Table of Contents

牙医教你 450 行代码自制编程语言

Description

没有系统学习过编译原理的同学可能会很好奇编程语言的编译器, Lexer & Parser, 虚拟机是怎么实现的. 而又苦于系统性的教材过于枯燥.

本教程教大家用 450 行 Go 代码实现一个简单的编程语言, 它的语法是这样的:

```
$a = "pen pineapple apple pen."
print($a)
```

看上去很简单是不是? 但是它包含了个手写的递归下降解析器和一个简单的解释器.

虽然该语言甚至不是图灵完备的. 但写这个语言和教程的主要目的是让编译原理初学者有一个预热, 简单了解一个编程语言是怎么构建的.

准备好了吗? 让我们开始!

Author

karminski-牙医

License

牙医教你 450 行代码自制编程语言 - 1, 从 EBNF 开始.md

@version 20210102:1
@author karminski work.karminski@outlook.com

打算新开个读书专栏, 主要写一些我读过的书的读书笔记和理解分享给大家. 本篇是其中的第一篇, 图书是《自己动手实现Lua: 虚拟机, 编译器和标准库》:

- 《自己动手实现Lua：虚拟机、编译器和标准库》

没有系统学习过编译原理的同学可能会很好奇编程语言的编译器, Lexer & Parser, 虚拟机是怎么实现的. 而又苦于系统性的教材过于枯燥.

那么其实本书作为系统学习编译原理的预热, 我觉得是非常适合的. 即使并不准备系统性的学习, 看完此书后, 也能对编译原理有个很不错的理解.

就这本书而言, 作者 张秀宏 利用 Go 语言实现了一个 Lua 虚拟机. 其代码简洁凝练, 非常易于学习.

这里我并不打算详细介绍这本书的内容, 反而我想利用这本书的内容给大家演示一下, 实现一个最简单的编程语言.

太长的教程作为例子不是很好, 所以我只用 450 行 Go 代码就完成了这个例子, 请看:

从 EBNF 开始

我们来实现一个最简单的编程语言, 就叫 pineapple 好了 (代码可以从 https://github.com/karminski/pineapple 下载). 它只有 450 行代码, 这个语言甚至都不是图灵完备的, 但作为演示已经足够了.

这个语言的功能只有给变量赋值, 然后打印变量, 类似这样:

```
$a = "pen pineapple apple pen."
print($a)
```

然后它会打印:

```
pen pineapple apple pen.
```

是不是很简单? 没错, 接下来我们就要实现它.

我们从 EBNF 开始, 下面的东西也许你看不懂, 没关系, 你马上就能看懂了!:

```
SourceCharacter ::=  #x0009 | #x000A | #x000D | [#x0020-#xFFFF]
Name            ::= [_A-Za-z][_0-9A-Za-z]*
StringCharacter ::= SourceCharacter - '"'
String          ::= '"' '"' Ignored | '"' StringCharacter '"' Ig
Variable        ::= "$" Name Ignored
Assignment      ::= Variable Ignored "=" Ignored String Ignored
Print           ::= "print" "(" Ignored Variable Ignored ")" Ign
Statement       ::= Print | Assignment
SourceCode      ::= Statement+
```

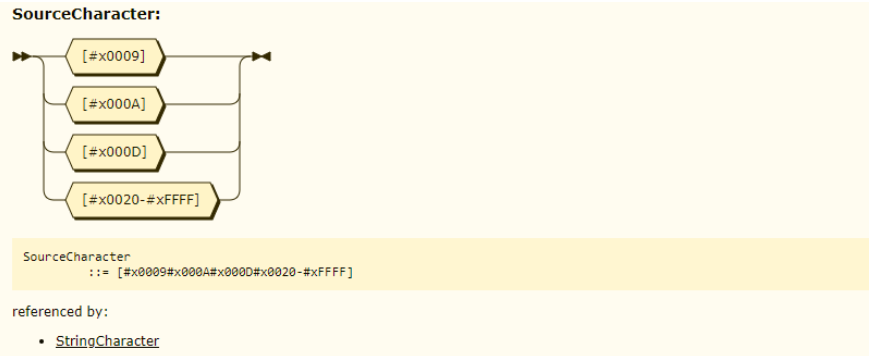
简单来讲, EBNF (Extended Backus–Naur form, 扩展巴科斯范式) 是一种描述语法的语言, 相当于编程语言语法的蓝图, 有了它, 我们就可以轻松实现一门语言的词法和语法解析器.

我们先来看第一行, SourceCharacter:

```
SourceCharacter ::=  #x0009 | #x000A | #x000D | [#x0020-#xFFFF]
```

这代表了, SourceCharacter 这个表达式 可以是 Unicode #x0009 或 #x000A 或 #x000D 或 #x0020-#xFFFF 这个范围. 这基本是可用字符串范围. 如果你看不懂, 没关系, 我们也不会这么去实现, 就大概知道是大部分的 Unicode 就可以了.

EBNF 定义可以生成语法图, 看起来是这样子的:

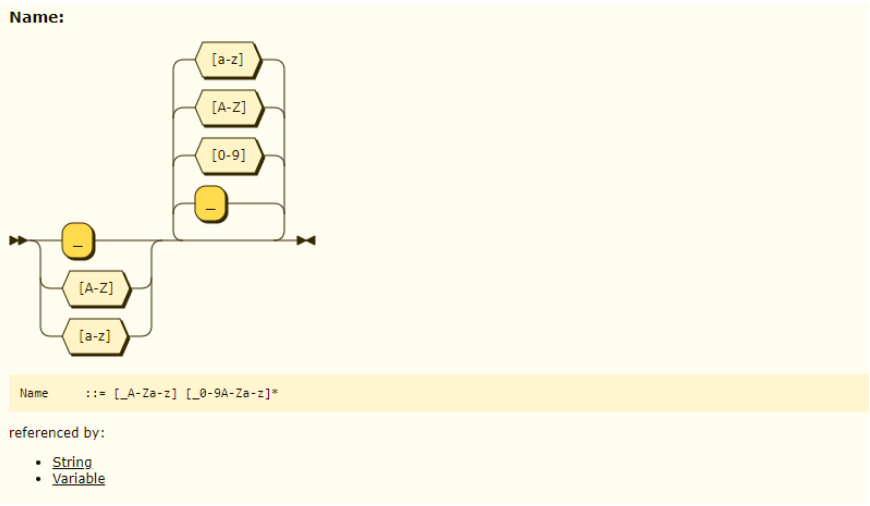


这图是从左向右的流程, 从左侧的双箭头开始, 每条分支代表都可以走的流程路线. 最后到右侧相对的箭头结束.

然后再来看 Name 表达式:

```
Name ::= [_A-Za-z][_0-9A-Za-z]*
```

这代表了, Name 这个表达式开头由 下划线 `_` 或大小写字母 `A-Za-z` 构成, 熟悉 正则表达式的同学立刻就会看懂. 然后后面由 下划线, 大小写字母, 数字构成 `_0-9A-Za-z` , 最后的 `*` 代表这些可以有0或多个. (名称不能以数字为开头, 一部分原因其实是为了规避词法解析中与数字冲突的问题, 因为数字是用数字开头的, 如果不是数字的类型 也用数字开头, 那么解析器的实现就会变得相当复杂)

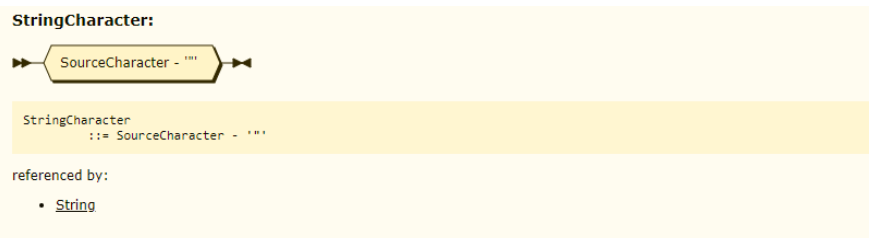


从左向右, 每条分支都代表可以构成的路径. 语法图 很好地图形化了我们的 EBNF 定义.
注意连接到中间的横线上的分支部分, 有的是向 内闭 合的, 有的是向 外闭合的, 向 内闭 合代表这可以循环, 即 `*` , 向 外闭 合则代表只能出现一次.

接下来是 StringCharacter 表达式:

```
StringCharacter ::= SourceCharacter - '"'
```

这里的 `- '"'` 代表不包含双引号 `"` , 即 StringCharacter 是 SourceCharacter 但不包含双引号. (String 要用双引号作为结束/闭 合的标记)



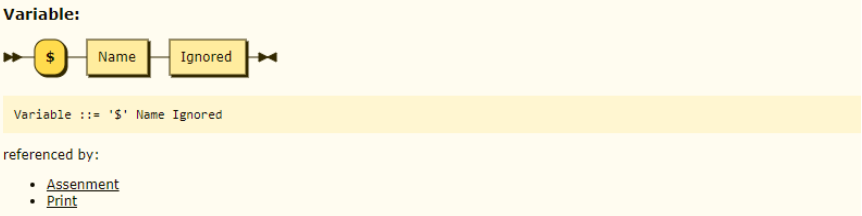
同样 String:

```
String ::= '"' '"' Ignored | '"' StringCharacter '"' Ignored
```

代表了 `String` 表达式由空字符串 `"" Ignored` (双引号里面是空的) 或不包含双引号的字符串 `"StringCharacter" Ignored` 构成. `|` 代表或, Ignored 代表可忽略的字符, 比如空格, tab, 换行, 注释等.

然后是 Variable 表达式:

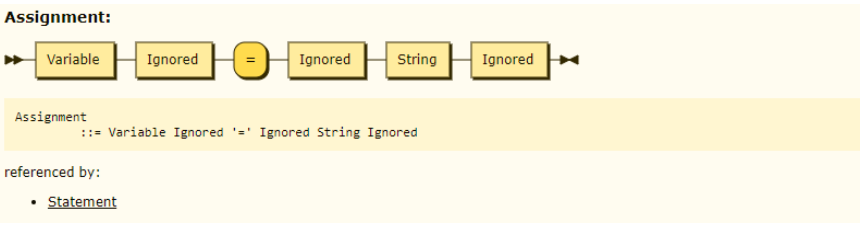
```
Variable ::= "$" Name Ignored
```



是不是已经能看懂了? 没错, Variable 由一个美元符号 $ 和 Name 表达式构成.

Assignment 表达式也很简单:

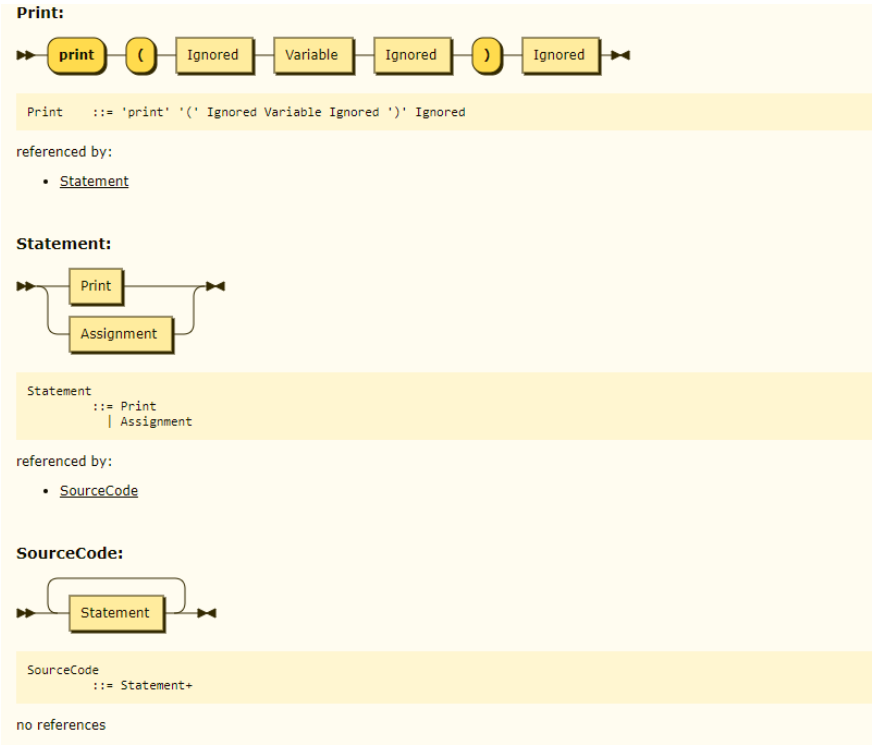
```
Assignment  ::= Variable Ignored "=" Ignored String Ignored
```



前面是 Variable, 然后跟着等号 `=`, 后面是 String. 中间有 Ignored 则代表, 可以用空格等将这些元素分开.

最后剩下的 Print, Statement, SourceCode:

```
Print       ::= "print" "(" Ignored Variable Ignored ")" Ignored
Statement   ::= Print | Assignment
SourceCode  ::= Statement+
```



Print 语句由 `print`, 左圆括号 `(`, Variable, 右圆括号 `)` 组成.

Statement 代表语法中的所有语句, 我们只有 Print 和 Assignment 是合法语句.

最后用 SourceCode 来封装 Statement, `+` 代表 SourceCode 里面可以有多个语句.

到这里, 我们就完成了 pineapple 语言的 EBNF 定义, 也了解了这门语言的语法.

告一段落

第一篇就到这里结束啦, 第二篇我们会教大家构建 Lexer (词法解析器). 敬请期待~

牙医教你 450 行代码自制编程语言 - 2, 两个魔法就可以实现永动机.md

@version 20210103:1
@author karminski work.karminski@outlook.com

上篇 牙医教你 450 行代码自制编程语言 - 1, 从 EBNF 开始, 我们简单演示了一门编程语言的语法设计, 以及 EBNF 是如何使用的. 现在我们已经有了名为 EBNF 的蓝图, 就可以开始写词法分析器 (Lexer) 了.

本教程的所有代码都可以在 https://github.com/karminski/pineapple 找到.

另外, 再次推荐这本书, 本教程就是类似这本书的简化版本, 想要仔细学习的话可以考虑看原作:

- 《自己动手实现Lua：虚拟机、编译器和标准库》

Lexer (词法分析器)

且慢, 啥是词法分析器?

简单来讲, 词法分析器就是将源代码解析成 EBNF 中定义的最小元素 (也叫 Token) 的过程. 比如如下 EBNF 定义 (看不懂的同学可以复习下上篇教程从 EBNF 开始):

```
IntPrefix ::= "int"
```

这里的 `int` 不可再分且是固定的, 就是 Token. 我们的目的就是先将所有代码解析成不可再分的最小元素 Token 的过程. (当然, 本文的 Token, 语句, 表达式等定义很不精确, 想要详细了解的话可以看教材. 我们为了简便, 不会过于纠结这些定义.)

定义 Token

我们仍然把上篇的 EBNF 定义放在这里:

```
SourceCharacter ::=  #x0009 | #x000A | #x000D | [#x0020-#xFFFF]
Name            ::= [_A-Za-z][_0-9A-Za-z]*
StringCharacter ::= SourceCharacter - '"'
String          ::= '"' '"' Ignored | '"' StringCharacter '"' Ig
Variable        ::= "$" Name Ignored
Assignment      ::= Variable Ignored "=" Ignored String Ignored
Print           ::= "print" "(" Ignored Variable Ignored ")" Ign
Statement       ::= Print | Assignment
SourceCode      ::= Statement+
```

然后我们开始定义 Token:

```
const (
    TOKEN_EOF          = iota  // end-of-file
    TOKEN_VAR_PREFIX           // $
    TOKEN_LEFT_PAREN           // (
    TOKEN_RIGHT_PAREN          // )
    TOKEN_EQUAL                // =
    TOKEN_QUOTE                // "
    TOKEN_DUOQUOTE                // ""
    TOKEN_NAME                 // Name ::= [_A-Za-z][_0-9A-Za-z]*
    TOKEN_PRINT                // print
)
```

可以看到我们定义了一堆常量, 包括变量的前缀 `$`, 包裹函数参数的括号 `(` `)`, 甚至还有函数名称 `print`.

这里有一些特殊的 Token, 比如 `TOKEN_EOF`, 这个 Token 用于标记代码的结束. 检测到这个 Token, 就不会继续解析下去了.

另外还有 `TOKEN_DUOQUOTE`, 两个双引号 `""`, 代表空字符串. 这里为了简化处理, 我们直接在 Lexer 阶段直接把空字符串 当成 Token 来处理了. 这样可以简化后续逻辑. 算是个小技巧.

最后还能看到, 说了需要写 **将所有代码解析成不可再分的最小元素**, 所以有 `TOKEN_NAME`, 但是为什么没有代表 `SourceCharacter` 的 `TOKEN_SOURCE_CHARACTER` 呢?

这是因为 `SourceCharacter` 的范围 实在是太大了, 每个字符都去按照范围 去匹配的话, 会影响 性能, 因此我们采用另外的方式去匹配 `SourceCharacter`, 一会大家就能看到了.

然后需要造一个永动机

等会, 刚才还计算机科学呢, 现在怎么就突然切换到民科了?

这里的永动机其实要说的是自动机, 编程语言的 Lexer 就是其中的一种. 我们并不会讲解自动机理论, 所以放松心情, 看我们来造两个神奇的函数:

```
func NextTokenIs(token int) (tokenName string) {}
func LookAhead() (token int) {}
```

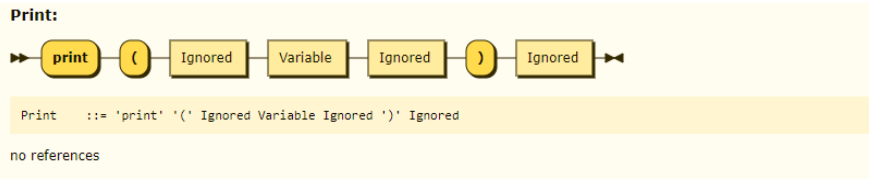
其中 `NextTokenIs()` 函数用于断言下一个 Token 是什么, 比如 `NextTokenIs(TOKEN_VAR_PREFIX)`, 我们断言下一个 Token 必然是 `$`, 如果不是, 就证明输入的代码出语法错误了.

而 `LookAhead()` 没有参数, 但执行它的时候, 他会 LookAhead (向前看) 一个 Token, 告诉我们下一个 Token 是什么.

也许你会问, 这俩函数有什么用呢? 来, 我们要开始咏唱魔法了!

我们现在定义如下 EBNF:

```
Print ::= "print" "(" Ignored Variable Ignored ")" Ignored
```



没错, 我们之前定义的 Print 语句. 它的开头必须是 `print` 这个字符串, 我们已经定义好了 Token: `TOKEN_PRINT`.

现在我们直接快进到 Parser, 为了解析 Print 语句, 我们可以这么写:

```
func parsePrint() {
    // "print"
    NextTokenIs(TOKEN_PRINT)
    ...
}
```

很简单对不对? 一旦解析 Print 的语法的时候, 我们就断言, Print 开头必须是 `TOKEN_PRINT` (即 `"print"`).

那么举一反三, 接下来是左括号 `(`, 然后是变量的语句 `Variable`, 最后是右括号 `)`. 那么就可以写成:

```
// Print ::= "print" "(" Ignored Variable Ignored ")" Ignored
func parsePrint() {
    // "print"
    NextTokenIs(TOKEN_PRINT)
    // "("
    NextTokenIs(TOKEN_LEFT_PAREN)
    // Variable
    parseVariable()
    // ")"
    NextTokenIs(TOKEN_RIGHT_PAREN)
}
```

是不是有一种渐渐的明白了什么的感觉? 对! 如果遇到了 Token 就直接断言 Token (`NextTokenIs(...)`), 如果遇到了语句, 就调用**解析这个语句的函数** (`parseXXXX()`).

这样, 不断地递归调用, 就完成了整个编程语言的 **Parser** (语法解析器)! 而这个解析器就叫做**递归下降解析器** (因为是按照 EBNF 从最开始的语句一直解析到最小的 Token, 这个过程是递归下降的, 所以叫递归下降解析器).

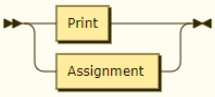
如果是你手写的, 不是机器生成的, 就叫 **手写递归下降解析器** 啦, 没错, 这个过程可以机器自动完成, 但为了给大家详细讲解, 我们并没有用 Flex/Bison 等自动化代码生成器 (大部分教材都用这个, 结果大家看得云里雾里很难受), 而是选择了手写.

等会, 那 `LookAhead()` 是用来干什么的呢?

看这个例子:

```
Statement       ::= Print | Assignment
Print           ::= "print" "(" Ignored Variable Ignored ")" Ign
Assignment      ::= Variable Ignored "=" Ignored String Ignored
Variable        ::= "$" Name Ignored
...
```

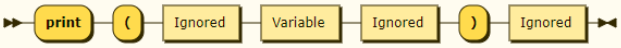
Statement:




```
Statement
        ::= Print
          | Assignment
```

no references

Print:

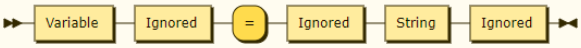



```
Print    ::= 'print' '(' Ignored Variable Ignored ')' Ignored
```

referenced by:

- Statement

Assignment:

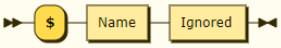



```
Assignment
        ::= Variable Ignored '=' Ignored String Ignored
```

referenced by:

- Statement

Variable:




```
Variable ::= '$' Name Ignored
```

referenced by:

- Assignment
- Print

这次我们要解析 Statement, 我们可以看到 Statement 可以是 Print 或 Assignment. 那怎么判断究竟是哪个呢? 可以这样:

```go
func parseStatement() () {
    switch LookAhead() {
    // "print"
    case TOKEN_PRINT:
        return parsePrint()
    // "$"
    case TOKEN_VAR_PREFIX:
        return parseAssignment()
    default:
        return nil, errors.New("parseStatement(): unknown Statem
    }
}
```

我们先用了 `LookAhead()` 来看看下一个 Token 是什么, 如果是 `TOKEN_PRINT` ("print"), 那么肯定就是 `Print` 语句了! 我们就调用 `parsePrint()` .

如果是 `TOKEN_VAR_PREFIX` ("$"), 那么就是 `Assignment` 了! 因为 `Assignment` 的第一个元素是 `Variable` , `Variable` 的第一个元素正好是 "$" (`TOKEN_VAR_PREFIX`). 是不是再次理解了什么是递归下降?

好了, 太长估计大家也消化不了, 所以每次的文章我会控制在 4000 字左右. 那么, 本篇告一段落. 我们下篇就要实现这两个重要的函数了. 祝你身体健康, 再见.

牙医教你 450 行代码自制编程语言 - 3, 实现 Lexer 上篇.md

@version 20210104:1
@author karminski work.karminski@outlook.com

上一篇 牙医教你 450 行代码自制编程语言 - 2, 两个魔法就可以实现永动机, 简单讲了构成 Lexer & Parser 的元素. 本期我们就要开始实现 Lexer了.

本教程的所有代码都可以在 https://github.com/karminski/pineapple 找到.

另外, KevinXuxuxu 同学看了我的 demo 后, 还实现了一个 Python 版本的, 地址在这里: https://github.com/KevinXuxuxu/pineapple-py

再次推荐这本书, 本教程就是类似这本书的简化版本, 想要仔细学习的话可以考虑看原作:

- 《自己动手实现Lua：虚拟机、编译器和标准库》

实现 Lexer 数据结构

我们直接来看 Lexer 的结构和 New 函数:

```
type Lexer struct {
    sourceCode          string
    lineNum             int
    nextToken           string
    nextTokenType       int
    nextTokenLineNum    int
}

func NewLexer(sourceCode string) *Lexer {
    return &Lexer{sourceCode, 1, "", 0, 0} // start at line 1 in
}
```

其中, `sourceCode` 就是我们的源代码, 直接读取源代码文件并输入进来就可以了.

然后 `lineNum` 用于记录当前执行到的代码的行号.

`nextToken` 即下一个 Token 的内容, `nextTokenType` 则是下一个 Token 的类型, 对应我们定义的 Token 类型常量. `nextTokenLineNum` 是下一个 Token 的行号.

为什么要这么设计? 其实本质上 Lexer 是一个状态机, 他只要能处理**当前状态**和**跳到下一个状态**, 就可以一直工作下去了. 我们接下来演示它是如何工作的.

看看是什么Token?

我们要实现的一个最重要的函数就是 **看看当前字符是什么Token**, 它的功能是查看当前字符, 然后识别是什么 Token, 实现是这样的:

```
func (lexer *Lexer) MatchToken() (lineNum int, tokenType int, to
    // check token
    switch lexer.sourceCode[0] {
    case '$' :
        lexer.skipSourceCode(1)
        return lexer.lineNum, TOKEN_VAR_PREFIX, "$"
    case '(' :
        lexer.skipSourceCode(1)
        return lexer.lineNum, TOKEN_LEFT_PAREN, "("
    case ')' :
        lexer.skipSourceCode(1)
        return lexer.lineNum, TOKEN_RIGHT_PAREN, ")"
    case '=' :
        lexer.skipSourceCode(1)
        return lexer.lineNum, TOKEN_EQUAL, "="
    case '"' :
        if lexer.nextSourceCodeIs("\"\"") {
            lexer.skipSourceCode(2)
            return lexer.lineNum, TOKEN_DUOQUOTE, "\"\""
        }
        lexer.skipSourceCode(1)
        return lexer.lineNum, TOKEN_QUOTE, "\""
    }

    // unexpected symbol
    err := fmt.Sprintf("MatchToken(): unexpected symbol near '%q
    panic(err)

    return
}
```

首先我们用 `switch lexer.sourceCode[0] {` 来实现取出当前字符, 然后用 `case '$' :` 这样的匹配来匹配 Token.

如果命中, 则返回当前行号, Token 类型, 和命中的字符, 即 `return lexer.lineNum, TOKEN_VAR_PREFIX, "$"` .

注意中间还有一行 `lexer.skipSourceCode(1)`，这个函数用来跳过指定长度的字符. 我们匹配完毕，当然要跳过当前的 Token，然后就可以匹配下一个了.

`skipSourceCode()` 的实现也很简单，是这样的:

```
func (lexer *Lexer) skipSourceCode(n int) {
    lexer.sourceCode = lexer.sourceCode[n:]
}
```

用了 Go 语言的 Slice 切片操作的小技巧.

那么多字符 Token 怎么办?

但是，这只能匹配单个字符的 Token，多个字符的 Token 该怎么办呢? 比如 `Name` Token，其实可以这样:

```
...
// check multiple character token
if lexer.sourceCode[0] == '_' || isLetter(lexer.sourceCode[0]) {
    token := lexer.scanName()
    if tokenType, isMatch := keywords[token]; isMatch {
        return lexer.lineNum, tokenType, token
    } else {
        return lexer.lineNum, TOKEN_NAME, token
    }
}
...

func isLetter(c byte) bool {
    return c >= 'a' && c <= 'z' || c >= 'A' && c <= 'Z'
}
```

我们判断如果是下划线 `lexer.sourceCode[0] == '_'` 或字母开头的 `isLetter(lexer.sourceCode[0])`，我们就运行 `token := lexer.scanName()` 把 Name 扫描出来.

`isLetter()` 实现也很简单，直接判断字符是否在大小写字符范围就可以了.

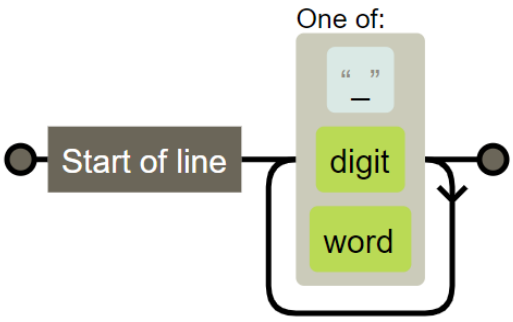
那么，重要的 `scanName()` 是怎么实现的呢? 答案是正则表达式.

```
var regexName = regexp.MustCompile(`^[_\d\w]+`)

func (lexer *Lexer) scanName() string {
    return lexer.scan(regexName)
}

func (lexer *Lexer) scan(regexp *regexp.Regexp) string {
    if token := regexp.FindString(lexer.sourceCode); token != ""
        lexer.skipSourceCode(len(token))
        return token
    }
    panic("unreachable!")
    return ""
}
```

我们的正则表达式是 `^[_\d\w]+`:



可以看到必须是行的开始, 然后是下划线 `_`, 数字 `\d`, 或者字符 `\w` 的循环 (`+` 代表一个或多个).

由于我们规定了 `Name ::= [_A-Za-z][_0-9A-Za-z]*` 只能是下划线或大小写字母开头, 我们在之前的 `if lexer.sourceCode[0] == '_' || isLetter(lexer.sourceCode[0]) {` 已经定义好了这一点. 所以不会出现正则扫描到数字开头的 Token 的情况.

多字符的实现方法 2

```
...
case '"' :
    if lexer.nextSourceCodeIs("\"\"") {
        lexer.skipSourceCode(2)
        return lexer.lineNum, TOKEN_DUOQUOTE, "\"\""
    }
    lexer.skipSourceCode(1)
    return lexer.lineNum, TOKEN_QUOTE, "\""
...

func (lexer *Lexer) nextSourceCodeIs(s string) bool {
    return strings.HasPrefix(lexer.sourceCode, s)
}
```

我们这里还演示了第二种判断多字符的方式, `nextSourceCodeIs()` 利用 `strings` 库的 `strings.HasPrefix()` 方法扫描了空字符串 (两个双引号 `""`) 的情况.

这种判断方式比正则表达式性能要好一些, 适合于确定性的 Token, 比如例子中扫描的就是两个双引号组成的代表空字符串 的 Token.

多字符的实现方法 3

那么不确定长度的 Token 怎么办? 就比如 `String ::= '"' StringCharacter '"'` 这种情况, 一大长串 用双引号包裹的字符串 该怎么办? 我们又实现了这样一个函数:

```
func (lexer *Lexer) scanBeforeToken(token string) string {
    s := strings.Split(lexer.sourceCode, token)
    if len(s) < 2 {
        panic("unreachable!")
        return ""
    }
    lexer.skipSourceCode(len(s[0]))
    return s[0]
}
```

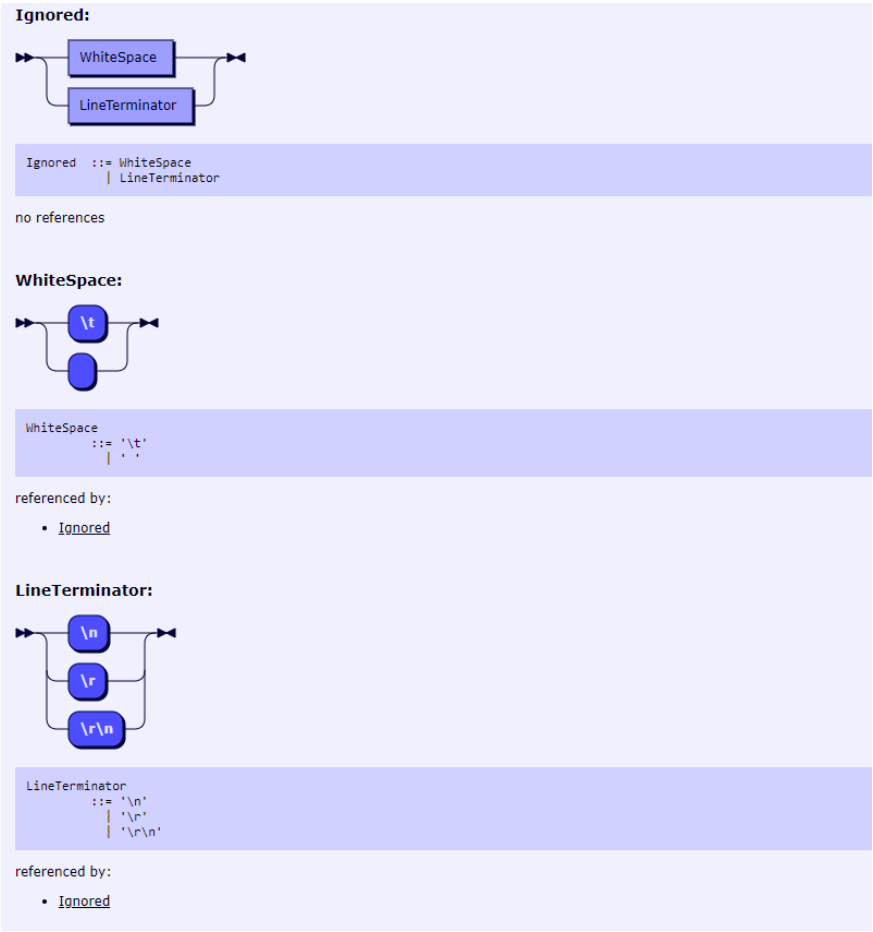
我们用了 `strings.Split()` 方法, 当检测到目标标记后, 就把字符串 分割开.

比如 `AAAA"BBBBBCCCC` 用 `"` 分割, 就会被分成 `AAAA` 和 `BBBBBCCCC` . 然后我们返回我们要的第一个就行了. 当然, 也可以逐个字符扫描, 直到检测到目标就停止, 这样性能会更好些, 感兴趣的同学可以改写下. 我这里犯懒了所以直接拍脑袋写了这个.

还有其他情况吗?

有的, 我们代码要有空格和换行的, 因此我们还需要一个 `Ignored` Token. 它的 EBNF 定义是这样的:

```
Ignored        ::= WhiteSpace | LineTerminator
WhiteSpace     ::= '\t' | ' ' /* ASCII: \t | Space, Horizontal T
LineTerminator ::= '\n' | '\r' | '\r\n'   /* ASCII: \n | \r\n |
```



简单来讲, 就是 tab, 空格, 和 换行 (`'\n' | '\r' | '\r\n'`) 都是 Ignored Token.

那么, 我们来这样检测 Ignored Token:

```
...
// check ignored
if lexer.isIgnored() {
    return lexer.lineNum, TOKEN_IGNORED, "Ignored"
}
...

func (lexer *Lexer) isIgnored() bool {
    isIgnored := false
    // target pattern
    isNewLine := func(c byte) bool {
        return c == '\r' || c == '\n'
    }
    isWhiteSpace := func(c byte) bool {
        switch c {
        case '\t', '\n', '\v', '\f', '\r', ' ':
            return true
        }
        return false
    }
    // matching
    for len(lexer.sourceCode) > 0 {
        if lexer.nextSourceCodeIs("\r\n") || lexer.nextSourceCod
            lexer.skipSourceCode(2)
            lexer.lineNum += 1
            isIgnored = true
        } else if isNewLine(lexer.sourceCode[0]) {
            lexer.skipSourceCode(1)
            lexer.lineNum += 1
            isIgnored = true
        } else if isWhiteSpace(lexer.sourceCode[0]) {
            lexer.skipSourceCode(1)
            isIgnored = true
        } else {
            break
        }
    }
    return isIgnored
}
```

还是简单的字符匹配, 然后跳过即可. 我这个实现判断了 Ignored Token, 顺便还跳过了一些其他的不需要的空白字符.

好了, 本期内容有些长, 大家可能要多消化一下, 而且本期内容是最难的, 下期开始就会变得特别简单了. 敬请期待.

牙医教你 450 行代码自制编程语言 - 4, 实现 Lexer 下篇.md

@version 20210104:1
@author karminski work.karminski@outlook.com

上一篇 牙医教你 450 行代码自制编程语言 - 3, 实现 Lexer 上篇, 讲了 Lexer 的 Token 识别方法, 接下来我们就要把它拼装起来了!

本教程的所有代码都可以在 https://github.com/karminski/pineapple 找到.

再次推荐这本书, 本教程就是类似这本书的简化版本, 想要仔细学习的话可以考虑看原作:

- 《自己动手实现Lua：虚拟机、编译器和标准库》

糖糖糖!

到目前为止, 我们已经实现了 Lexer 的核心功能, 接下来我们需要一些糖, 将他们包裹起来, 作为一种抽象, 帮助我们后续更好地构建 Parser.

抽象能力是很重要的能力, 他能直接能决定构建上层逻辑的复杂度. 抽象得越好, 上层实现越简洁, 越易于理解. 原作这一点实现的很好. 好的代码是易于理解的, 而不是只有编写者自己才能看懂的.

我们首先要实现的就是 `GetNextToken()`:

```
func (lexer *Lexer) GetNextToken() (lineNum int, tokenType int,
    // next token already loaded
    if lexer.nextTokenLineNum > 0 {
        lineNum                = lexer.nextTokenLineNum
        tokenType              = lexer.nextTokenType
        token                  = lexer.nextToken
        lexer.lineNum          = lexer.nextTokenLineNum
        lexer.nextTokenLineNum = 0
        return
    }
    return lexer.MatchToken()

}
```

该函数其实就是上一篇中 `MatchToken()` 的封装, 返回值是一样的.

但不同之处在于, 我们判断了一下 `nextTokenLineNum` 的值, 这个值表面上是下一个 Token 的行号. 但实际上, 这个值一旦大于 0, 即意味着 nextTokenType 已经加载了, 我们获取下一个 Token 的时候 (即调用 `GetNextToken()`) 就可以直接返回上次已经加载好的值. 避免了再次调用 `MatchToken()` 浪费性能.

举例来讲, 比如我们调用了 `LookAhead()` 来获取下一个 Token 是什么, 判断是我们要的 Token 之后, 必然会使用 `NextTokenIs(TOKEN)` 进行断言.

例如解析 Variable (`Variable ::= "$" Name Ignored`), 需要先判断是美元符号 `$` , 然后是 `Name` :

```
token := LookAhead()
if token == TOKEN_VAR_PREFIX {
    NextTokenIs(TOKEN_VAR_PREFIX)
    parseName()
}
```

这样在 `LookAhead()` 中调用了 `GetNextToken()` , 调用 `NextTokenIs(TOKEN_VAR_PREFIX)` 的时候, 就不用再次调用 `GetNextToken()` 了. 存储了上一次的执行结果, 提升了性能.

最后三个函数

至此, 所有的基础函数做完了, 我们要实现三个用于抽象的函数. 首先是 `NextTokenIs()` :

```
func (lexer *Lexer) NextTokenIs(tokenType int) (lineNum int, tok
    nowLineNum, nowTokenType, nowToken := lexer.GetNextToken()
    // syntax error
    if tokenType != nowTokenType {
        err := fmt.Sprintf("NextTokenIs(): syntax error near '%s
        panic(err)
    }
    return nowLineNum, nowToken
}
```

这个函数用于断言下一个 Token 是什么. 并且由于内部执行了 `GetNextToken()` , 所以游标会自动向前移动.

然后是 `LookAhead()` :

```
func (lexer *Lexer) LookAhead() int {
    // lexer.nextToken* already setted
    if lexer.nextTokenLineNum > 0 {
        return lexer.nextTokenType
    }
    // set it
    nowLineNum                := lexer.lineNum
    lineNum, tokenType, token := lexer.GetNextToken()
    lexer.lineNum              = nowLineNum
    lexer.nextTokenLineNum     = lineNum
    lexer.nextTokenType        = tokenType
    lexer.nextToken            = token
    return tokenType
}
```

我们也见过很多次了, 这个函数用于返回下一个 Token 是什么, 不过它并不会将游标向 前移动 (准确的说是移动了, 然后又移了回来, 在 `lineNum, tokenType, token := lexer.GetNextToken()` 后面那一堆代码.).

头部的 `if lexer.nextTokenLineNum > 0 {` 则像我们之前说的, 检测是否曾 经获取过下一个 Token, 如果获取过, 直接返回 `lexer` 结构体内部缓存好的结果.

最后是 `LookAheadAndSkip()` :

```
func (lexer *Lexer) LookAheadAndSkip(expectedType int) {
    // get next token
    nowLineNum                := lexer.lineNum
    lineNum, tokenType, token := lexer.GetNextToken()
    // not is expected type, reverse cursor
    if tokenType != expectedType {
        lexer.lineNum              = nowLineNum
        lexer.nextTokenLineNum     = lineNum
        lexer.nextTokenType        = tokenType
        lexer.nextToken            = token
    }
}
```

它相对于 `LookAhead()` 多了个参数, 仍然是先看一下下一个 Token 是什么, 如果跟输入的相同就会跳过去, 如果不同则不会跳过去. 这个函数是为了 `Ignored` Token 特别定制的.

趁热打铁

好的, 接下来我们来迅速熟悉下我们定义的这些函数的使用方法, 我们直接来看 `Print` 语句是如何解析的:

```
// Print ::= "print" "(" Ignored Variable Ignored ")" Ignored
func parsePrint(lexer *Lexer) (*Print, error) {
    var print Print
    var err   error

    print.LineNum = lexer.GetLineNum()
    lexer.NextTokenIs(TOKEN_PRINT)
    lexer.NextTokenIs(TOKEN_LEFT_PAREN)
    lexer.LookAheadAndSkip(TOKEN_IGNORED)
    if print.Variable, err = parseVariable(lexer); err != nil {
        return nil, err
    }
    lexer.LookAheadAndSkip(TOKEN_IGNORED)
    lexer.NextTokenIs(TOKEN_RIGHT_PAREN)
    lexer.LookAheadAndSkip(TOKEN_IGNORED)
    return &print, nil
}
```

我们先不要管看不懂的细节, 直接来看使用了我们刚才定义的那些函数的部分, 首先 `Print` 语句的开头必须是字符 `"print"`, 我们已经定义好了 Token (`TOKEN_PRINT`), 对于这种确定的场景, 直接用 `NextTokenIs()` 来进行断言即可.

同样, 然后的左括号 (`TOKEN_LEFT_PAREN`) 也直接断言, 然后就是 `Ignored` 了, 我们的语法允许函数里面有空格或者换行. 即这样写也是合法的:

```
print(
    $a
)
```

那么, 这里我们就需要用 `LookAheadAndSkip(TOKEN_IGNORED)` 来判断接下来是不是 `Ignored`, 如果是就跳过, 如果不是就不跳过.

剩下就没有新鲜的了. 我们已经重复很多次了.

到此为止

全部的 lexer.go 文件见 Github:
https://github.com/karminski/pineapple/blob/main/src/lexer.go.

牙医教你 450 行代码自制编程语言 - 5, 递归下降语法解析器.md

@version 20210105:1
@author karminski work.karminski@outlook.com

上一篇 牙医教你 450 行代码自制编程语言 - 4, 实现 Lexer 下篇, 我们终于完成了 Lexer, 本篇我们就来讲 Parser 怎么实现.

本教程的所有代码都可以在 https://github.com/karminski/pineapple 找到.

再次推荐这本书, 本教程就是类似这本书的简化版本, 想要仔细学习的话可以考虑看原作:

- 《自己动手实现Lua：虚拟机、编译器和标准库》

按图索骥

我们再次把已经看到烦的 EBNF 贴在这里, 接下来用这个, 我们就能飞速的构建 Parser 了. 最难的 Lexer 我们已经搞定, 接下来请系好安全带, 我们开始发车了!

```
SourceCharacter ::=  #x0009 | #x000A | #x000D | [#x0020-#xFFFF]
Name            ::= [_A-Za-z][_0-9A-Za-z]*
StringCharacter ::= SourceCharacter - '"'
String          ::= '"' '"' Ignored | '"' StringCharacter '"' Ig
Variable        ::= "$" Name Ignored
Assignment      ::= Variable Ignored "=" Ignored String Ignored
Print           ::= "print" "(" Ignored Variable Ignored ")" Ign
Statement       ::= Print | Assignment
SourceCode      ::= Statement+
```

Name

首先是 `Name` 语句, 很简单, 我们的 Lexer 能自动解析出 `Name` Token, 因此:

```
// Name ::= [_A-Za-z][_0-9A-Za-z]*
func parseName(lexer *Lexer) (string, error) {
    _, name := lexer.NextTokenIs(TOKEN_NAME)
    return name, nil
}
```

5 行搞定! 甚至还包含了一行注释! 我们只需要断言 `parseName()` 的时候, 接下来的 Token 必须是 `Name` 就可以了! 如果不是怎么办? 不是那就是输入的代码语法错误了! 与我们无关~

然后返回的真正的 name 是个 string 类型, 直接返回就好了.

String

`String` 复杂一些, 它要么是空字符串, 要么是双引号包裹的字符串 `StringCharacter`. 而 `StringCharacter` 是由 不包含双引号的 `SourceCharacter` 构成.

```
// String ::= '"' '"' Ignored | '"' StringCharacter '"' Ignored
func parseString(lexer *Lexer) (string, error) {
    str := ""
    switch lexer.LookAhead() {
    case TOKEN_DUOQUOTE:
        lexer.NextTokenIs(TOKEN_DUOQUOTE)
        lexer.LookAheadAndSkip(TOKEN_IGNORED)
        return str, nil
    case TOKEN_QUOTE:
        lexer.NextTokenIs(TOKEN_QUOTE)
        str = lexer.scanBeforeToken(tokenNameMap[TOKEN_QUOTE])
        lexer.NextTokenIs(TOKEN_QUOTE)
        lexer.LookAheadAndSkip(TOKEN_IGNORED)
        return str, nil
    default:
        return "", errors.New("parseString(): not a string.")
    }
}
```

我们来按照这两种情况书写, 既然有两种情况, 我们只好先 `LookAhead()` 一下, 看看接下来的是什么 Token.

首先, 如果是 `TOKEN_DUOQUOTE` 空字符串 的情况, 那么就断言是空字符串 `lexer.NextTokenIs(TOKEN_DUOQUOTE)`. 然后跳过后面的 `Ignored` Token 即: `lexer.LookAheadAndSkip(TOKEN_IGNORED)`.

然后如果只是个单引号 `TOKEN_QUOTE`, 那接下来就是字符串 本体了, 我们用 `scanBeforeToken()` 扫描直到遇到下一个单引号. 然后断言双引号, 最后跳过 `Ignored`.

如果都不是, 那我们直接返回错误, 这不是 string. 不用怀疑, 肯定是出语法错误了, 让写代码的人自己检查去吧. 哈哈哈, 是不是有种翻身了的感觉?

Variable

同理, `Variable` 是美元符号 `TOKEN_VAR_PREFIX` 和 `Name` 构成:

```
// Variable ::= "$" Name Ignored
func parseVariable(lexer *Lexer) (*Variable, error) {
    var variable Variable
    var err      error

    variable.LineNum = lexer.GetLineNum()
    lexer.NextTokenIs(TOKEN_VAR_PREFIX)
    if variable.Name, err = parseName(lexer); err != nil {
        return nil, err
    }
    lexer.LookAheadAndSkip(TOKEN_IGNORED)
    return &variable, nil
}
```

这里有些不同, 我们声明了 `Variable` 类型, 它是这样的:

```
type Variable struct {
    LineNum int
    Name    string
}
```

可见很忠实的还原了 EBNF 定义的结构, `Variable` 包含了 `Name` .

回到代码, 我们这里还用 `variable.LineNum = lexer.GetLineNum()` 获取了行号, 这个主要是提供给源代码编写者报错用的. 可以不必太在意. 后续优化的话可以给到精确的行号和列号.

这里除了开头是美元符号, 我们用了 `variable.Name, err = parseName(lexer)` . 这样就把解析 `Name` 的工作交给了刚刚完成的 `parseName()` 方法. 这个过程就是递归下降的过程. 把很大的目标拆分成递归形式的小目标, 可以让我们更好的完成工作.

SourceCode & Statements

由于 `parseAssignment()` , `parsePrint()` 很相似, 我们就不赘述了. 我们只把他们的数据结构定义写下来:

```
type Assignment struct {
    LineNum   int
    Variable *Variable
    String    string
}

type Print struct {
    LineNum   int
    Variable *Variable
}
```

接下来看 `parseStatement()`:

```
// Statement ::= Print | Assignment
func parseStatement(lexer *Lexer) (Statement, error) {
    switch lexer.LookAhead() {
    case TOKEN_PRINT:
        return parsePrint(lexer)
    case TOKEN_VAR_PREFIX:
        return parseAssignment(lexer)
    default:
        return nil, errors.New("parseStatement(): unknown Statem
    }
}
```

`Statement` 可以是 `Print` 或 `Assignment`, 需要先 `LookAhead()` 一下, 如果是 `TOKEN_PRINT` 那么就解析 `Print`, 如果是 `TOKEN_VAR_PREFIX` 就解析 `Assignment`.

然后我们看 `SourceCode` 的定义 `SourceCode ::= Statement+` 它可以是多个 `Statement`. 所以我们这样定义:

```
type SourceCode struct {
    LineNum      int
    Statements []Statement
}
```

对, 我们定义了一个 `Statements []Statement` 一个盛放 Statement 的 Slice (不会 Go 的同学理解成数组也行). 然后 `Statement` 是这样的:

```
type Statement interface{}

var _ Statement = (*Print)(nil)
var _ Statement = (*Assignment)(nil)
```

由于 `Statement` 可以是不同数据类型，因此我们定义成了 `interface{}`。然后 Go 的初学者可能看不懂接下来的是什么。

简单来讲，`var _ Statement = (*Print)(nil)` 可以简单理解为 `*Print` 属于 `Statement`。这样 `Statement` 的具体实现就被限定到了 `*Print`，`*Assignment` 这两种类型。对后续的编写安全是一种很好的保障，否则 interface{} 满天飞，很容易出错。

定义完毕之后，我们来看如何搞定 `SourceCode ::= Statement+` 中的 `Statement+`：

```
func parseStatements(lexer *Lexer) ([]Statement, error) {
    var statements []Statement

    for !isSourceCodeEnd(lexer.LookAhead()) {
        var statement Statement
        var err       error
        if statement, err = parseStatement(lexer); err != nil {
            return nil, err
        }
        statements = append(statements, statement)
    }
    return statements, nil
}

func isSourceCodeEnd(token int) bool {
    if token == TOKEN_EOF {
        return true
    }
    return false
}
```

我们设置了个有条件的死循环，`for !isSourceCodeEnd(lexer.LookAhead()) {` 不断向前判断 Token，直到遇到源码末尾。判断末尾的 `isSourceCodeEnd()` 也很简单，直接判断是不是 `TOKEN_EOF` 就可以了。

然后不断 `statement, err = parseStatement(lexer)` 然后把结果放到 Slice 里面 `statements = append(statements, statement)`。这样，就完成了！

长舒一口气

最后，我们把调用方法封装一下：

```
func parse(code string) (*SourceCode, error) {
    var sourceCode *SourceCode
    var err         error

    lexer := NewLexer(code)
    if sourceCode, err = parseSourceCode(lexer); err != nil {
        return nil, err
    }
    lexer.NextTokenIs(TOKEN_EOF)
    return sourceCode, nil
}
```

到此, 我们就完成了编译器的前端 (对, 相对后端的都叫前端. 编译器的前端一般就指 Lexer 和 Parser).

Parser 的源码详见: https://github.com/karminski/pineapple/blob/main/src/parser.go.

相关数据结构的定义详见: https://github.com/karminski/pineapple/blob/main/src/definition.go.

下一节我们就要开始写编译器的后端了! 就可以让代码跑起来了! 我们下一篇再见~

牙医教你 450 行代码自制编程语言 - 6, 后端.md

@version 20210105:1
@author karminski work.karminski@outlook.com

上一篇 牙医教你 450 行代码自制编程语言 -5, 递归下降 语法解析器, 我们终于完成了编译器前端, 本篇我们就来讲编译器后端怎么实现.

本教程的所有代码都可以在 https://github.com/karminski/pineapple 找到.

再次推荐这本书, 本教程就是类似这本书的简化版本, 想要仔细学习的话可以考虑看原作:

- 《自己动手实现Lua：虚拟机、编译器和标准库》

我们的目的是什么?

回想一下, 我们的这门 语言是这样的:

```
$a = "pen pineapple apple pen."
print($a)
```

把一个字符串变量赋值, 然后打印出来. 由于上下文只有这一层, 因此变量的存储很简单, 我们给它存到一个大 map 里面, 需要的时候去查找就好了.

```
type GlobalVariables struct {
    Variables  map[string]string
}

func NewGlobalVariables() *GlobalVariables {
    var g GlobalVariables
    g.Variables = make(map[string]string)
    return &g
}
```

定义的 `GlobalVariables` 里面装了个 `map[string]string` 结构. 变量名称作为索引, 变量值作为值就搞定了.

`NewGlobalVariables` 则作为全局变量存储的初始化方法.

执行!

接下来我们就编写执行代码的本体函数:

```
func Execute(code string) {
    var ast *SourceCode
    var err  error

    g := NewGlobalVariables()

    // parse
    if ast, err = parse(code); err != nil {
        panic(err)
    }

    // resolve
    if err = resolveAST(g, ast); err != nil {
        panic(err)
    }
}
```

首先这个函数需要传入 `code`, 即源代码. 然后交给 `Parser`, 执行 `parser`. 这样就得到了 AST (抽象语法树). AST 是这个样子的:

```
(*pineapple.SourceCode)({
    LineNum: (int) 1,
    Statements: ([]pineapple.Statement) (len=2 cap=2) {
        (*pineapple.Assignment)({
            LineNum: (int) 1,
            Variable: (*pineapple.Variable)({
                LineNum: (int) 1,
                Name: (string) (len=1) "a"
            }),
            String: (string) (len=24) "pen pineapple apple pen."
        }),
        (*pineapple.Print)({
            LineNum: (int) 2,
            Variable: (*pineapple.Variable)({
                LineNum: (int) 3,
                Name: (string) (len=1) "a"
            })
        })
    }
})
```

有了这个语法树的蓝图, 我们接下来就可以编写相关的 resolve 方法了.

解析 AST

首先我们看到 AST 的顶层就是 `*pineapple.SourceCode` 而我们之前 EBNF 的定义 `SourceCode ::= Statement+`, 它是由 多个 `Statement` 构成的. 我们也可以看到, 的确是这样: `Statements: ([]pineapple.Statement) (len=2 cap=2)` .

我们仍然用类似递归下降 的思路去完成解析器:

```
func resolveAST(g *GlobalVariables, ast *SourceCode) error {
    if len(ast.Statements) == 0 {
        return errors.New("resolveAST(): no code to execute, ple
    }
    for _, statement := range ast.Statements {
        if err := resolveStatement(g, statement); err != nil {
            return err
        }
    }
    return nil
}
```

遍历 `ast.Statements` 然后将得到的 `statement` 交给后续的 `resolveStatement` 方法就可以了.

解析 Statement

同样, `Statement` 是由 两种类型 `*Print` 和 `*Assignment` 构成的, 那么我们这样做:

```
func resolveStatement(g *GlobalVariables, statement Statement) e
    if assignment, ok := statement.(*Assignment); ok {
        return resolveAssignment(g, assignment)
    } else if print, ok := statement.(*Print); ok {
        return resolvePrint(g, print)
    } else {
        return errors.New("resolveStatement(): undefined stateme
    }
}
```

我们直接对传入的 `statement` 进行断言, 如果是 `*Assignment` 那就执行 `resolveAssignment()` , 如果是 `*Print` , 那就执行 `resolvePrint()` . 是不是简单极了!

解析 Assignment

Assignment 是给变量赋值的过程. 而我们已 经定义好了装变量的全局变量表 `GlobalVariables` . 只要装进去就行了:

```
func resolveAssignment(g *GlobalVariables, assignment *Assignmen
    varName := ""
    if varName = assignment.Variable.Name; varName == "" {
        return errors.New("resolveAssignment(): variable name ca
    }
    g.Variables[varName] = assignment.String
    return nil
}
```

这里判断变量名是否不为空, 如果不为空, 那就装进去 `g.Variables[varName] = assignment.String` .

解析 Print

Assignment 是给变量赋值的过程. 而 Print 是打印变量的过程, 那么很容易就推断出, 实现方法就是在全局变量表找到目标变量, 然后打印 出它的值就行了:

```
func resolvePrint(g *GlobalVariables, print *Print) error {
    varName := ""
    if varName = print.Variable.Name; varName == "" {
        return errors.New("resolvePrint(): variable name can NOT
    }
    str := ""
    ok  := false
    if str, ok = g.Variables[varName]; !ok {
        return errors.New(fmt.Sprintf("resolvePrint(): variable
    }
    fmt.Print(str)
    return nil
}
```

查找 `str, ok = g.Variables[varName]` , 打印 `fmt.Print(str)` . 秒杀~

还缺点什么?

当然, 我们还却一个入口 文件:

```
package main

import (
    "os"
    "fmt"
    "io/ioutil"
    "github.com/karminski/pineapple/src"
)

func main() {

    // read file
    args := os.Args
    if len(args) != 2 {
        fmt.Printf("Usage: %s filename\n", args[0])
        return
    }
    filename := args[1]
    code, err := ioutil.ReadFile(filename)
    if err != nil {
        fmt.Printf("Error reading file: %s\n", filename)
        return
    }

    // execute
    pineapple.Execute(string(code))

}
```

读取文件, 然后执行代码 `pineapple.Execute(string(code))` 就可以了!

激动人心的时刻:

```
karminski@DESKTOP-5K6J40R /works/github.com/public/pineapple/examples/pineapple
$ go build

karminski@DESKTOP-5K6J40R /works/github.com/public/pineapple/examples/pineapple
$ ./pineapple.exe hello-world.pineapple
pen pineapple apple pen.
```

恭喜你! 到此为止, 你已经简单掌握了**编译器前端**和一个**解释器后端**的编写方法了! 感兴趣的同学可以 fork 这个项目, 试试给它添加你想要的功能, 比如实现字符串拼接, 实现数值类型变量, 实现四则运算, 甚至实现函数! 这样一个真正图灵完备的语言就实现了!

当然, 如果你遇到了问题, 这本书是你最好的朋友.

牙医教你 450 行代码自制编程语言 - 7, 后续该如何学习编译原理.md

@version 20210105:1
@author karminski work.karminski@outlook.com

上一篇 牙医教你 450 行代码自制编程语言 - 6, 后端, 我们终于完成了整个编译器, 本篇则是一点点的经验分享, 讲一讲该如何系统学习编译原理.

本教程的所有代码都可以在 https://github.com/karminski/pineapple 找到.

Why

有同学会问, 学这玩意真的不是屠龙之术么? 答案是不是的. 编译原理在很多工程实践中都可以用得到. 我来举几个例子:

我刚开始工作的时候, 是在 so.com, 360 搜索工作. 当时要收录一些新闻, 我想做一个正文抽取工具. 我的一个思路就是, 分析 HTML 中的元素, 然后判断其中哪个元素视觉面积最大, 那肯定就是正文了.

而想要实现这一点, 需要一个 HTML 和 CSS 的 Parser, 然后还要有个后端去解析语法树, 找到视觉面积最大的节点.

当然, 这个项目最后是用 NLP 做的, 因为用 NLP 很成熟. 我的想法遇到的最大问题是, 并不一定视觉元素最大的就是正文. 但这个思路本身提供了一种新的正文抽取的可能性. 作为未来的研究是非常不错的. 比如, 如果它的正文面积不够大, 我们就认为是垃圾文章. 做一个页面排版评分器. 又或者作为一种轻量化的无头浏览器 (Headless Brwoser) 去完成其他工作.

学习新技术并不单只是去应用, 去完成业务. 更重要的是给你一个更高的视角, 让你知道会有超脱于传统的, 更有效的, 更高效率的方法, 让你去了解业界的变革和方向. 这也是不断学习的意义与价值.

好的, 例子就举这么多, 那么该怎样去学习呢?

How

首先如果你喜欢看视频的话, 可以直接去 B 站看上面的编译原理相关课程. 上面我粗略看了一眼, 国内比较好的大学的编译原理视频课程都有. 可以放心看.

我主要推荐的是书籍, 而且写作这个专栏的目的也是为了推荐一些我觉得非常棒的书籍和书单. 相对于视频, 书籍更好翻阅和查找, 并且能更快速地学习. 视频的优点在于有讲解, 而随着你的学习深入, 总有一天你会遇到没有相关视频, 只有相关书籍的情况, 这时候没有很好的看书自学能力, 就很难继续学习下去了.

第一本仍然推荐我最推荐的 张秀宏 写的这本《自己动手实现Lua：虚拟机、编译器和标准库》. 如果说 2020 年我看过的最好的书, 那么第二名就是它了. 它从 0 开始 用 Go 语言实现了一个真正可用的 Lua 虚拟机. 它的实现非常凝练, 代码易于理解. 本系列教程也是根据这本书的的思想写作而成. 但这本书注重实现一个 Lua 虚拟机, 而没有系统的讲解编译原理需要的知识, 而且它实现的是个解释器, 因此编译器最难的代码生成部分, 并没有介绍. 所以后续学习还要看别的书.

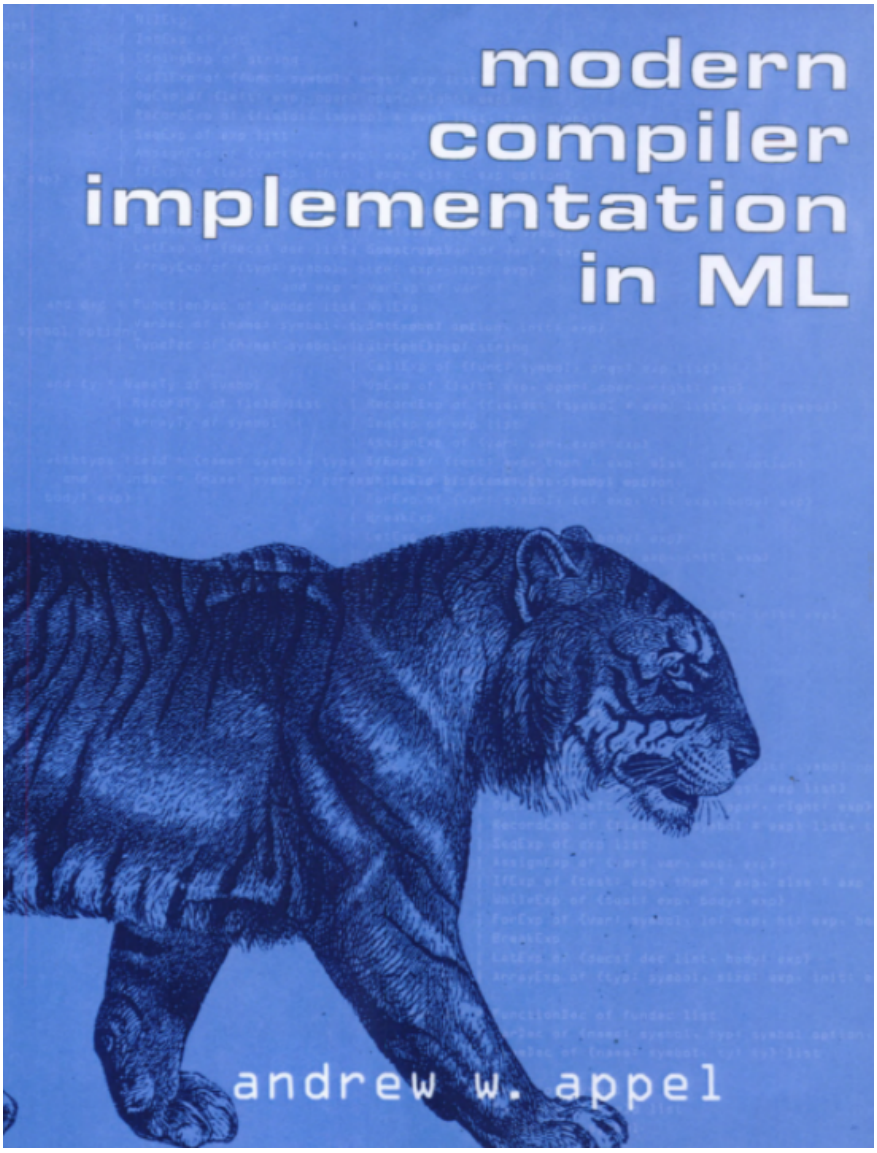
- 《自己动手实现Lua：虚拟机、编译器和标准库》

第二本则是你想系统的学习编译原理推荐的书 《编译器设计(第2版)》. 这本书的优点在于, 它相比传统经典的"龙书", "虎书", 出版时间比较晚, 因此内容也比较新. 而且内容分配我认为也比较合理. 是我最推荐的系统学习编译原理的书籍.

- 《编译器设计(第2版)》

第三本是著名的虎书《现代编译原理 C语言描述》. 这本书其实当你有了一定的编译原理知识后, 作为一个索引是很不错的, 它比这些系统讲解编译原理的书都薄 (不到400页). 因此一些细节不是很翔实. 另外有些同学觉得难受的是, 这本书本来是用 ML 语言写的, 这本 C语言描述 (还有 Java 语言描述), 看起来更像 ML 移植的, 看起来不舒服. 我倒觉得只当作索引就还好啦, 真的用它来实现相关代码没什么必要.

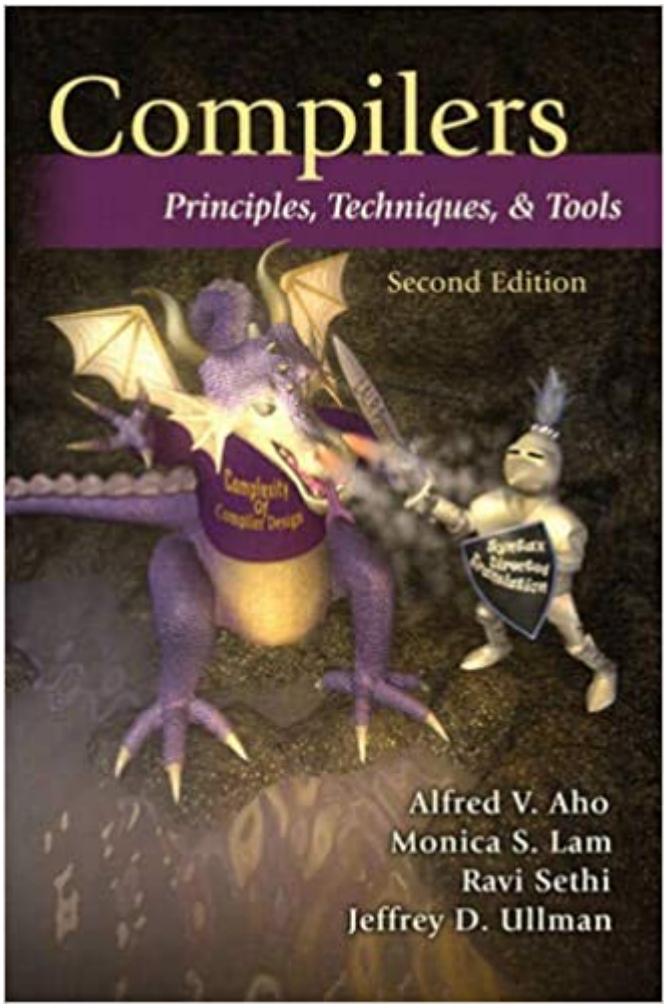
另外, 它之所以叫"虎书", 是因为原书封面是个老虎:



- 《现代编译原理 C语言描述》

第四本果然还是龙书《编译原理(第2版)》. 这本书是国内很多高校的教材. 大家普遍反映它的前端部分写得非常棒. 但是后端部分缺乏现代化的内容 (这本书是2006年出版的). 因此后端的内容可以结合其他书去阅读.

这本书叫龙书当然也是因为它的封面有龙...

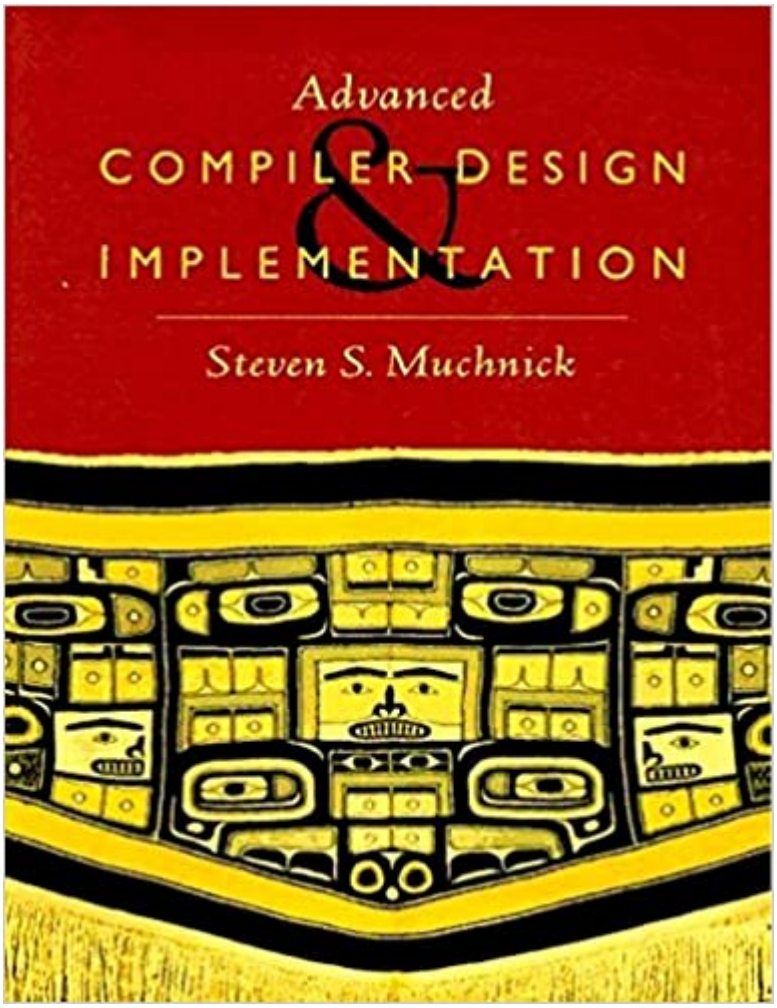


顺便, 这本书的最新第三版也出版了, 不过还没有中文翻译版.

- 《编译原理(第2版)》

第五本那就是鲸书《高级编译器设计与实现》. 这本书全是讲 IR 和后端优化的, 如果你有志从事编译器相关工作, 那么还是需要看一看的.

当然, 这本书叫鲸书是因为原书封面是鲸鱼张大的嘴, 我觉得是比较抽象了...



- 《高级编译器设计与实现》

然后是《垃圾回收的算法与实现》. 这本书主要是讲 GC 的. 对 GC 感兴趣的朋友可以看一看, 普通作为阅读的话, 我觉得掌握三色标记就可以面试的时候跟面试官稍微讲一下了.

- 《垃圾回收的算法与实现》

最后, 这本书什么时候都可以读, 只要你学习计算机科学, 那这本书就跑不了了. 《深入理解计算机系统(原书第3版)》.

如果你没有计算机的基础知识, 那么这本书很可能每一页都会震撼到你.

什么是进程?什么是线程?

什么是CPU上下文?

并发和并行的区别是什么?

CPU是怎么运算数据的?

编译器都做了些什么?

L1/L2 cache 有什么用?

什么是物理地址?页表有什么用?

malloc函数是怎么实现的?

什么是锁?什么是信号量?

这本 CSAPP (Computer Systems: A Programmer's Perspective) 会给你最好的解答.

- 《深入理解计算机系统(原书第3版)》

End

当然, 我并不是专业工作于编译器领域的, 所以更深入的学习还需要各位自己去探索. 这一系列文章如果能激起一个人学习编译原理的兴趣, 哪怕有一个人看了看我的例子, 哪怕有一个人实现了一个简单的编译器, 便是我最大的欣慰了.

书山有路勤为径. 各位共勉.