编程任务选题列表

- 词法分析器，根据正则表达式生成 DFA
  - 将输入的正则表达式转化为“简化正则表达式”
  - 将每一个“简化正则表达式”转化为 NFA
  - 将一组 NFA 转化为一个 DFA，使得能依据“匹配字符串长度第一优先”、“匹配规则先后次序第二优先”进行词法分析
  - 依据 DFA 将输入的字符串转化为分段结果和分类结果。

  其中，输入的正则表达式中可能包含的语法结构有：字符的集合（如 `[a-z0-9]` ）、可选的情形（即 $?r$）、多种循环（即 $r*$ 与 r^+）、字符串（如 `"ab\n"` ）、单字符、并集与连接；“简化正则表达式”中可能包含的语法结构有字符的集合、空字符串、星号表示的循环（即 $r*$）、并集与连接。本任务中，用于存储正则表达式、NFA 与 DFA 的数据结构已经在 lang.h 中提供了（详见`regexp_NFA_DFA.zip`）。
- 语法分析器，根据上下文无关语法生成语法分析器
  - 依据输入的上下文无关语法计算 first 集合与 follow 集合
  - 依据输入的上下文无关语法生成生成状态转移表
  - 基于状态转移表，处理输入终结符序列，输出移入规约过程

  其中，本任务默认输入的上下文无关语法中，每一条产生式右侧的符号串都非空；状态转移表中的每一个节点是扫描线左侧结构 NFA 的节点集合；状态转移表应描述在每个节点上遇到每个不同终结符（这个符号为此时扫描线右侧的符号）时的动作，这个动作或为移入（此时应指明移入后的状态节点）或为规约（此时应指明规约所使用的产生式）。本任务中，用于的上下文无关语法以及状态转移表的数据结构已经在 cfg.h 中提供了（详见`shift_reduce.zip`）。
- C 语言中 struct/union/enum 的定义与声明的词法分析与语法分析

  考虑 C 语言中 struct/union/enum 的定义与声明，基于 typedef 的类型定义，以及变量的定义。下面是它们的语法（不需要考虑一条语句定义多个变量的情形，也不需要考虑变量定义同时初始化的情形）：

```
STRUCT_DEFINITION ::= struct STRUCT_NAME { FIELD_LIST } ;
   STRUCT_DECLARATION ::= struct STRUCT_NAME ;
   UNION_DEFINITION ::= union UNION_NAME { FIELD_LIST } ;
   UNION_DECLARATION ::= union UNION_NAME ;
   ENUM_DEFINITION ::= enum ENUM_NAME { ENUM_ELE_LIST } ;
   ENUM_DECLARATION ::= enum ENUM_NAME ;
   TYPE_DEFINITION ::= typedef LEFT_TYPE NAMED_RIGHT_TYPE_EXPR ;
   VAR_DEFINITION ::= LEFT_TYPE NAMED_RIGHT_TYPE_EXPR ;
```

  本任务中，约定 struct 与 union 的域列表允许为空，但 enum 的元素列表不得为空。

```
FIELD ::= LEFT_TYPE NAMED_RIGHT_TYPE_EXPR ;
   FIELD_LIST ::= FIELD FIELD ... FIELD
   ENUM_ELE_LIST ::= ENUM_ELE, ENUM_ELE, ... , ENUM_ELE
```

这里提到的 `STRUCT_NAME` 、`UNION_NAME` 、`ENUM_NAME` 、`ENUM_ELE` 以及下面会提到的 `IDENT` （标识符）都表示以字母或下滑线开头且仅包含字母数码与下划线的名字。需要特别注意的是，C 语言的中变量定义与域定义中，变量类型与域类型都是通过两部分进行描述的：左半部分是基础类型，右半部分是包含变量名或域名的一个表达式。例如，`int * x` 这个定义可以分为 `int` 与 `* x` 两个部分，它表示 `* x` 的值（即存储在 `x` 地址的内容）为整数类型。这就是上面提到的：

`LEFT_TYPE` `NAMED_RIGHT_TYPE_EXPR`

本任务中需要考虑指针类型、数组类型、函数类型的情形，在基础类型方面只考虑 `int` 与 `char` 两个类型：

```
LEFT_TYPE ::= struct STRUCT_NAME { FIELD_LIST }
              | struct { FIELD_LIST }
              | struct STRUCT_NAME
              | union UNION_NAME { FIELD_LIST }
              | union { FIELD_LIST }
              | union UNION_NAME
              | enum ENUM_NAME { ENUM_ELE_LIST }
              | enum { ENUM_ELE_LIST }
              | enum ENUM_NAME
              | int | char | IDENT
```

```
NAMED_RIGHT_TYPE_EXPR ::= IDENT
                          | * NAMED_RIGHT_TYPE_EXPR
                          | NAMED_RIGHT_TYPE_EXPR [ NAT ]
                          | NAMED_RIGHT_TYPE_EXPR ( ARGUMENT_TYPE_LIST )
   ANNON_RIGHT_TYPE_EXPR ::= EMPTY
                          | * ANNON_RIGHT_TYPE_EXPR
                          | ANNON_RIGHT_TYPE_EXPR [ NAT ]
                          | ANNON_RIGHT_TYPE_EXPR ( ARGUMENT_TYPE_LIST )
   ARGUMENT_TYPE ::= LEFT_TYPE ANNON_RIGHT_TYPE_EXPR
   ARGUMENT_TYPE_LIST ::= ARGUMENT_TYPE, ..., ARGUMENT_TYPE
```

在 C 语言中，表达式（这里提到的 `NAMED_RIGHT_TYPE_EXPR` 与 `ANNON_RIGHT_TYPE_EXPR` ）后缀（数组与函数）的结合优先级高于前缀的结合优先级，并且允许使用圆括号改变优先级。例如，`int * x[10]` 表示 `int * (x[10])` ，定义了一个整数指针的数组；函数参数类型的语法也是类似的，例如 `int * [10]` 表示整数指针的数组类型，`int ( * )(int)` 表示整数一元函数的函数指针类型。本题约定，函数的参数类型列表可以为空。

本任务中，用于所有定义与声明的抽象语法树的 C 语言存储结构以及辅助构造函数、调试函数已经在 lang.h 与 lang.c 中提供了，main.c 程序也是固定的（详见 `struct_union_enum.zip`），请使用 flex 与 bison 实现词法分析与语法分析。

- 带类型 WhileD 语言的词法分析、语法分析和类型分析

  带类型 WhileD 语言的语法应当包含所有 WhileD 语言的语法、带类型的变量声明、以及类型转化，变量声明的语法是：`type_name var_name; command` ，类型转换的语法是 `(type_name) expression` 。应当支持的类型有：`short` 、`int` 、`long` 、`long long` 、它们的指针类型，包括指针的指针、指针的指针的指针... 在这个任务中，你需要：

  – 设计带类型 WhileD 语言的 C 语言实现

  - 实现该程序语言的词法分析和语法分析
  - 检查每个变量是否都先声明后使用
  - 在上述程序语言抽象语法树的基础上，设计带隐式类型转换的抽象语法树
  - 分析原先程序语法树，进行类型检查并添加必要的隐式类型转换

  具体在实现的过程中，含隐式类型转换的语法树和不含隐式类型转换的语法树，可以选用相同或者不同的 C 语言存储结构。在最终的提交文档中，应当包含添加隐式类型转换规则，规则的制定应当具有合理性，可以参考 C 标准（不一定要完全相同），例如：

  - `int` 类型的 `a` 与 `long` 类型的 `b` 相加要先将 `a` 转化为 `long` 类型，
  - 常量的取值如果不超出 `int` 的范围则其类型默认为 `int` ，等等。

  在最终的提交文档中，还应当包含类型检查规则，例如：

  - 指针类型不能参与乘法运算，
  - 指针与指针不能相加，
  - 取地址操作只能用于左值表达式，
  - 非指针类型不能解引用，等等。

- 带引用类型与函数调用的程序语言词法分析、语法分析和引用类型消去

  在这个任务中，你需要在 WhileD 语言中添加以下语言特性并完成词法分析和语法分析：

  - 普通变量声明：`var x; c` ，其中 `var` 是保留字，`x` 是变量名，`c` 是程序语句
  - 引用变量声明：`ref x = e; c` ，其中 `ref` 是保留字，`e` 是用于引用初始化的左值表达式
  - 函数调用：`f(e1, e2, ..., en)` ，这既可以是表达式，也可以是语句
  - 函数定义，以下都是合法的函数定义语法（即参数可以是引用类型参数也可以是普通类型参数）：`func f(x1, &x2) { c }` ，`func f(x) { c }` ，`func f(& x, & y) { c }` ... 其中 `func` 是保留字，`c` 是函数体，一个完整的程序由若干个函数定义构成

  除了词法分析和语法分析之外，你还要作语法变换，并消去所有引用变量和引用参数，用取地址操作和解引用操作将它们改为普通指针变量。

- 简单数学表达式的词法分析、语法分析和简单等性判断

  - 定义含整数常数、变量（单个字母）、加、减、乘（乘号可能省略）、除、幂运算、开根号、自然对数、三角函数的以及括号（仅限小括号）的数学表达式语法树
  - 实现这类表达式的词法分析和语法分析
  - 将包含加减乘以外任何运算符的子式看作一个整体，利用多项式的变换判定两个表达式是否相等，例如 $1+x$ 与 $x+1$ 应当被判定为相等，但是你的程序不必识别出 $1-1/x$ 与 $(x-1)/x$ 之间式相等的
  - 将指数为常数 2、常数 3 的情况也纳入上述等性判断的处理范围

- 一阶逻辑表达式的词法分析、语法分析和简单语法树处理

  $R(x, f(x))$、$\forall x.\ P(x)$、$\forall x.\exists y.\ P(g(y)) \wedge Q(x, y)$ 等等都是典型的一阶逻辑命题。具体而言：

  - 一阶逻辑命题的语法由量词（$\forall\exists$）、逻辑连接词 $\wedge\vee\rightarrow\leftrightarrow\neg$ 和原子命题构成

  - 原子命题具有形式 $P(t_1, t_2, ..., t_n)$，其中 P 是一个谓词的名字（由字母和下划线构成），$t_1 t_2 ... t_n$ 都是“项”
  - “项”要么是变量（由字母和下划线构成）要么具有形式 $f(t_1, t_2, ..., t_n)$，其中 f 是一个函数名（由字母和下划线构成）
  - 变量名、函数名、谓词名之间都不会重名

  在这个任务中，你需要

  - 对一阶逻辑命题进行词法分析和语法分析
  - 对一阶逻辑命题进行简单的语法变换：将所有的 $\phi \leftrightarrow \psi$ 替换为 $(\phi \to \psi) \wedge (\psi \to \phi)$
  - 实现正出现量词与负出现量词的识别与处理

  下面是一个量词正出现和负出现的例子，在

  $$(\forall x.\ P(x)) \to \big((\forall x.\ Q(x)) \wedge \neg(\exists x.\ R(x))\big)$$

  中，$\forall x.\ P(x)$ 与 $\exists x.\ R(x)$ 是负出现，而 $\forall x.\ Q(x)$ 是正出现。在这个任务中，识别并处理正出现量词与负出现量词的 C 函数应当以两个函数指针为参数，其中一个处理被识别为正出现的量词（以及这个量词为语法树根节点的子命题），另一个处理被识别为负出现的量词（以及这个量词为语法树根节点的子命题），这两个函数指针都以一阶逻辑命题语法树的节点指针为参数。例如，在处理上方一阶逻辑命题示例时，会调用处理正出现的函数指针一次，调用处理负出现的函数指针两次。本任务中，用于一阶逻辑项与命题语法的数据结构已经在 syntax.h 中提供了（详见 `FOL.zip`）。

- 从 WhileD 语言程序的语法树生成控制流图

  这个任务中，你的输入是 WhileD 程序，输出是若干个基本块，以及基本块中的程序语句。你需要：

  - 完成表达式拆分
  - 完成基本块构建
  - 正确处理短路求值
  - 正确处理 while 循环语句

- 寄存器分配与汇编代码生成

  - 在控制流图上计算每个指令的 use 和 def，要考虑算数运算加减乘、大小比较与内存读写
  - 基于 use 和 def 完成活性分析，计算 in 和 out
  - 基于活性分析的结果生成 interference graph
  - 基于 interference graph，通过 simplify、spill 操作完成寄存器分配
  - 基于 spill 结果重构控制流图
  - 实现将 start-over 考虑在内的寄存器分配

- 带结构化宏的程序语言的词法分析、语法分析与宏展开

  这个任务中，你需要在 WhileDB 语言中加入函数过程调用与结构化的宏。我们称一个宏是结构化的，如果宏参数的语法树在宏展开之后不会被破坏。具体而言，一个结构化的宏要么是一个表达式宏（它的参数都是表达式）要么是一个程序语句宏（它的每个参数可以是变量名、表达式或者语句）。具体而言，你需要

  - 定义宏定义的语法
  - 定义含宏的 WhileDB 语言语法
  - 在不展开宏、将结构化的宏也当作特定语法结构的前提下，完成词法分析、语法分析，并能输出语法树用于调试
  - 在上述语法树上实现宏展开，并能够输出展开后的语法树。

- 带标注 SimpleWhile 语言程序的词法分析和语法分析

  带标注的 SimpleWhile 语言程序包括：

  - 开头的 `require` 和 `ensure` 条件
  - 带 `inv` 标注的 SimpleWhile 程序语句，一个合法的带标注程序语句中每个 `while` 语句之前都应该有一个循环不变量

  `require` 条件、`ensure` 条件和 `inv` 断言都是整数算数运算的一阶逻辑命题

  - 断言由量词（`forall` 与 `exists` ）、逻辑连接词（`&&` 、`||` 、`->` 、`<->` 与 `!` ）和原子命题构成
  - 原子命题是项之间的大小比较：`t1 <= t2` ，`t1 < t2` ，`t1 == t2` ，`t1 != t2` ，`t1 >= t2` ，`t1 > t2` ，其中 `t1` 和 `t2` 是“项”
  - “项”要么是程序变量，要么是逻辑变量，要么是常数，要么是算数运算 `t1 + t2` ，`t1 - t2` 和 `t1 * t2`

  在这个任务中，你需要对带标注的 SimpleWhile 语言做词法分析和语法分析，另外，存储带标注 SimpleWhile 语言语法的数据结构以及一些辅助构造函数已经在 lang.h 和 lang.c 中提供了（详见`annotated_simple_while.zip`）。

- 带标注 While 语言的符号执行和 VC 生成

  在这个任务中，你需要对带标注的 SimpleWhile 语言进行符号执行并且生成 VC。你不需要对带标注的 SimpleWhile 语言进行词法分析和语法分析，你的 C 程序以语法树 AST 为输入参数，以 VC 列表为输出。存储带标注 SimpleWhile 语言语法的数据结构以及一些辅助构造函数已经在 lang.h 和 lang.c 中提供了（详见`annotated_simple_while.zip`），这些辅助构造函数可以用于构建测试数据。

- 伪代码程序的词法分析、语法分析和 monad 程序生成

  在这个任务中，你需要设计伪代码程序的语法，对下面这样的伪代码程序进行词法分析和语法分析，并将其打印为 Coq 中的 SetMonad 程序。

```
a1 <- average(S1);;
a2 <- average(S2);;
cmp(a1, a2)
```

```
loop_init:
    b <- 1;;
    a <- x
loop_body:
    if (n > 0):
        if (n % 2 == 1):
            b <- b * a;;
        a <- a * a;;
        n <- n / 2;;
        continue
    else:
        break
```