实现语法分析器

杨侯哲 李煦阳

November 2020

目录

1 实验描述

学期已过半，我们实现编译器的征程也终于来到最有趣最关键的地方。

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。

下面将以一个最简单的语言为例，说明代码的一种组织方式。

1.1 简单语言上的样例

1.1.1 一些分析与注意

数据结构：树结点 我们唯一需要面对的数据结构就是语法树。需要注意的是，我们随着语法分析的进行，构建的是一棵抽象语法树（AST），因为我们并不需要保留产生式推导过程中的每一层，这对应着具体语法树（CST）。我们要设计的就是语法树的结点。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。直观上，结点的类型被分为词法分析得到的叶节点、表达式、基本语句（不嵌套地有子语句的语句）、复合语句（比如 `for`，`if-else`，`{}`，函数），以及象征完成程序的根节点。为了支持基本的输入输出、计时（如果要做程序的优化），基本语句中还要包含特殊的函数调用语句。如果你想要支持一般的函数，表达式的设计可能会变得复杂。

类型系统的复杂性 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，甚至函数本身作为变量，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

[1] 这一部分是私货。

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”。

算法：词法分析、语法分析与树的构建 词法分析得到的，实质是语法树的叶子结点[2]，语法树所有内部节点均由语法分析创建。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节点。在 `yacc` 每次确定一个产生式发生 reduce 时，我们会 `new` 出父节点、根据子结点正确设置父节点的属性、记录继承关系。每个结点的孩子数量是不统一的，我们可以设计一个 `sibling` 链表，或者一个宽裕的子节点数组，记录继承关系。

工程难点：lex 与 yacc 的连结、个人项目文件的引入 `lex` 与 `yacc` 的连结依靠某些全局变量，如 `yytext` 等，相信最近两次作业已让你对它们十分熟悉。此外，你可能希望将自己的数据结构定义放在独立的`.c|h` 文件之中，供 `lex` 与 `yacc` 使用，为此你需要正确的区分头文件与源文件中代码的内容，避免编译时出现 `link error`。

项目的管理与错误的定位 本次开始的最后三次作业，建议使用 `git` 保管好项目的记录。因为很有可能在某一次代码功能的更新或重构后，你的代码改的太乱、已经痛苦地不再可用，而需要进行版本回溯。也为了防止可能的虚拟机崩溃，希望你可以托管到远程仓库中。同时，我们也会以你的 `commit` 记录评判你的作业完成情况[3]。为了方便你自己的调试、测试，建议你实现 `yyerror` 函数、为每个结点标记其出现的首行行号、设计良好的有信息价值的 `logging` 等。

1.1.2 示例语言

实验课的演示以 **Figure. 1** 中语法为例。

(*Var*) x	$\in$	*ID*
(*Type*) T	::=	`int` | `char`
(*Expr*) E	::=	$x \mid n \mid E + E \mid \ldots$
(*BExp*) B	::=	`true` | `false` | $E == E \mid !B \mid \ldots$
(*Instr*) c	::=	$T\ x = E \mid x = E \mid \textbf{printf}(E) \mid \textbf{printf}(B) \mid \textbf{scanf}(x)$
(*Stmt*) C	::=	ϵ
		c
		if (B) C **else** C
		while (B) C
		$\{S\}$
(*Stmts*) S	::=	$C \mid C; C$
(*Prog*) W	::=	S

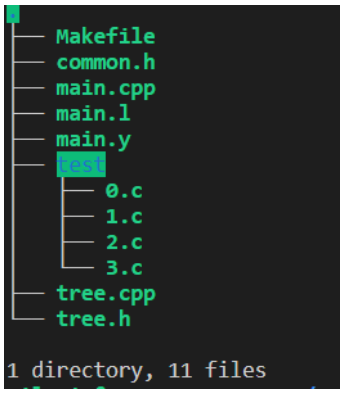
Figure. 1 示例语言语法

1.1.3 代码

我们提供一份根据示例语言的简易实现供参考使用。我们示例中的项目结构如下图所示。

[2]运算符在 CST 的意义上也可以算作叶子结点

[3]回答问题不好、又没有良好的 `git log`，我们会优先判定为抄袭。



首先需要说明的是，**我们并没有给出全部的实现文件，提供的示例也缺失很多必要的功能**。示例的主要目的是提供实现思路，了解 flex 与 bison 是如何结合起来与我们自己设计的结构一起工作的。

这其中 common 文件引用了我们自定义的树结构头文件并定义 YYSTYPE 为树结构结点，*main.cpp* 作为主函数引用了 bison 编译出的头文件与其他库控制整体流程，*tree* 文件实现了我们自定义的语法树结构，在 l 文件与 y 文件中被使用。

语法树结点 通过观察代码，可以知道到我们的示例中有一个完整的建立语法树并输出的过程，中间每个步骤的赋值细节同学们可以根据自己定义的树结构自行修改，当然除了对当前结点的赋值外这其中还用到语法树结构的 *addChild,addSibling* 功能，这里值得说明的是每次添加的位置不一定是当前调用函数的结点，同学们应该根据自己定义的结点形式自行考虑。

定义不同的类型 在 y 文件中我们对不同结点赋予了不同类型的值，我们的实现思路是用下文中的方法来实现：

```
enum NodeType{
    NODE_A,
    NODE_B,
    NODE_C,
    ...
};
enum OpType{
    OP_A,
    OP_B,
    OP_C,
    ...
}
struct TreeNode{
    ...
    NodeType nodeType;//结点类型
    OpType opType;//操作符具体类型 结点是操作符时写入具体类型
    ...
    TreeNode(NodeType type);//在构造结点时直接填入类型
    ...
```

```
}
```

示例中需要说明的事情

- 我们没有任何函数的实现，但同学们需要至少完成 main 函数的语法结构实现，对于普通函数调用的实现对同学们来说是可选项。
- 对于 IO 函数的操作，示例中将其作为特例来实现，并且未实际支持真正的函数多参数的调用，而对于同学们来说要如何实现 IO 函数也是由自己决定的，另外，为了最终检测的方便性，我们建议同学们统一使用 `printf`、`scanf` 作为 IO 函数使用，如果实在无法完成也可以自行修改样例。
- 我们没有添加符号表，但是对于作业来说一个全局的符号表是必须的，如果有实现作用域功能的同学可能需要配合使用多层符号表。
- 对于变量的类型，我们可以只实现一个 int 型的变量，但字符常量和字符串常量是我们应尽量识别的目标。
- 编译运行的命令可以直接通过 `make run` 进行编译，Makefile 会自动检测哪些文件发生了变化，整体的编译过程同学们可以自己运行观察。

1.1.4 实验效果

通过 `make test` 命令，Makefile 会编译并将 test 文件夹下的 c 文件逐个测试，输出到对应的 res 文件中，对于最简单的示例文件[4]的最简单输出如下：

```
0   program     child:  1 5 9 23
1   statement   child:  2 3 4
2   type        child:
3   variable    child:
4   const       child:
...
```

注意，**我们的示例输出只输出了非常少的内容**，同学们可以自行添加更多的内容如结点具体的值，作用域，类型系统输出，对应哪行等信息。

另外，对于不同的大作业级别，我们提供几份基础样例，供同学们自行检查，同学们也可以自行添加更多的测试样例来测试自己的语法分析器。

2 作业的分级要求与测试样例

本次作业与最终作业紧密相关，不同级别选择造成的实现难度的区分会在这三次作业都有体现。**但注意，在本次作业和下次作业中，只要实现最基本要求的全部功能就可以得到满分**。但若你的大作业要支持更多的功能，也必然需要补充这两次作业。这一部分你可以在未来逐渐补充。

下面简单阐述你可能要完成的工作。

本次作业（实现语法分析器）满分 5 分，下次作业实现类型检查满分 2 分，大作业（完成编译器）满分 14 分。

[4] 即 0.c

2.1 级别一要求及测试样例

要求：

1. 数据类型：int，char, 常量字符串。
2. 变量声明，注意正确区分不同作用域的同名变量。
3. 语句：赋值（=）、表达式语句、语句块、`if`、`while`、`for`、`return`.
4. 表达式：算术运算（+、-、*、/、%，其中 +、-都可以是单目运算符）、关系运算（==，>，<，>=，<=，!=）和逻辑运算（(与)、||（或）、!（非））
5. 注释.
6. 简单的输入输出（依 SysY 运行时库或自行定义词法、语法、语义均可，最好可以支持有“格式控制符”的 `printf`, `scanf`）。

下次类型检查作业中，你需要支持变量未声明、重声明错误检查，类型错误检查（比如字符串、布尔值不能参与某些运算、输入输出函数也有参数类型要求）、并在进行隐式类型转换时给出提示。

```
/*
    I'm level 1 test.
 */
void main() {
    int a, s;
    a = 10;
    s = 0;
    char ch;
    scanf("\%d", &ch);
    while(a>0 && a<=10 || a%100==10 && !a==10) {
        a -= 1;
        int a;
        a = 10;
        s += a;
        if(-s < -10) {
            printf("result is: %d\n", s);
            int b;
            b = 10;
            for(int i=0; i<b; i++) {
                printf("Have fun: %d\n", i);
            }
        }
    }
}
// No more compilation error.
```

2.2 级别二要求及测试样例

进阶要求的每一项，我们会根据实现难度、实现效果给予分数。也就是说，每一个级别，在保证一定的工作量前提下，你可以只实现你最喜欢的一部分。

要求：

1. 支持每种类型的 const 常量的声明（与初始化），对于 int/char 类型，支持十进制、八进制、十六进制数。同时支持变量的定义与初始化。
2. 支持任意维数组。
3. 支持结构体类型。
4. 支持指针，支持引用。

对于语法分析，你将需要实现常量声明与初始化、数组、结构体声明与使用、对指针等语法结构的支持。对各进制数在词法分析中作识别。为了支持非基本数据类型，你需要修正你的类型系统。

类型检查作业中，你需要检查常量的未初始化错误，对常量的重赋值错误。对于非基本类型，实现关于它们的类型检查。

```
/*
    I'm level 2 test. Without pointer.
 */
struct Matrix {
    int id;
    int arr[10][10];
} m1, m2, m3;
const int len = 10;
void main() {
    m1.id = 1, m2.id = 2, m3.id = 3;
    for(int i=0; i<len; i++) {
        for(int j=0; j<len; j++) {
            m1.arr[i][j] = i;
            m2.arr[i][j] = j;
            m3.arr[i][j] = m1.arr[i][j] + m2.arr[i][j];
        }
    }

    for(int i=0; i<len; i++) {
        for(int j=0; j<len; j++) {
            printf("<%d>[%d][%d] %d\t",m3.id, i, j, m3.arr[i][j]);
        }
        printf("\n");
    }
}
```

2.3 级别三要求及测试样例

要求：

1. 支持 `break/continue`
2. 支持任意数量基本类型参数的函数。此项为必须项。
3. 支持对运行时库中函数的调用。

对于语法分析，你主要需要实现对**有基本类型参数的函数**、对 break/continue 语句的支持。

类型检查作业中，你需要实现对函数调用作参数检查以及对 `break/continue` 语句作 `within loop` 检查。此外，鉴于函数本质也是变量，也要被纳入类型检查中。

```
/* I'm level 3 test, With no runtime */
int s = 0;
int f(int x, int y) {
    s += x*x + y*y;
    return s;
}
void main(){
    int i=0;
    int a=1, b=1;
    int line;
    scanf("%d", &line);
    if(line > 10000) line = 10000;
    while(true) {
        if(f(a++, b++)<line) {
            printf("sum is: %d\n", s);
        } else {
            printf("result is:%d\n", s);
            break;
        }
    }
}
```

2.4 级别四要求

大作业的最终级别！

1. 你需要支持中间代码生成（可以想象，你需要额外设计 CFG 的数据结构，它由三地址码构成；并设计算法完成 AST 到 CFG 的转换。）
2. 并在其上实现代码优化。优化借助计时函数衡量。

这一部分不在本次或下次作业中具有要求。