作业 1（主要部分）:算术表达式

1.0 版

截止时间:2022 年 9 月 16 日星期五 22:00

本作业的目的是获得一些初步的动手编程经验
与哈斯克尔。在开始工作之前,请通读整个作业文本。

注意该作业还包括一系列简单的热身练习,如 Absalon 中所述。对于这些,您只需要提交您的工作代码,而不是单独的设计/实施文档、评估或测试证据。如果您想就热身练习的解决方案交流任何其他内容,请将您的评论作为注释放在源代码中。

1 简单的算术表达式

考虑以下代数数据类型,表示简单的算术表达式:

```
数据Exp = Cst
        整数|添加经验
    Exp |子经验 Exp |
    Mul Exp Exp | Div
    Exp 经验| Pow Exp
    Exp ...（附加构造函
    数;见下一节）
--
```

也就是说,算术表达式要么是（无界）整数常量,要么是对两个子表达式的以下五种运算之一:加法、减法、乘法、除法或幂（又名求幂）。请注意,没有规定在表示中包含显式括号,因为 Exp 类型值的树结构已经编码了预期的分组:通常写为 2 × (3 + 4) 的算术表达式表示为 Mul (Cst 2) (Add (Cst 3) (Cst 4)),而 (2 × 3) + 4 对应于 Add (Mul (Cst 2) (Cst 3)) (Cst 4)。

1.1 打印表达式

定义一个函数

```
showExp :: Exp ->字符串
```

将算术表达式呈现为字符串,使用普通的数学/Haskell 中缀表示法,如上面的示例所示。使用“+”、“-”、“*”、“`div`”和“^”来表示五个算术运算符。 （在 Haskell 中,运算符“/”表示除以可能的小数结果,这不是这里的意图。）在输出中包含足够的括号以

确保输出字符串可以作为 Haskell 表达式被正确读取和评估,并且任何两个不同的 Exp 树具有不同的渲染（即使它们会评估相同的结果,例如“(2+3)+4”和“2+(3+4)”）。明确允许您在输出中插入名义上冗余的（根据通常的数学约定)括号,例如“((2*3)+4)”。

如果要打印的表达式不是上述六种形式之一（即,如果它属于上述数据 Exp 声明的 ... 部分）,您的代码应该使用适当的消息显式报告问题（使用标准函数错误）,而不是因非详尽的模式匹配错误而崩溃。

1.2 评估表达式

算术表达式可以计算为数值结果。在这个作业中,我们只考虑整数算术,其中 n div m (for m 6= 0) 定义为 b n
$\frac{}{*}$c,其中 brc（r 的下限)是
小于或等于 r 的最大整数。 （所以 b Pow 操作中的指数（第 $\frac{7}{2}$c = 3,而 b $\frac{-7}{2}$c = -4。)此外,我们要求
二个子表达式)是非负的,我们指定
0^n = 1 对于所有整数 n,包括 0。
定义一个 Haskell 函数

```
evalSimple :: Exp ->整数
```

这样 evalSimple e 在上面指定的算术运算符的解释下计算 e 的值。如果在内置操作中发生错误（例如,除以零）,只需使用相关的 Haskell 运行时错误中止即可。而且,与打印一样,不在 Exp 的“简单”片段中的表达式应该明确报告为错误。如果一个表达式中有多个错误,那么报告哪一个都没有关系。

2 扩展算术表达式

我们现在考虑由完整数据类型给出的更丰富的表达式类：

```
data Exp = ...
--      （上面的 6 个构造函数）
    |如果{test, yes, no :: Exp}
    |变量 VName |让
    {var :: VName, def, body :: Exp}
    | Sum {var :: VName, from, to, body :: Exp}

类型VName =字符串
```

这里,表达式形式 If e1 e2 e3 （或者,更详细地说,If {test = e1, yes = e2, no = e3})表示条件表达式（类似于C 中的e1 ? e2 : e3 ）。也就是说,如果e1的计算结果为非零数,则它的值要么是e2的值;要么是 e2 的值。或e3 的值,如果e1的计算结果为零。仅评估选定的分支e2或e3 ;例如,计算表达式

```
如果{test = Sub (Cst 2) (Cst 2),yes = Div (Cst 3)
        (Cst 0),no = Cst 5}
```

应该返回 5,并且不会因除以零而中止。

表达式 Var v,其中 v 是变量名（表示为 Haskell 字符串），返回变量 v 的当前值（如下所示）。如果变量没有当前值,则会发出错误信号。

相反,表达式 Let v e1 e2用于（仅)在计算 e2 期间将变量 v绑定到e1的值;之后,v 的先前绑定（如果有)被重新实例化。因此,例如,表达式

```
让{var =  x  , def = Cst 5,
      body = Add (Let {var =  x  , def = Add (Cst 3) (Cst 4), body = Mul (Var  x  ) (Var
                                          x  )})
                  (var  x  )}
```

对应于更易读的 Haskell 符号

```
让x = 5 in (让x = 3 + 4 in x * x) + x
```

应该计算为 (3 + 4)2 + 5 = 54。但是,与 Haskell 不同,Let 绑定是非递归的;也就是说,在e1中出现的任何 v 都指的是 v 的可能外部绑定,而不是e1本身。

如果绑定变量 v 未在主体e2中实际使用,是否应通知定义表达式e1中发生的错误,这是故意未指定的（即,您作为实现者可以决定）。例如,

```
让  x  (Div (Cst 4) (Cst 0)) (Cst 5)
```

允许评估为 5,或因被零除错误而中止（但仅此而已）。

最后,表达式形式 Sum v e1 e2 e3对应于数学求和符号Pe2 e3。也就是说,它首先将e1和e2计算为数字n1和n2,然后计算计算 e3 的结果之和,其中 v 依次绑定到每个值n1、 n1 + 1、 ...。例如,表达式

n2

```
总和“x”  (Cst 1) (加(Cst 2) (Cst 2))
      (Mul (Var  x  ) (Var  x  ))
```

计算结果为12 + 22 + 32 + 42 = 30。如果n1 > n2,则和定义为 0（并且根本不应计算e3）。

我们跟踪环境中的变量绑定,将变量名映射到它们的值（如果有的话）。我们将环境表示为功能值：

```
输入Env = VName -> Maybe Integer
```

也就是说,对于环境 r :: Env 和变量 v :: VName,如果 v 在 r 中没有绑定,则应用程序 rv 返回 Nothing,如果 v 绑定到整数 n,则返回 Just n。所有变量未绑定的环境可以简单地写为 initEnv = \v -> Nothing,而变量“ans”绑定到 42（并且所有其他变量都未绑定)的环境将表示为以下函数：

```
\v -> if v ==  ans  then Just 42 else Nothing
```

定义一个函数

```
extendEnv :: VName -> Integer -> Env -> Env
```

这样 extendEnv vnr 返回一个新环境 r 其他变量具有与它们在 r 中相同的绑定（如 。 其中 v 绑定到 n,并且所有果有的话）。

然后定义一个函数

```
evalFull :: Exp -> Env -> Integer
```

计算给定环境中的表达式。和以前一样,错误应该用错误发出信号,现在可能 除了错误的算术运算 包括访问未绑定变量的尝试。另一方面,现在应该涵盖 Exp 中的所有表达式形式。

请务必在报告中说明您选择如何处理***中不需要的部分中的错误
让表达式,以及为什么。 (实现的简单性是一个完全可以接受的理由。)

3 返回显式错误

用 Haskell 错误立即中止评估是一个相当激烈的步骤,尤其是无法从概念上非致命的问题中优雅地恢复。一种更灵活的方法使评估器函数返回一个明确指示出了什么问题,并让评估器的用户决定下一步该做什么。因此,我们首先列举一些可能的故障:

```
数据ArithError =
      EBadVar VName -- 未绑定变量
    | EDivZero -- 尝试除以零
    | ENegPower 试图提升到负能量
    | EOther String -- 任何其他错误,如果相关
```

定义一个 Haskell 函数,

```
evalErr :: Exp -> Env -> Either ArithError Integer
```

这样 evalErr er 会尝试在环境 r 中评估 e,就像在 evalFull 中一样,但现在返回错误值(例如 Left ENegPower)或正确结果(例如 Right 42)。 evalErr永远不会导致 Haskell 运行时错误(除非内存不足)。

对于 evalErr,我们还明确指定要从左到右计算所有子表达式,因此,例如,如果我们正在计算 Add e1 e2,并且e1返回错误,则不应计算e2 。但是,仍然未指定(意思是:您应该选择)是否应该报告或忽略未使用的 Let 绑定中的错误。再次,在报告中证明您的选择是正确的。

***

提示: evalErr 的代码可能会变得有些冗长和重复;尝试将通用代码片段抽象为(高阶)辅助函数,这样您只需编写一次。不要尝试使用 Haskell 内置的不精确异常工具:它非常挑剔,很可能无法满足您的需求。

4 可选扩展

本节中的问题更具挑战性,因此不是强制性的,但如果您快速完成主要任务,仍建议您进行额外练习。它们是独立的,所以你可以做它们的任何子集。

请注意,您在可选问题上的表现不会影响您是否通过作业:这里的不完整或错误的解决方案不会拖累强制性部分的其他可接受的解决方案;相反,您无法通过成功解决一个或多个可选问题来挽救主要问题的失败解决方案。

4.1 取幂中的严格误差传播

取幂的语义在上面指定为所有 n 的 n。但是,在一般表达式 Pow e1 e2 中, $0^0 = 1$ 对于所有整数 n,就像 $n \times 0 = 0$
如果对子表达式e1的求值导致错误（被零除、未绑定变量等），我们仍然希望对整个表达式的求值报告相同的错误,即使e2求值为 0（因此实际上并不需要 e1 的实际值）。让您的实现特别是 evalSimple 和 evalFull 确保这一点,这样,例如 (Pow (Div (Cst 0) (Cst 0)) (Cst 0)) 会导致错误,而不是1 的成功结果。（如果您的evaluator 已经发生了这样的行为,你不需要在这里做任何事情！）。简要说明您的解决方案如何/为何起作用。

***

提示:考虑如何确保 Haskell 将某个子表达式计算为一个值,即使该值对整个表达式的最终结果无关紧要。您可以使用内置函数 seq :: a -> b -> b,但这里实际上并不需要它。

4.2 用最少的括号打印

定义 showExp 函数的变体，

```
showCompact :: Exp ->字符串
```

它使用最少数量的括号（并且没有额外的空格)打印一个简单的算术表达式,以便它仍然可以被 Haskell 正确读取和评估。该函数还必须是一对一的,即不应将 Exp 类型的两个不同值呈现到同一个字符串中。您应该假设算术运算符具有传统的优先级和关联性（例如 Add (Cst 2) (Mul (Cst 3) (Cst 4))应该打印为“2+3*4”，并且 Add (Cst 2) (Add (Cst 3) (Cst 4)) 为“2+(3+4)”)。特别注意 2^3^4 corre 2(34)对应于23 ,这通常意味着

^{4}, ，不是 $(23)^4$.

提示:您可能希望根据辅助递归函数定义 showCompact,该函数将要打印的表达式和一些附加数据作为参数,以确定是否需要在它周围使用显式括号,给定它出现的上下文。简要解释您的解决方案如何工作。

***

4.3 显式渴望/惰性语义

在 evalErr 中,Let 表达式的确切语义部分未指定。事实上,表达式 Let v e1 e2 有两种自然的解释:急切的,其中e1总是主动求值,不管 v 是否在e2 中使用;和懒惰的,其中e1仅在计算e2需要其值时才被评估。

因此,定义两个函数，

```
evalEager :: Exp -> Env -> Either ArithError Integer evalLazy :: Exp -> Env ->
Either ArithError Integer
```

实现这两个变体（对于第一个参数中出现的所有 Lets）。例如,调用 evalLazy (Let x (Var y) (Cst 0)) initEnv 现在应该返回 Right 0,而 evalEager (Let x (Var y) (Cst 0)) initEnv 应该返回左（EBadVar“y”）

提示:如果强制部分中的 evalErr 已经精确地实现了这两种行为之一,则可以直接将其用作相关定义。再一次,您可能希望用另一个更通用的函数来表达上面的一个或两个评估函数,该函数具有附加和/或不同类型的参数;特别是,您的内部函数可能对环境使用稍微不同的类型。简要说明你的方法。

***

5 上交什么

5.1 代码

表单为了促进人工和自动反馈,请务必严格遵循本节中的代码打包说明。我们为热身和主要部分中的所有请求功能提供骨架/存根文件。这些存根文件打包在分发的 code.zip 中。它包含一个目录 code/,有几个子目录组织为 Stack 项目。您应该按照指示编辑提供的存根文件,并保持其他所有内容不变。

至关重要的是,您不要更改任何导出函数的提供类型,因为这会使您的代码与我们的测试框架不兼容。此外,不要删除您未实现的任何功能的绑定;只需将它们保留为未定义即可。

提交作业时,将您的代码再次打包为单个 code.zip （不是.rar、.tar.gz 或类似文件）,其结构与原始文件完全相同。重建 code.zip 时,请注意仅包含构成您实际提交的文件:您的源代码和支持文件（构建配置、测试等）,但不要包含过时/实验版本、备份、编辑器自动保存文件、修订版-控制元数据、.stack-work 目录等。如果您的最终 code.zip 比分发的版本大很多,那么您可能包含了一些您不应该包含的内容。

对于热身部分,只需将您的函数定义放在 code/part1/src/Warmup.hs 中,如果有指示。

对于主要部分,您的代码必须放在文件 code/part2/src/Arithmetic.hs 中。它应该只导出请求的功能。任何测试或示例都应放在 code/part2/tests/ 下的单独模块中。为了获得灵感,我们在 code/part2/tests/Test.hs 中提供了一个非常简约（而且远非完整)的测试套件。如果您正在使用 Stack（为什么不使用？）,您可以通过目录 code/part2/ 中的堆栈测试来构建和运行套件。

此分配的定义（例如类型 Exp)可在文件 .../src/Definitions.hs 中找到。
您应该只从该模块导入,而不是直接将其内容复制到 Arithmetic。
同样,不要修改定义中的任何内容。

内容与往常一样,您的代码应该有适当的注释。特别是,尝试为您定义的任何辅助“帮助”函数提供简短的非正式规范,无论是本地的还是全局的。另一方面,避免用英文改写代码在 Haskell 中已经清楚表达的内容的琐碎注释。尝试使用一致的缩进样式,并避免超过 80 个字符的行,因为这些通常包含在打印列表中（或者如果有人使用比您更窄的编辑器窗口）,使它们难以阅读。

您可以（但不应该为此作业)仅从核心 GHC 库导入附加功能:您的解决方案代码应使用提供的 package.yaml 使用从目录 code/partn/ 发布的堆栈构建进行编译。对于测试,您可以选择使用 Stack LTS 发行版的课程强制版本中的附加包,例如测试框架。在课程的稍后部分,我们将使用 Tasty 和 QuickCheck。）

理想情况下,您的代码在使用 ghc(i) -W 编译时应该不会发出警告;否则,请添加注释,解释为什么任何此类警告在每个特定情况下都是无害或不相关的。
如果代码中的某些问题根本无法编译整个文件,请务必在提交之前注释掉有问题的部分,否则所有自动化测试都会失败。

5.2 报告

除了代码之外,您还必须提交一份简短的（通常为 1-2 页)报告,包括以下两点,仅用于主要（非热身)部分：

- 记录您所做的任何（重要的)设计和实施选择。这包括但不一定限于回答作业文本中明确提出的任何问题（在空白处标有*** ,以加强强调）。专注于高层次的方面和想法,并解释为什么你做了一些不明显的事情,而不仅仅是你做了什么。在报告中逐个功能地进行详细的代码演练是非常不合适的;关于函数如何工作的技术注释属于代码中的注释。

·对您提交的代码的质量以及它满足作业要求的程度（尽您的理解和知识)进行诚实、合理的评估。务必清楚地解释任何已知或可疑的缺陷。

记录评估的基础是非常重要的（例如,一厢情愿、零散的例子、系统测试、正确性证明？）。包括您对源代码提交所做的任何自动测试,明确如何运行它们,并在报告中总结结果。如果您的代码的某些方面或属性无法以自动化方式轻松测试,请解释原因。

我们建议（并且可能会在以后强制要求)您将评估分为以下小节/段落：

完整性是否至少在原则上实现了所有要求的（以及可选的)功能,即使不一定完全工作?如果没有,您对如何实现缺少的部分有任何具体的想法吗？

正确性所有实现的功能是否都能正常工作,或者是否存在已知的错误或其他限制?在后一种情况下,您对如何潜在地解决这些问题有任何想法吗？

效率您的代码的运行时时间和空间使用（正如您期望它由 Haskell 执行的那样;您不需要实际对其进行基准测试)是否合理地匹配,至少渐近地,从正确的实现中自然会假设什么?如果没有,您是否有关于如何显着提高代码性能的想法？

可维护性通用代码片段是否通过参数化辅助定义合理共享,或者是否存在大量复制粘贴片段形式的代码重复,并进行了微小的更改？ （请注意,这个问题可能也适用于您的测试套件！)否则代码是否处于您认为良好的状态（正确布局、注释等）？

其他您认为值得一提的其他任何事情,无论是正面的还是负面的。

前两点应该通过参考你的正式测试的结果来证实。对于其他人,您还应该通过相关示例或其他证据来证明您的评估是正确的。

您的报告文档应该是一个名为 report.pdf 的 PDF 文件,并与Absalon 中的代码分开提交。

5.3 时间表

在今年的 AP 课程中,我们特别关注学生在课程中的工作量。为了帮助我们获得比课程结束时的正式课程评估提供的更详细、完整和及时的画面,我们要求您每周填写一份简短的时间表,详细说明您在各种课程上花费了多少时间作业的部分和方面,以及其他与课程相关的活动。

时间表模板位于主代码目录中的单独文件 timesheet.txt 中,并且是机器处理的,因此正确填写它很重要。

特别是,对于每个时间类别,您应该以自然格式报告您花费的时间(小时和/或分钟),例如“2 小时”、“15 米”、“120 米”、“1 小时 30 米”等。记住包含单位特别重要,因为没有默认值。对于花费不到一个小时左右的活动,尽量做到精确;不要只是将它们四舍五入到最接近的小时。

如果您不知道或不记得您在特定类别中花费了多少时间,请给出您的最佳估计。如果您对一个或多个类别没有有意义的数字,或者不想说,只需将时间写为单个“x”。如果您以 2 人小组的形式提交作业,请报告每个类别的平均时间使用情况(即,您个人贡献的总和除以 2)。最后,如果你以前学过 AP(或者知道其他地方的材料),那么了解你本周在课程上花费的实际时间对我们来说比你估计需要多少时间更有用。一切都是第一次。

您可以包括注释(以“#”开头并一直运行到行尾)解释或详细说明您的数字,特别是如果您认为它们可能是非典型的。但是,时间表主要用于自动处理,因此任何文本评论都可能无法系统地注册。如果您有任何重要的观点或意见想要引起我们的注意,您应该直接向教学人员提出。

为帮助确保对类别的解释一致,请使用以下指南
用于计算各种课程相关活动的行:

分配(不是一个实际的类别。留为数字 1 以供识别)。

安装GHC/Stack 在您的平台上安装和运行所花费的时间。不要包括不必要的设置调整(例如让语法突出显示
在您最喜欢的编辑器中工作,更不用说尝试正确配置任何花哨的 IDE 功能;本课程不需要这些。)

本周花在推荐阅读材料上的时间,即使与作业没有直接关系,以及您专门为在作业中取得进展而进行的任何补
充阅读(可能来自建议来源以外)。

讲座花在听讲座上的时间。如果您出于某种原因跳过(部分)讲座,这可能少于标称的每周 4 小时。

探索在非强制性活动中花费在 Haskell 编程上的时间,无论是基于建议的练习还是您自己的实验。这还包括
作业的可选部分的工作,无论您最终是否将其交给反馈。

热身作业热身部分所花费的时间。

开发时间花在编写和调试分配的主要部分的代码上
ment,包括您在开发期间进行的任何集成或临时测试。

测试时间,超出上述开发时间,专门用于以自动化测试套件的形式记录您的测试。请注意,如果您在代码之前或与代码一起编写了相关的测试用例,这可能（接近)为零。

报告编写设计/实施决策和评估的时间。

其他上述类别未正确涵盖的任何其他与课程相关的活动。如果您在这里报告了重要的时间,那么在评论中包含一个简短的解释性说明会很有帮助。

如果上述方向中的任何内容不清楚（并且会严重影响您报告的数字）,请在论坛上寻求澄清。请注意,类别和/或其描述可能会根据需要为以后的分配进行调整。

您的时间表编号（或明显缺少编号)不会影响您的作业成绩,也不会被助教评论。但我们希望您能尽可能准确和完整地回答,以帮助我们正确了解哪些活动对 AP 工作量有显着影响,以及哪些具体的纠正措施可以为以后的分配和/或未来运行课程。

你不必用秒表或类似的东西给自己计时,但一定要尝试在名义上专门用于特定活动的时间段中考虑任何重大的暂停或中断（例如,午休时间）。您可能会发现一个专用的时间跟踪工具（例如 clockify.me 或任何数量的免费应用程序)通常有用且信息丰富,不仅适用于 AP。

5.4 概述

详细的上传说明,特别是关于团体提交的物流,可以在 Absalon 提交页面上找到。

我们还期望提供一个自动化系统,对您计划提交的代码提供初步反馈,包括形式、正确性、样式等问题。详细信息将在 Absalon 上公布。强烈建议您利用此机会验证您的提交,并在必要时修复或以其他方式解决（例如,通过记录已知缺陷)它发现的任何合法问题。

但是请注意,通过我们的自动化测试并不能替代进行和记录您自己的测试。您的评估必须能够独立进行,而无需依赖我们工具的输出。