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

2020 年 11 月—2024 年 10 月

目录

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。

```
;; Function main (null)
;; enabled by -tree-original

@1      bind_expr       type: @2       body: @3
@2      void_type       name: @4       algn: 8
@3      statement_list
@4      type_decl       name: @5       type: @2
@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 CPP-ARM 框架代码中，我们只实现了 int、void 和函数类型，如果你想要实现 const、数组等其他类型，你需要设计相关的数据结构。

在 CPP-RISCV 框架代码中，我们提供了 float，double，void，ptr，bool，int 类型以及单独的 const 标记，你可以在 include/type.h 中找到相关定义。

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。

CPP-ARM 框架代码中，我们定义了三种类型的符号表项：用于保存字面值常量属性值的符号表项、用于保存编译器生成的中间变量信息的符号表项以及保存源程序中标识符相关信息的符号表项。代码已经实现了符号表的插入函数，你需要在 SymbolTable.cpp 中实现符号表的查找函数。

CPP-RISCV 框架中，语法分析阶段没有进行符号表的实现，我们在语义分析阶段再考虑具体实现符号表。

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 CPP-ARM 框架代码（CPP-RISCV 框架同理）为例：

```
class Node
{
```

[1] 这一部分是私货。

```
private:
    static int counter;
    int seq;
public:
    Node();
    int getSeq() const {return seq;};
    virtual void output(int level) = 0;
};

class ExprNode : public Node
{
protected:
    SymbolEntry *symbolEntry;
public:
    ExprNode(SymbolEntry *symbolEntry) : symbolEntry(symbolEntry){};
};

class Id : public ExprNode
{
public:
    Id(SymbolEntry *se) : ExprNode(se){};
    void output(int level);
};
```

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 Id。Node 类中声明了纯虚函数 output，用于输出语法树信息，派生出的具体子类均需要对其进行实现。

对于 CPP-ARM 框架，你需要根据 SysY 语言特性设计其他结点类型，如 while 语句、函数调用等。

对于 CPP-RISCV 框架，SysY 涉及到的语言特性的语法树类均已实现，但是你需要完全读懂已有的代码（包括语法树节点的输出），在线下检查时助教会提问这部分代码。

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：

```
IfStmt
    : IF LPAREN Cond RPAREN Stmt %prec THEN {
        $$ = new IfStmt($3, $5);
    }
    | IF LPAREN Cond RPAREN Stmt ELSE Stmt {
```

```
        $$ = new IfElseStmt($3, $5, $7);
    }
    ;
```

你可能会对代码中的%prec THEN 感到疑惑，这是为了解决悬空-else 文法的二义性问题, 考虑下面的 if 语句文法:

stmt → **if** *expr* **then** *stmt*

stmt → **if** *expr* **then** *stmt* **else** *stmt*

在语法分析处于如下状态时:

if *expr* **then if** *expr* **then** *stmt* · **else** *stmt*

我们可以将终结符 **else** 移入，也可以使用产生式 *stmt* → **if** *expr* **then** *stmt* 进行归约，这时发生了移入/归约冲突，而正确的做法是将 **else** 移入。

在 yacc 中我们可以给终结符声明优先级:

%precedence **then**

%precedence **else**

这样终结符 **else** 的优先级高于终结符 **then**。产生式的优先级和右部最后一个终结符的优先级相同，即产生式 *stmt* → **if** *expr* **then** *stmt* 的优先级和终结符 **then** 的优先级相同。在发生移入/归约冲突时，通过比较向前看符号和产生式的优先级来解决冲突，若向前看符号的优先级更高，则进行移入，若产生式的优先级更高，则进行归约。这里 **else** 的优先级更高，因此会将 **else** 移入。

SysY 语言中的 if 语句并没有终结符 **then**, 在 yacc 中我们可以使用%prec 关键字, 将终结符 **then** 的优先级赋给产生式。

3.5 CPP-ARM 框架

3.5.1 目录结构

本次实验 框架代码 的目录结构如下:

```
./
├── include
│   ├── Ast.h
│   ├── SymbolTable.h
│   └── Type.h
├── src
│   ├── Ast.cpp ..... 抽象语法树
│   ├── lexer.l ..... 词法分析器
│   ├── parser.y ..... 语法分析器
│   ├── main.cpp
│   ├── SymbolTable.cpp ..... 符号表
│   └── Type.cpp ..... 类型系统
├── sysyruntimelibrary ..... SysY 运行时库
├── test ..... 测试用例
├── .gitignore
├── example.sy ..... SysY 语言样例程序
└── Makefile
```

3.5.2 实验效果

以下面的 SysY 语言源程序为例：

```
int a;

int main()
{
    int a;
    a = 1 + 2;
    if(a < 5)
        return 1;
    return 0;
}
```

如果你已经正确实现了符号表的查找函数，通过 make run 命令，会输出如下的语法树：

```
program
    DeclStmt
        Id    name: @a    scope: 0    type: i32
    FunctionDefine function name: @main, type: i32()
        CompoundStmt
            DeclStmt
                Id    name: %a    scope: 2    type: i32
            AssignStmt
                Id    name: %a    scope: 2    type: i32
                BinaryExpr    op: add
                    IntegerLiteral    value: 1    type: i32
                    IntegerLiteral    value: 2    type: i32
            IfStmt
                BinaryExpr    op: less
                    Id    name: %a    scope: 2    type: i32
                    IntegerLiteral    value: 5    type: i32
                ReturnStmt
                    IntegerLiteral    value: 1    type: i32
            ReturnStmt
                IntegerLiteral    value: 0    type: i32
```

3.5.3 Makefile 使用

框架代码 makefile 的使用如下：

- 编译：

```
make
```

编译出我们的编译器。

- 运行:

```
make run
```

以 example.sy 文件为输入，输出相应的语法树到 example.ast 文件中。

- 调试:

```
make gdb
```

使用 gdb 调试我们的编译器。

- 测试:

```
make testlab2
```

该命令会默认搜索 test 目录下所有的.sy 文件，逐个输入到编译器中，生成相应的抽象语法树.ast 文件到 test 目录中。你还可以指定测试目录:

```
make testlab2 TEST_PATH=dirpath
```

- 清理:

```
make clean
```

清理所有可执行文件和测试输出。

3.6 CPP-RISCV 框架

请注意，和词法分析一样，在实验开始前，你需要先仔细阅读框架中的 README.md 文档，框架如何使用等信息均写在了文档中

3.6.1 实验效果

```
1   int a = 5;
2   int main()
3   {
4       putf("hello world\n");
5       return a;
6   }
```

```
1   Program
2     VarDecls   Type: Int
3       VarDef   name:a   scope:-1
4         init:
5           VarInitVal_exp
6             Intconst   val:5   Type: Void
7     FuncDef   Name:main   ReturnType: Int
8       Block   Size:2
9         ExpressionStmt:   Type: Void
10          FuncCall   name:putf   Type: Void
11            FuncRParams:
12              StringConst   type:string   val:hello world\n
13        ReturnStmt:
14          Lval   Type: Void      name:a   scope:-1
```

图 3.1: 语法分析实验结果示例

可以看到，在语法树的输出中，大部分 Type 均为 Void，并且 scope 作用域的值均为-1，这是由于我们只建立了语法树，但是没有对语法树进行任何处理。我们会在语义分析中确定每个节点的类型以及作用域。实际上对语法树进行一次遍历即可完成类型和作用域的确定，你在语法分析环节中就可以思考一下具体如何实现。（图片中的 StringConst 仅仅是一个示例，实际我们不要求实现）

3.6.2 运行编译器

假设你更新了框架中的 SysY_parser.y 文件，使用如下命令进行编译

```
make parser
make -j   # 如果你更新的不是 SysY_parser.y 和 SysY_lexer.l 文件，直接 make -j 即可
```

假设你成功编译后想要对项目根目录下的 example.sy 进行测试，使用如下命令：

```
./bin/SysYc -parser -o example.out example.sy
# 查看 example.out 来检验你的语法分析实现
# 在中间代码生成之前，框架不会提供测试脚本，需要同学们自行检验实现的正确性
```

3.7 注意事项

1. 虽然本次实验只要求能实现上级大作业总体要求中的基本要求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. 请说明 yylval，yytext 以及词法分析中的返回值在语法分析阶段是如何被使用的。(如果被使用到的话)
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。
3. 请说明语法树的根节点是什么类型，该根节点的子结点可能有哪些类型。
4. SysY_parser.y 中，%union，%token, %type 分别是什么意思，对应上下文无关文法的哪些部分(如果有对应的话)。
5. 变量定义的数组维度可以出现 0 次或多次，变量声明 decl 的 vardef 可以出现一次或多次，但是不能 0 次。你在编写 SysY_parser.y 时是怎么处理这种语法的。
6. 请说明文法设计时为什么要使用 addExp，mulExp，relExp 等一系列 Exp 并加上复杂的语法推导，而不直接使用一个 Exp，然后 Exp → Exp ('+' | '−' | '*' | '&&' | ...) Exp