COMP3173 24 f

项目描述

目录

序言

项目名称:设置代数计算器

分组:

两个学生在一个小组，不需要在同一个小组。分组完成后，学生不允许改变分组。请按照 10 月 8 日 Lily 邮件中的指示，在 **AUTOLAB** 上完成分组。只有组内两名学生均确认，分组才算完成。

摘要目的:

在这个项目中，你将开发一个小型的集合代数计算器。这个代数被简化了，它只由整数、整数集合、谓词、整数运算符、集合运算符、逻辑运算符和关系运算符组成。这种语言被设计得直截了当、简单明了，让你更容易实现。所以，有些功能看起来很连线。如果你看到了，请接受它们，并认为它们是有意设计的选择。在开始项目工作之前，你必须彻底审阅这份文件。请注意，这篇文档很**长**。

输入将是一个源代码文件，其中可能包含零个或多个变量声明(整数或集合)和一个单一的计算表达式。然后计算器将分析在集合代数中表述的源代码。分析器应该检测并突出显示源程序中出现的任何错误;如果没有发现错误，计算器将继续进行评估。

本文档的第一部分用例子解释了语言的特性，就像一个编程语言的用户手册。通过阅读它，学生将获得对该语言的全面理解，使他们能够用该语言编写表达式，并识别源代码中的错误。文件的后续部分概述了不同阶段的执行说明。学生需要通过依次完成

这 3 个阶段来完成项目。

实现语言:

C 或 Python，你可以自己选择。两种语言的启动代码和指令都在包中给出。

评分:

- 提交- 5%
- 汇编- 5%
- 第一阶段- 40%
- 第二阶段- 30%
- 第三阶段打字- 10%
- 第三阶段评估- 10%
- 奖励 1:不用分配律的“简化”——2%
- 奖励 2 用分配律“简化”- 2%
- 加分 3 函数或不等式阶测试- 1%

DDL:

我们没有为每个阶段放置 ddl。整个项目的 DDL 将在 **12 月 17 日午夜**，最后一天上课。但是，**不要等到最后一分钟**。请预计这个项目将耗费 **40 个工时**。学生将向 **AUTOLAB** 提交他们的作品。(后续会给出更多说明。)标记将完全基于 AUTOLAB 的测试用例。

测试点:

更多的测试用例将不会与此项目描述。因为学生可以在 Set Algebra Playground 上检查正确的结果。**(**←请试试这个!!!)同学们也可以自己写测试用例，更好地理解语言。

语言功能

列出了该语言的使用手册。

关键词:

- “`let`”:初始化变量声明。
- “`be`”:在其他编程语言中充当赋值运算符。
- "`int` ":在声明时指定变量是**整数类型**。
- “`set`”:在声明时指定变量是 **set 类型**。
- “`show`”:在其他编程语言中充当主函数，初始化一个计算。

数据类型:

- **整数**:
  - 基本数据类型
  - 个位数整数是任意的十进制数。
  - 多位数整数以一个非零的小数开始，后面跟着任意的小数序列。

- 用户只能声明非负整数。负整数是通过减法运算来构造的。

- **算术表达式**:
  - 构造的数据类型
  - 用户**不**允许**声明**算术表达式。语言中没有针对算术表达式的特定数据类型关键字。这个数据类型只存在于编译器中。你也可以检查关键字。算术表达式没有数据类型关键字。
  - **原子算术表达式**要么是整型常量，要么是整型变量。
  - 复合算术表达式由由算术运算符连接的两个算术表达式组成(加法“+”，减法“-”，乘法“*”)。而圆括号则用来定义表达式中的子结构。例如,

    ```
    1 + 2 - 3 * 4
    ```

    解析为:

    ```
    (1 + 2) - (3 * 4)
    ```

- **谓词**
  - 构造的数据类型
  - 与算术表达式一样，谓词**不**直接**由**用户声明，而是由编译器管理。
  - **原子谓词**是一种关系比较，可以有两种类型:
    - **整数值比较**:涉及两个整数的比较，使用关系运算符小于(“<”)、大于(“>”)或相等(“=”);或
    - 隶属度测试**(Membership testing)**:使用隶属度运算符“@”来确定某个元素是否是集合的成员。
  - **复合谓词**是通过使用逻辑运算符组合“较小的”谓词(原子谓词或复合谓词)而形成的:
    - **二元逻辑运算符**:由连词(“&”)或析取(“|”)连接的两个(原子谓词或复合谓词)
    - **一元逻辑运算符**:在另一个谓词之前的否定(“!”)。

    例如,

    ```
    P & q
    ```

    和

    ```
    !R
    ```

    其中 P、Q 和 R 是谓词。
  - 括号也用来定义谓词中的子结构。括号对于定义谓词内操作的优先级和分组至关重要，确保复杂表达式的正确求值。

- **Bool**:
  - 基本数据类型
  - 不能由用户**声明**
  - 只有两个常量:“true”和“false”
  - 布尔值是由一个谓词的求值产生的，没有未初始化的变量。例如,

    ```
    X > 5
    ```

    是一个谓词，表示变量 x 尚未初始化。但如果 x 之前已经初始化为 3，那么谓词就变成

    ```
    3 > 5
    ```

并且可以被评估为布尔值“false”。“>”的行为将在本文档的后面解释。

- **设置**:
  - 构造的数据类型
  - 可以由用户声明
  - 集合是在语言中使用这种语法定义的

    {x: P(x)}

    在哪里
    - 集合定义包含在花括号“{}”中;
    - “x”是一个叫做“representative”的变量名，其作用范围限定在**这个集合**定义之内;
    - “:”是另一个标点符号，将代表 x 与定义的其余部分分开;
    - “P(x)”是一个谓词，应用于变量 x，作为集合的特征函数，对 x 进行逻辑测试，如果 P(x)的计算结果为真，则 **x 是集合的一个**元素**;** 否则，x 不在集合中。
  - 本项目仅关注整数集合。其他类型的集合，如字符串的集合、对的集合或集合的集合不包括在内。这种限制是有意为之，旨在简化实现过程。Goliath 试图让你的生活变得简单!

- **无效**:
  - 基本数据类型
  - 不能由用户**声明**
  - 用于没有任何类型的子表达式

标识符:由英文小写字母组成的任意字符串，不保留关键字。

运营商:

- **算术运算符**:
  - “+”:整数加法，计算两个整数的和
  - “-”:整数减法，计算两个整数之差
  - “*”:整数乘法，计算两个整数的乘积
  - 乘法优先级最高。加法和减法的优先级相等，优先级比乘法低。

- **关系运算符(用于整数)**:
  - “<”(小于):如果左边的整数小于右边的整数，则返回“true”
  - “>”(大于):如果左边的整数**大于**右边的整数，返回“true”
  - “=”:(等于)，如果左边的整数和右边的整数**相等**，返回“true”

- **关系运算符(成员关系)**:
  - “@”:这个运算符检查左边的元素是否是右边集合的成员。如果元素**在集合中**，则返回“true”，否则返回“**false**”。

- **逻辑运算符**:
  - 连词“&”，析取“|”，否定“!的行为如下所示。

逻辑运算符的真值表:

&	真正的	假
真正的	真正的	假
假	假	假

结合表

|	真正的	假
真正的	真正的	真正的
假	真正的	假

析取表

x	真正的	假
x !	假	真正的

否定表

  - 否定的**优先级**最高，然后是合取，然后是析取。

- **运营商**:
  - “I”:设置交集，计算两个集合的**交集**
  - “U”:设置并集，计算两个**集合**的并集
  - 交集的**优先级**高于并集。

句子:

- 每个**源代码**包含 0 个、1 个或多个**变量声明**;以及**一个精确的计算表达式**。

- 每个**变量声明**的语法如下

  设 T id 为 E。

  在哪里
  - T 是类型名称(int 或 set)，
  - Id 是变量标识符，
  - E 是一个将特定类型的值(集合定义也是一个“值”，即使它不是数字)赋给变量的表达式。
  - 句点"标志着宣言的结束。

  例如,

  设 int a 为 5。

  定义一个整数 a，其值为 5。和

  设 s 为{x: x = 5}。

  定义一个只包含一个整数 5 的集合 s。

- **计算表达式**以关键字“show”开始，后面跟着**代数表达式，代数表达式可以是**算术表达式、布尔表达式(所有变量初始化的谓词)，也可以是集合代数表达式。计算表达式也以句号“。”结束。例如,

  - 计算集合 S1 和 S2 的并集:show S1 U S2。
  - 计算整数 1 和 2 的和:show 1 + 2。
  - 要测试整数 3 是否在 S1 中:显示 3 @ S1。

  输出:计算结束后，程序将结果(类型和值)打印到屏幕上。(见下面的例子)。对

于集合运算符，“show”语句并没有简化特征函数。例如,

```
show {x: x > 3} U {x: x > 5}。
```

将输出

```
{x:(x > 3) | (x > 5)}
```

而不是

```
{x: x > 3}
```

即使两个集合是等价的。这个要求会使这个项目变得容易。谓词的简化是一个高级功能，将是一个额外的好处，稍后会解释。

例子

- 示例 1:

```
show 3 .
```

它只是打印整数 3。所以，预期的结果是

```
(int) 3
```

- 示例 2:

```
let int x be 3 .
show x + 1 .
```

例 2 声明了一个整数 x，其值为 3。它做了一个简单的计算。所以，结果是

```
(int) 4
```

- 示例 3

```
let int x be 3 .
let int y be 4 .
let set s be { a : a = 3 | a = 4 } .
show x @ s .
```

这个例子简单地测试了整数 x 是否在集合 s 中。结果是

```
(bool) true
```

- 示例 4

```
let set x be { a : a > 3 } .
let set y be { a : a < 5 } .
show x I y .
```

这个例子声明了两个集合并计算交集。答案是

```
(set) { a : ( a > 3 ) & ( a < 5 ) }
```

- 示例 5

子集也可以构造。

```
let set x be { a : a > 3 } .
let set y be { a : a < 5 & a @ x } .
let set z be { a : a > 0 } .
show z I y .
```

集合 y 是 x 的子集。

- 例子 6

```
This is a lexical error .
```

例 6 有一个词法错误，因为字母“T”不在字母表中。“This”不是关键字，所以它必须是标识符。然而，标识符是由英文小写字母组成的字符串。不包括大写字母。

- 例 7

```
this is a syntax error .
```

在这个例子中，每个单词都将被标记为标识符。因此，没有词法错误。但是，它的结构是错误的。

- 示例 8

```
let set x be 4 .
let set y be 3 .
show x @ y.
```

这是一个类型错误的例子。x 和 y 都是整数，但关系运算符“@”需要第二个运算符是集合。

- 例 9

“show”语句也可以计算布尔表达式(没有未初始化变量的谓词)。例如,

```
show 3 < 5 & 2 > 4 .
```

可以评价为

```
(bool) false
```

第一阶段-词法分析

时间:学生可以在“第 3 课-有限自动机”之后开始第一阶段的工作。

描述:

在第一阶段，学生需要为这个项目实现一个词法分析器。词法分析器将源代码读取为字符流，将其切割为词素，将词素分类为标记，并为一些词素决定属性。如果源代码中有词法错误(拼写错误)，词法分析器也会报告。标记先前在语言用户手册中有定义，

并在下面再次总结。每个只包含一个词位的词例是

- 关键字:let,be,show,int, set
- 标点符号:。， (，)， {，}，:
- 算术运算符:+、-、*
- 关系运算符:@、<、>
- 逻辑运算符:&、|、!
- 集合运算符:U、I

而包含无限词素的 token 是

- 整数常量:num
- 变量名:id

空格符号“ ”是一个特殊字符，词法分析器会忽略它，但会强行终止其他词素。例如,

- “let be”被认为是两个 token“let”和“be”，因为空格符号终止了词法“let”。
- “letbe”被识别为一个 token“id”，因为它是一串低阶英文字母，而不是关键字。
- “be123+a”被识别为四个 token“be”、“num”、“+”和“id”。

符号表在这一阶段也需要实现。它保存了用户声明的所有变量名(id)。每个变量名都有另外两个属性:

- 变量的类型(“int”或“set”);
- 变量的值。

注意，set 变量的值不是一个数字，而是一个 set 定义。例如“设集合 s 为{x: x = 5}”，“s”的值为“{x: x = 5}”。但是在第 1 阶段，词法分析器无法为集合决定一个值。

词法分析器被解析器操作为函数调用“next_token()”。它从输入中读取几个字符，并返回第一个识别出的 token。

- 如果标记是一个整型常量 num，函数也返回它的值。
- 如果标记是一个变量名 id，函数将这个变量名存储在一个符号表中，并返回符号表中的变量位置(例如 C 语言中的一个点)。
- 对于其他符号，函数只返回符号名称。

阶段 2 -语法分析

时间：学生可以在“第 8 课-SLR 解析器”之后开始第二阶段的工作。

描述:

这是编译器的第二阶段。学生应该为这种语言实现一个 **SLR(1)解析器**。语言语法在“SLR 语法。txt”中给出。SLR(1)解析表也在“SLR 解析表。csv”中给出（您的程序

可以将该文件作为文本文档读取）。解析器**使用栈工作，栈**中包含已经**解析过的配置**。解析器还需要维护一个**语法树**来呈现源代码的语法结构。当解析器缩减其堆栈时，就会创建树的子结构。语法树中的每个节点都有**两个属性:类型**和**值**。属性在阶段 2 中未初始化，但将在阶段 3 语义分析中使用。

阶段 3 -语义分析

时间：学生可以在“第 12 讲-类型系统第二部分”之后开始第三阶段的工作。事实上，第 11 讲和第 12 讲是用于类型检查的语义分析的例子，这相当简单。我们鼓励学生自学，并尽早开始第三阶段的工作。

描述:

语义分析是编译器前端的最后一个阶段，也是这个项目的最后一部分。在这一部分中，语义分析器检查类型并评估源代码。如果有类型错误，分析器也会报告。

语义分析器应该被实现为一个函数，这

- 包含所有语法规则的语义定义，和
- 每当解析器使用相应的语法规则减少其堆栈时，执行一个特定的语义计算。

语义分析的结果存储在栈中每个结构的属性和解析器的解析树中。(回想一下，在阶段 2 中，语法的每个结构都与两个属性相结合:类型和值。)因此，你的解析器需要稍作修改才能与你的语义分析器进行交互。

类型检查规则已经在“type Checker.docx”中给出。学生可以直接使用它们。但对于价值评估，学生需要自己设计。

输出

编译器的输出包括以下内容。

1. **屏幕上**按标准输出**的一条消息**

   如果源代码有错误(词法、语法或类型)，编译器应该识别它，并显示错误消息(**“词法错误!**、**“语法错误!**”，或者**“类型错误!**”)，并停止分析过程。这个项目**没有错误恢复**。此外，如果检测到任何错误，以下 json 文件将保留为**空**(创建但为空)。
   如果源代码没有错误，编译器需要显示一条消息(**“词法分析完成!”)。**、**“句法分析完成!”**或**“语义分析完成!”，**分别表示已实现的阶段。
   此外，如果你已经实现了阶段 3 的评估，你的编译器还需要在屏幕上**打印评估的结果**。

2. **一个**名为“**lexer_out.”的文件。Json**”通过文件写入

   这个文件包含词法分析器的结果(**词法和**相应的**标记**)作为**字典的列表**。对象列表由一对方括号“[]”括起来。而字典由两对组成，并由一对大括号“{}”括

起来。第一对是一个字符串“token”和 token 的名称。第二对是字符串“词法位”和词法位。例如,

```
show 3 .
```

“lexer_out。Json”应该是

```
[
    {
        "token": "show",
        "lexeme": "show"
    },
    {
        "token": "num",
        "lexeme": "3"
    },
    {
        "token": ".",
        "lexeme": "."
    }
]
```

请注意，空格、缩进和换行不是必需的。它们只是用来提高可读性的。因此，上面的例子也可以

```
[{"token":"show","lexeme":"show"},
{"token":"num","lexeme":"3"},{"token":".","lexeme":"."}]
```

3. **一个**名为“**parser_out.”的文件。Json**”的文件写入

第二阶段的结果必须写入“**syntax_out. json”。Json**”使用圆括号来呈现解析树的层次结构。解析树中的每个非终结符都是一个由两对组成的字典。第一对表示非终结符的名称。第二组显示的是儿童列表。每个终端都以与词法分析器输出相同的方式呈现。例如,

```
show 3.
```

“syntax_out。Json”应该是

```
{
  "name": "S",
  "children": [
    {
      "name": "C",
      "children": [
        {
          "token": "show",
          "lexeme": "show"
        },
        {
          "name": "A",
          "children": [
            {
              "name": "E",
              "children": [
                {
                  "name": "E'",
                  "children": [
                    {
                      "name": "E''",
                      "children": [
                        {
                          "token": "num",
                          "lexeme": "3"
                        }
                      ]
                    }
                  ]
                }
              ]
            }
          ]
        }
      ]
    },
    {
      "token": ".",
      "lexeme": "."
    }
  ]
}
```


同样，空格、缩进和换行符只是为了可读性。

4. 名为“**typing_out”的文件。Json**”的文件写入
   这个文件包含了与“syntax_out.json”结构相同的类型检查解析树。但是树中的每个节点都包含一个名为“**type**”的属性。对于终端来说，属性的顺序是“token”、“lexeme”和“type”。对于非终端，顺序是“名称”、“类型”和“子女”。对于上面的例子，“typing_out.”Json”将是

```
{
  "name": "S",
  "type": "calculation",
  "children": [
    {
      "name": "C",
      "type": "calculation",
      "children": [
        {
          "token": "show",
          "lexeme": "show",
          "type": "void"
        },
        {
          "name": "A",
          "type": "calculation",
          "children": [
            {
              "name": "E",
              "type": "integer",
              "children": [
                {
                  "name": "E'",
                  "type": "integer",
                  "children": [
                    {
                      "name": "E''",
                      "type": "integer",
                      "children": [
                        {
                          "token": "num",
                          "lexeme": "3",
                          "type": "integer"
                        }
                      ]
                    }
                  ]
                }
              ]
            }
          ]
        }
      ]
    },
    {
      "token": ".",
      "lexeme": ".",
      "type": "void"
    }
  ]
}
```


5. 名为 evaluation_out. json 的文件。**Json**”的文件写入

类似于“typing_out. json”Json”，这个文件保存着求值的结果。所以，每个解析树节点都有属性“`evaluation`”(省略“`type`”)。

注意:语言设计得很好。语法为 SLR(1)语法，语义为 l 属性。因此，这三个阶段可以同时工作。一旦词法分析器标记一个词法位，解析器就可以开始解析，同时类型检查器可以检查类型。因此，你需要一个临时的内存位置来保存中间结果(词法、标记、解析树等)。如果整个源代码都没有发现错误，这些中间结果就会被写入 json 文件。记住，如果发现任何错误，json 文件是空的。

奖金

“这个奖励就像冰淇淋底部的那一口巧克力。”

-CGY

“`show`”语句只能做一些计算，但不能简化集合定义中的谓词。例如,

```
show { a : 0 = 0 } .
```

仅返回

```
(set) { a : 0 = 0 }
```

然而，“`0=0`”是重言式(总是为真)。ℤ因此，这个集合实际上与所有整数的集合是相同的。此外，一个集合可以有多种形式的定义。例如,

```
show { a : a < 10 | a > 5 } .
```

也定义了，但与前面的例子不同。ℤ

为了克服这个问题，我们的语言被扩展了一个新的关键字“`simplify`”，它可以在以下规则下简化集合定义中的谓词。

- 一个空集合总是以{a: false}的形式呈现，因为“`false & P`”与支配律中的“false”是相同的。例如,

  ```
  simplify { a : 0 = 1 & a > 0 } .
  ```

  返回

  ```
  (set) { a : false }
  ```

- 全集总是以ℤ{a: true}表示，类似于空集。例如,

```
simplify { a : 0 = 0 | a > 0 } .
```

返回

```
(set) { a : true }
```

- 谓词应简化为析取范式（DNF）。一般来说，DNF 中的谓词由一组最小项组成，并由析取连接。而每个 minterm 都由一堆原子谓词(有或没有否定)组成，并由连词连接。例如,

```
A = 1
(a = 1) | (a = 2)
(a > 1) & (a < 3)
((a > 1) & (a < 3))A > 5)
```

在 DNF，但是

```
(a = 1) & (a = 2 | a =3)
!((a > 1) & (a < 3))
```

不在 DNF 中。在使用 DNF 之后，由于优先级的关系，可以毫无困难地删除所有括号。上述 DNF 表达式可为

```
A = 1
A = 1 | A = 2
A > 1 & A < 3
A > 1 & A < 3 | !A > 5
```

在我们的语言中，每个 minterm 都是 either

- 单个较大或较小的比较(例如“A > 0”，或“A < 3”)或
- 较大或较小的连词(例如“a > 5 & a < 10”)。大的比较总是在小的比较之前。
- 此外，minterms 需要按照下界递增的顺序排序。例如,

  ```
  A > 10 & A < 15 | A < 3 | A > 5 & A < 8
  ```

  重新排列为

  ```
  A < 3 | A > 5 & A < 8 | A > 10 & A < 15
  ```

  因为“a < 3”、“a > 5 & a < 8”、“a > 10 & a < 15”的下界分别为负无穷、5、10。
- 而两个 minterms 定义的范围不相交。例如,

  ```
  A > 10 & A < 15 | A < 3 & A > 8
  ```

  应该简化为

  ```
  A > 3 & A < 15
  ```

为了完成这些特征，需要定义一些评估规则。

- “a = b”求值为“a > b - 1 & a < b + 1”。
- “a > b1 & a < b2 | a > b3 & a < b4”评估为
  - “a > min(b1,b3) & a < max(b2,b4)”，如 b1b3b2b4≤<≤或 b1b3b4b2≤<≤;
  - “a > b1 & a < b2 | a > b3 & a < b4”否则。
- 如果“x”是一个初始化的变量，它被赋值为“x.value”。例如，“a>x”

变成“a>x .value”。

- 如果“x”是一个初始化的集合(其形式为{a:P(a)})，则谓词“a @ x”被求值为“P(a)”。例如,

设 s 为{a: a > 10}。

设 set t 为{a: a > 5 | a @ s}。

集合 t 的评价为{a: a > 5 | a > 10}，进一步评价为{a: a > 5}。

- 注意，为了简化表达式，在某些情况下需要使用分布律(将合取词分布到析取词)。例如，集合交集

{a: a > 5} I {a: a < 10 | a > 15}

等同于

{a:(a > 5) & (a < 10 | a > 15)}

但在 DNF 中没有。所以，可以简化为

{a: a > 5 & a < 10 | a > 5 & a > 15}

并进一步简化为

{a: a > 5 & a < 10 | a > 15}

- 即使有上述实现，“简化”在某些情况下也无法正常工作。算术表达式可能有乘法，这可以在集合定义的谓词中。例如，集合可以是

{a: a * a * a + 2 * a * a + 5 < 10}

要简化集合定义，必须解三次不等式，这已经不容易了。那么，高阶不等式呢? 因此，我们的语言排除了所有高于 1 阶的函数或不等式。而你的语义分析器需要计算函数的阶数。如果阶数高于 1，“simplify”会报告一个语义错误;否则，表达式可以正常简化。

奖金等级

- 2%是“简化”陈述，没有分配法
- 2%适用分配法
- 1%用于函数或不等式阶测试(此功能将单独测试)