吉林大学学士学位论文（设计）承诺书

本人郑重承诺：所呈交的学士学位毕业论文（设计），是本人在指导教师的指导下，独立进行实验、设计、调研等工作基础上取得的成果。除文中已经注明引用的内容外，本论文（设计）不包含任何其他个人或集体已经发表或撰写的作品成果。对本人实验或设计中做出重要贡献的个人或集体，均已在文中以明确的方式注明。本人完全意识到本承诺书的法律结果由本人承担。

学士学位论文（设计）作者签名：

2018 年 5 月 20 日

摘要

中文标题

中文摘要。

关键字：甲, 乙, 丙

Abstract

English Title

Engligh abstract goes here.

Keywords: a, b, c

目　　录

第 1 章　绪论

1.1 课题的研究背景和现状

正则表达式最早由 Stephen C.Kleene 在他 1956 年的论文中提出[1]，在这篇论文中他试图通过正则表达式来刻画神经网络和有限自动机的行为。Ken. Thompson 在 1968 年的论文[?] 中首先对正则表达式的匹配进行了描述。Aho 的著作[2] 中对正则表达式和有限自动机的理论进行的综合的叙述。正则表达式不但在计算机的基础理论研究中有重要的地位，在实际应用中也有广泛的应用。正则表达式为我们提供了简单、有力而效率高的工具，允许我们用比较短的程序去刻画字符串中复杂的字符串结构。Sedgewick[3] 中指出，经典正则表达式匹配算法的时间复杂度为 $O(mn)$, 其中 n 是字符串的长度而 m 是正则表达式模式的长度。对于正则表达式而言，我们可以将其看作一个比较简单的程序设计语言，那么我们就可以使用编译原理中所学习的知识去处理正则表达式。我们需要考虑对正则表达式进行词法分析、语法分析然后生成抽象语法树，最后将抽象语法树转化为有限自动机，再和给定的字符串进行匹配。这些经典的步骤的设计与实现在[4] 中有详细的叙述。

随着计算机科学的发展以及对文本处理需求的提高，正则表达式本身处理能力也不断地提高。除了经典正则表达式中的并、连结、克莱因星号等操作以外，现有的程序设计语言中为了提高正则表达式的效率，还支持回溯的操作。这同时也对正则表达式的处理技术提出了更高的要求。研究人员考虑并行化处理生成的有限自动机，称为标签化有限自动机 (”Tagged–Determined-Finite-Automata”)，即” TDFA”，Aaron Karper[5] 在他 2014 年的硕士毕业论文里给出了一个基于 TDFA 的新匹配算法，将正则表达式的平均性能提升到了 $O(m)$ 级别，并且给出了在 Java 程序设计语言中的实现。但是，在他的实现中，我们也可以看出经典的命令式程序设计语言的问题：代码极其冗长，达到了上千行。从软件工程的角度说，这无疑是对于软件可靠性极大的隐患之一。

和以 C 语言为代表的命令式程序设计语言相对的，还有以 Lisp、Ocaml、Haskell 为代表的函数式程序设计语言。函数式程序设计语言强调数据之间的映射和变换。那么正则表达式的处理过程可以看作从输入代码到目标代码之间的映射。实际上，在 Robert Harper 的一个非正式的手稿中，他仅用了不到 200 行

代码就实现了经典正则表达式的全部功能。在 2010 年的 ICFP 上，来自德国基尔大学的 Fischer 等学者基于 Haskell 程序设计语言提出了函数式高效的正则匹配算法[6]。

1.2 研究基本内容和意义

本文着重考虑软件的高效性和正确性，本文基于 2010 年的 ICFP 中 Fischer 等人的文章，基于 Standard ML 语言实现一个正则表达式解析引擎，并尝试使用 HOL4 交互式定理证明器来对程序的正确性进行形式化验证。Standard ML 语言是经过小步语义规范定义的轻量级的函数式程序设计语言，并且在 Linux 和 Windows 平台上都有良好的运行支持。HOL4 是基于 Standard ML 程序设计语言开发的、基于高阶逻辑的交互式定理证明器，可以用于证明程序的正确性。和各种各样由人操作保证软件工程的手段相比，由数理逻辑规律所保证的可靠性应当是最强的。

1.3 研究内容与论文结构

第一章主要介绍课题的研究背景、研究现状和论文的整体内容和结构

第二章是论文中的基本概念，主要用递归定义的方法定义正则表达式，编译原理中词法分析、语法分析和抽象语法树的概念简介，Standard ML 程序设计语言简介，以及 HOL4 定理证明器进行形式化验证的基础。

第三章将会从一个简单的递归下降语法分析程序开始，给出一个 Standard ML 正则库的简单实现，然后介绍 Fischer 论文中的算法[6]

第四章利用形式化的开发方法实现 Fischer 论文中的算法

第五章将会介绍基于形式化验证开发的正则库的测试结果。

第六章将会进行展望和总结。

第七章为附录，主要补全对本文展开思路没有影响的证明。

第 2 章　插入算法的示例

可用如下方式插入算法流程：

Algorithm 1 ACP 调度器

输入： 要调度的作业集合 J 和资源集合 R

for each r ∈ R **do**
  使用基准测试工具对 r 进行测试
  获取 r 的 $storage_r$ 和 $compute\_power_r$ 的值
  将 $storage_r$ 和 $compute\_power_r$ 写入到资源限制中
end for
while true **do**
  根据上次运行的结果，更新任务约束
  for each r ∈ R **do**
    if r.schedular ≠ NULL **then**
      if r.runningtask ≠ NULL **then**
        rt ← r.runningtask
        st ← rt.starttime
        et ← rt.endtime
        pd ← rt.performancedemand
        sr ← rt.scheduledresource
        根据 (rt, st, et, pd, sr) 更新约束条件
        // 根据任务的开始时间，结束时间，性能需求和其运行资源的性能添加一条约束
      else
        r.runningtask ← NULL
        // 将该任务设置为已完成
        将 r.runningtask 从 r.schedular 中移除
        pj ← r.parentjob
        if pj.schedular = NULL **then**
          从 J 中删除 pj
        end if
      end if

end if

end for

新建一个带有资源约束的模型

将任务约束加入到 OPL 模型中

将 J 和 R 作为约束添加到数据中

求解 OPL 模型

按照得出的最优解对任务进行分配

等待新的事件出现

end while

第 3 章　基本概念和技术简介

3.1 本章内容概述

本章主要介绍论文中一些重要的基本概念。本章分为四节，第一节介绍正则表达式的定义。第二节介绍编译原理的部分基本概念。第三节介绍 Standard ML 程序设计语言。第四节为 HOL4 定理证明器简介。

3.2 正则表达式

3.2.1 正则表达式的递归定义

本节主要参考 Sipser[7] 计算理论导引中的内容，正则表达式作为和有限自动机相等价的模型，我们不妨从更直观的有限自动机出发来对正则表达式进行定义。考虑篇幅和不影响思路的原因，本节略去了证明，完整的证明放在第七章附录里。

定义 3.1 (有限自动机)**.** 一个有限自动机是一个五元组 $(Q, \Sigma, \delta, q_0, F)$, 其中

1. Q 是一个有限集被称作状态集。
2. Σ 是一个有限集被称作字母表。
3. $\delta : Q \times \Sigma \to Q$ 是转移函数。
4. $q_0 \in Q$ 是初始状态集。
5. $F \subseteq Q$ 是接受状态集。

在给出有限自动机的定义以后，我们继续对有限自动机识别字符串给出一个严格的定义。从直观上讲，只要输入的字符串一步一步地按照状态图规定的状态运行，最终到达接受状态，我们就可以说有限自动机接受了这个输入的字符串。

定义 3.2 (有限自动机接受语言)**.** 令 $M = (Q, \Sigma, \delta, q_0, F)$ 是一个有限自动机，令 $w = w_1 w_2 ... w_n$ 是一个字符串，其中每一个 w_i 都在字母表 Σ。则我们称 M 接受 w 当且仅当一个状态序列满足下列条件：

1. $r_0 = q_0$
2. 对所有的 $i = 0...n-1$ 都有 $\delta(r_i, w_{i+1}) = r_{i+1}$
3. $r_n \in F$

我们称有限自动机 M 识别一个语言 A 当且仅当 $A = \{w|M接受w\}$。

定义 3.3 (正则语言)**.** 我们称一个语言为正则语言当且仅当存在有限自动机识别这个语言。

就像在算术中，我们可以对阿拉伯数字定义 $+\times$ 这样的操作，组成 $2\times(3+2)$ 这样的算术表达式。在正则语言中，我们也可以对正则语言定义一些操作，我们可以称之为正则操作。

定义 3.4. 令 A 和 B 是两个正则语言，我们定义并、连结和克莱因星号操作如下：

1. 并：$A \cup B = \{x|x \in A或x \in B\}$

2. 连结: $A \circ B = \{xy|x \in A且y \in B\}$

3. 克莱因星号 $A^* = \{x_1x_2...x_k|k \geq 0并且所有的x_i \in A\}$

在优先级上，克莱因星号是最高级的，连结其次，并的优先级是最低的

我们可以证明下面的内容：

定理 3.1. 所有的正则操作都是在正则语言里封闭的，也就是说，所有的正则语言经过正则操作以后都可以被有限自动机所识别

继续刚才算术中的例子，正如我们可以用算术操作符组合出算术表达式，我们也可以将正则语言用正则操作组合出正则表达式。我们不妨递归地进行定义。

定义 3.5 (正则表达式的递归定义)**.** 我们称 R 是一个正则表达式，如果 R 是

1. a 对字母表 Σ 中的任意字母 a

2. ϵ

3. $\emptyset$

4. $(R_1 \cup R_2)$, 其中R_1和R_2都是正则表达式

5. $(R_1 \circ R_2)$, 其中R_1和R_2都是正则表达式

6.(R_1^*), 其中 R_1是一个正则表达式

其中，表达式 ϵ 代表语言的是一个空字符串，$emptyset$ 代表的是空集，即其中不包含任何字符串。

我们也可以证明:

定理 3.2. 这个正则表达式的定义和有限自动机是等价的

3.2.2 用上下文无关文法描述正则表达式

基于定义 3.5，我们可以使用上下文无关文法递归地描述正则表达式。但是在此之前，我们需要修改一下正则表达式约定的符号。由于计算机命令行程序难以处理"()"，并且没有"∪" 符号和 ∘，我们约定："(" 和")" 修改为 "[" 和"]"。并且，我们用"+" 代表" 并" 操作，即"∪"，用"r_1r_2"，也就是两个正则表达式直接连写来代表连结操作。

现在，我们可以把定义 3.5 改写成如下的形式

$$r ::= \quad \emptyset \quad | \quad \epsilon \quad | \quad a \quad | \quad r_1 r_2 \quad | \quad r_1 + r_2 \quad | \quad r^* \tag{3-1}$$

我们首先从优先级最低的并操作开始，我们约定，上下文无关文法的起始符是 rexp, 那么，我们可以得到：

$$rexp \quad ::= \quad rterm \quad | \quad rterm + rexp \tag{3-2}$$

起始符的推导有两种情况，如果有并操作的话，这种情况可以分解为两个子表达式，另外一种情况是直接推导出下一个子表达式。对于在并操作下的情况，为了避免在后面的递归下降分析出现左递归的情况，我们规定"+" 符号左边是新的非终结符。如果后续的表达式中还会出现"+" 操作符，那继续展开右边的 $rexp$ 即可。

对于优先级居中，而且也是双目操作的连结操作，我们可以像连结操作一样处理。这是因为：根据递归定义，我们可以把优先级更高、暂时不需要展开的正则表达式视作一个整体。在这种情况下，有更高等级的双目操作符可以抽象为同一个情况。在这种情况下，我们有：

$$rterm \quad ::= \quad rfac \quad | \quad rfac \quad rterm \tag{3-3}$$

对于优先级最高，而且是单目操作符的克莱因星号，用上下文无关文法描述是比较容易的：

$$rfac \quad ::= \quad ratom \quad | \quad ratom^* \tag{3-4}$$

根据递归定义，最低级的非终结 $ratom$ 对应两种情况，第一种情况是直接推导出非终结符，第二种情况是推导出括号下的作为一个整体的正则表达式。

$$ratom \quad ::= \quad @ \quad | \quad ! \quad | \quad a \quad | \quad [\quad rexp \quad] \tag{3-5}$$

其中 @ 对应的是空集，! 对应的是空串，a 对应的是字母表中任意一个字母，[$rexp$] 对应的是可能存在是括号内作为一个整体的正则表达式。

综上所述，我们得到了正则表达式的上下文无关文法的定义：

$$\begin{aligned} rexp &::= rterm \mid rterm + rexp \\ rterm &::= rfac \mid rfac \quad rterm \\ rfac &::= ratom \mid ratom^{*} \\ ratom &::= @ \mid ! \mid a \mid [\quad rexp \quad] \end{aligned} \tag{3-6}$$

从 (3 – 6) 中我们不难看出，每一个非终结符都能推导出下一个非终结符，而最后一个非终结符 $ratom$ 可以推导出终结符，这样的话，我们可以说3-6中的上下文无关文法是可达的。

3.3 词法分析和语法分析简介

这一节中主要对编译原理从词法分析到抽象语法树生成的过程和部分重要概念作简要的介绍。

3.4 词法分析简介

词法分析

3.5 Standard ML 语言简介

Standard ML 语言最早是为了爱丁堡大学的可计算函数逻辑定理证明器而设计出来的。后来逐渐用于编译器设计、交互式验证等领域。而在这一节中，我们主要参考的是 Standard ML 语言规约文档[8]。是由卡耐基梅隆大学的 Robert Harper 等学者基于小步语义所制定出来的，这个文档严格地刻画了 Standard ML 语言的行为，后来 Standard ML 程序设计语言所有的编译器和运行环境的实现都是基于这个规约文档。

3.5.1 Standard ML 语言的实现和运行环境

Standard ML 语言有两种运行模式，一种是由编译器直接编译产生可执行代码，另外一种运行模式是函数式程序设计语言所特有的“读入-计算-输出”循环 (英语：”Reading-Evaluating-Print” Loop)，也被称作为交互式顶层构件 (英语：” Interactive Toplevel”)，是一个比较简单的，和程序的解释器、编译器进行交互的运行环境[9]。

Standard ML 语言的实现有很多种，其中本论文主要涉及两个实现：第一个

是 SML/New Jersey，由美国的普林斯顿大学开发维护，被广泛应用于程序设计的课堂教学，以 REPL 环境为主，在本章中的所有代码都基于这个编译器；第二个则是 Poly/ML 编译器，这个编译器用于实现两个大型定理证明器项目——由慕尼黑工业大学和剑桥大学支持开发的 Isabelle/HOL 和剑桥大学和查尔姆斯理工学院支持的 HOL4，这个编译器既支持 REPL 环境又可以编译出一个可执行文件，下一章的内容都是基于这个编译器运行的。

3.5.2 Standard ML 语言中的类型

Standard ML 是静态强类型语言，也就是说每一个变量都必须有对应的类型，否则程序不可能通过编译器的类型检查，这就要求我们无论是在声明变量还是在编写函数的时候都要注意变量和参数的类型。下面为我们可以看几个在 REPL 环境中的例子。例如：

```
1  val pi = 3.14;
2  > val pi = 3.14 : real
```

这就说明变量 *pi* 的类型是 *real*。和其他的程序设计语言类似，Standard ML 语言也有诸如字符 *char*、字符串、(、布尔值 (”bool”) 等类型。这些类型之间也可以相互转换。例如：

```
1  val what = true;
2  > val what = true : bool
3  (* 调用类型转换函数 *)
4  Bool.toString(true)
5  > val it = "true" : string
6  (* it 是返回值的引用 *)
```

刚才的程序片段中出现了将 *bool* 类型转化为 *string* 类型的函数，实际上，Standard ML 中的函数可以看作从从类型到类型的映射，比如刚才的 *toString* 函数，就可以当作 *bool* 到 *string* 的一个映射，在 Standard ML 语言中表示为:

```
1  > val toString : bool -> string
```

除了程序语言规范中给出的类型以外，我们还可以在 Standard ML 语言中自行定义数据类型。[10] 这里不妨以本文的研究对象正则表达式作为一个例子，根据上文对正则表达式的定义，我们可以定义一个正则表达式的数据类型 *Regex*

```
1  datatype regexp = None
```

```
    | Empty
    | Char of char
    | AnyChar
    | Or of regexp * regexp
    | Concat of regexp * regexp
    | Star of regexp;
```

上面这个声明了八个对象，数据类型 *regexp*，和这个数据类型的七个构造子 *Empty*、*Char*、*AnyChar*、*Or*、*Concat* 和 *Star*，以及每个构造子所对应的类型。例如，*Or* 构造子必须由两个 *regex* 类型的元素作为参数去构造。每一个数据类型的变量包含且仅包含其构造子所构造的值。

3.5.3 Standard ML 语言函数式特性概览

函数式程序设计中以数据的变换取代了状态，因而在函数式编程中程序没有数组、循环和赋值语句。相应的，作为支持函数式编程的语言，Standard ML 有一系列的特性来解决问题。

3.5.3.1 元组

元组 (*tuple*) 是 Standard ML 中最简单的数据结构，例如

```
(* 两个 int 类型变量组成的元组 *)
val pair : int * int = (2,3)
```

其中，元组的类型可以不同，元组中还可以嵌套元组。

Standard ML 只能支持函数返回一个参数，但是基于元组，我们可以让函数返回多个参数。

3.5.3.2 表

表 (*list*)。数据集可以组织成为下面的表这样的形式来处理。

```
[a,b,c,d,e]
```

表支持顺序访问，从左到右地进行扫描。和 Lisp 系语言一样，Standard ML 也可以对表进行处理。在本文中，对 *list* 最重要的操作符是:: 操作符：

```
val (op ::) : typ * typ list -> typ list
```

其中，*op* 的意思是这是一个函数。:: 操作符的意义是，给定一个表 it，$h :: t$ 会把这个表分为两个部分，表头第一个元素 h 和剩下的部分 t。 函

数 (*function*), Standard ML 中的表达式主要由函数调用构成的。函数的参数可以是任何类型，包括函数本身。这部分特性在介绍 CPS 和高阶函数时会进行更详细的说明。Standard ML 语言的垃圾收集功能[11] 支持了这个特性。

3.5.3.3 模式匹配

模式匹配 (*pattern matching*)，这是 Standard ML 语言中非常重要的一个特性，这个特性一般用于多个分支的时候，跳转到不同的分支情况，这个特性可以代替命令式程序语言中的 *case* 语句。模式匹配在 Standard ML 中有两种形式。

第一种形式被称作定义函数表达式，一般的形式如下：

```
fun pat_1 => exp_1
        |...
        |pat_n => pat_n
```

其中每一个 pat_i 是一个模式而每一个 exp_i 是一个表达式。其中的每一个元素 $pat => exp$，被称作为一个定义，或者是一条规则。所有的规则组合起来被称作一个匹配。

匹配对模式和表达式的类型都有严格的要求，模式要求每一个规则的模式都是相同的类型 typ_1 而每一个表达式的类型也是相同的 typ_2。这样才能建立起来映射 $typ_1 \to typ_2$。

这里我们可以给模式匹配举一个简单的例子：

```
(*字符串元素计数*)
fun length []= 0
        | length (x::xs) =1 + length xs
```

此外，模式匹配还有另外一种格式，如果对一个表达式 *exp* 的各种情况进行匹配，那么也考虑下面这种形式：

```
case exp
of pat1 => exp1
| ...
| patn => expn
(*是下面这种情况的简写*)
(fn pat1 => exp1
| ...
```

```
8                   | patn => expn)
9                  exp.
```

程序会首先计算 *exp* 的值，然后在 *caseof* 语句下的各种情况进行匹配。

3.5.3.4 高阶函数

高阶函数 (*Higher Order Function*)。

参 考 文 献

[1] KLEENE S. Representation of events in nerve nets and finite automata[J]. Automata Studies, 1956 : 3 – 42.

[2] AHOA, ULLMAN. Theory of Parsing, Translation and Compiling, Vol I: Parsing[M]. [S.l.] : Prentice-Hall, Englewood Cliffs, N.J., 1972 : 146.

[3] SEDGEWICK R. Algorithms in C++[M]. [S.l.] : Addison Wesley, 1992.

[4] APPEL A W. Modern compiler implementation in C[M]. [S.l.] : Cambridge university press, 2004.

[5] KARPER A. Efficient regular expressions that produce parse trees[M]. [S.l.] : Universität Berne, 2014.

[6] FISCHER S, HUCH F, WILKE T. A play on regular expressions: functional pearl[C] // Proceedings of the 15th ACM SIGPLAN international conference on Functional programming. 2010 : 357 – 368.

[7] SIPSER M. Introduction to the Theory of Computation[M]. [S.l.] : Course Technology, 2005 : 31 – 65.

[8] HARPER R, MACQUEEN D, MILNER R. Standard ml[M]. [S.l.] : Department of Computer Science, University of Edinburgh, 1986.

[9] ABELSON H, SUSSMAN G J. Structure and interpretation of computer programs[M]. [S.l.] : The MIT Press, 1996.

[10] PAULSON L C, PAULSON L C. ML for the Working Programmer[M]. 1996 : 124.

[11] PAULSON L C, PAULSON L C. ML for the Working Programmer[M]. 1996 : 5 – 7.

致　　谢

本科生涯看似漫长却又一晃而过，回首走过的岁月，我感慨良多。从最初的论文选题、思路梳理到研讨交流、反复修改直至最终完稿，都离不开老师、同学和亲人们的支持和无私帮助，在此我要向他们表达我最诚挚的谢意。

...

求学生涯暂告段落，但求知之路却永无止境。我将倍加珍惜大学生活给予我的珍贵财富，不忘初心，砥砺前行！

本科期间发表论文和科研情况

...