret i32 0
}
```

- 函数声明

函数声明就是使用 declare 关键词替换函数定义的 define，如：

declare i32 @foo(i32 %a, i64 %b)

- 函数调用

在 LLVM IR 中，函数的调用与高级语言几乎没有什么区别。使用 call 指令可以像高级语言那样直接调用函数。如：

%1 = call i32 @foo(i32 1, i64 2)

5.1.3 借助 LLVM 生成中间代码

1. 撰写生成 AST 代码

这里我们选择自己撰写代码生成便于我们撰写 codeGen 的 AST，同时这减少了中间代码生成和语义分析的耦合度，中间代码将基于语法分析树提取 AST，而语义分析将直接使用语法生成树。

- proc_CompUnit：解析开始符号 CompUnit 对应节点，并调用解析其生成的节点，是执行 AST 生成的入口：

```
std::vector<
    std::variant<std::unique_ptr<FunctionAST>, std::unique_ptr<ExprAST>>>
proc_CompUnit(const grammarTree *r);
```

- get_FuncDef_AST：解析 FuncDef 对应节点，并调用解析其生成的节点：

```
std::unique_ptr<FunctionAST> get_FuncDef_AST(const grammarTree *r);
```

- get_Decl_AST：解析 Decl 对应节点，并调用解析其生成的节点：

```
std::unique_ptr<VarDefAST> get_Decl_AST(const grammarTree *r,
                                        bool isGlbl = false);
```

- get_Stmt_AST：解析 Stmt 对应节点，并调用解析其生成的节点：

```
std::unique_ptr<ExprAST> get_Stmt_AST(const grammarTree *r);
```

- get_Block_AST：解析 Block 对应节点，并调用解析其生成的节点：

```
std::unique_ptr<BlockAST> get_Block_AST(const grammarTree *r);
```

- get_Exp_AST：解析 Exp 对应节点，并调用解析其生成的节点：

```
std::unique_ptr<ExprAST> get_Exp_AST(const grammarTree *r);
```

- get_LVal_AST：解析 LVal 对应节点，并调用解析其生成的节点：

```
std::unique_ptr<VariableExprAST> get_LVal_AST(const grammarTree *r);
```

- get_BinExpr_AST：解析 BinExpr 对应节点，并调用解析其生成的节点：

```
std::unique_ptr<BinaryExprAST>
get_BinExpr_AST(const grammarTree *r,
                std::stack<std::unique_ptr<ExprAST>> &out);
```

2. 基于 AST 撰写每个 AST 结构对应的 codegen()

- BlockAST::codegen()：调用 Block 的孩子节点的 codegen()

```
Value *BlockAST::codegen()
```

- NumberExprAST::codegen()：调用 llvm Build 创建一个 int 数值

```
Value *NumberExprAST::codegen()
```

- VariableExprAST::codegen()：调用 llvm Build 创建对应变量

```
Value *VariableExprAST::codegen()
```

- UnaryExprAST::codegen()：层层向下解析 AST，并调用 llvm Build 创建单目运算代码

```
Value *UnaryExprAST::codegen()
```

- BinaryExprAST::codegen()：层层向下解析 AST，并调用 llvm Build 创建双目运算代码

```
Value *BinaryExprAST::codegen()
```

- VarAssignAST::codegen()：调用 llvm Build 创建赋值代码

```
Value *VarAssignAST::codegen()
```

- ReturnAST::codegen()：调用 llvm Build 创建 return 代码

```
Value *ReturnAST::codegen()
```

- CallExprAST::codegen()：调用 llvm Build 创建函数调用代码

```
Value *CallExprAST::codegen()
```

- IfAST::codegen()：调用 llvm Build 创建 if 控制语句代码

```
Value *BlockAST::codegen()
```

- WhileAST::codegen()：调用 llvm Build 创建 while 控制语句代码

```
Value *WhileAST::codegen()
```

- GotoAST::codegen()：调用 llvm Build 创建 goto 控制语句代码

```
Value *GotoAST::codegen()
```

- VarDefAST::codegen()：调用 llvm Build 创建变量声明代码

```
Value *VarDefAST::codegen()
```

- GlblVarDefAST::codegen()：调用 llvm Build 创建全局变量声明代码

```
Value *GlblVarDefAST::codegen()
```

LLVM IR 的生成整体上采取 **macro expansion** 的手法。中间代码作为 AST 的 综合属性。

Getting Started

我们使用 LLVM 提供的实用类 `IRBuilder`. 这使得我们不用手动管理插入中间代码的位置，同时指令选择更便利。IR 生成的接口如下

```
virtual Value *codegen() = 0;
```

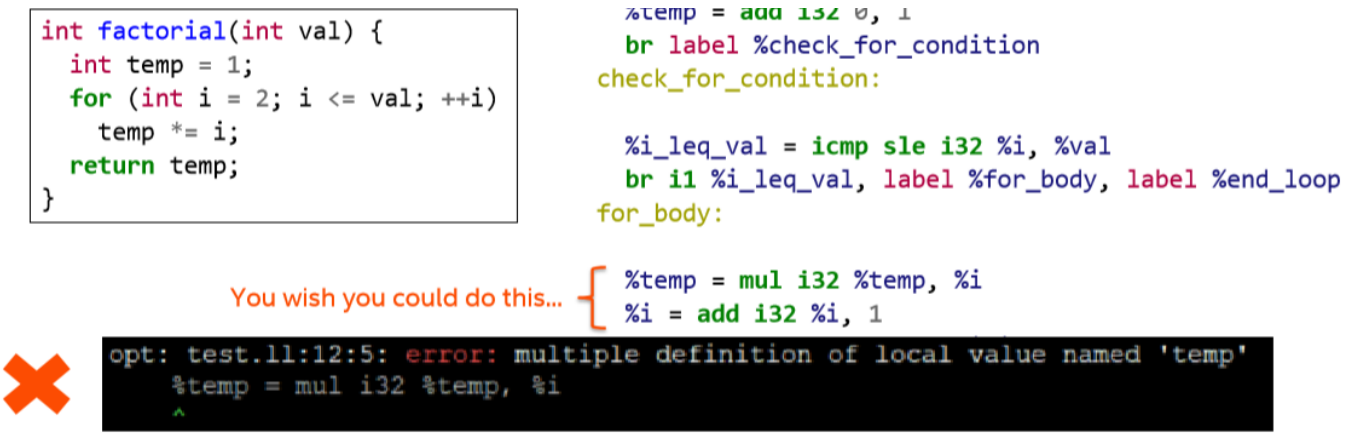
- 它将为该 AST 节点调用 `IRBuilder` 插入中间代码
- 如果语义上该节点有一个“值”，例如对常数、变量的引用，则返回这个值的表示；否则返回值为空指针
- `Value` 表示一个类型化值，可以用作（但不限于）指令操作数。有许多不同类型的 `Values` ，例如 `Constants` , `Arguments` . 甚至 `Instructions` 和 `Functions` 都是 `Values` . 这一良好的类型层次结编码提供了很好的抽象。 `Value` 对象的生命周期由 LLVM 管理（并跟踪 `User` ）， `Value` 类对外提供一个 `get` 静态方法返回 `Value *`

对于叶子节点，接口实现简单：

```
std::map<std::string, Value *> NamedValues; // a simple symtab
...
Value *NumberExprAST::codegen() {
  return Builder->getInt32(Val);
}

Value *VariableExprAST::codegen() {
  // Look this variable up in the function.
  Value *var = NamedValues[Name];
  // reference variable in memory
  return Builder->CreateLoad(Builder->getInt32Ty(), var, Name);
}
```

LLVM IR 要求 静态单赋值 , 即：只允许带初值的变量声明，高级语言中的变量赋值不能够直接翻译，如下图所示。



我们使用 Clang 的实践：先把所有变量都视作内存变量（参考 代码优化 一节）。这解释了为什么上面的变量引用会发出 load 指令。对应地，变量赋值翻译为向内存存值

```
Value *VarAssignAST::codegen() {
  auto val = RHS->codegen();
  auto var = NamedValues[LHS->Name];
  Builder->CreateStore(val, var);
  ...
```

函数返回直接对应一条 IR.

```
Value *ReturnAST::codegen() {
  if (Value *RetVal = RHS->codegen()) {
    // Finish off the function.
    Builder->CreateRet(RetVal);
  ...
```

算术运算与关系运算

语言的主要能力是通过单目和双目运算符实现的。以双目运算符为例，codegen 时，首先对两个 操作数 做 codegen，并可以信任两个方法会返回 操作数 的值表示。然后枚举运算符

```
Value *BinaryExprAST::codegen() {
  Value *L = LHS->codegen();
  Value *R = RHS->codegen();

  if (Op == "+") {
    return Builder->CreateAdd(L, R, "addtmp");
  ...
```

处理关系运算符时，要遵守 SysY 的语言标准，应保证返回类型 i32

```
    // relation op: i32 in {0, 1}
    auto it = srel2pred.find(Op);
    if (it != srel2pred.end()) {
      R = RHS->codegen();
      L = Builder->CreateICmp(it->second, L, R, "cmptmp");
      return Builder->CreateZExt(L, Builder->getInt32Ty(), "i1toi32_");
    }
```

{Block} 嵌套作用域

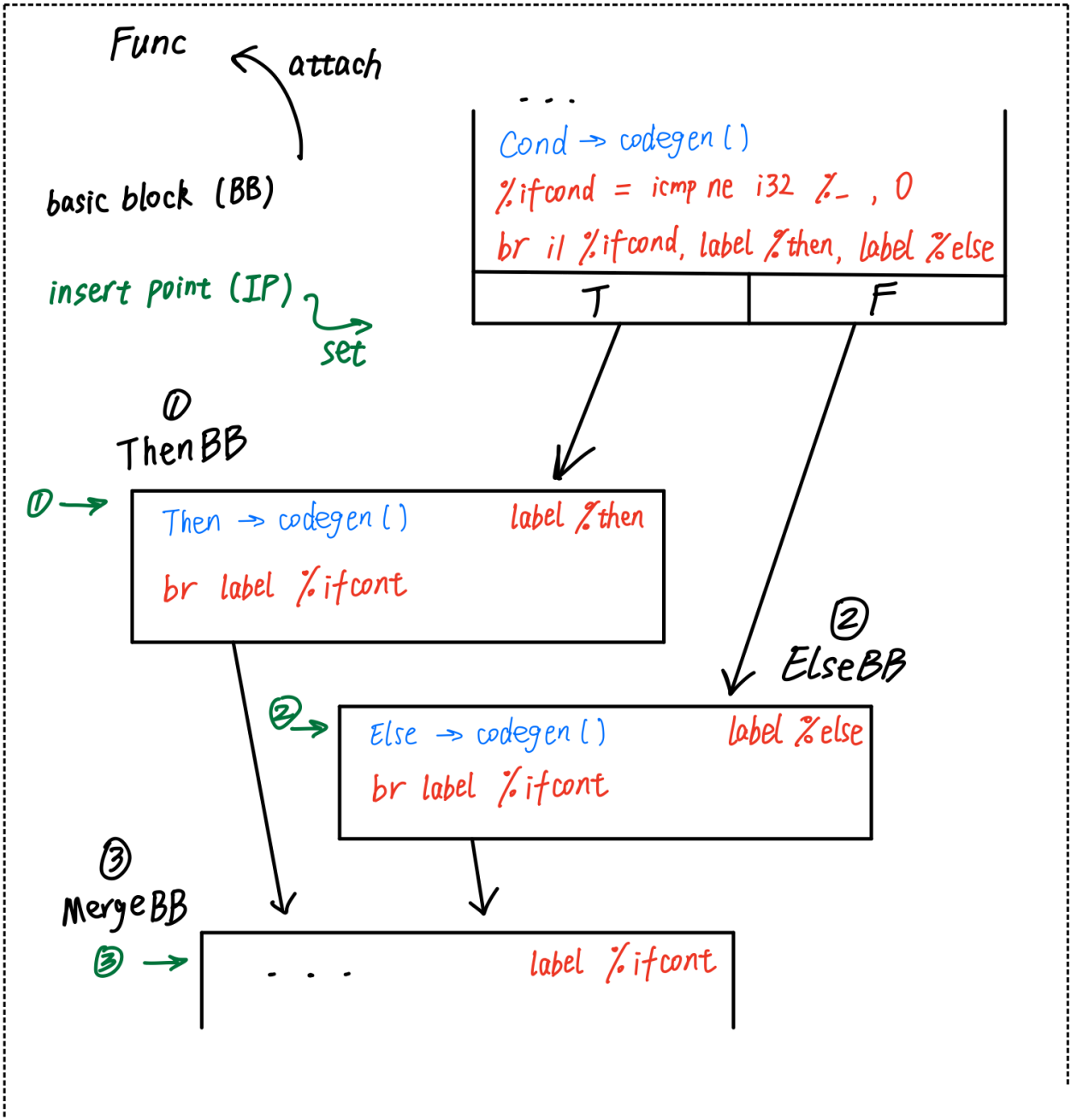
以 BlockAST 囊括这些信息。BlockAST::items 表示了一对大括号内的所有声明/语句，codegen 时顺序生成。这样一来，符号表可以逐层嵌套

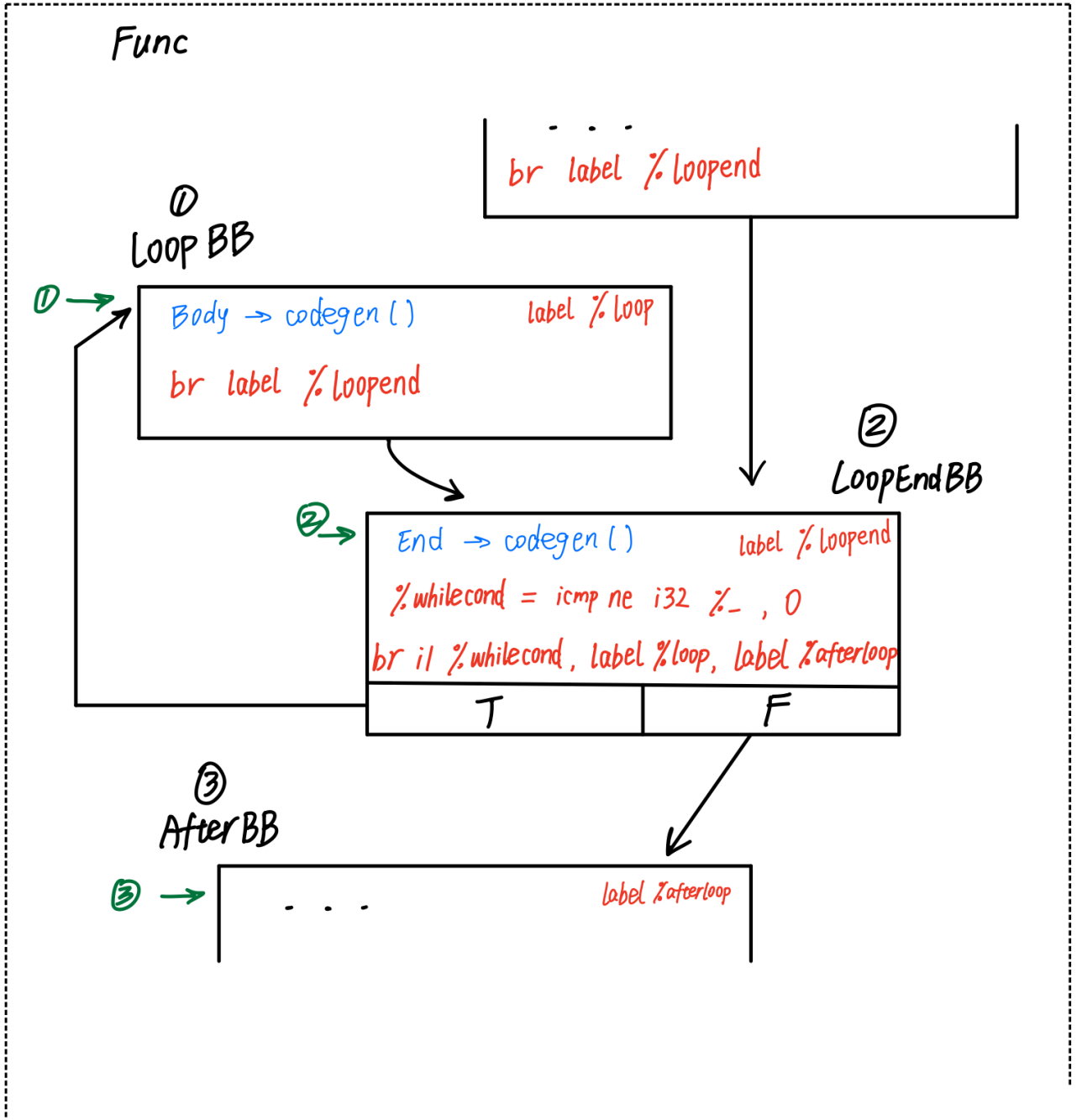
```
Value *BlockAST::codegen() {
  // new symtab for each scope
  auto symtab_stash = NamedValues;
  for (auto &&pAST : items) {
    pAST->codegen();
  }
  // restore symtab
  NamedValues = symtab_stash;
  return nullptr;
}
```

控制流

LLVM 用 `BasicBlock` 类做 IR 的基本容器，呼应汇编中的 基本块 概念 (a single entry single exit section of the code). 实现条件分支与循环，需要创建多个基本块，在其中分别插入适当的 IR ，并用 terminator instructions (可以简单理解为跳转) 连接这些基本块。

- `IRBuilder` 管理了当前插入 IR 的基本块，以及插入位置，称为 `InsertPoint` . 顺序生成 IR 时不需要考虑它，但切换基本块插入 IR 时需要 `SetInsertPoint`
- `BasicBlock` 最后必须用 `insertInto` 方法插入 `Function` 中。一串 `BasicBlock` s 组成了一个 `Function` 的 body.
- 使用 `CreateCondBr` 和 `CreateBr` 插入跳转指令


Func
attach
basic block (BB)
insert point (IP)
set
...
Cond -> codegen()
%ifcond = icmp ne i32 %_ , 0
br i1 %ifcond, label %then, label %else
T
F
① ThenBB
Then -> codegen()
label %then
br label %ifcont
② ElseBB
Else -> codegen()
label %else
br label %ifcont
③ MergeBB
...
label %ifcont



IfAST::codegen 和 WhileAST::codegen 显得有些冗长，但主要是因为要实现的过程有三四个步骤；方法所做的事情就是在当前函数体中插入上图所示的一个结构。如果思路清晰，是不难实现的。

break 与 continue

LLVM IR 直接支持跳转到基本块，因此处理结构化的控制语句是简单的。我们利用 GotoAST 类来 cover 到 break 和 continue 语句。问题简化为在 GotoAST::codegen 时提供位置信息。采用了类似 python 魔术变量的 trick, 简单在 Body->codegen() 之前复制一份符号表，往里面塞了两个编译器内部使用的标签

```
Value *WhileAST::codegen() {
  ...entering
  // establish magic labels
  auto symtab_stash = NamedValues;
  NamedValues["__WEND__"] = AfterBB;
  NamedValues["__WHILE__"] = LoopBB;
  ...gen body
  // restore symtab
  NamedValues = symtab_stash;
```

```
Value *GotoAST::codegen() {
  if (auto *tgtBB = dyn_cast<BasicBlock>(NamedValues[label])) {
    Builder->CreateBr(tgtBB);
```

逻辑运算符 短路求值

短路求值即 如果当前结果已能够确定逻辑运算式的真值，就不能继续求值右侧的操作数。未实现短路求值的后果包括 意外的指针访问，全局变量改写，I/O调用等。短路求值 可以改写成 **if-else** 的形式，如 `x && y` 等价于

```
if (evaluate x) is true
  then use (evaluate y)
  else use x
```

- 不能再像前面那样一开始一起求取两个操作数
- 与 IfAST::codegen 的逻辑类似
- 为了遵循 SSA 且不使用 alloca, 利用 PHI Node 根据控制来源，选择正确的值返回

```
Value *BinaryExprAST::codegen() {
  Value *L = LHS->codegen(), *R; // codegen of RHS postponed due to Logic Op
  ...
    } else {
      // logic op: short-circuit evaluation
      BasicBlock *EntryBB = Builder->GetInsertBlock();
      Value *CondV = Builder->CreateICmpNE(L, Builder->getInt32(0), "short");
      // ...Fill 'then' block
      R = RHS->codegen();
      R = Builder->CreateICmpNE(R, Builder->getInt32(0), "i32toi1_");
      Builder->CreateBr(MergeBB);
      // ...Finish merge block.
      PHINode *PN = Builder->CreatePHI(Builder->getInt1Ty(), 2, "logictmp");
      PN->addIncoming(CondV, EntryBB);
      PN->addIncoming(R, ThenBB);
      return Builder->CreateZExt(PN, Builder->getInt32Ty(), "i1toi32_");
```

数组支持

编译时常量求值

Short Answer: No need to DIY.

本来思考基于 递归匹配 + dynamic_cast 实现，手敲了一半，基本是在对常数重新实现加减乘除运算。后来发现这部分工作，其实在最开始实现单双目运算符时已经做了 Orz. LLVM 使用了一套自洽的，基于模板，高度优化的 RTTI 系统。一大福利是，实现了非常 优雅 的常数折叠：包含在 `IRBuilder` 创建指令的过程中了！

IRBuilder 示例 https://llvm.org/doxygen/IRBuilder_8h_source.html#l01603

```
 Value *CreateNot(Value *V, const Twine &Name = "") {
   if (auto *VC = dyn_cast<Constant>(V))
     return Insert(Folder.CreateNot(VC), Name);
   return Insert(BinaryOperator::CreateNot(V), Name);
 }
```

需要某个编译时可计算的常量时，先像一般的 Expr 节点一样调用codegen(), 再直接使用 LLVM RTTI 设施 cast 获得常数

数组声明

- 局部作用域的数组使用 alloca 创建

```
// alloc stack space
auto alloc = Builder->CreateAlloca(Builder->getInt32Ty(), len, vn.Name);
```

- 全局变量需要单独处理，创建 llvm::GlobalVariable
- 全局值一定要使用 ExternalLinkage. 为此花费了 2 hr debug

```
// array: len must be constant
auto len = cast<ConstantInt>(vn.len->codegen());
auto ulen = len->getZExtValue();
auto ArrTy = ArrayType::get(Builder->getInt32Ty(), ulen);
// using zero initializer
auto init_ref = std::vector<Constant *>(ulen, Builder->getInt32(0));
if (vn.iv) {
  auto p = static_cast<BlockAST *>(vn.iv.get());
  for (int i = 0, e = p->items.size(), f = init_ref.size();
       i < e && i < f; i++) {
    // each must be constant
    init_ref[i] = cast<ConstantInt>(p->items[i]->codegen());
  }
}
gv = new GlobalVariable(
    *TheModule, ArrTy, isConst, GlobalVariable::ExternalLinkage,
    ConstantArray::get(ArrTy, ArrayRef<Constant *>(init_ref)), vn.Name);
```

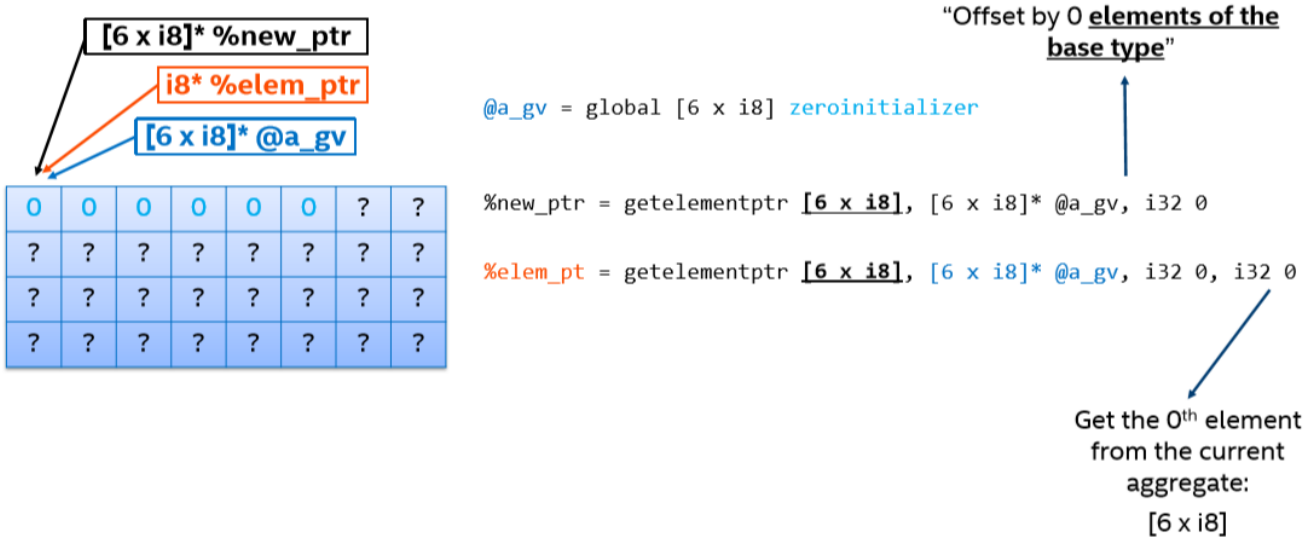
数组索引

用到比较难的 GEP (get element ptr) 指令. 对于局部数组，可以类比 x86 asm lea 指令，此时 alloca 是一个 i32*；而全局变量非常坑，因为它总是一个指针；特别地，对于全局数组，它是 ArrayType* 而不是 i32*

Manipulating pointers

这意味着 GEP 的 IdxList 事实上有两个元素: {0, idx->codegen()}

Manipulating pointers

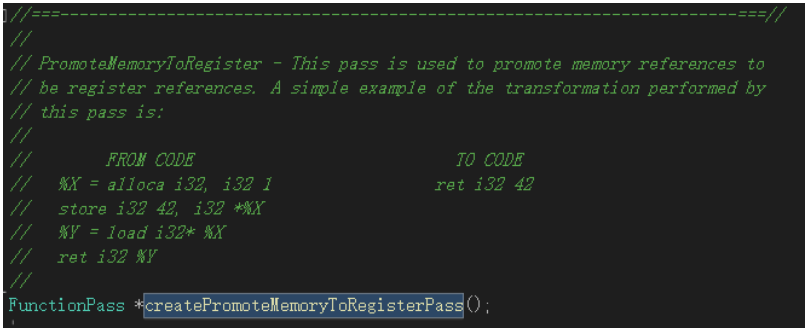



```
Value *VariableExprAST::codegen() {
  if (idx) {
    std::vector<Value *> Idx{};
    // cause: llvm gv is always a ptr (to ArrTy) while alloca simply ret i32*
    if (var->getType()->getContainedType(0)->isArrayTy()) {
      Idx.push_back(Builder->getInt32(0));
    }
    Idx.push_back(idx->codegen());
    var = Builder->CreateGEP(var, Idx, "arridx");
  }
  return Builder->CreateLoad(Builder->getInt32Ty(), var, Name);
```

```
}
```

代码优化

调用 LLVM 提供的优化 pass. 特别地，mem2reg pass 使用 *iterated dominance frontier* 算法消除我们之前无脑 alloca 出的变量



此外还使用了窥孔优化，表达式重排，公共子表达式消除与控制流简化。由于均是调 API, 不阐述原理了。

运行时环境

引入了官方教程的 JIT, 可以在内存中动态编译 SysY 源程序。同时使用动态链接库导出函数，为 SysY 提供 I/O 库。示例

```
#ifdef _WIN32
#define DLLEXPORT __declspec(dllexport)
#else
#define DLLEXPORT
#endif

extern "C" DLLEXPORT int getint() {
  int rv;
  scanf("%d", &rv);
  return rv;
}
```

LLVM JIT 和同一编译环境保证了 ABI 兼容

```
---Compile file ../test/functional_test\12_getint.sc:---

; ModuleID = 'SysY--'
source_filename = "SysY--"
target datalayout = "e-m:w-p270:32:32-p271:32:32-p272:64:64-i64:64-f80:128-n8:16:32:64-S128"

define i32 @main() {
entry:
  %calltmp = call i32 @getint()
  ret i32 %calltmp
}

declare i32 @getint()
250
Target main() exited on 250
```

参考

教程

- https://llvm.org/docs/tutorial/MyFirstLanguageFrontend/index.html
- 二手中文 https://llvm-tutorial-cn.readthedocs.io/en/latest/index.html
- 系列 逐步讲解 https://www.bilibili.com/video/av626821579/

开发 & 测试

- LLVM IR 指令速学 https://zhuanlan.zhihu.com/p/64427829

参考

- 稍微友好一些 https://llvm.org/docs/ProgrammersManual.html
- https://llvm.org/docs/LangRef.html
- https://llvm.org/doxygen/classllvm_1_1IRBuilderBase.html
- mapping-high-level-constructs-to-llvm-ir.readthedocs.io

前人工作借鉴

- w/ English report: https://github.com/huangyangyi/Our-Pascal-Compiler
- 介绍了完整流水线 http://www.cppblog.com/woaidongmao/archive/2009/11/11/100693.aspx https://github.com/lsegal/my_toy_compiler

5.2 目标代码生成：基于 LLVM

在中间代码生成中，我们生成了 LLVM IR 的可视化文件.ll 文件，这里以 main.ll 文件为例演示如何生成汇编代码：

```
llvm-as main.ll -o main.bc
llc main.bc -o main.s
```

其中生成的 main.bc 是 LLVM IR 的字节码格式，main.s 是生成的汇编代码。

同时我们也可以直接运行中间代码：

```
lli --dlopen="x64/Debug/libsysy.dll" .\main.ll
```

这里我们必须链接我们的输入输出用的动态库 libsysy.dll。

第六章、测试案例　　（每个语句成分的测试案例，至少两个复杂语句组合后的测试案例）

6.1 语法分析功能性测试

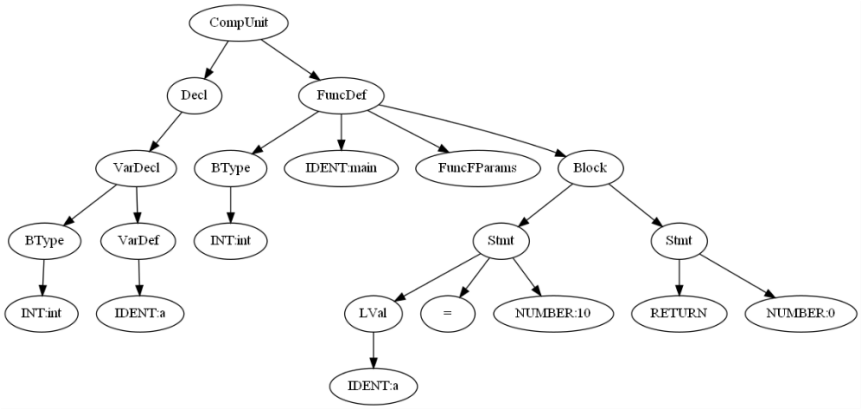
语法分析测试所有的测试样例在文件./code/functional_test /文件夹内，而输出的可视化语法树均生成在文件夹./code/viewTree/内，测试时在 Window 环境下将 VS 生成的 exe 文件拷入 code 目录，然后在 code 目录下命令行运行 .\test_syntax.bat。运行过程中有些文件需要输入，这些输入在./code/functional_test/input.txt 内提供。此处仅展示几个较简单的测试案例（因为报告篇幅原因，语法树可能放不下）

6.1.1 样例 1：

输入代码：01_var_defn.c--main 函数+声明+赋值

```
1  int a;
2  int main(){
3      a=10;
4      return 0;
5  }
```

输出语法树：

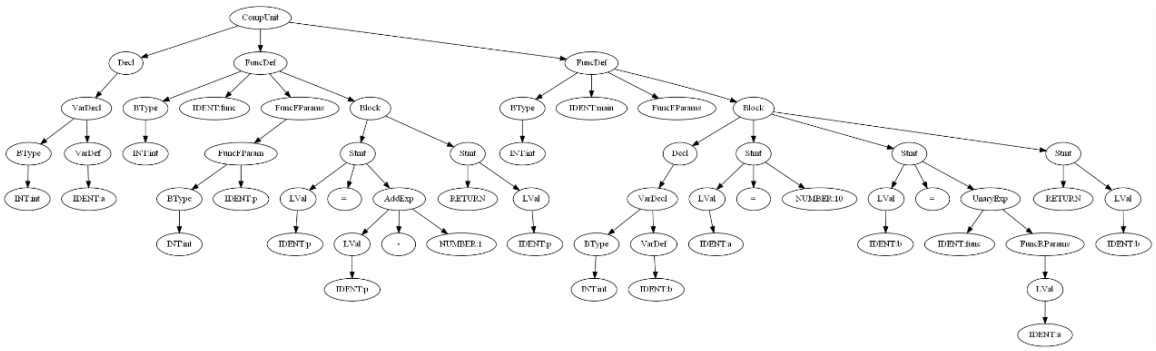


6.1.2 样例 2：

输入代码：04_func_defn.c--函数定义+函数调用+声明+运算

```
1   int a;
2   int func(int p){
3       p = p - 1;
4       return p;
5   }
6   int main(){
7       int b;
8       a = 10;
9       b = func(a);
10      return b;
```

输出语法树：

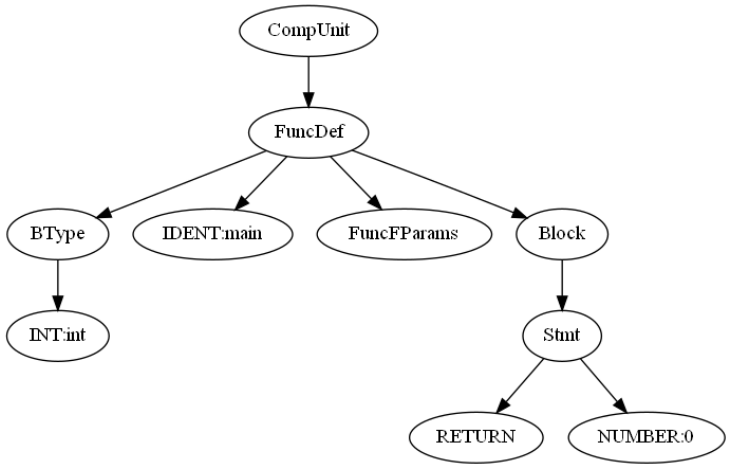


6.1.3 样例 3：

输入代码：08_comment2.c--注释

```
1   int main(){
2       /*
3       The comment block;
4       int a=5;
5       int b[10];
6       int c;
7       int d=a*2;
8       */
9       return 0;
```

输出语法树：

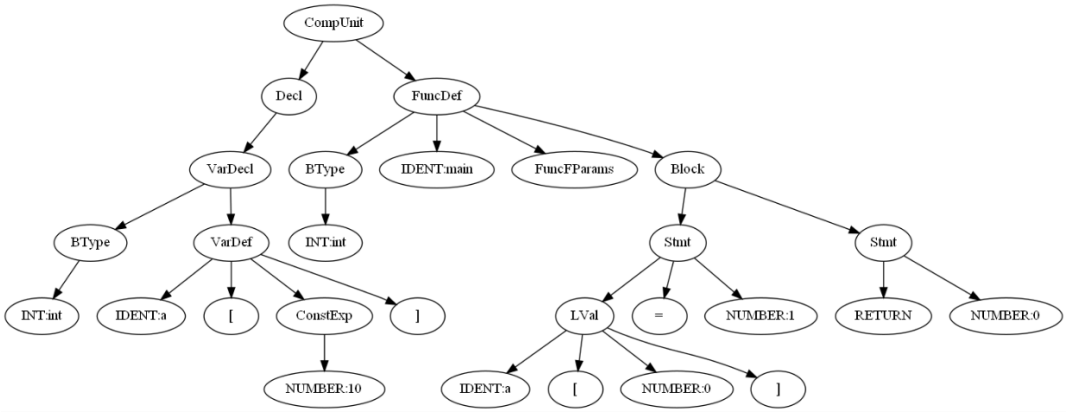


6.1.4 样例 4：

输入代码：08_arr_assign.c--数组声明及赋值

```
int a[10];
int main(){
    a[0]=1;
    return 0;
}
```

输出语法树：

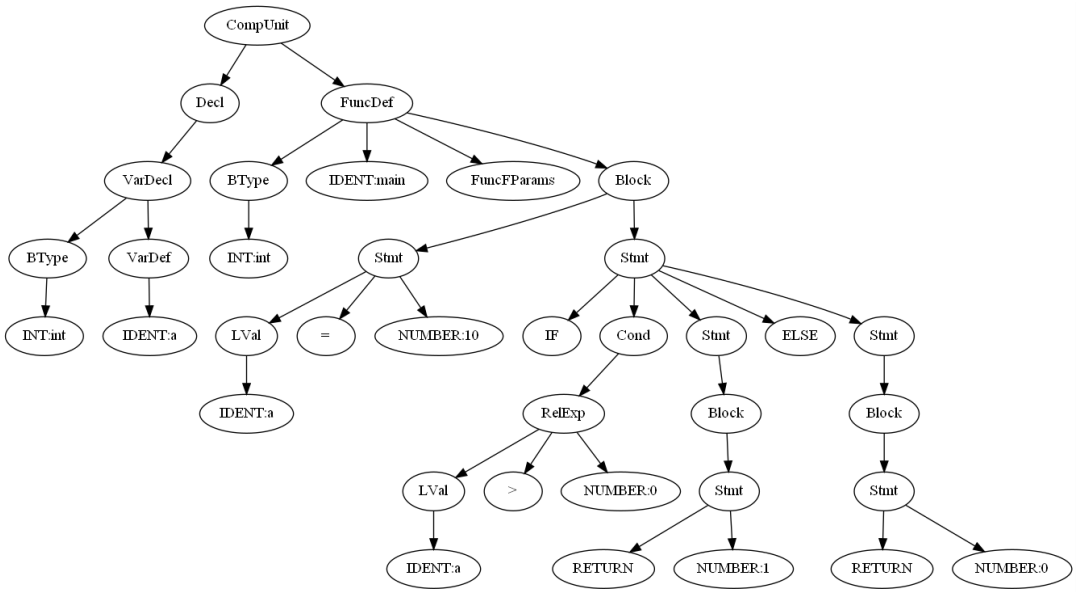


6.1.5 样例 5：

输入代码：10_if_else.c--if-else

```
int a;
int main(){
    a = 10;
    if( a>0 ){
        return 1;
    }
    else{
        return 0;
    }
```

输出语法树：

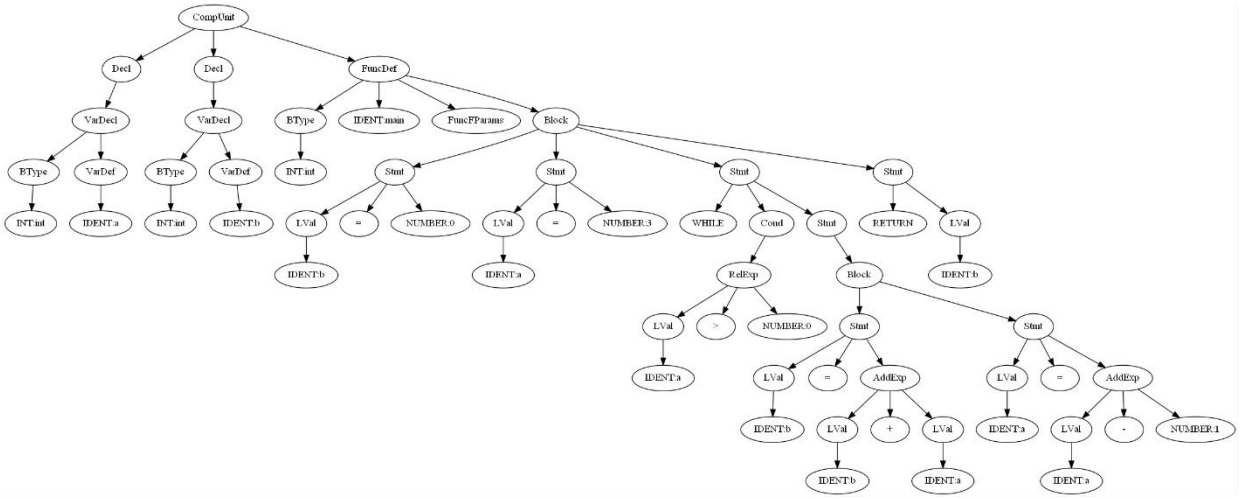


6.1.6 样例 6：

输入代码：11_while.c--while

```
1   int a;
2   int b;
3   int main(){
4       b=0;
5       a=3;
6       while(a>0){
7           b = b+a;
8           a = a-1;
9       }
10      return b;
```

输出语法树：



6.2 错误检测测试（即语义分析测试+语法词法纠错测试）

6.2.1 样例 1：

输入错误代码：出现我们语言不支持的 C 关键字

```
1   int main()
2   {
3       float c;
4       char d;
5       double e;
6       bool f;
7
8       int a;
9       int b[2];
10  }
```

输出错误：

```
---Compile file ./error_test/00_undef_keyword.c:---
Error [Lexical] at Line 3, Col 10: Undefined data type 'float'.
Error [Lexical] at Line 4, Col 9: Undefined data type 'char'.
Error [Lexical] at Line 5, Col 11: Undefined data type 'double'.
Error [Lexical] at Line 6, Col 9: Undefined data type 'bool'.
4 errors occured when compiling.
```

6.2.2 样例 2：

输入错误代码：未知的符号

```
1   int main()
2   {
3       int i = 1;
4       int j = ~i;
5   }
```

输出错误：

```
---Compile file ./error_test/01_mys_~.c:---
Error [Lexical] at line 4, Col 13: Mysterious character '~'.
1 error occured when compiling.
```

6.2.3 样例 3：

输入错误代码：语法错误混合未定义关键字错误

```
1   int main()
2   {
3       float a[10][2];
4       int i;
5       a[5, 3] = 1.5;
6       if (a[1][2] == 0)
7           i = 1 else i = 0;
8   }
```

输出错误：

```
---Compile file ./error_test/02_syntax_error.c:---
Error [Lexical] at Line 3, Col 10: Undefined data type 'float'.
Error [Syntax] at Line 5, Col 9: parse error ','.
Error [Syntax] at Line 6, Col 14: parse error '['.
Error [Syntax] at Line 7 Col 26: Syntax error ';'.
4 errors occured when compiling.
```

6.2.4 样例 4：

输入错误代码：错误的数据表示形式及不支持的数据常量

```
1   int main()
2   {
3       int i = 092;
4       int j = 0x3G;
5       int a = 0;
6       int b = 1.5e-03;
7       int k = 00010;
8       int c = 1.6234;
9       int d = '5';
10      int e = "s";
11      int f = true;
12      int g = false;
13  }
```

输出错误：

```
---Compile file ./error_test/03_wrong_data_format.c:---
Error [Lexical] at Line 3, Col 16: Illegal octal number '092'.
Error [Lexical] at Line 4, Col 17: Illegal hex number '0x3G'.
Error [Lexical] at Line 6, Col 20: Unsupported scientific format '1.5e-03'.
Error [Lexical] at Line 7, Col 18: Confusing decimal format '00010'?
Error [Lexical] at Line 8, Col 19: Unsupported float/double format '1.6234'.
Error [Lexical] at line 9, Col 13: Unsupported char/string.
Error [Lexical] at line 10, Col 13: Unsupported char/string.
Error [Lexical] at Line 11, Col 17: Unsupported bool format 'true'.
Error [Lexical] at Line 12, Col 18: Unsupported bool format 'false'.
9 errors occured when compiling.
```

6.2.5 样例 5：

输入错误代码：错误注释

```
1   int main()
2   {
3       /*
4       /*
5       */
6       */
7           int i = 0;
8       //
9       // asdf adsfrg
10      /*asdf
11
12
13  }
```

输出错误：

```
---Compile file ./error_test/04_wrong_comments.c:---
Error [Lexical] at Line 6, Col 5: Unmatched comment for "*/".
Error [Lexical] at Line 13, Col 9: Unmatched comment for "/*".
2 errors occured when compiling.
```

6.2.6 样例 6：

输入错误代码：错误的常量定义（我们要求常量必须在定义时初始化）+不支持的数据常量+错误的常量数组定义（我们要求常量数组必须在定义时初始化，并且要求必须指明数组长度，而且长度必须为整数）

```
1   int main()
2   {
3       const int a;
4       const int b = 1;
5       const int c = 3.5;
6       const int d[];
7       const int e[2];
8       const int i[2.5] = {};
9   }
```

输出错误：

```
---Compile file ./error_test/05_wrongdef_const.c:---
Error [Syntax] at Line 3, Col 17: Define const var without initialization.
Error [Lexical] at Line 5, Col 22: Unsupported float/double format '3.5'.
Error [Syntax] at Line 6, Col 18: Define const array without space specification.
Error [Syntax] at Line 7, Col 20: Define const array without initialization.
Error [Lexical] at Line 8, Col 20: Unsupported float/double format '2.5'.
5 errors occured when compiling.
```

6.2.7 样例 7：

输入错误代码：不支持的数据常量+错误的整型数组定义（我们要求整型数组必须指明数组长度，而且长度必须为整数）

```
1   int main()
2   {
3       int a;
4       int b = 1;
5       int c = 3.5;
6       int d[];
7       int e[2];
8       int i[2.5] = {};
9   }
```

输出错误：

```
---Compile file ./error_test/06_wrongdef_var.c:---
Error [Lexical] at Line 5, Col 16: Unsupported float/double format '3.5'.
Error [Syntax] at Line 6, Col 12: Define an array without space specification.
Error [Lexical] at Line 8, Col 14: Unsupported float/double format '2.5'.
3 errors occured when compiling.
```

6.2.8 样例 8：

输入错误代码：未初始化完全或多初始化的常量数组(我们要求常量数组必须按长度初始化)+数组长度出现非数字

```
1   int main()
2   {
3       int b = 2;
4       const int c = 3;
5       const int f[2] = {};
6       const int g[2] = {1};
7       const int h[2] = {1, 2, 3};
8       const int j[b] = {1};
9       const int k[b] = {1, 2};
10      const int l[2] = {1, 2};
11      const int m[l] = {1, 2};
12      const int o[2] = l;
13      const int j[l[5]] = {1, 2, 3};
14      const int p[2] = {l[1], 2};
15  }
```

输出错误：

```
---Compile file ./error_test/07_wrongdef_const.c:---
Error [Semantic] at Line 5: Define const array main:f without initialization.
Error [Semantic] at Line 6: Define const array main:g with incomplete initialization size 1 < 2.
Error [Semantic] at Line 7: Define array main:h with initialization size 3 > 2.
Error [Semantic] at Line 8: Can't define an array's size with NOT-NUMBER.
Error [Semantic] at Line 9: Can't define an array's size with NOT-NUMBER.
Error [Semantic] at Line 11: Can't define an array's size with NOT-NUMBER.
Error [Semantic] at Line 12: Can't initialize an array with NOT-NUMBER.
Error [Semantic] at Line 13: Can't define an array's size with NOT-NUMBER.
Error [Semantic] at Line 14: Can't initialize an array with NOT-NUMBER.
```

6.2.9 样例 9：

输入错误代码：未初始化完全或多初始化的整型数组(我们要求整型数组初始化可以可变长度，但是不能超出数组长度限制)+数组长度出现非数字

```
1   int main()
2   {
3       int b = 2;
4       int c = 3;
5       int f[2] = {};
6       int g[2] = {1};
7       int h[2] = {1, 2, 3};
8       int j[b] = {1};
9       int k[b] = {1, 2};
10      int l[2] = {1, 2};
11      int m[l] = {1, 2};
12      int o[2] = l;
13      int j[l[5]] = {1, 2, 3};
14      int p[2] = {l[1], 2};
15  }
```

输出错误：

```
---Compile file ./error_test/08_wrongdef_var.c:---
Error [Semantic] at Line 7: Define array main:h with initialization size 3 > 2.
Error [Semantic] at Line 8: Can't define an array's size with NOT-NUMBER.
Error [Semantic] at Line 9: Can't define an array's size with NOT-NUMBER.
Error [Semantic] at Line 11: Can't define an array's size with NOT-NUMBER.
Error [Semantic] at Line 12: Can't initialize an array with NOT-NUMBER.
Error [Semantic] at Line 13: Can't define an array's size with NOT-NUMBER.
Error [Semantic] at Line 14: Can't initialize an array with NOT-NUMBER.

7 errors occured when compiling.
```

6.2.10 样例 10：

输入错误代码：未声明/定义的数组、变量引用

```
1   int f1(int a, int d[])
2   {
3       a;
4       b = 1;
5       d = 2;
6       e[2];
7   }
8
9   int main()
10  {
11      a;
12      b = 1;
13      e[2];
```

```
14
15      if (1)
16      {
17          a;
18          b = 1;
19          e[2];
20      }
21      while (1)
22      {
23          a;
24          b = 1;
25          e[2];
26      }
27  }
```

输出错误：

```
---Compile file ./error_test/09_undef_var.c:---
Error [Semantic] at Line 4: Undefined reference to VAR b.
Error [Semantic] at Line 6: Undefined reference to ARRAY e.
Error [Semantic] at Line 11: Undefined reference to VAR a.
Error [Semantic] at Line 12: Undefined reference to VAR b.
Error [Semantic] at Line 13: Undefined reference to ARRAY e.
Error [Semantic] at Line 23: Undefined reference to VAR a.
Error [Semantic] at Line 24: Undefined reference to VAR b.
Error [Semantic] at Line 25: Undefined reference to ARRAY e.

8 errors occured when compiling.
```

6.2.11 样例 11：

输入错误代码：将非数组当作数组使用

```
1   int main()
2   {
3       int i;
4       int b[2];
5       i[0];
6       i[5 + 3 - b[0 / 0]];
7   }
```

输出错误：

```
---Compile file ./error_test/10_wrong_use.c:---
Error [Semantic] at Line 5: i is not an ARRAY.
Error [Semantic] at Line 6: i is not an ARRAY.

2 errors occured when compiling.
```

6.2.12 样例 12：

输入错误代码：未声明/定义的函数引用

```
1   int main()
2   {
3       f1();
4       f2();
5   }
```

输出错误：

```
---Compile file ./error_test/11_undef_func.c:---
Error [Semantic] at Line 3: Undefined reference to FUNC f1.
Error [Semantic] at Line 4: Undefined reference to FUNC f2.

2 errors occured when compiling.
```

6.2.13 样例 13：

输入错误代码：常量的重复定义错误

```
1   const int b = 1;
2   const int f[2] = {1, 2};
3   int f1()
4   {
5       const int b = 1;
6       const int f[2] = {1, 2};
7       const int b = 1;
8       const int f[2] = {1, 2};
9   }
10  int main()
11  {
12      const int b = 1;
13      const int f[2] = {1, 2};
14      const int b = 1;
15      const int f[2] = {1, 2};
16      if (1){
17          const int b = 1;
18          const int f[2] = {1, 2};
19      }
20      while (1){
21          const int b = 1;
22          const int f[2] = {1, 2};
23      }
24  }
```

输出错误：

```
---Compile file ./error_test/12_reapdef_const.c:---
Error [Semantic] at Line 7: CONST name f1:b has already been declared.
Error [Semantic] at Line 8: CONST name f1:f has already been declared.
Error [Semantic] at Line 14: CONST name main:b has already been declared.
Error [Semantic] at Line 15: CONST name main:f has already been declared.

4 errors occured when compiling.
```

6.2.14 样例 14：

输入错误代码：整型变量的重复定义错误

```
1   int a;
2   int b = 1;
3   int e[2];
4   int f[2] = {1, 2};
5   int f1(){
6       int a;
7       int b = 1;
8       int e[2];
9       int f[2] = {1, 2};
10
11      int a;
12      int b = 1;
13      int e[2];
14      int f[2] = {1, 2};
15  }
```

```c
int main(){
    int a;
    int b = 1;
    int e[2];
    int f[2] = {1, 2};
    int a;
    int b = 1;
    int e[2];
    int f[2] = {1, 2};
    if (1){
        int a;
        int b = 1;
        int e[2];
        int f[2] = {1, 2};
    }
    while (1){
        int a;
        int b = 1;
        int e[2];
        int f[2] = {1, 2};
    }
}
```

输出错误：

```
---Compile file ./error_test/13_reapdef_var.c:---
Error [Semantic] at Line 13: VAR name f1:a has already been declared.
Error [Semantic] at Line 14: VAR name f1:b has already been declared.
Error [Semantic] at Line 15: VAR name f1:e has already been declared.
Error [Semantic] at Line 16: VAR name f1:f has already been declared.
Error [Semantic] at Line 26: VAR name main:a has already been declared.
Error [Semantic] at Line 27: VAR name main:b has already been declared.
Error [Semantic] at Line 28: VAR name main:e has already been declared.
Error [Semantic] at Line 29: VAR name main:f has already been declared.

8 errors occured when compiling.
```

6.2.15 样例 15：

输入错误代码：函数名的重复定义错误

```c
int f1()
{
}

int f1()
{
}

int main()
{
}
```

输出错误：

```
---Compile file ./error_test/14_reapdef_func.c:---
Error [Semantic] at Line 5: FUNC name f1 has already been declared.

1 error occured when compiling.
```

6.2.16 样例 16：

输入错误代码：break、continue 的错误使用（我们规定 break、continue 只能再循环题中使用）

```c
int main(){
    if (1){
        while (1){
            if (1){
                break;
            }
            else{
                continue;
            }
            break;
        }
    }
    if (1){
        break;
        continue;
        if (1){
            break;
            continue;
        }
        else{
            break;
            continue;
        }
    }
}
```

输出错误：

```
---Compile file ./error_test/18_break_continue_error.c:---
Error [Semantic] at Line 14: 'break' can be only used in loop.
Error [Semantic] at Line 15: 'continue' can be only used in loop.
Error [Semantic] at Line 17: 'break' can be only used in loop.
Error [Semantic] at Line 18: 'continue' can be only used in loop.
Error [Semantic] at Line 21: 'break' can be only used in loop.
Error [Semantic] at Line 22: 'continue' can be only used in loop.

6 errors occured when compiling.
```

6.2.17 样例 17：

输入错误代码：表达式的错误使用

```
1   int main()
2   {
3       int a;
4       int b = 1;
5       int c[2] = {1, 2};
6
7       // Array can not be LVal
8       c = 1;
9       c = a;
10      c = b;
11
12      // Array can not in cal
13      a = b + c;
14      a = b - c;
15      a = b / c;
16      a = b * c;
17      a = b % c;
18
19      a = 1 + c;
20      a = 1 - c;
21      a = 1 / c;
22      a = 1 * c;
23      a = 1 % c;
24
25      // Const value
26      const int o = 1;
27      o = 5;
28      a = o;
29
30      // Complex exp
31      a = c[((b * 1 / 2) + 4 + (a - b) - (b + 3) % 2) % c];
32
33      // Redudand operator
34      a = +-b + 1;
35      a = --1;
36
37      // Compare
38      int m;
39      int n[2];
40      if (m + a - 1 + 1 - 1 + 1 * 1 / 1 % 1 == n)
41      {
42          return 0;
43      }
44      if (m + 1 != n)
45      {
46          return 0;
47      }
48  }
```

输出错误：

```
---Compile file ./error_test/15_exp_error.c:---
Error [Semantic] at Line 8: Can't mix ARRAY main:c in INT expression.
Error [Semantic] at Line 9: Can't mix ARRAY main:c in INT expression.
Error [Semantic] at Line 10: Can't mix ARRAY main:c in INT expression.
Error [Semantic] at Line 13: Can't mix ARRAY main:c in INT expression.
Error [Semantic] at Line 14: Can't mix ARRAY main:c in INT expression.
Error [Semantic] at Line 15: Can't mix ARRAY main:c in INT expression.
Error [Semantic] at Line 16: Can't mix ARRAY main:c in INT expression.
Error [Semantic] at Line 17: Can't mix ARRAY main:c in INT expression.
Error [Semantic] at Line 19: Can't mix ARRAY main:c in INT expression.
Error [Semantic] at Line 20: Can't mix ARRAY main:c in INT expression.
Error [Semantic] at Line 21: Can't mix ARRAY main:c in INT expression.
Error [Semantic] at Line 22: Can't mix ARRAY main:c in INT expression.
Error [Semantic] at Line 23: Can't mix ARRAY main:c in INT expression.
Error [Semantic] at Line 27: Can't mix a LVal main:o in calculation.
Error [Semantic] at Line 31: ARRAY main:c can't be another array's index.

15 errors occured when compiling.
```

6.2.18 样例 18：

输入错误代码：函数调用格式有误

```
1   int f1(int a, int b[])
2   {
3   }
4
5   int main()
6   {
7       int a;
8       int b[2];
9       f1();
10      f1(a);
11      f1(a, a);
12      return 0;
13  }
```

输出错误：

```
---Compile file ./error_test/16_func_call_error.c:---
Error [Semantic] at Line 1: FUNC f1 doesn't return.
Error [Semantic] at Line 9: The number of function call arguments 0 does not match the definition 2.
Error [Semantic] at Line 10: The number of function call arguments 1 does not match the definition 2.
Error [Semantic] at Line 11: Func f1 call main:a(BASIC) mismatches with definition f1:b(ARRAY).

4 errors occured when compiling.
```

6.2.19 样例 19：

输入错误代码：函数返回错误

```
1   int f1()
2   {
3   }
4
5   int f2()
6   {
7       int a[10];
8       return a[1];
9       return a;
10  }
```

输出错误：

```
---Compile file ./error_test/17_func_ret_error.c:---
Error [Semantic] at Line 1: FUNC f1 doesn't return.
Error [Semantic] at Line 9: Return type ARRAY mismatchs with FUNC f2 definition type BASIC.

2 errors occured when compiling.

---Compile file ./error_test/18_break_continue_error.c:---
Error [Semantic] at Line 14: 'break' can be only used in loop.
Error [Semantic] at Line 15: 'continue' can be only used in loop.
Error [Semantic] at Line 17: 'break' can be only used in loop.
Error [Semantic] at Line 18: 'continue' can be only used in loop.
Error [Semantic] at Line 21: 'break' can be only used in loop.
Error [Semantic] at Line 22: 'continue' can be only used in loop.
Error [Semantic] at Line 1: FUNC main doesn't return.

7 errors occured when compiling.
```

6.2.20 样例 20：

输入错误代码：空文件

```
1
```

输出错误：

```
---Omit blank file ./error test/19 blank test.c:---
```

6.3 中间代码及目标代码生成测试

6.3.1 样例 1：

输入类 C 代码： 01_var_defn.c--main 函数+声明+赋值

```
1   int a;
2   int main(){
3       a=10;
4       return 0;
5   }
```

输出中间代码：

```
---Compile file ./functional_test/01_var_defn.c:---

; ModuleID = 'SysY--'
source_filename = "SysY--"
target datalayout = "e-m:w-p270:32:32-p271:32:32-p272:64:64-i64:64-f80:128-n8:16:32:64-S128"

@a = common global i32 0

define i32 @main() {
entry:
  store i32 10, i32* @a, align 4
  ret i32 0
}
Target main() exited on 0
```

输出目标代码：

```
1       .text
2       .def     @feat.00;
3       .scl    3;
4       .type   0;
```

```
    .endef
    .globl  @feat.00
.set @feat.00, 0
    .file   "SysY--"
    .def     main;
    .scl    2;
    .type   32;
    .endef
    .globl  main                # -- Begin function main
    .p2align    4, 0x90
main:                           # @main
# %bb.0:                        # %entry
    movl    $10, a(%rip)
    xorl    %eax, %eax
    retq
                                # -- End function
    .bss
    .globl  a                   # @a
    .p2align    2
a:
    .long   0                   # 0x0
```

6.3.2 样例 2：

输入类 C 代码：04_func_defn.c--函数定义+函数调用+声明+运算

```
int a;
int func(int p){
    p = p - 1;
    return p;
}
int main(){
    int b;
    a = 10;
    b = func(a);
    return b;
}
```

输出中间代码：

```
---Compile file ./functional_test/04_func_defn.c:---

; ModuleID = 'SysY--'
source_filename = "SysY--"
target datalayout = "e-m:w-p270:32:32-p271:32:32-p272:64:64-i64:64-f80:128-n8:16:32:64-S128"

@a = common global i32 0

define i32 @func(i32 %p) {
entry:
  %arithtmp = add i32 %p, -1
  ret i32 %arithtmp
}

define i32 @main() {
entry:
  store i32 10, i32* @a, align 4
  %calltmp = call i32 @func(i32 10)
  ret i32 %calltmp
}
Target main() exited on 9
```

输出目标代码：

```
    .text
    .def     @feat.00;
    .scl    3;
    .type   0;
    .endef
    .globl  @feat.00
.set @feat.00, 0
    .file   "SysY--"
    .def     main;
    .scl    2;
    .type   32;
    .endef
    .globl  main                            # -- Begin function
    .p2align    4, 0x90
main:                                       # @main
# %bb.0:                                    # %entry
    movl    $4, %eax
    retq
                                            # -- End function
    .section    .rdata,"dr"
    .globl  x                                   # @x
    .p2align    2
x:
    .long   4                                   # 0x4
```

6.3.3 样例 3：

输入类 C 代码：08_comment2.c--注释

```
int main(){
    /*
    The comment block;
    int a=5;
    int b[10];
    int c;
    int d=a*2;
    */
    return 0;
```

输出中间代码：

```
---Compile file ./functional_test/08_comment2.c:---

; ModuleID = 'SysY--'
source_filename = "SysY--"
target datalayout = "e-m:w-p270:32:32-p271:32:32-p272:64:64-i64:64-f80:128-n8:16:32:64-S128"

define i32 @main() {
entry:
  ret i32 0
}
Target main() exited on 0
```

输出目标代码：

```
    .text
    .def     @feat.00;
    .scl    3;
    .type   0;
    .endef
    .globl  @feat.00
.set @feat.00, 0
    .file   "SysY--"
    .def     main;
    .scl    2;
    .type   32;
    .endef
    .globl  main                            # -- Begin function
    .p2align    4, 0x90
main:                                       # @main
# %bb.0:                                    # %entry
    xorl    %eax, %eax
    retq
                                            # -- End function
```

6.3.4 样例 4：

输入类 C 代码：08_arr_assign.c--数组声明及赋值

```
int a[10];
int main(){
    a[0]=1;
    return 0;
}
```

输出中间代码：

```
---Compile file ./functional_test/08_arr_assign.c:---

; ModuleID = 'SysY--'
source_filename = "SysY--"
target datalayout = "e-m:w-p270:32:32-p271:32:32-p272:64:64-i64:64-f80:128-n8:16:32:64-S128"

@a = common global [10 x i32] zeroinitializer

define i32 @main() {
entry:
  store i32 1, i32* getelementptr inbounds ([10 x i32], [10 x i32]* @a, i64 0, i64 0), align 4
  ret i32 0
}
Target main() exited on 0
```

输出目标代码：

```
    .text
    .def     @feat.00;
    .scl    3;
    .type   0;
    .endef
    .globl  @feat.00
```

```
    .set @feat.00, 0
        .file   "SysY--"
        .def     main;
        .scl    2;
        .type   32;
        .endef
        .globl  main                            # -- Begin function
        .p2align    4, 0x90
    main:                                       # @main
    # %bb.0:                                    # %entry
        movl    $1, a(%rip)
        xorl    %eax, %eax
        retq
                                                # -- End function
        .bss
        .globl  a                                   # @a
        .p2align    4
    a:
        .zero   40
```

6.3.5 样例 5：

输入类 C 代码：10_if_else.c--if-else

```
int a;
int main(){
    a = 10;
    if( a>0 ){
        return 1;
    }
    else{
        return 0;
    }
```

输出中间代码：

```
---Compile file ./functional_test/10_if_else.c:---

; ModuleID = 'SysY--'
source_filename = "SysY--"
target datalayout = "e-m:w-p270:32:32-p271:32:32-p272:64:64-i64:64-f80:128-n8:16:32:64-S128"

@a = common global i32 0

define i32 @main() {
entry:
  store i32 10, i32* @a, align 4
  ret i32 1
}
Target main() exited on 1
```

输出目标代码：

```
        .text
        .def     @feat.00;
        .scl    3;
        .type   0;
        .endef
        .globl  @feat.00
    .set @feat.00, 0
        .file   "SysY--"
        .def     main;
        .scl    2;
        .type   32;
```

```
    .endef
    .globl  main                            # -- Begin function
    .p2align    4, 0x90
main:                                       # @main
# %bb.0:                                    # %entry
    movl    $10, a(%rip)
    movl    $1, %eax
    retq
                                            # -- End function
    .bss
    .globl  a                                   # @a
    .p2align    2
a:
    .long   0                                   # 0x0
```

6.3.6 样例 6：

输入类 C 代码：11_while.c--while

```
int a;
int b;
int main(){
    b=0;
    a=3;
    while(a>0){
        b = b+a;
        a = a-1;
    }
    return b;
```

输出中间代码：

```
---Compile file ./functional_test/11_while.c:---

; ModuleID = 'SysY--'
source_filename = "SysY--"
target datalayout = "e-m:w-p270:32:32-p271:32:32-p272:64:64-i64:64-f80:128-n8:16:32:64-S128"

@a = common global i32 0
@b = common global i32 0

define i32 @main() {
entry:
  store i32 0, i32* @b, align 4
  br label %loopend

loop:                                             ; preds = %loopend
  %b = load i32, i32* @b, align 4
  %a = load i32, i32* @a, align 4
  %arithtmp = add i32 %b, %a
  store i32 %arithtmp, i32* @b, align 4
  %arithtmp2 = add i32 %a, -1
  br label %loopend

loopend:                                          ; preds = %loop, %entry
  %storemerge = phi i32 [ 3, %entry ], [ %arithtmp2, %loop ]
  store i32 %storemerge, i32* @a, align 4
  %cmptmp = icmp sgt i32 %storemerge, 0
  br i1 %cmptmp, label %loop, label %afterloop

afterloop:                                        ; preds = %loopend
  %b4 = load i32, i32* @b, align 4
  ret i32 %b4
}
Target main() exited on 6
```

输出目标代码：

```
    .text
    .def     @feat.00;
    .scl    3;
    .type   0;
    .endef
    .globl  @feat.00
.set @feat.00, 0
    .file   "SysY--"
    .def     main;
    .scl    2;
    .type   32;
    .endef
    .globl  main                            # -- Begin function
    .p2align    4, 0x90
main:                                       # @main
# %bb.0:                                    # %entry
    movl    $0, b(%rip)
    movl    $3, %eax
    .p2align    4, 0x90
.LBB0_2:                                    # %loopend
                                            # =>This Inner Loop
    movl    %eax, a(%rip)
    testl   %eax, %eax
    jle .LBB0_3
# %bb.1:                                    # %loop
                                        #in Loop: Header=BB0_2 Depth=1
    movl    a(%rip), %eax
    addl    %eax, b(%rip)
    decl    %eax
    jmp .LBB0_2
.LBB0_3:                                     # %afterloop
    movl    b(%rip), %eax
    retq
                                            # -- End function
    .bss
    .globl  a                                   # @a
    .p2align    2
a:
    .long   0                                   # 0x0

    .globl  b                                   # @b
    .p2align    2
b:
    .long   0                                   # 0x0
```