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

0 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。

1 需求分析

1.1 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。

1.2 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。

1.2.1 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。

1.2.1.1 类C 文法

本程序使用的类 C 文法如下（运行后产生的 grammar.txt）：

========== 文法打印 ==========

名称: 类 C 文法

终结符:

"void", "标识符", ";", "(", ")", "int", "{", "}", "=", "return", "while", "if", "else", ">", "<", "<=", ">=", "==", "!=", "+", "-", "*", "/", ",", "||", "&&", "!", "true", "false", "整数", "[EOF]"

非终结符:

程序, 跳转 main 动作, 声明串, 声明, 声明类型, 初始化函数动作, 形参, 形参列表, 具体形参, 语句块, 初始化语句块动作, 内部变量声明串或空, 内部变量声明串, 内部变量声明, 语句串, 语句, 表达式布尔化动作, 函数调用, 函数调用具体形式, 实参列表, 记录下一指令编号动作, 跳转待填地址动作, 或级表达式, 与级表达式, rel 级表达式, rel, 加减级表达式, 加减运算符, 项, 乘除运算符, 非级表达式, 因子

起始符号: 程序

产生式:

0. 程序 → 跳转 main 动作 声明串

1. 跳转 main 动作 → ε

2. 声明串 → 声明

3. 声明串 → 声明 声明串

4. 声明 → 声明类型 标识符 ;

5. 声明 → 声明类型 标识符 (形参) 初始化函数动作 语句块

6. 声明类型 → int

7. 声明类型 → void

8. 初始化函数动作 → ε

9. 形参 → 形参列表

10. 形参 → void

11. 形参列表 → 具体形参

12. 形参列表 → 形参列表 , 具体形参

13. 具体形参 → int 标识符

14. 语句块 → { 初始化语句块动作 内部变量声明串或空 语句串 }

15. 初始化语句块动作 → ε

16. 内部变量声明串或空 → ε

17. 内部变量声明串或空 → 内部变量声明串

18. 内部变量声明串 → 内部变量声明 内部变量声明串

19. 内部变量声明串 → 内部变量声明

20. 内部变量声明 → int 标识符 ;

21. 语句串 → 语句 记录下一指令编号动作 语句串

22. 语句串 → 语句

23. 语句 → 语句块

24. 语句 → 标识符 = 或级表达式 ;

25. 语句 → return 或级表达式 ;

26. 语句 → return ;

27. 表达式布尔化动作 → ε

28. 语句 → while 记录下一指令编号动作 (或级表达式 表达式布尔化动作) 记录下一指令编号动作 语句

29. 语句 → if (或级表达式 表达式布尔化动作) 记录下一指令编号动作 语句

30. 语句 → if (或级表达式 表达式布尔化动作) 记录下一指令编号动作 语句 跳转待填地址动作 else 记录下一指令编号动作 语句

31. 语句 → 函数调用 ;

32. 函数调用 → 函数调用具体形式

33. 函数调用具体形式 → 标识符 ( )

34. 函数调用具体形式 → 标识符 (实参列表)

35. 实参列表 → 或级表达式

36. 实参列表 → 实参列表 , 或级表达式

37. 记录下一指令编号动作 → ε

38. 跳转待填地址动作 → ε

39. 或级表达式 → 或级表达式 表达式布尔化动作 || 记录下一指令编号动作 与级表达式 表达式布尔化动作

40. 或级表达式 → 与级表达式

41. 与级表达式 → 与级表达式 表达式布尔化动作 && 记录下一指令编号动作 rel 级表达式 表达式布尔化动作

42. 与级表达式 → rel 级表达式

43. rel 级表达式 → 加减级表达式 rel 加减级表达式

44. rel 级表达式 → 加减级表达式

45. rel → <

46. rel → >

47. rel → <=

48. rel → >=

49. rel → ==

50. rel → !=

51. 加减级表达式 → 加减级表达式 加减运算符 项

52. 加减级表达式 → 项

53. 加减运算符 → +

54. 加减运算符 → -

55. 项 → 项 乘除运算符 非级表达式

56. 项 → 非级表达式

57. 乘除运算符 → *

58. 乘除运算符 → /

59. 非级表达式 → true

60. 非级表达式 → false

61. 非级表达式 → ！非级表达式

62. 非级表达式 → - 非级表达式

63. 非级表达式 → + 非级表达式

64. 非级表达式 → 因子

65. 因子 → 整数

66. 因子 → (或级表达式)

67. 因子 → 标识符

68. 因子 → 函数调用

1.2.2 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。

1.2.3 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>

中间代码是为生成 MIPS 的目标机器代码而设计的，考虑了栈式存储分配、MIPS 的调用约定。

1.2.4 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。

1.2.5 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。

1.3 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。

1.4 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。

2 概要设计

2.1 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

2.2 数据类型

2.2.1 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
…
```

2.2.2 NFA 状态

在用户配置中体现。如：

```
states:
  start:
    name: start
    accept: false
    delta:
      '': []
      "/":
      - afterSlash
```

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。

2.2.3 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。

2.2.4 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
…
```

2.2.5 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
  }
…
```

2.2.6 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

  }
```

2.3 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

2.4 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。

3 详细设计

3.1 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/CLRTranslatorGenerator.js	描述一个 CLR 翻译器的生成器。

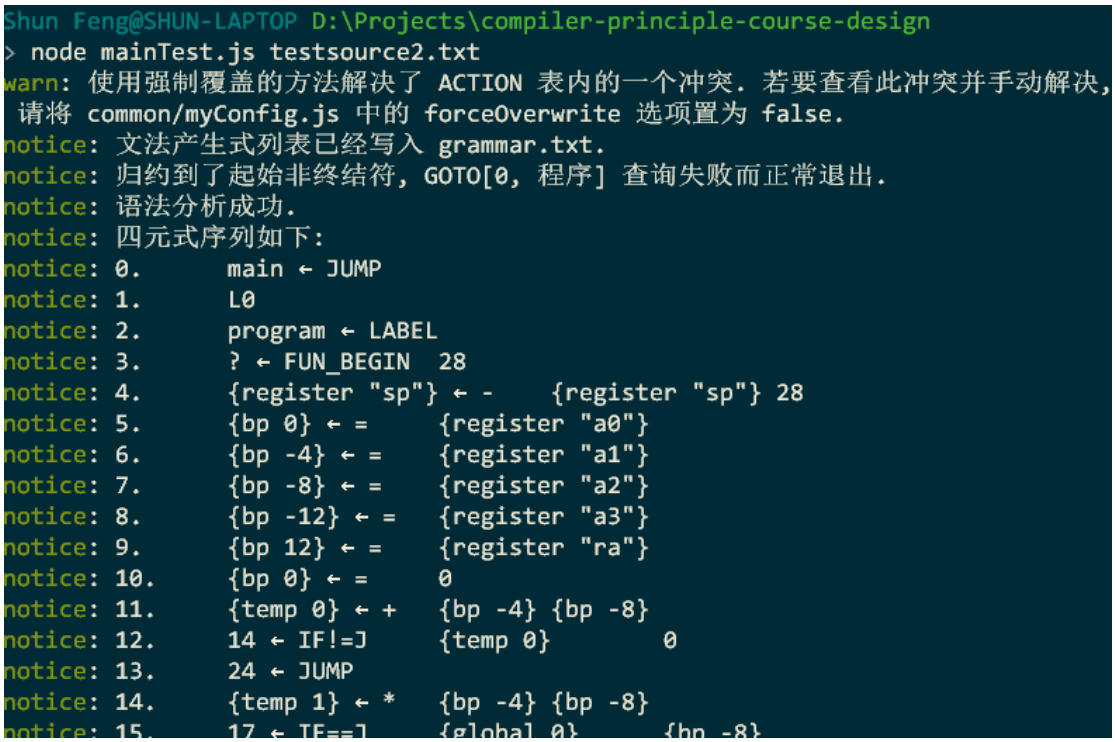
4 调试分析

4.1 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。

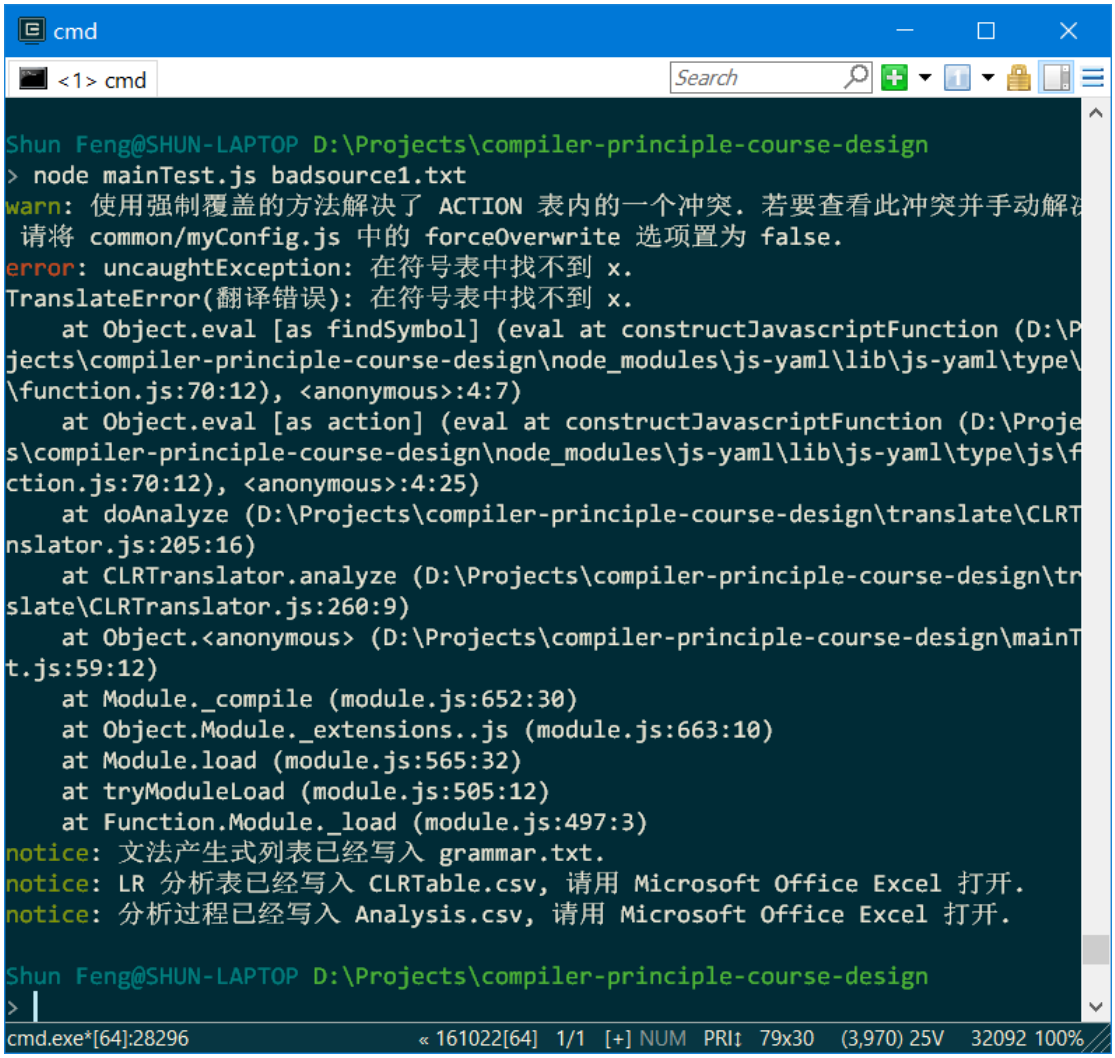
加载含布尔表达式和算术表达式混合使用的源代码 testsource2.txt：



输出仍然正确。

4.2 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。

4.3 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。

4.4 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。

5 用户使用说明

5.1 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。

5.2 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
- 在代码根目录下，打开命令提示符/Shell，运行 npm install
- 等待完成

也可以在命令行下运行根目录随附的可执行文件 compiler-principle-course-design-win.exe。

5.3 运行程序

- 在命令提示符下执行 node mainTest.js 或 compiler-principle-course-design-win.exe，后可以跟需要分析的源文件路径作为可选参数。如不指定，则默认为在 common/myConfig.js 中配置的源文件路径。
- 观察输出。

6 课程设计总结

6.1 过程总结

本次课设是将上学期的词法分析器、语法分析器及其生成器用 JavaScript 重写，并加上语义动作处理的结果。由于设计得当，过程进展较为顺利，我也重温了上学期编译原理的知识和 JavaScript 的知识。

6.2 遇到难点

6.2.1 布尔表达式和普通表达式的杂糅

布尔表达式和算术表达式编入文法时，由于其语法上的相似性，编写后的文法不是 LR(1)的（如遇到“标识符 •)||”的情况，由于只能前瞻到右括号，不能确定应该把标识符归约为算术表达式还是布尔表达式），不能投入使用。

因此，只能将布尔表达式和算术表达式合并为以优先级为界划分的各级表达式，并在需要的时候调用 boolize 和 exprize 函数进行布尔性质和算术性质的相互转换。

6.2.2 悬空 else 问题

若允许“if () if () doSth(); else doSth();”这类句子对应的文法，在第一个 else 之前会出现归约-归约冲突，导致文法不是 LR(1)的。本课设中编写的文法也确实不是 LR(1)的，在制作 LR(1)的分析表时，强制让“else 合并到最近的 if”分支进入分析表，另一分支不进入分析表，才构造了可用的 LR(1)分析表。

6.2.3 遵守 MIPS 调用约定的问题

中间代码的产生考虑到了 MIPS 的调用约定。由于 MIPS 没有 BP 寄存器，只有 SP 寄存器，在确定栈帧（活动记录）中某元素的位置时，若缺少栈帧大小则很难判断；但栈帧大小必须要在函数结束时才能确定（因为中间可能有语句块分配了新的局部变量）。采用回填技术部分解决这个问题；另外，将中间代码的地址分为“BP”类和“SP”类，“SP”类的地址以当前 SP 为基础进行计算，“BP”类的地址以假想的 BP 寄存器为基础进行计算。在生成目标代码时，对 BP 类地址进行适当的后期计算处理，反映真实地址。

6.3 将来改进

需要生成目标代码，增加寄存器分配算法等。

7 参考资料

- 陈火旺《程序设计语言编译原理》

- 紫龙书（《编译原理》）