

实现语法分析器

杨侯哲 李煦阳

杨科迪 孙一丁

韩佳迅 朱璟钰

华志远

唐显达

2020 年 11 月—2025 年 10 月

目录

1 实验描述	3
2 实验要求	3
3 实验流程	4
3.1 类型系统	4
3.2 符号表	4
3.3 抽象语法树	5
3.4 语法分析与语法树的创建	6
3.5 语法规则定义	8
3.6 实验框架	9
3.6.1 目录结构	9
3.6.2 实验效果	10
3.6.3 编译与运行	11
3.7 注意事项	12
3.8 线下检查提问示例	12

1 实验描述

如果你还记得本学期初探索编译器的时候，我们曾使用`-fdump-tree-original-raw`获得 gcc 构建的语法树。对于`void main() {}`，它的输出如下。

```
1  ;; Function main (null)
2  ;; enabled by -tree-original
3
4  @1      bind_expr      type: @2      body: @3
5  @2      void_type      name: @4      algn: 8
6  @3      statement_list
7  @4      type_decl      name: @5      type: @2
8  @5      identifier_node strg: void    lngt: 4
```

我们知道，输出的每一行可以理解为语法树上的一个结点。每一个结点有其自身的类型、属性，以及数个子结点。本次作业便是要求构建这样一棵树并输出。

可以想象，gcc 采取了更复杂的语法定义去构建这棵树，并使用一些压缩算法处理这棵树。本次实验，我们只要求以最简洁最直观的方式将这棵树构建出来、展示结果。

构建出树后，我们之后的所有操作，比如树上各结点信息的获取与流动、类型检查、翻译至中间代码，都可以理解为对该树进行一次遍历。同时值得一提的是，若一些操作需要考虑语法，比如构建**作用域树**，那么通过语法树上一次遍历，便可以很容易完成。

2 实验要求

1. 根据 SysY 文法定义，借助 Yacc 工具实现语法分析器：
 - 语法树数据结构的设计：结点类型的设计，不同类型的节点应保存的信息。
 - 扩展上下文无关文法，设计翻译模式。
 - 设计 Yacc 程序，实现能构造语法树的分析器。
 - 以文本方式输出语法树结构，验证语法分析器实现的正确性。
2. 无需撰写完整研究报告，但需要在雨课堂上提交本次实验的 gitlab 链接。
3. 上机课时，以小组为单位，向助教讲解程序。

3 实验流程

3.1 类型系统

变量的**类型**，仿佛只是简单作为变量结点的一个属性而已，但仔细考虑会发现它可以极其复杂。直观上，我们有 `struct`、`union` 构造复合类型，函数本身作为变量，它也有其自身的特殊类型。**类型系统**是编程语言理论的一个重要一部分。很有趣的一点是，类型系统与数理逻辑紧密相关，类型的检查可以视为定理的证明，这一关系被称为**Curry-Howard Correspondence**。比如 `struct` 可以视为**合取**，`union` 可以视为析取，函数的输入类型与输出类型可以视为蕴含¹。为了进行静态类型检查，你需要根据你的目标语言设计好与你需要的类型系统有关的数据结构。你可能还要考虑如何插入“类型转换”。

在实验框架中，类型系统的定义位于 `interfaces/frontend/ast/ast_defs.h` 中，提供了一些基础类型定义（其中的指针类型用于处理数组，如果你不打算实现数组特效，可以忽略它）。对 `const` 修饰符的处理则交给了 AST 的变量声明语句节点，你可以在 `frontend/ast/decl.h` 的 `VarDeclaration` 类中找到 `isConstDecl` 属性。

3.2 符号表

符号表是编译器用于保存源程序符号信息的数据结构，这些信息在词法分析、语法分析阶段被存入符号表中，最终用于生成中间代码和目标代码。符号表条目可以包含标识符的词素、类型、作用域、行号等信息。

符号表主要用于作用域的管理，我们为每个语句块创建一个符号表，块中声明的每一个变量都在该符号表中对应着一个符号表条目。在词法分析阶段，我们只能识别出标识符，不能区分这个标识符是用于声明还是使用。而在语法分析阶段，我们能清楚的知道一个程序的语法结构，如果该标识符用于声明，那么语法分析器将创建相应的符号表条目，并将该条目存入当前作用域对应的符号表中，如果是使用该标识符，将从当前作用域对应的符号表开始沿着符号表链搜索符号表项。

但综合考虑后，实验框架将符号表的实现放在了语义分析阶段。在本次实验中，你只需要理解符号表的基本概念即可，具体实现将在后续实验中进行讲解。在 AST 节点中，会使用符号表项来保存一些信息，具体见 AST 节点的定义。

¹这一部分是私货。

3.3 抽象语法树

语法分析的目的是构建出一棵抽象语法树（AST），因此我们需要设计语法树的结点。结点分为许多类，除了一些共用属性外，不同类结点有着各自的属性、各自的子树结构、各自的函数实现。我们可以简单用 struct 去涵盖所有需要的内容，也可以设计复杂的继承结构。结点的类型大体上可以分为表达式和语句，每种类型又可以分为许多子类型，如表达式结点可以分为词法分析得到的叶结点、二元运算表达式的结点等；语句还可以分为 if 语句、while 语句和块语句等。

以框架代码为例：

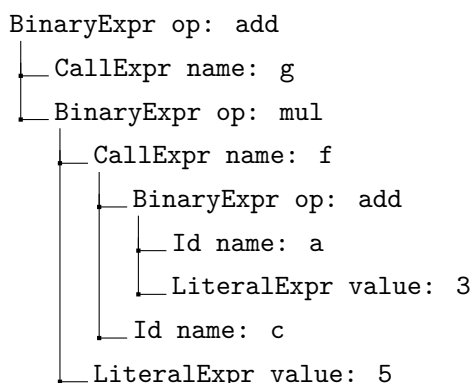
```
class Node
{
    public:
        int      line_num;
        int      col_num;
        NodeAttr attr;
};

class ExprNode : public Node
{
    public:
        size_t trueTar;
        size_t falseTar;
};

class CallExpr : public ExprNode
{
    public:
        Entry*      func;
        std::vector<ExprNode*>* args;
};

class BinaryExpr : public ExprNode
{
    public:
        Operator op;
        ExprNode* lhs;
        ExprNode* rhs;
};
```

Node 为 AST 结点的抽象基类, ExprNode 为表达式结点的抽象基类, 从 ExprNode 中派生出 CallExpr 与 BinaryExpr。这样, 我们就可以使用下面这个树形结构来表达 $g() + f(a + 3, c) * 5$:



目前实验框架中已经给出了一些常用的语法树结点的定义, 足以覆盖 SysY 语言的特性。当然, 如有需要, 你也可以自行添加更多的结点类型。但需要注意在添加后, 向 interfaces/frontend/ast/ast_visitor.h 中添加相应的类型声明。

3.4 语法分析与语法树的创建

词法分析得到的, 实质是语法树的叶子结点的属性值, 语法树所有结点均由语法分析器创建。在自底向上构建语法树时 (与预测分析法相对), 我们使用孩子结点构造父结点。在 yacc 每次确定一个产生式发生归约时, 我们会创建出父结点、根据子结点正确设置父结点的属性、记录继承关系。下面是语法分析中一个特殊的例子:

FUNC_CALL_EXPR:

```

IDENT '(' ')' {
    Entry* entry = Entry::getEntry($1);
    $$ = new CallExpr(entry, nullptr, @1.begin.line, @1.begin.column);
}
| IDENT '(' EXPR_LIST ')' {
    Entry* entry = Entry::getEntry($1);
    $$ = new CallExpr(entry, $3, @1.begin.line, @1.begin.column);
}
;
  
```

这里实际定义了两个产生式, 分别对应函数调用时无参数和有参数的两种情况。在每个产生式归约时, 我们都创建了一个 CallExpr 结点, 并根据不同的情况传入不同的参数列表。在 bison 中, \$\$ 表示当前归约产生的非终结符的属性, \$1、\$2 等表示产生式右侧第 1、第 2 个符号的属性, @1、@2 等表示产生式右侧第 1、第 2 个符号在源代码中的位置信息。具体到此处第二个产生式中, \$1 表示 IDENT 的属性, 即识别到的函数名字符串; \$3 表示 EXPR_LIST 的属性, 即参数表达式列表。

这里需要额外提醒一下关于 if 语句的二义性问题。当文法中同时存在两种 if 语句时:

匹配的 if if (cond) stmt else stmt

不匹配的 if if (cond) stmt

对于像 if (c1) if (c2) s1 else s2; 这样的嵌套结构, 语法分析器在解析时会遇到歧义。

这个歧义具体表现为“移入/归约冲突”。当分析器已经处理完 `if (c1) if (c2) s1` 部分, 并且向前看到下一个符号是 `else` 时, 它面临两种选择:

移入 : 将 `else` 符号移入分析栈。这个选择意味着 `else` 将与最近的、尚未匹配的 `if` (即 `if (c2)`) 进行匹配。这会产生如下的语法树结构, 也是大多数语言所期望的:

```

IfStmt
├── cond: c1
├── thenStmt: IfStmt
│   ├── cond: c2
│   ├── thenStmt: s1
│   └── elseStmt: s2
└── elseStmt: nullptr

```

归约 : 分析器发现分析栈的栈顶内容 (即 `if (c2) s1`) 已经可以匹配“不匹配的 `if`”规则 (`if (cond) stmt`), 于是将它归约为一个 `stmt`。如果执行归约, 那么这个新生成的 `stmt` 就结束了内部的 `if` 语句, 随后的 `else` 将只能与外部的 `if` (即 `if (c1)`) 匹配。这会产生如下的语法树结构:

```

IfStmt
├── cond: c1
├── thenStmt: IfStmt
│   ├── cond: c2
│   ├── thenStmt: s1
│   └── elseStmt: nullptr
└── elseStmt: s2

```

由于分析器在同一个状态下, 面对同一个输入符号 (`else`) 时, 既可以“移入”又可以“归约”, 就产生了移入/归约冲突。包括 `SysY` 在内的大多数语言规定应优先选择“移入”, 即 `else` 与最近的 `if` 匹配。你的任务就是通过设计语法规则来消除这种歧义, 引导分析器做出正确的选择。

对于该问题, 通过语法结构来解决会使文法变得复杂。此时, `bison` 提供了更直接的方式: 使用 `%left`, `%right`, `%nonassoc` 和 `%precedence` 指令来明确解决冲突。当 `bison` 遇到移入/归约冲突时, 它会比较向前看符号的优先级和待归约产生式的优先级:

- 如果向前看符号的优先级更高, 则选择移入。
- 如果产生式的优先级更高, 则选择归约。
- 如果优先级相同, 则根据结合性 (`%left` 或 `%right`) 来决定。

对于该问题, 我们希望移入 `else` 的动作, 其优先级高于归约不带 `else` 的 `if` 语句。但问题是, 产生式 `if (cond) stmt` 的结尾没有一个像 `+` 或 `*` 那样可以代表其优先级的终结符。

为了解决这个问题, `bison` 提供了一个特殊的指令 `%prec`, 它允许我们为一个产生式手动指定一个虚拟的终结符来决定其优先级。你可以定义一个不存在于词法分析中的虚拟符号 (例如 `THEN`), 并将其优先级设置得比 `ELSE` 更低, 然后使用 `%prec THEN` 将这个较低的优先级赋予不带 `else` 的 `if` 产生式。这样, `bison` 在面临冲突时, 就会因为 `ELSE` 的优先级更高而选择移入, 从而解决了二义性。

3.5 语法规则定义

本次实验中并不强行要求你使用某一套语法定义，你可以根据自己的理解设计一套适合自己的语法定义，也可以参考 SysY 语言文法定义（见 doc/SysY2022 语言定义-V1.pdf）。但无论如何，你都需要确保你的语法定义能够覆盖 SysY 语言的所有特性，并且能够正确处理二义性问题。考虑到实验框架编写时并未严格按照 SysY 语言文法定义，因此此处给出实验框架中使用的语法定义，供参考：

```

PROGRAM      -> STMT_LIST [ END ]
STMT_LIST    -> STMT { STMT }
STMT          -> EXPR_STMT | VAR_DECL_STMT | BLOCK_STMT | FUNC_DECL_STMT
               | RETURN_STMT | WHILE_STMT | FOR_STMT | IF_STMT
               | BREAK_STMT | CONTINUE_STMT | ';' | SLASH_COMMENT

BLOCK_STMT    -> '{' [ STMT_LIST ] '}'
FUNC_BODY     -> '{' [ STMT_LIST ] '}'          // 该定义实际上是冗余的，你也可以直接使用 BLOCK_STMT
VAR_DECL_STMT -> VAR_DECLARATION ';'          // 此处定义冗余的 VAR_DECLARATION 主要用于 for 语句的 init
VAR_DECLARATION -> [ CONST ] TYPE VAR_DECLARATOR { ',' VAR_DECLARATOR }

VAR_DECLARATOR -> LEFT_VAL_EXPR [ '=' INITIALIZER ]
               | IDENT '[' ']' [ ARRAY_DIMENSION_EXPR_LIST ] [ '=' INITIALIZER ]
               | IDENT ARRAY_DIMENSION_EXPR_LIST [ '=' INITIALIZER ]

INITIALIZER   -> NOCOMMA_EXPR | '{' [ INITIALIZER { ',' INITIALIZER } ] '}'

FUNC_DECL_STMT -> TYPE IDENT '(' [ PARAM_DECLARATOR { ',' PARAM_DECLARATOR } ] ')' FUNC_BODY
PARAM_DECLARATOR -> TYPE IDENT [ '[' ']' [ ARRAY_DIMENSION_EXPR_LIST ] ]
                  | TYPE IDENT ARRAY_DIMENSION_EXPR_LIST

RETURN_STMT   -> RETURN [ EXPR ] ';'
WHILE_STMT    -> WHILE '(' EXPR ')' STMT
FOR_STMT      -> FOR '(' (VAR_DECLARATION | EXPR) ';' EXPR ';' EXPR ')' STMT
IF_STMT       -> IF '(' EXPR ')' STMT [ ELSE STMT ]

EXPR_STMT     -> EXPR ';'
EXPR          -> NOCOMMA_EXPR { ',' NOCOMMA_EXPR }
NOCOMMA_EXPR  -> ASSIGN_EXPR | LOGICAL_OR_EXPR    // 此项用于将单表达式与可能的逗号表达式区分开来
ASSIGN_EXPR   -> LEFT_VAL_EXPR '=' NOCOMMA_EXPR

LOGICAL_OR_EXPR -> LOGICAL_AND_EXPR { '||' LOGICAL_AND_EXPR }
LOGICAL_AND_EXPR -> EQUALITY_EXPR { '&&' EQUALITY_EXPR }
EQUALITY_EXPR  -> RELATIONAL_EXPR { ('==' | '!=') RELATIONAL_EXPR }
RELATIONAL_EXPR -> ADDSUB_EXPR { ('>' | '>=' | '<' | '<=') ADDSUB_EXPR }
ADDSUB_EXPR    -> MULDIV_EXPR { ('+' | '-') MULDIV_EXPR }
MULDIV_EXPR    -> UNARY_EXPR { ('*' | '/' | '%') UNARY_EXPR }

UNARY_EXPR     -> BASIC_EXPR | UNARY_OP UNARY_EXPR
BASIC_EXPR     -> LITERAL_EXPR | LEFT_VAL_EXPR | '(' EXPR ')' | FUNC_CALL_EXPR
FUNC_CALL_EXPR -> IDENT '(' [ EXPR { ',' EXPR } ] ')'
```

```

LEFT_VAL_EXPR  -> IDENT [ ARRAY_DIMENSION_EXPR_LIST ]
ARRAY_DIMENSION_EXPR_LIST -> '[' NOCOMMA_EXPR ']' { '[' NOCOMMA_EXPR ']' }

LITERAL_EXPR   -> INT_CONST | LL_CONST | FLOAT_CONST
TYPE           -> INT | FLOAT | VOID
UNARY_OP       -> '+' | '-' | '!'

```

其中，中括号在此处表示可选项，大括号表示可重复 0 次或多次。上述语法定义覆盖了 SysY 语言的所有特性并有所扩展，你可以选择参考它来完成本次实验，也可以自行设计一套语法定义。

3.6 实验框架

3.6.1 目录结构

本次实验主要涉及到以下文件：

```

framework
├── frontend
│   ├── parser
│   │   └── yacc.y ..... 语法分析器 Bison 定义
│   ├── ast ..... AST 节点定义
│   │   ├── ast.{h|cpp}
│   │   ├── expr.{h|cpp}
│   │   ├── stmt.{h|cpp}
│   │   ├── decl.{h|cpp}
│   │   ├── visitor
│   │   └── printer ..... AST 打印机
│   ├── symbol ..... 符号表
│   │   └── symbol_entry.{h|cpp} ..... 符号表项定义
│   └── interfaces ..... 接口定义
│       ├── frontend
│       │   ├── ast
│       │   │   └── ast_defs.h ..... AST 类型定义
│       │   └── symbol
│       │       └── symbol_entry.h ..... 符号表项
│       └── doc
│           └── SysY2022 语言定义-V1.pdf
└── main.cpp

```

在本次实验中，你主要需要为 **yacc.y** 编写语法规则，将词法分析实验中得到的 token 逐步规约为 AST。抽象语法树的结点类型定义详见 `frontend/ast` 目录。视你的需求，可以自行添加更多结点。框架提供了简易的类型系统，详见 `interfaces/frontend/ast/ast_defs.h`，视你的需求，可自行扩展。

语法树的打印方法已经提供，详见 `frontend/ast/visitor/printer` 目录，可借此对访问者模式先做了解，框架代码后续的类型检查、代码生成等工作也依赖于此。视你的需求，可自行修改打印逻辑。

3.6.2 实验效果

以下面的 SysY 语言源程序为例：

```
const int len = 10;
int arr[len] = {1, 2};

int main()
{
    if (arr[0] < arr[1]) putint(arr[1]);
    else putint(arr[0]);
    return 0;
}
```

注意：在语法分析阶段，我们只需要构建出语法树即可，不需要进行符号表的处理和语义检查。因此在本阶段的输出中，部分信息可能尚未确定，如数组类型我们只要求确认到基础类型和维度信息即可，更详细的类型处理我们可以选择交由类型检查阶段完成。

完成语法分析器后，使用命令 `bin/compiler -parser -o output.txt input.sy` 如果你已经正确实现要求的功能，可得到类似下面的输出

```
ASTree
|-- VarDeclStmt
|   |-- VarDeclaration, BaseType: int
|   |   |-- VarDeclarator
|   |   |   |-- Var:
|   |   |   |   |-- LeftValueExpr len
|   |   |   |   |-- Init:
|   |   |   |   |   |-- Initializer
|   |   |   |   |   |   |-- literal int: 10
|   |-- VarDeclStmt
|   |   |-- VarDeclaration, BaseType: int
|   |   |   |-- VarDeclarator
|   |   |   |   |-- Var:
|   |   |   |   |   |-- LeftValueExpr arr
|   |   |   |   |   |   |-- LeftValueExpr len
|   |   |   |   |-- Init:
|   |   |   |   |   |-- InitializerList
|   |   |   |   |   |   |-- Initializer
|   |   |   |   |   |   |   |-- literal int: 1
|   |   |   |   |   |   |   |-- Initializer
|   |   |   |   |   |   |   |   |-- literal int: 2
|   |-- FuncDecl main() -> int, line: 4
|   |   |-- BlockStmt, line: 5
|   |   |   |-- IfStmt
|   |   |   |   |-- Condition:
|   |   |   |   |   |-- BinaryExpr <
|   |   |   |   |   |   |-- LeftValueExpr arr
```

```

| | | | `-- literal int: 0
| | | | `-- LeftValueExpr arr
| | | | `-- literal int: 1
| | |-- Then:
| | | | `-- ExprStmt line: 6
| | | | `-- Call putint
| | | | `-- Arg 0:
| | | | | `-- LeftValueExpr arr
| | | | | `-- literal int: 1
| | `-- Else:
| | | | `-- ExprStmt line: 7
| | | | `-- Call putint
| | | | `-- Arg 0:
| | | | | `-- LeftValueExpr arr
| | | | | `-- literal int: 0
|-- ReturnStmt
    `-- literal int: 0

```

在 `testcase/parser/` 目录下提供了完整的测试用例及其预期输出，你可以参考这些用例来验证你的实现。语法分析阶段暂不提供自动化测试脚本，需要同学们自行使用检查输出结果的合理性。不需要和给出的预期输出完全一致，只需要能合理的反映语法结构即可。

3.6.3 编译与运行

编译项目 实验框架的 `Makefile` 已配置好各文件的依赖关系。当你修改了 `yacc.y` 文件后，只需执行：

```
make
```

即可完成项目的构建。`Makefile` 会自动检测修改并重新生成相应的词法/语法分析器代码。

如果实验完成正确，执行 `make` 时将不再出现 Bison 产生的警告信息。

理解 Makefile 的依赖规则 为帮助同学们更好地使用构建系统，这里简要说明 `Makefile` 中的依赖机制。

在框架的 `Makefile` 中有如下规则：

```

$(BISON_H) $(BISON_C) $(LOC_H): $(BISON_SRC)
    @bison -d --language=c++ --defines=$(BISON_H) -o $(BISON_C) $(BISON_SRC)
...

```

该规则声明了 `yacc.h`、`yacc.cpp`、`location.hh` 三个生成文件依赖于 `yacc.y` 源文件。如果源文件没有被修改（通过文件修改时间戳判断），那么 `make` 工具会认为无需重新执行依赖于它的目标。

如果在特定条件下你通过 `make` 生成了错误的，或者与框架期望接口不符的代码（如在 bison 3.5 版本中，生成的 token 类名为 `token::yytokentype`，而非 `parser` 中期望的 `symbol_kind`），那么后续编译就可能报错。且在你修改 `BISON_SRC` 之前，`BISON_H`、`BISON_C`、`LOC_H` 都不会被更新，因此 `make` 会一直报错。

解决方法是使用 `Makefile` 提供的清理目标：

```
# 清理生成的词法/语法分析器文件，强制重新生成
make clean-lexer
# 清理所有编译产物，强制从头开始构建整个项目
make clean
```

如果你在后续实验中遇到了一些奇怪的问题，可以先试试清理后重新编译来排除过时文件的影响。

3.7 注意事项

1. 虽然本次实验只要求能实现上机大作业总体要求中的基本要求（除数组，浮点数，字符常量外的 SysY 特性）的语法分析即可获得满分，但是如果你想做进阶语法要求，请尽量在本次实验就完成你想实现文法的语法分析。否则到后面再回来补可能会面临着非常大的工作量，甚至可能需要对代码进行重构。
2. 和词法分析一样，框架已有的代码也是要求完全读懂的，编写代码时一定要不要照着示例代码生搬硬套就完成了实验，否则在语义分析阶段你会完全无从下手。

3.8 线下检查提问示例

1. 若你使用框架代码完成实验，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()` 说明。
2. 请说明你是如何处理 `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。
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？
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. 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？
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，而不直接使用一个 `EXPR`，然后定义成 `EXPR → EXPR ('+' | '-' | '*' | '&&' | ...) EXPR`？
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？请举例说明。
8. 请说明你（或框架代码）是如何将 AST 打印出来的。