编译原理课程设计

——类C编译器设计与实现 报告

1652270 冯舜 指导老师：卫志华

# 目的

1. 掌握使用高级程序语言实现一个一遍完成的、简单语言的编译器的方法。
2. 掌握简单的词法分析器、语法分析器、符号表管理、中间代码生成的实现方法。

# 需求分析

## 程序任务范围

使用高级语言实现一个编译器的生成器。具体来说，需要实现词法分析器的生成器、LR(1)（也叫CLR）语法制导的翻译器的生成器。再用这个生成器生成一个类C语言（含函数调用）的编译器，能将源代码编译为中间代码。

程序使用JavaScript实现，因为用它来实现文法符号的属性比较容易。用JavaScript重写了上学期的C++代码。

## 程序输入与输出

**注：未经特殊说明，所有对文件的输入输出默认是UTF-8，Unix格式换行（仅含LF）。请不要使用系统自带的记事本打开文件，而应使用较高级的文本编辑器如Notepad++等。**

### 配置输入

配置部分的输入使用YAML格式。YAML格式方便书写，并且与JavaScript内部对象可以无缝转换，甚至可以以特殊标签的形式向YAML添加JavaScript函数、方法，实现求值器和语义动作。

* 用户提供的词法分析器配置（lex/lex-c-style.yaml）。
  + 词法分析器的配置由多个“词法单元识别器”（Token Recognizer，TR）的配置组成；每个TR分别识别不同类别的词法单元（Token）。
    - 一个TR由一个用户定义的NFA（识别有效字符串）和一个求值器（Evaluator，将词法单元对应的字符串转换为机器内部表示）。NFA的配置细节请参考配置文件。求值器作为JavaScript函数标签插入。
* 用户提供的CLR语法配置（translate/ grammar-c-style.yaml）。包括：
  + 各产生式
    - 产生式的左边、右边符号列表
    - 产生式归约时执行的语义动作
  + 所有终结符（非终结符可以从产生式左边统计得到）
  + 起始符号
  + 辅助语义动作执行的辅助对象（Auxiliary Object，auxObj）。语义动作需要暂存、取用的数据可以放在这里，同时也可以调用auxObj下属的函数方法，做到代码重用。
* 一个程序顶层配置（common/myConfig.js），指定了其他配置文件的路径，以及本程序可调节的一些选项。

### 程序源码输入

要编译的源码文件。在程序顶层配置中可以配置其路径。

### 中间代码输出（屏幕、日志）

是一个JavaScript对象数组，每一个对象代表一个中间代码四元式。在程序最后，这个数组被转换为字符串输出，格式如下：

<结果地址> ← <运算符> <操作数1地址> <操作数2地址>

### 日志输出（debug.log）

日志输出包含了其他中间信息，如文法编译过程、确定化后的DFA、语法分析树等。

### 文法清单（grammar.txt）、CLR分析表（CLRTable.csv）和分析过程表（Analysis.csv）

文法清单简单的列举了文法的简要情况，以及其产生式列表。

CLR分析表包含每个状态下遇到每个符号所执行的ACTION或GOTO动作。

分析过程表详细展示了语法分析过程栈的变化。

## 程序功能

* 读取用户输入。
* 根据用户的配置，生成类C词法分析器和CLR语法制导的翻译器。
* 调用翻译器。翻译器开始翻译，按需调起词法分析器以获取下一个词法单元；获得词法单元后运行翻译机制，生成一个中间代码数组。
* 输出中间代码数组和其他输出数据。

## 测试数据

在代码文件夹下的testsource\*.txt包含所有正确的测试源代码。

在代码文件夹下的badsource\*.txt包含所有错误的测试源代码。badsource1~3分别有“使用未声明的符号”、“语法错误”、“调用不存在的函数”错误。

# 概要设计

## 任务分解

根据任务的阶段划分，首先可以分解出如下子任务：

* 词法分析、词法分析器的生成（lex文件夹）
* 语法制导翻译、语法制导翻译器的生成（translate文件夹）

同时，词法分析和CLR的语法分析都需要构造NFA及将其确定化，因此分解出子任务：

* FA处理

语义动作涉及到大量代码，因此将翻译器生成器配置文件中的语义动作部分算作一个任务：

* 进行语义动作

## 数据类型

### NFA

|  |
| --- |
| class NFA {  constructor(StateClass, obj) {  this.name = "(Unnamed NFA)"; // NFA名  this.alphabet = []; // NFA 字母表  this.categories = {}; // NFA 字母表分类  this.catMap = {}; // NFA 字母分类映射  this.enablesElse = false; // NFA 是否处理未在字母表的字母  this.states = {}; // 所有 NFA 状态  this.initial = ""; // 起始状态名  this.StateClass = StateClass; // NFA 处理的状态类  … |

### NFA状态

在用户配置中体现。如：

|  |
| --- |
| 绿框内为一个状态。   * name: 状态名 * accept：是否为可接受状态 * delta：一个Object，由字符串（字母）映射到转换状态名列表。 |

### Token（词法单元）

|  |
| --- |
| 继承自Symbol（文法符号）。其属性有：   * type：小分类名 * category：大分类名 * lexeme：词法单元对应的字符串 * 其他语义动作赋予的属性。 |

### Grammar（文法）

|  |
| --- |
| class Grammar {  constructor(obj) {  this.auxObj = null; // 辅助对象  this.terminals = []; // 终结符列表  this.nonTerminals = []; // 非终结符列表  this.startSymbol = null; // 起始符号  this.productions = []; // 产生式列表  … |

### CLRItem（CLR文法项目）

|  |
| --- |
| class CLRItem {  constructor(prod, dotPos, lookAhead) {  this.production = prod ? prod : null; // 关联的产生式  this.dotPosition = dotPos !== undefined && dotPos !== null ? dotPos : null; // “点”的位置  this.lookAhead = lookAhead !== undefined && lookAhead !== null ? lookAhead : null; // 前瞻符号  }  … |

### CLRTable（CLR分析表）

|  |
| --- |
| class CLRTable {  constructor() {  this.terminals = []; // 终结符  this.nonTerminals = []; // 非终结符  this.initialState = null; // 起始状态  this.states = null; // 所有状态    this.ACTION = {}; // ACTION[][] 表，二级嵌套Object  this.GOTO = {}; // GOTO[][] 表，二级嵌套Object  } |

## 主程序流程

主程序在mainTest.js中。流程如下；

* 载入配置文件
* 载入源文件，获得一个读取流，利用读取流构造一个“获取一些字符”的函数
* 载入词法分析器的生成器的配置文件，生成一个词法分析器
* 将“获取一些字符”函数载入词法分析器
* 载入翻译器的生成器的配置文件，生成一个翻译器
* 将词法分析器的引用传入翻译器
* 启用翻译器（调用analyze方法）
  + 翻译器按需调用词法分析器的getNextToken方法，获取词法单元
    - 词法分析器按需调用“获取一些字符”函数，获取字符用来识别和构造词法单元

## 模块间调用关系

翻译模块调用词法模块、FA模块、语法和语义动作模块。

词法模块调用FA模块。

# 详细设计

## 各具体类所在源文件及解释

|  |  |  |
| --- | --- | --- |
| **类** | **文件** | **解释** |
| Token | common/Token.js | 词法单元的数据结构 |
| Lexer | lex/Lexer.js | 描述一个词法分析器 |
| LexerGenerator | lex/LexerGenerator.js | 描述一个词法分析器的生成器。调用generate()方法可以得到一个Lexer对象 |
| TokenRecognizer | lex/TokenRecognizer.js | 描述一个词法单元识别器（NFA+求值器）。其实例通常由配置文件里的结构直接转换而来。 |
| NFA | automata/NFA.js | 描述NFA的结构。其实例通常由配置文件里的结构直接转换而来。 |
| DFA | automata/DFA.js | 描述DFA的结构。创建方式只有一种：在构造函数中传入一个NFA对象对其进行确定化。 |
| State, CLRItemState | automata/states.js和translate/CLRItemState.js | 描述FA中的状态。 |
| Grammar | translate/grammar/ Grammar.js | 描述一个CLR文法。其实例通常由配置文件里的结构直接转换而来。 |
| CLRItem | translate/CLRItem.js | 描述一个CLR文法项目。 |
| CLRTranslator | translate/CLRTranslator.js | 描述一个CLR翻译器。 |
| CLRTranslatorGenerator | translate/CLRTranslator Generator.js | 描述一个CLR翻译器的生成器。 |

# 调试分析

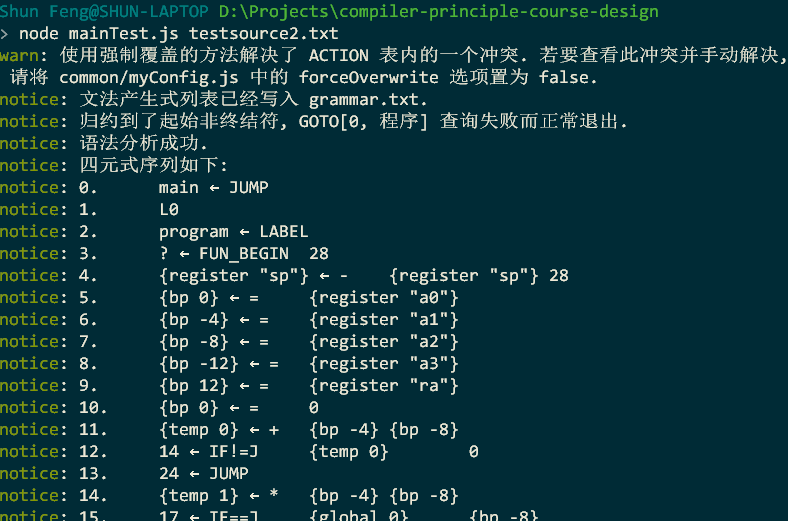
## 应用正确测试用例

默认加载正确实例testsource1.txt：



正常输出了四元式序列。

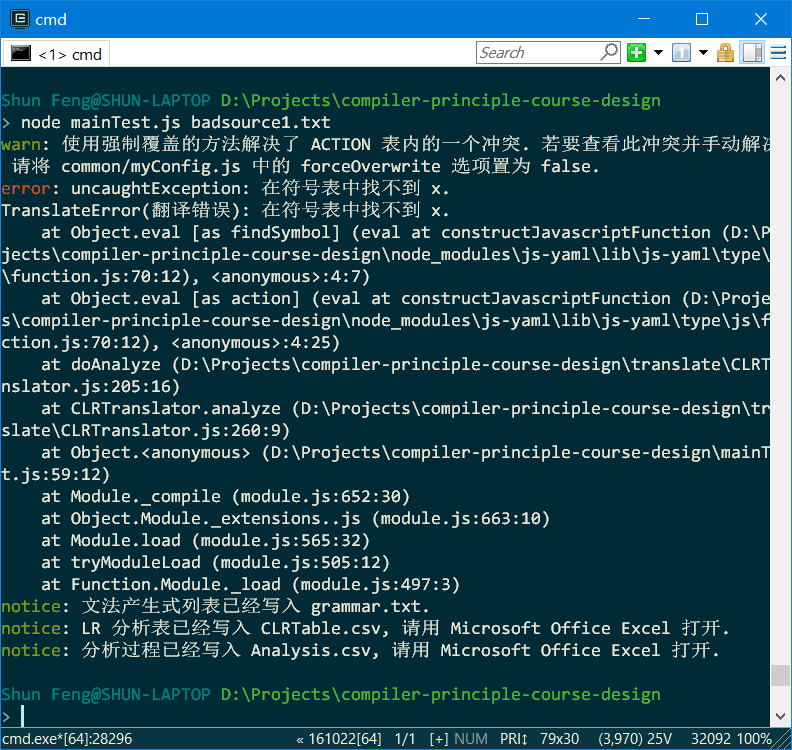
加载含布尔表达式和算术表达式混合使用的源代码testsource2.txt：



输出仍然正确。

## 应用错误测试用例

使用badsource1.txt：



可见以error的等级打印出了检查到的异常：“在符号表中找不到x”。

加载其他错误测试用例可以看到类似的结果。

## 时间复杂度分析

词法分析器的生成器：O(kx)。其时间大部分用于构造DFA。式中，k为TR的个数，x为每个TR NFA的状态数。由于编写的NFA有规律可循，所以在确定化时，涉及到的状态数和NFA的状态数在同一数量级。

词法分析器：O(kn)，其中n为字符数，k为TR的个数。因为每一个字符都要被每一个TR接受，以判断它最后被哪个TR识别。

翻译器的生成器：视输入文法而定。大致为O(yz)，其中y为每个产生式的长度，z为产生式个数。

翻译器：为O(n)，其中n为词法单元数。读入词法单元后，只需查找CLR分析表进行操作即可，所以是O(1)的。每个语义动作也是O(1)的。

## 调试存在问题

由于设计合理、日志详尽、调试工具高效，调试期间没有遇到较大问题。

# 用户使用说明

## 编写配置

你可以直接使用已经编写完毕的配置，或模仿它们写自己的配置：

* common/myConfig.js，在此修改默认源文件名、词法分析器配置路径、翻译器配置路径、日志文件名。其他选项不建议修改。
* lex/lex-c-style.yaml，类C词法。你需要懂YAML的基本语法。
* translate/grammar-c-style.yaml，类C语法和语义动作。你需要懂YAML的基本语法。

## 配置环境

本软件使用JavaScript编写，需要在Node.JS的环境下运行，同时需要几个NPM包的安装。

安装Node.JS 和NPM

## 运行程序

## 查看结果

# 课程设计总结

## 过程总结

## 遇到难点

## 将来改进

# 参考资料

* 陈火旺《程序设计语言编译原理》
* 紫龙书（《编译原理》）