编译原理

实验（三）自顶向下的语法分析器

姓　　名	熊恪峥
学　　号	22920202204622
日　　期	2023年4月19日
学　　院	信息学院
课程名称	编译原理

实验（三）自顶向下的语法分析器

目录

1 实验目的

掌握语法分析器的构造原理，掌握递归下降法的编程方法。

2 实验内容

用递归下降法编写一个语法分析程序，使之与词法分析器结合，能够根据语言的上下文无关文法，识别输入的单词序列是否文法的句子。

3 运行结果

运行结果如图 1，完整输出见附录：完整输出。

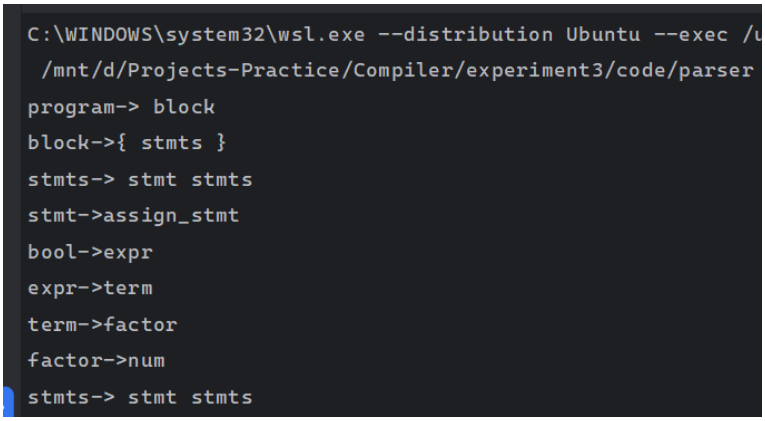

图 1: 运行结果

4 Pratt Parser: 比LL(1)更优雅、同等高效地处理表达式

Pratt Parser也被称为“优先表达式分析”（Precedence Parsing）。这种算法由美国计算机科学家Vaughan Pratt于1973年提出，在JSLint这一著名的JavaScript语法检查工具中得到了应用。

Pratt Parser对于操作符的优先级和结合性进行处理，通过定义不同的优先级和绑定（左结合、右结合等）来处理表达式。这种算法可以被认为是一种自顶向下的分析方法，与传统的LL（1）和LR（1）方法类似，但具有更好的灵活性和适应性。

与基于BNF语法来决定动作的LL(k)不同，Pratt Parser是基于Token的种类根据优先级和结合性来决定动作的。这种方法的优点是，在很容易地处理操作符的优先级和结合性的同时，最大限度地避免回溯，因而比自顶向下的分析更为高效。例如考察要求中给定的文法中的表达式部分：

```
bool → expr relop expr | expr
expr → expr + term | expr - term | term
term → term * factor | term / factor | factor
factor → ( expr ) | id | num
```

假设给定一个简单的单变量赋值语句$a = b;$，为了分析右侧的变量表达式b，普通的自顶向下分析需要依次调用`bool`、`expr`、`term`，`factor`，才能选定给定的输入串b对应的产生式`factor -> id`。在这一个例子中，给定的表达式仅仅造成了4次函数调用，但是在实际的编程语言中，表达式的复杂性往往会造成更多的函数调

用，在一个递归过程中，更多的函数调用会造成更多的栈空间的使用，从而造成更多的内存开销，也限制了表达式的处理深度。

产生这一现象的原因在于，为了让文法能够有效表达优先级和结合性，并且消除左递归，BNF文法被大大复杂化了。而普通的自顶向下分析，或者 LL(1)预测分析都依赖于文法决定分析动作，因此这是无法避免的。

在其论文中*Top Down Operator Precedence*，Vaughan Pratt详细描述了这一现象：

> The traditional mechanism for assigning meanings to programs is to associate semantic rules with phrase-structure rules, or equivalently, with classes of phrases. This is inconsistent with the following reasonable model of a programmer.
>
> The programmer has in mind a set of semantic objects. His natural inclination is to talk about them by assigning them names, or tokens. He then makes up programs using these tokens, together with other tokens useful for program control, and some purely syntactic tokens. (No clear-cut boundary separates these classes.) This suggests that it is more natural to associate semantics with tokens than with classes of phrases.

当程序员写程序时，与普通的自顶向下分析器不同，他们并非根据语法规则决定语义进而构造程序，而是将语义对象赋予名字，然后他们就成为了代码中的Token。这样一来，根据Token的种类来决定语义动作就成为了一种更为自然的方法。

并且在其论文中也提到了LL(k)分析的缺陷，这些缺陷也跟我实现真实世界中的编译器[1]时的经验相吻合：

> A number of down-to-earth issues are not satisfactorily addressed by their system – deficiencies which we propose to make up in the approach below; they are as follows.
>
> From the point of view of the language designer, implementer or extender, writing an LL(k) grammar, and keeping it LL(k) after extending it, seems to be a black art, whose main redeeming feature is that the life-support system can at least localize the problems with a given grammar. It would seem preferable, where possible, to make it easier for the user to write acceptable grammars on the first try, a property of the approach to be presented here.
>
> There is no "escape clause" for dealing with non-standard syntactic problems (e.g. Fortran format statements). The procedural approach of this paper makes it possible for the user to deal with difficult problems in the same language he uses for routine tasks.
>
> The life-support system must be up, running and debugged on the user's computer before he can start to take advantage of the technique. This may take more effort than is justifiable for one-shot applications. We suggest an approach that requires only a few lines of code for supporting software.
>
> Lewis and Stearns consider only translators, in the context of their LL(k) system; it remains to be determined how effectively they can deal with interpreters. The approach below is ideally suited for interpreters, whether written in software, firmware or hardware.

这包括：不易于扩展、无法处理上下文无关文法无法处理的问题（例如Python中的缩进，以及原文提到的Fortran格式化语句）、需要大量的辅助代码等。

因此，在我的实现中，虽然输出结果和自顶向下分析的语法定义保持了一致，但是实际分析过程中，对于表达式，我使用Pratt Prasing进行了实现。

它需要为标志着表达式的Token定义下面的内容：

- 优先级

[1]开源在GitHub上：`https://github.com/SmartPolarBear/clox`

- 在整个表达式中作为表达式前缀的处理函数
- 在整个表达式中作为表达式中缀的处理函数

然后，直接使用优先级就可以驱动分析过程，而不必依赖于文法。

例如，在我的实现中定义了代码 1的优先级和结合性规则。 而Pratt Parsing的驱动程序如代码 2。首先根

代码 1 优先级和结合性规则表

```
std::unordered_map<token_type, rule> expr_rules{
        {token_type::TK_LPAREN, rule{PREC_NONE, &parser::grouping, nullptr}},
        {token_type::TK_PLUS, rule{PREC_TERM, &parser::unary, &parser::binary}},
        {token_type::TK_MINUS, rule{PREC_TERM, &parser::unary, &parser::binary}},
        {token_type::TK_DIVIDE, rule{PREC_FACTOR, nullptr, &parser::binary}},
        {token_type::TK_MULTIPLY, rule{PREC_FACTOR, nullptr, &parser::binary}},
        {token_type::TK_NEQ,rule{PREC_EQUALITY, nullptr, &parser::binary}},
        {token_type::TK_EQ,rule{PREC_EQUALITY, nullptr, &parser::binary}},
        {token_type::TK_GT,rule{PREC_COMPARISON, nullptr, &parser::binary}},
        {token_type::TK_GTE,rule{PREC_COMPARISON, nullptr, &parser::binary}},
        {token_type::TK_LT,rule{PREC_COMPARISON, nullptr, &parser::binary}},
        {token_type::TK_LTE,rule{PREC_COMPARISON, nullptr, &parser::binary}},
        {token_type::TK_NUM, rule{PREC_NONE, &parser::unary, nullptr}},
        {token_type::TK_ID, rule{PREC_NONE, &parser::unary, nullptr}},
};
```

据当前Token选择前缀规则，然后向右侧继续选择中缀规则，直到遇到优先级低于当前优先级的Token。从代码中易见得，为了分析前述的例子中的b，仅需要一次函数调用即可完成。这大大提高了效率。

需要注意的是，这种方式虽然和LL(1)分析达到了近乎一致的效果，但它在工程上的优势是显著的：

真实世界中的语言对优先级、结合性的是相当复杂的，为此构建文法、消除左递归、建立分析表的过程将会相当复杂，这导致深层嵌套的语法，还有难以保证文法预处理正确的问题。而Pratt Parsing则只需要这是因为在LL(1)分析中，我们需要将这些信息隐式地编码在文法中，而Pratt Parsing则将这些信息显式地编码在了分析表中，这使得我们可以更加直观地理解和修改这些信息。这带来了极大的工程上的优势

代码 2 Pratt Parsing

```
// pratt parsing
void parser::all_expr(precedence prec)
{
        // Get the next token
        auto token = advance();

        // Get the prefix parse function for the token
        auto prefix = expr_rules[token->type].prefix;

        // Check if there is a prefix parse function for the token
        if (!prefix)
        {
                throw std::runtime_error("Unexpected token: " + token->lexeme);
        }

        // Parse the prefix expression
        auto left = (this->*prefix)();

        // Loop until the precedence is lower than the current precedence
        while (prec <= expr_rules[peek()->type].pred)
        {
                // Get the infix parse function for the next token
                auto in_tk = advance();
                auto infix = expr_rules[in_tk->type].infix;

                // Parse the infix expression
                left = (this->*infix)(left);
        }

        // Return the parsed expression
//        return left;
}
```

A 附录：完整输出

```
program-> block
block->{ stmts }
stmts-> stmt stmts
stmt->assign_stmt
bool->expr
expr->term
term->factor
factor->num
stmts-> stmt stmts
stmt->assign_stmt
bool->expr
expr->term
term->factor
factor->num
stmts-> stmt stmts
stmt->while(expr)stmt
bool->expr
expr->term
term->factor
factor->id
bool->expr
expr->term
term->factor
factor->num
bool->expr<=expr
stmt->block
block->{ stmts }
stmts-> stmt stmts
stmt->assign_stmt
bool->expr
expr->term
term->factor
factor->id
bool->expr
expr->term
term->factor
factor->id
bool->expr
expr->expr+term
stmts-> stmt stmts
stmt->assign_stmt
bool->expr
expr->term
term->factor
factor->id
bool->expr
expr->term
```

```
term->factor
factor->num
bool->expr
expr->expr+term
stmts-> stmt stmts
stmt->if(expr)stmt
bool->expr
expr->term
term->factor
factor->id
bool->expr
expr->term
term->factor
factor->num
bool->expr
expr->term
term->term/factor
bool->expr
expr->term
term->factor
factor->num
bool->expr==expr
stmt->break;
```