

# 语法分析答辩问题

## 语法分析答辩问题

- (1) nextToken() 返回什么？怎么带类型和属性？
- (2) 是如何处理 if 和 if-else 的移进-规约冲突的。
- (3) 请说明语法树的根节点是什么类型 (Root) ？
- (4) 在 bison 中，%token、%nterm 分别是什么意思，对应上下文无关文法的哪些部分？
- (5) 你在定义文法规则时是如何处理这种“零次或多次”、“一次或多次”的语法结构的？
- (6) 请说明文法设计时为什么要使用 LOGICAL\_OR\_EXPR、LOGICAL\_AND\_EXPR、ADDSUB\_EXPR、MULDIV\_EXPR 等一系列不同优先级的表达式，并用递归推导
- (7) 在 yacc.y 中创建 AST 节点时，\$\$、\$1、\$2 分别代表什么？@1.begin.line 和 @1.begin.column 的作用是什么？
- (8) 请说明你是如何将 AST 打印出来的

下面是你这八个答辩问题的汇总版，全部基于你刚才给的代码结构说的。引用放最后。

### 1. nextToken() 返回什么？怎么带类型和属性？

- 你在 scanner.h 里把签名写死成：FE::YaccParser::symbol\_type scanner::nextToken()，所以它返回的是 bison 生成的 symbol\_type。
- 在 lexer.l 里每条规则匹配到词法单元后，都会写成 return YaccParser::make\_xxx(值, loc);。这里的 make\_xxx 会把三样东西装进 symbol\_type：
  1. token 的类型（哪一个 %token，比如 IF、IDENT）；
  2. token 的属性/值（比如整数的具体数值、标识符的字符串）；
  3. token 的位置信息 loc（有行列）。
- 语法分析器拿到这个 symbol\_type 后，可以用 kind() 看类型，用 value.as<T>() 拿值，用 location.begin.line 拿行号。

### 2. 怎么处理 if / if-else 的移进-归约冲突？

- 你在 yacc.y 里把 if 分成了两条：

```
1 | IF_STMT:
2 |   IF '(' EXPR ')' STMT %prec THEN
3 |   | IF '(' EXPR ')' STMT ELSE STMT
4 | ;
```

- 问题是 if (a) if (b) s1; else s2; 到 else 时，既能“归约成没有 else 的 if”，也能“移进 else 变成有 else 的 if”，这就是移进-归约冲突。
- 你给第一条加了 %prec THEN，并且让 ELSE 的优先级高于 THEN，这样 bison 在看到 else 时会选择“移入 else”，于是 else 永远跟最近的 if 绑定，冲突就消失。

### 3. 语法树的根节点是什么类型？里面有什么？子节点是什么？

- 在 PROGRAM: STMT\_LIST { \$\$ = new Root(\$1); parser.ast = \$\$; } 里你创建了根，所以根节点类型是 FE::AST::Root。
- Root 里有一个成员：std::vector<StmtNode\*>\* stmts；，就是整段程序的顶层语句列表。

- 这些子节点都是各种语句类：`IfStmt`、`WhileStmt`、`ForStmt`、`BlockStmt`、`VarDeclStmt`、`ExprStmt`、`ReturnStmt`...它们全都继承自 `StmtNode`，而 `StmtNode` 和 `Root` 都继承自最顶层的 `Node`。

#### 4. `%token`、`%nterm` 是什么，对应文法哪里？

- `%token` 用来声明“终结符”，也就是词法能直接识别出来的符号，文法里的终结符集合  $\Sigma$ ，比如 `IF`, `ELSE`, `INT_CONST`, `IDENT`。bison 会为这些生成 `TOKEN_IF`、`make_IF(...)` 这一类东西。
- `%nterm <类型> 名字` 用来声明“非终结符”及其语义值类型，对应文法里的非终结符集合  $V \setminus \Sigma$ ，比如 `STMT_LIST`、`LOGICAL_OR_EXPR`、`ARRAY_DIMENSION_EXPR_LIST`，你在这里写了指针类型，这样语义动作里就能返回 AST 节点。

#### 5. “零次或多次”“一次或多次”怎么写的？

- 数组维度“可以没有也可以有很多”是这样做的：

```

1 | VAR_DECLARATOR : IDENT
2 |           | IDENT ARRAY_DIMENSION_EXPR_LIST
3 | ;
4 | ARRAY_DIMENSION_EXPR_LIST
5 |   : ARRAY_DIMENSION_EXPR
6 |   | ARRAY_DIMENSION_EXPR_LIST ARRAY_DIMENSION_EXPR
7 | ;

```

上面一条负责“0 次或多次”里的“0 次/有无”，下面一条负责“来了一定至少 1 次而且还能继续来”。

- 逗号分隔的声明符列表是：

```

1 | VAR_DECLARATOR_LIST
2 |   : VAR_DECLARATOR
3 |   | VAR_DECLARATOR_LIST COMMA VAR_DECLARATOR
4 | ;

```

第一条保证至少一个，第二条不断加，得到“一次或多次、逗号分隔”。

#### 6. 为什么要 `LOGICAL_OR_EXPR`、`LOGICAL_AND_EXPR`、`ADDSUB_EXPR`、`MULDIV_EXPR` 这一层一层？

- 如果你只写一条 `EXPR : EXPR op EXPR`，看到 `a + b * c` 会有移进-归约冲突，因为不知道先算加还是先算乘。
- 你现在一层一层写：加减这一层只连乘除层；乘除这一层只连一元层；逻辑或这一层只连逻辑与层。这样优先级就自然对了：乘除高于加减，`&&` 高于 `||`。
- 再加一点，逻辑运算单独一层，后面要做短路或生成跳转时能直接看到这是逻辑运算，不用在一条规则里区分所有操作符。

#### 7. 在 `yacc.y` 中创建 AST 节点时，`$$`、`$1`、`$2` 是什么？`@1.begin.line/@1.begin.column` 干嘛的？

- `$1`, `$2`, `$3`...表示这条产生式右边第 i 个符号已经算好的“语义值”（在你这一般是某个 AST 指针），比如 `IF (...) STMT` 里 `$3` 就是条件表达式的 AST，`$5` 就是 then 语句的 AST。
- `$$` 表示这条产生式左边要返回的那个值，你 new 出来的 AST 节点要放到这里，让上层能接到。
- `@1.begin.line`、`@1.begin.column` 是右边第 1 个符号在源码里的行列，你在构造 AST 时把它们传进去，这样节点就带位置了，打印和报错能指到行列。

## 8. AST 怎么打印出来的？

- 你有一个 `ASTPrinter` (在 `ast_printer.cpp`) 作为访问者，从根节点 `Root` 开始打印：先输出 `ASTree`，然后遍历 `Root` 里的每个 `StmtNode*`。
- 每个具体节点 (if、二元表达式、变量声明) 都有对应的 `visit(...)`，里面先打印当前节点的名字，再递归打印子节点，并用前缀函数打印出树形的 `|--`、``--`。
- 所以语法分析只负责“new 出 AST”，打印机负责“一层层走这棵树按结构输出”。

这八条就是你这份工程里语法分析部分要说的点。

## (1) `nextToken()` 返回什么？怎么带类型和属性？

词法分析器 (Scanner) 通过 `nextToken()` 方法返回什么类型的对象？

该对象如何携带 token 的类型和属性信息？请结合 `scanner.h` 和 `lexer.l` 中的 `YaccParser::make_XXX`

### 1. bison 决定要什么类型的 token

在 `yacc.y` 里你用了 C++ 模式的 bison，它生成的解析器类叫 `YaccParser`，并且生成了一个统一的返回类型：`YaccParser::symbol_type`。以后语法分析阶段只认这种对象。

### 2. scanner.h 把 `nextToken` 的返回类型定死了

你在 `scanner.h` 里写了：

```
1 #define YY_DECL FE::YaccParser::symbol_type FE::Scanner::nextToken()
```

这句话的意义就是：以后词法分析函数 `nextToken()` 必须返回 `YaccParser::symbol_type`。也就是说，词法阶段就要把“类型+值+位置”都准备好。

### 3. flex/lexer.l 里每条规则匹配到文本

当输入里来了标识符、数字、关键字，flex 对应的规则会被触发，动作代码开始执行。动作里能拿到的原始东西有两个：

- `yytext`：匹配到的原始字符串
- 当前的位置信息变量 `loc` (你在生成的 `lexer.cpp` 里看到，每次匹配都会 `loc.columns(yylen)`，所以行列是更新过的)

### 4. 在动作里先算“值”

比如匹配到数字，你先把 `yytext` 转成 `int/long long`：

```
1 long long v = convertToInt(yytext, '\0', isLL);
```

这一步是你自己写的。也就是说“token 的属性信息”是你在 `.l` 里主动算出来的，不是框架帮你猜的。

### 5. 在动作里选择“这是哪个 token”

你要告诉 bison：这是 IDENT，还是 INT\_CONST，还是 LPAREN。做法就是选对应的工厂函数：

```
1 return YaccParser::make_IDENT(str, loc);
2 return YaccParser::make_INT_CONST(v, loc);
3 RETT(LPAREN, loc); // 宏里也是调用 make_LPAREN
```

你选了哪个 `make_XXX`, 这个 token 的“类型”就是什么。

## 6. 把三样东西一起塞进 `symbol_type`

这些 `make_XXX(值, loc)` 是 `bison` 自动生成的函数 (`yacc.h`), 长这样:

```
1 static symbol_type make_INT_CONST(int v, location_type l) {  
2     return symbol_type(token::TOKEN_INT_CONST, std::move(v),  
3     std::move(l));  
4 }
```

可以看到:

- 第1个参数: token 的种类
- 第2个参数: 你算出来的值
- 第3个参数: 位置

所以一调用 `make_XXX(...)`, 就得到一个 **同时带类型、带值、带位置** 的 `symbol_type`。

## 7. `nextToken()` 把这个 `symbol_type` 返回给语法分析器

你的 `scanner::nextToken()` 最后就是 `return YaccParser::make_XXX(...)`, 所以它返回的正是这个装好东西的对象。

## 8. `parser` 这边怎么取

在 `parser.cpp` 里你看到这样用:

```
1 auto tok = scanner.nextToken();  
2 tok.kind();           // 知道是哪个token  
3 tok.value.as<int>(); // 取出刚才塞进去的值  
4 tok.location.begin.line; // 取位置
```

这段说明了三件事:

1. `nextToken()` 回来的东西有 `kind()`, 说明它知道“我是 INT\_CONST 还是 IDENT”;
2. 有 `location.begin.line`, 说明它里面自带位置信息;
3. 有 `value.as<int>()`, 说明它里面还能把真正的词法值取出来。

## 9. 总结成一句话

- 我们自己写的只有: 匹配→算值→选 `make_XXX`→传 `loc`
- 生成代码干的事是: 把这几样封装成 `symbol_type`
- 所以“这个对象如何携带 token 的类型和属性信息”就是: 在 `.1` 的动作里显式地把“类型”(选哪个 `make_XXX`)、“值”(算出来的 `v`)、“位置”(当前的 `loc`) 当参数传给 `make_XXX`, `bison` 生成的 `make_XXX` 把它们全装进 `symbol_type` 返回。

这样链路就闭合了, 没有隐形步骤。

## (2) 是如何处理 if 和 if-else 的移进-规约冲突的。

举一个会出现移进-规约冲突的 if-else 例子, 并解释为什么你的方案可以解决该问题。

语法分析器在读到一段输入时, 经常会遇到“我现在有两个都能走的选项”:

1. 移进 (shift) : 再多读一个词进来, 看看后面能不能拼成更长的式子。
2. 归约 (reduce) : 不读了, 把栈顶这几项合起来, 当成已经识别好的一个语法成分

举一个例子：

```
1 if (a)
2     if (b)
3         s1;
4     else
5         s2;
```

走到 `else` 时，分析器看到：

- 前面那个 `if (b) s1` 已经能归约成一条完整的“不带 `else` 的 `if`”
- 但当前又来了一个 `else`，也能“移进”进去，和最近的那个 `if` 拼成 “`if (...) ... else ...`”

两个都行，就冲突了。

怎么处理？

1. 在文件头部声明了两个优先级标记，大概像这样：

```
1 %nonassoc THEN
2 %nonassoc ELSE
```

告诉 bison ‘THEN 的优先级比 ELSE 低’。

在只有 `then` 的那条 `if` 上面加了一个优先级覆盖：

```
1 IF_STMT:
2     IF LPAREN EXPR RPAREN STMT %prec THEN {
3         $$ = new IfStmt($3, $5, nullptr, @1.begin.line, @1.begin.column);
4     }
5     | IF LPAREN EXPR RPAREN STMT ELSE STMT {
6         $$ = new IfStmt($3, $5, $7, @1.begin.line, @1.begin.column);
7     }
8 ;
```

这一行的关键就是：`%prec THEN`

而上面你又说了 `ELSE` 的优先级更高，所以当分析器看到 `else` 时，会觉得：

- “我要是现在归约成没有 `else` 的 `if`，优先级低”
- “我要把这个 `else` 移进去，去匹配‘有 `else` 的 `if`’，优先级高”  
→ 那就选移进

这样就把 `else` 强行绑到了最近的那个 `if` 上，冲突就没了。

`%prec` 的意思是：把这一条产生式的优先级，强行改成括号后面这个符号的优先级

所以一句话解释 `%prec THEN`：

“这条没有 `else` 的 `if`，请用 `THEN` 这个比较低的优先级来对比，让我在看到 `else` 的时候不要先归约，而是先把 `else` 吃进来。”

### (3) 请说明语法树的根节点是什么类型 (Root) ?

该根节点包含哪些成员变量，其子节点可能有哪些类型？这些子节点都继承自什么基类？

回答：

根节点是 `FE::AST::Root`，它里面主要就挂了一条“顶层语句列表”  
(`std::vector<FE::AST::StmtNode*>*`)。这个列表里的每一个元素都是某种具体语句节点，比如  
`if`、`while`、`for`、块、声明、返回、表达式语句，这些具体节点都继承自同一个基类  
`FE::AST::StmtNode`。

```
1 PROGRAM: STMT_LIST {
2     $$ = new Root($1);
3     parser.ast = $$;
4 }
```

## 1. 根节点是什么类型

```
1 class Root : public Node
2 {
3     private:
4         std::vector<StmtNode*>* stmts;
5     ...
6 };
```

所以根节点类型是 `FE::AST::Root`，继承自最顶层的 `Node`。`Node` 里有行号、列号、还有一个 `NodeAttr`，所有节点都会有这些公共属性。

## 2. 根节点里边装了什么成员

只有一个真正有用的成员：

```
1 std::vector<StmtNode*>* stmts;
```

也就是“程序的顶层语句列表”。构造函数也是直接收这个：

```
1 Root(std::vector<StmtNode*>* stmts) : Node(-1, -1), stmts(stmts) {}
```

说明 `parser` 在归约 `PROGRAM -> STMT_LIST` 的时候，就把那一整串语句打包成 `vector<StmtNode*>`，然后塞进 `Root`。析构函数里也是把这串语句一个个 `delete`，证明这就是它唯一关心的东西。

## 3. 其子节点可能是什么类型

`stmts` 是 `std::vector<StmtNode*>`，所以它下面只能放“语句类”节点。语句类你都放在 `stmt.h` 里了，有：

- `ExprStmt` 表达式语句
- `FuncDeclStmt` 函数声明语句
- `VarDeclStmt` 变量声明语句
- `BlockStmt` 复合语句 { ... }
- `ReturnStmt`
- `WhileStmt`

- `Ifstmt`
- `Breakstmt`
- `Continuestmt`
- `Forstmt`

这些全都是 `class XXX : public StmtNode` 这种写的，所以都能进那个 `vector<StmtNode*>`。

换句话说：`Root` 下面挂的就是“整个程序的每一条顶层语句/声明”。

#### 4. 这些子节点继承自什么？

这些具体语句都继承自 `StmtNode`，而 `StmtNode` 又继承自最顶层的 `Node`：

```
1 | class StmtNode : public Node { ... };
```

所以继承链是：

`Node` → `StmtNode` → 具体语句(`Ifstmt / Whilestmt / VarDeclstmt / ...`)

`Root` 自己也是从 `Node` 继承的，只是它专门当“根”。

#### 小结一句话

- 根： `FE::AST::Root`
- 成员： 一条 `std::vector<StmtNode*>* stmts`
- 子节点： 各种语句节点 (if、while、for、函数、变量声明、块、return...)
- 全部都继承自 `StmtNode`，而 `StmtNode` 继承自 `Node`

所以这棵树的形状是“`Root` → 一串 `StmtNode*` → 各种更细的语句/表达式节点”。

|   |                         |   |
|---|-------------------------|---|
| 1 | <code>Node</code>       | ← 顶层基类，放行号、列号、属性                                      |
| 2 | <code>└ Root</code>     | ← 专门当语法树根的节点，里面是 <code>vector&lt;StmtNode*&gt;</code> |
| 3 | <code>└ StmtNode</code> | ← “所有语句的基类”， <code>if/while/for/decl</code> 都继承它      |

- `Root : public Node`
- `StmtNode : public Node`

但 `Root` 不是 `StmtNode`， `StmtNode` 也不是 `Root`。只是兄弟节点，共同的父类都是 `Node`。

## (4) 在 bison 中，%token、%nterm 分别是什么意思，对应上下文无关文法的哪些部分？

`%token` 是终结符，`%nterm` 是非终结符。

- `%token`： 告诉 bison“这是词法能直接识别出来的符号”。对应文法里的终结符号集合  $\Sigma$ ，比如 `IF`, `ELSE`, `INT_CONST`, `IDENT`, `'+'`, `'('`。这些都是 lexer 直接 `return YaccParser::make_XXX(...)` 出来的。
- `%nterm`： 告诉 bison“这是语法归约出来的符号，而且我还想给它绑一个 C++ 类型”。对应文法里的非终结符集合  $V \setminus \Sigma$ ，比如 `EXPR`, `STMT`, `STMT_LIST`, `IF_STMT`。这些都是在产生式左边出现的、要靠别的符号推出来的。

所以一句话：

- `%token` ↔ 上下文无关文法里的终结符

- `%nterm` ↔ 上下文无关文法里的非终结符

### `%token`在哪?

在 `yacc.y` 中, 声明了 `%token IF ELSE FOR WHILE ...`, bison 帮我在 `yacc.h` 里生成了这些 `TOKEN_xxx` 和 `make_xxx`。

在 `yacc.h` 中

```

1 | static symbol_type make_IF(location_type l) { return
2 |     symbol_type(token::TOKEN_IF, std::move(l)); }
3 | static symbol_type make_ELSE(location_type l) { return
4 |     symbol_type(token::TOKEN_ELSE, std::move(l)); }
5 | static symbol_type make_WHILE(location_type l) { return
6 |     symbol_type(token::TOKEN_WHILE, std::move(l)); }
7 | ...

```

### `%nterm`在哪?

看生成的 `yacc.cpp` 里 reduce 时的类型分支

```

1 | case symbol_kind::S_STMT_LIST: // Stmt_LIST
2 |     value.move<std::vector<FE::AST::StmtNode*>>(that.value);
3 |     break;
4 | case symbol_kind::S_LOGICAL_OR_EXPR: // Logical_OR_EXPR
5 |     value.move<FE::AST::ExprNode*>(that.value);
6 |     break;
7 | case symbol_kind::S_ARRAY_DIMENSION_EXPR_LIST: // Array_Dimension_EXPR_LIST
8 |     value.move<std::vector<FE::AST::ExprNode*>>(that.value);
9 |     break;

```

这些名字 `Stmt_LIST`, `Logical_OR_EXPR`, `Array_Dimension_EXPR_LIST` 都是你在 `yacc.y` 里写的:

```

1 | %nterm <std::vector<FE::AST::StmtNode*>> Stmt_LIST
2 | %nterm <FE::AST::ExprNode*> Logical_OR_EXPR
3 | %nterm <std::vector<FE::AST::ExprNode*>> Array_Dimension_EXPR_LIST

```

这种声明生成的。生成代码里能看到它确实给这些非终结符绑定了 C++ 类型, 按你写的类型在 move。

- `%token` 对应的例子: `IF`、`ELSE`、`FOR`、`WHILE`、`IDENT`、`INT_CONST`, 在 `yacc.h` 里都变成了 `TOKEN_IF`、`TOKEN_ELSE`、`TOKEN_IDENT`。
- `%nterm` 对应的例子: `Stmt_LIST`、`IF_STMT`、`Logical_OR_EXPR`、`AddSub_EXPR`, 在 `yacc.cpp` 里都作为 `symbol_kind::S_xxx` 的非终结符出现, 而且带了你声明的指针类型。

### `TOKEN_xxx` 是什么

这是 bison 给你生成的“终结符的枚举值”。在你声明了

```
1 | %token IF ELSE WHILE IDENT INT_CONST
```

之后, 生成的 `yacc.h` 里就会出现类似:

```
1 TOKEN_IF      = 291,
2 TOKEN_ELSE    = 292,
3 TOKEN WHILE   = 293,
4 TOKEN_IDENT   = 294,
5 TOKEN_INT_CONST = 295,
6 ...
```

这些就是“这个 token 在语法分析表里的编号”。语法分析阶段靠这个编号判断你现在看到的是 IF 还是 ELSE。

### make\_xxx(...) 是什么

这是 bison 给你生成的“造一个符号对象的工厂函数”。同一组 %token 会生成一组函数，比如：

```
1 static symbol_type make_IF(location_type l) {
2     return symbol_type(token::TOKEN_IF, std::move(l));
3 }
4 static symbol_type make_INT_CONST(int v, location_type l) {
5     return symbol_type(token::TOKEN_INT_CONST, std::move(v), std::move(l));
6 }
```

你在 lexer.l 里就用这个：

```
1 return YaccParser::make_INT_CONST(v, loc);
```

意思是：我要一个“种类是 TOKEN\_INT\_CONST、值是 v、位置是 loc”的符号。TOKEN\_xxx 是编号，make\_xxx 是帮你把“编号+值+位置”打包成 symbol\_type。

### %nterm 在 yacc.cpp 里是干嘛的

你在 .y 里写了：

```
1 %nterm <FE::AST::ExprNode*> LOGICAL_OR_EXPR
2 %nterm <std::vector<FE::AST::StmtNode*>*> STMT_LIST
```

意思是：这些是“非终结符”，而且它们的语义值类型是什么。生成后的 yacc.cpp 里你就会看到对应的 case：

```
1 case symbol_kind::S_STMT_LIST:
2     value.move<std::vector<FE::AST::StmtNode*>*>(that.value);
3     break;
4 case symbol_kind::S_LOGICAL_OR_EXPR:
5     value.move<FE::AST::ExprNode*>(that.value);
6     break;
```

也就是说 %nterm 的信息被用来告诉解析器：“**当我在传递/归约这个非终结符的时候，要搬运哪一种 C++ 类型**”。这跟 %token 不同，%token 是告诉它“这是哪个终结符号”；%nterm 是告诉它“这个非终结符的语义值具体是什么类型”。

一句话对应关系：

- %token → 生成 TOKEN\_xxx 枚举 + 对应的 make\_xxx(...)

- `%nterm <T> NAME` → 告诉 bison：非终结符 `NAME` 的语义值是 `T`，在 `yacc.cpp` 里就按这个类型搬来搬去。

## (5) 你在定义文法规则时是如何处理这种“零次或多次”、“一次或多次”的语法结构的？

变量声明中的数组维度可以出现 0 次或多次（如 `int a` 或 `int a[10][20]`），变量声明符可以用逗号分隔出现一次或多次（如 `int a, b, c`）。你在定义文法规则时是如何处理这种“零次或多次”、“一次或多次”的语法结构的？

### 1. 数组维度出现0次或多次

```
1 VAR_DECLARATOR
2   : IDENT
3   | IDENT ARRAY_DIMENSION_EXPR_LIST
4   ;
```

写了两个分支：

- 只写 `IDENT` → 对应 `int a`，也就是“0 次数组维度”
- 写 `IDENT ARRAY_DIMENSION_EXPR_LIST` → 对应 `int a[10][20]`，也就是“有很多次数组维度”

“能不能没有维度”是在这一层解决的，不是在 `list` 那一层解决的。

```
1 ARRAY_DIMENSION_EXPR_LIST
2   : ARRAY_DIMENSION_EXPR
3   | ARRAY_DIMENSION_EXPR_LIST ARRAY_DIMENSION_EXPR
4   ;
```

这一层是典型的“1 次或多次”左递归：第一条先有 1 个，第二条在前面基础上继续加。所以它本身不负责 0 次，0 次是上面那个“带不带 `list` 的分支”做的。

### 2. 一次或多次、逗号分隔”是怎么做到的

```
1 VAR_DECLARATOR_LIST //一串声明符
2   : VAR_DECLARATOR //只写了一个声明就走这条
3   | VAR_DECLARATOR_LIST COMMA VAR_DECLARATOR
4   ;
```

- 第一行 `: VAR_DECLARATOR`：如果你只写了一个声明符就走这条，比如 `int a;` 里的 `a`
- 第二行 `| VAR_DECLARATOR_LIST COMMA VAR_DECLARATOR`：如果前面已经有一串了，再来一个逗号，再加一个新的声明符，比如从 `a` 变成 `a, b`，再变成 `a, b, c`
- 举例对应：

- `int a;`  
→ 刚好匹配第一行
- `int a, b;`  
→ 先匹配第一行得到一个 `VAR_DECLARATOR_LIST` 是 `a`  
→ 再用第二行：`(a)`，`(b)` 变成更长的 `VAR_DECLARATOR_LIST`

- `int a, b, c;`
- `(a)`
- `(a) , (b)`
- `(a, b) , (c)`

所以这三行的目的就是：用文法表达“至少一个，逗号隔开，多来也行”。

## (6) 请说明文法设计时为什么要使用 `LOGICAL_OR_EXPR`、`LOGICAL_AND_EXPR`、`ADD_SUB_EXPR`、`MUL_DIV_EXPR` 等一系列不同优先级的表达式，并用递归推导

请说明文法设计时为什么要使用 `LOGICAL_OR_EXPR`、`LOGICAL_AND_EXPR`、`ADD_SUB_EXPR`、`MUL_DIV_EXPR` 等一系列不同优先级的表达式，并用递归推导？而不直接使用一个 `EXPR`，然后定义成 `EXPR → EXPR ('+' | '-' | '*' | '&&' | ...) EXPR?`

回答：

我把表达式分成很多层：逻辑或、逻辑与、比较、加减、乘除、一元。每一层只处理自己那一类操作符，并且递归到更高优先级的层，这样可以：

1. 避免 `EXPR → EXPR op EXPR` 带来的移进-归约冲突；
2. 自然体现运算符优先级和结合性，比如乘除高于加减，`&&` 高于 `||`；
3. 逻辑运算单独一层，后面做短路或生成跳转代码时能直接看出来是哪一类逻辑运算，不用在一条规则里区分所有操作符。

现在的写法是这一串分层的非终结符：

- `LOGICAL_OR_EXPR`
- `LOGICAL_AND_EXPR`
- (中间还有关系、等号那一层)
- `ADD_SUB_EXPR`
- `MUL_DIV_EXPR`
- 再往下是一元、基本表达式

**不分层会产生移进-归约冲突**

如果只写：

```

1  EXPR : EXPR '+' EXPR
2      | EXPR '*' EXPR
3      | EXPR ANDAND EXPR
4      ...

```

遇到 `a + b * c`，bison 会卡住：是先把 `a + b` 归约还是先把 `b * c` 归约？两个都合法，就出现移进-归约冲突。你现在的分层写法是：

```

1 | LOGICAL_OR_EXPR

```

```

2   : LOGICAL_AND_EXPR
3   | LOGICAL_OR_EXPR "||" LOGICAL_AND_EXPR
4   ;
5
6 LOGICAL_AND_EXPR
7   : EQUALITY_EXPR
8   | LOGICAL_AND_EXPR "&&" EQUALITY_EXPR
9   ;
10
11 ADDSUB_EXPR
12   : MULDIV_EXPR
13   | ADDSUB_EXPR '+' MULDIV_EXPR
14   ;
15
16 MULDIV_EXPR
17   : UNARY_EXPR
18   | MULDIV_EXPR '*' UNARY_EXPR
19   ;

```

这种写法有两个共同点：

1. 每一层只管一种“级别”的运算符
2. 每一层的两边都递归到“更高优先级”的一层

所以它的意思是：

- 做加减的时候，两边已经是“做完乘除的东西”了
- 做乘除的时候，两边已经是“做完一元的东西”了
- 做 `||` 的时候，两边已经是“做完 `&&` 的东西”了

这样优先级就自动出来了。

### 为什么这样写优先级就对了

看这个：`a + b * c`

- `b * c` 能匹配 `MULDIV_EXPR '*' UNARY_EXPR`，所以先变成一个整体 `MULDIV_EXPR`
- 上面一层 `ADDSUB_EXPR -> ADDSUB_EXPR '+' MULDIV_EXPR` 再把 `a` 和那个整体拼起来
- 所以一定是先乘后加

如果只写一条：

```
1 | EXPR : EXPR '+' EXPR | EXPR '*' EXPR
```

到 `a + b * c` 就会卡住：先归约 `a + b` 还是先归约 `b * c`，这就是你不想要的移进/归约冲突。

### 分层就自然写出了“\* 比 + 高”“&& 比 || 高”

现在是：

- 加减那层只接 `MULDIV_EXPR`
- 乘除那层只接更底的一元

所以 `a + b * c` 只能先走到乘除那层把 `b * c` 变成一个整体，再回到加减层去接到 `a + (...)` 上。你不用额外写优先级声明，它就有优先级了。这就是你文法里多写几个名字的直接收益。

## AST 好建

在这些产生式的动作里都是直接：

```
1 | $$ = new BinaryExpr($1, OP, $3, ...);
```

如果是单一的 `EXPR -> EXPR op EXPR`，你要在一条动作里区分是哪一类操作符，再决定塞到哪一层 AST 里面，代码会又长又容易错；现在你一层一个节点类型，哪层建哪层的二元表达式，很直观。

### 短路运算要单独一层

逻辑或、逻辑与 (`|| / &&`) 本来就跟算术运算不一样，你后面跑语义或生成代码时要做短路，所以它们单独是 `LOGICAL_OR_EXPR`、`LOGICAL_AND_EXPR` 两层，这正是你文件里的名字。要是你全塞到一条 `EXPR` 里，后面区分哪种运算做短路就麻烦了

逻辑运算跟算术运算不一样的点在这里：需要短路。

- `A && B`：如果 A 是假，B 根本不用算
- `A || B`：如果 A 是真，B 根本不用算

为了做到“B 根本不用算”，很多编译器在生成代码时要把这两种运算符当成“控制流”来翻译，比如生成跳转。那就必须能在语法树上清楚地看到“这是一个逻辑或”“这是一个逻辑与”，不能混在一堆加减乘除里面。

所以我们写为

```
1 | LOGICAL_OR_EXPR -> LOGICAL_AND_EXPR
2 |           | LOGICAL_OR_EXPR "||" LOGICAL_AND_EXPR
3 |
4 | LOGICAL_AND_EXPR -> EQUALITY_EXPR
5 |           | LOGICAL_AND_EXPR "&&" EQUALITY_EXPR
```

好处有两个：

1. 优先级对：`&&` 比 `||` 高，因为 `||` 的两边必须先变成 `LOGICAL_AND_EXPR`
2. 语义好写：你在动作里一看这是 `LOGICAL_OR_EXPR` 分支就知道“这里是或，要短路”；看见 `LOGICAL_AND_EXPR` 分支就知道“这里是与，要短路”

如果你全写成一条：

```
1 | EXPR : EXPR '+' EXPR
2 |   | EXPR '*' EXPR
3 |   | EXPR "&&" EXPR
4 |   | EXPR "||" EXPR
```

那你在语义动作里只看到“这是 EXPR 和 EXPR 做了个运算”，根本分不出这是算术还是逻辑，还得再去看操作符再分类，一条动作要写一堆 if，很难看，也容易错。

### 所以写法的核心思路是

- “一层只处理一种优先级的运算符”
- “这一层的两边都递归到更高优先级的那层”
- “真正允许出现多次运算的地方用左递归”

# (7)在 yacc.y 中创建 AST 节点时，\$\$、\$1、\$2 分别代表什么？@1.begin.line 和 @1.begin.column 的作用是什么？

回答：

在 yacc.y 的语义动作里，

\$\$ 表示当前产生式左部非终结符的语义值，用来保存新建的 AST 节点；\$1、\$2、\$3...表示当前产生式右部第 1、2、3...个符号已经计算好的语义值，用来当作子节点传给构造函数。

@i.begin.line 和 @i.begin.column 表示第 i 个符号在源程序中的起始行、起始列，我在构造 AST 节点时把这两个位置传进去，这样 AST 节点就能记录源码位置，便于后续打印和错误定位。

## \$\$、\$1、\$2 是什么

```
1 IF_STMT:  
2     IF LPAREN EXPR RPAREN STMT %prec THEN {  
3         $$ = new IfStmt($3, $5, nullptr, @1.begin.line, @1.begin.column);  
4     }  
5     | IF LPAREN EXPR RPAREN STMT ELSE STMT {  
6         $$ = new IfStmt($3, $5, $7, @1.begin.line, @1.begin.column);  
7     }  
8 ;
```

- \$\$：这条产生式左边那个符号的语义值，这里左边是 IF\_STMT，所以 \$\$ 就是“我要返回的这个 if 语句的 AST 节点”。你最后要把一个 StmtNode\*（实际上是 IfStmt\*）放进 \$\$ 里。
- \$1：右边第 1 个符号的语义值，这里第 1 个是 IF，但关键字一般没语义值，所以你没用它。
- \$3：右边第 3 个符号，这里是 EXPR，也就是 if 的条件，所以你写成了 new IfStmt(\$3, ...)。
- \$5：右边第 5 个符号，这里是 STMT，也就是 then 分支，所以是第二个参数。
- 在第二条规则里你又用了 \$7，因为第 7 个符号是 STMT，也就是 else 分支。

同样，在开头你还有一条：

```
1 PROGRAM: STMT_LIST {  
2     $$ = new Root($1);  
3     parser.ast = $$;  
4 }
```

这里的含义一样：

- 左边是 PROGRAM，所以 \$\$ 是整个程序的根节点
- 右边第 1 个是 STMT\_LIST，所以 \$1 是“很多语句组成的 vector”
- 你就拿 \$1 去构造 Root，然后放进 \$\$

所以一句话：\$\$=要交给上层的结果，\$i=右边第 i 个符号已经算好的结果。

`$1` 是“右边第1个东西已经算出来的结果”，`$2` 是第2个，`$$` 是“**这条规则最后要交出去的结果**”。“语义值”就是这个“结果”，在你这就是各种 AST 节点指针或列表。

## 1. 为什么会有“语义值”

语法分析不只是认单词顺序，还要“带着东西往上走”。

比如你识别了一个表达式，就应该顺手建一个 `ExprNode*`；识别一个 `if`，就该建一个 `Ifstmt*`。这个“识别完后带着走的东西”就叫**语义值**。

- 对标识符来说，语义值可能是一个 `std::string`
- 对表达式来说，语义值是一个 `ExprNode*`
- 对语句来说，语义值是一个 `StmtNode*`

所以“语义值”=“这个符号真正代表的那份数据”。

### 用你的 if 这条看

你写的是：

```
1 IF_STMT:  
2   IF LPAREN EXPR RPAREN STMT %prec THEN {  
3     $$ = new Ifstmt($3, $5, nullptr, @1.begin.line, @1.begin.column);  
4   }  
5   | IF LPAREN EXPR RPAREN STMT ELSE STMT {  
6     $$ = new Ifstmt($3, $5, $7, @1.begin.line, @1.begin.column);  
7   }  
8   ;
```

这一条右边有这些符号：

1. `IF`
2. `LPAREN`
3. `EXPR`
4. `RPAREN`
5. `STMT`
6. (第二种写法还有) `ELSE`
7. `STMT`

bison 会给**每一个**符号挂一份“语义值”。于是你就可以写：

- `$3` → 第3个符号的语义值 → 就是那个表达式的 AST，类型是 `ExprNode*`
- `$5` → 第5个符号的语义值 → 就是 `then` 那条语句的 AST，类型是 `StmtNode*`
- `$7` → `else` 那条语句的 AST

现在你要造一条完整的 `if` 语句，就用这几份现成的值去 `new`：

```
1 $$ = new Ifstmt($3, $5, $7, ...);
```

这里的 `$$` 就是**这条规则左边的那个符号 `IF_STMT` 的语义值**。也就是说：

“我这条规则识别出了一个 `if` 语句，它的值就是我刚 `new` 出来的这个 `Ifstmt*`。”

所以：

- `$3` = 子节点
- `$5` = 子节点
- `$$` = 父节点

就是往上拼树。

---

## 再看你最上面的程序那条

```

1 | PROGRAM: STMT_LIST {
2 |     $$ = new Root($1);
3 |     parser.ast = $$;
4 |

```

- 右边只有一个符号 `STMT_LIST`, 所以它的语义值就是 `$1`
- `$1` 是“一串语句的 vector”
- 你用它来 `new` 一个根节点
- 把 `new` 出来的根节点放进 `$$`
- `$$` 就成了 `PROGRAM` 这个符号的语义值

这说明：`$1` 是右边第1个东西的结果, `$$` 是这条产生式左边整个东西的结果。

## 再说一遍：语义值是什么

在你这份代码里，语义值就是这些东西里的一个：

- `ExprNode*`
- `StmtNode*`
- `std::vector<StmtNode*>*`
- `std::vector<ExprNode*>*`

也就是你在 `%nterm <...>` 里声明的那些类型。bison 把它们挂到 `$1`、`$2` 上，你再从里面取出来拼 AST。没有神秘含义，就是“我这个符号要带的数据”。

在 `stmt.h` 中

```

1 | // if 语句, 如 if (cond) { ... } else { ... }
2 | // 其中的 cond 是条件表达式, thenstmt 是条件为真时执行的语句, elsestmt 是条件为假
3 | // 时执行的语句 (可选)
4 | class Ifstmt : public StmtNode
5 | {
6 |     public:
7 |         ExprNode* cond;
8 |         StmtNode* thenstmt;
9 |         StmtNode* elsestmt;
10 |        ...
11 | };

```

这说明 `Ifstmt` 是你定义好的一个语句节点类型，表示一条 if 语句。

在 `yacc.y` 里写

```
1 | $$ = new Ifstmt($3, $5, $7, @1.begin.line, @1.begin.column);
```

这里的意思是：

- 用刚才语法分析出来的条件 `$3`、`then` 分支 `$5`、`else` 分支 `$7`
- 调用 `Ifstmt` 这个构造函数
- 在堆上创建一个 `Ifstmt` 对象
- 把这个对象的指针作为这条产生式的语义值，放到 `$$` 里

为什么要 `new`？

因为 AST 要在整个编译期间都活着，不能出了这个语义动作的作用域就没了，所以用 `new` 分配在堆上，让后面别的节点也能指到它，最后统一 `delete`。

所以一句话：

`new Ifstmt(...)` = “按 if 的结构，创建一个 if 语句的 AST 节点对象，返回它的指针”。

---

`@1.begin.line` 和 `@1.begin.column` 干嘛的

bison 会跟着每个符号一起传“位置信息”(location)，所以你可以写 `@1`、`@2` 去拿第 `i` 个符号在源码里的位置

```
1 | new Ifstmt($3, $5, nullptr, @1.begin.line, @1.begin.column);
```

解释：

- `@1` 指的是右边第 1 个符号，也就是那个 `IF` 关键字
- `@1.begin.line` 就是这个 `if` 在源文件的行号
- `@1.begin.column` 就是这个 `if` 在这一行的列号

你把这两个传进 `Ifstmt(...)`，就是让 AST 里的这个 if 节点记住“我是在第几行第几列出现的”。后面打印 AST 或报错时就能说“line 12, column 3: ...”。

别的规则也是同理，比如：

```
1 | IF LPAREN EXPR RPAREN STMT ELSE STMT {  
2 |     $$ = new Ifstmt($3, $5, $7, @1.begin.line, @1.begin.column);  
3 | }
```

还是用第 1 个符号的行列，因为 if 语句应该记在 if 开头的位置，不是记在 else 那里。

## (8) 请说明你是如何将 AST 打印出来的

回答：

我这份代码里，语法分析完成后会得到一棵以 `FE::AST::Root` 为根的 AST。

框架提供了 `FE::AST::ASTPrinter` 这个访问者类，它实现了对所有节点类型（表达式、语句、声明）的 `visit(...)` 函数。

在打印时，从 `visit(Root)` 开始，先输出“ASTree”，然后对根节点下的每个语句调用 `apply` 递归打印。

每个 `visit` 里都会先输出当前节点的名字，再用 `withChild(...)` 配合一个前缀栈 `lastStack` 打印 `|--`、``--` 这样的树形前缀，最后递归到子节点。

整个过程就是典型的访问者模式实现的 AST pretty-print。

打印入口：

在 `ast/printer/ast_printer.cpp` 中 `ASTPrinter::visit(Root& node, std::ostream* os)` 是入口。它先打印一行标题，再把根下面的每个语句打印出来。代码：

```
1 void ASTPrinter::visit(Root& node, std::ostream* os)
2 {
3     lastStack.clear();
4     *os << "ASTree\n";
5
6     auto* stmts = node.getStmts();
7     if (!stmts) return;
8
9     size_t cnt = stmts->size();
10    for (size_t i = 0; i < cnt; ++i)
11    {
12        if (!(*stmts)[i]) continue;
13        withChild(i + 1 == cnt, [&]() { apply(*this, *(*stmts)[i], os); });
14    }
15 }
16
```

怎么“往下走一层”

看它的工具函数：

```
1 void ASTPrinter::emitPrefix(std::ostream& os) const { ... } // 打印前面的 "|--"
2 或 "`-- "
3 void ASTPrinter::withChild(bool isLast, const std::function<void()>& fn)
4 {
5     pushLast(isLast);
6     fn();
7     popLast();
8 }
```

`withChild(...)` 就是“我要去打印一个子节点了，先把当前是不是最后一个孩子记下来，打印完再弹出来”。这样就能打印出像：

```
1 | ASTree
2 | |-- VarDeclStmt
3 |   `-- FuncDecl ...
```

这种树形结构

各种节点的 visit 里只管打印自己+递归孩子

举例子：

### 1. 表达式节点

在 expr\_printer.cpp 中

```
1 | void ASTPrinter::visit(BinaryExpr& node, std::ostream* os)
2 | {
3 |     emitHeader(*os, std::string("BinaryExpr ") + toString(node.op));
4 |     if (node.lhs) withChild(false, [&]() { apply(*this, *node.lhs, os); });
5 |     if (node.rhs) withChild(true, [&]() { apply(*this, *node.rhs, os); });
6 | }
```

先打印一行说“这是个二元表达式 + 号/ \* 号”，然后把左右子树递归打印。

### 2. if 语句

```
1 | void ASTPrinter::visit(IfStmt& node, std::ostream* os)
2 | {
3 |     emitHeader(*os, "Ifstmt");
4 |     ... // 打印 Condition:
5 |     ... // 打印 Then:
6 |     ... // 打印 Else:
7 | }
```

它不重新造树，只是把你再 yacc 里 new 出来的 Ifstmt 里的 cond/then/else 再走一遍。

### 3. 变量声明

```
1 | void ASTPrinter::visit(VarDeclaration& node, std::ostream* os)
2 | {
3 |     emitHeader(*os, "VarDeclaration, BaseType: ...");
4 |     ... // 遍历所有 declarator
5 | }
```

一层套一层打印