PPL 2022 Final Practice
参考答案

发布： 2022 年 12 月 27 日
截止： 2023 年 1 月 10 日 23:59
不允许补交

本作业共 17 页，共 4 道大题，满分 100 分。

提交方式

请你在学在浙大的测试中找到「期末作业」，在其中提交你的答案。你可以在截止时间前提交任意次，得分以最后一次提交为准。

为了方便批改，请你将答案分为 4 个 PDF 文件，分别上传到 4 个题目的附件中。你可以选择手写或者使用 Markdown、LaTeX 等任意方式作答，但你应保证上传的附件均为 PDF 格式。

不符合上述要求的提交会被酌情扣分。

作答规则

1. 请独立思考，独立完成，在作业截止前不与其他同学交流本作业。
2. 你可以查阅笔记、书本、网络资料，但答案必须用你自己的语言亲手写下。

答题建议

1. 当遇到不懂的概念、记号时，建议首先查阅课本 Practical Foundations for Programming Languages (PFPL)。
2. 当题目不要求你写出过程，但你对自己的答案不够有信心时，可以把过程写出。
3. 如对题目有疑问，可在钉钉上联系任何一位助教询问。

祝大家好运！

Problem 1 : Review (40 Points)

这一部分旨在帮助大家回顾课程中的**部分**内容（尤其是较为形式化的部分），并考查大家对相关知识的理解和掌握程度。大家如果对内容已经比较熟悉，直接找 Question 做就可以了。

注意： 请用尽可能简洁的语言回答问题。没有要求说明理由的题目，不需要说明。

（λ 演算相关内容，请大家阅读之前的资料自行复习。）

判断与规则

我们学习了抽象语法和具体语法的区别。

我们引入了抽象语法树 (AST)。抽象语法树是一棵有序树，其叶子结点为变量或者没有参数的运算符，其内部节点是运算符。

Question 1 (1 point)**.** 请在课程所学语言 (*E, EF, ED, T* (包括和类型和积类型扩展), *M, PCF*) 的范围内，举一个「没有参数的运算符」的例子。

提示： 请确定你的答案在课本中对应语言的「语法表」中是有出现的。请注意用某种语言定义出的其他类型并不属于该语言的一种，例如用「和类型扩展的 *System T*」定义的 *option* 类型并不属于和类型扩展的 *System T* 本身的一部分。在后面的题目中也需要注意这一点。

Answer: E 中的 num, str；T 和 PCF 中的 nat, z；积类型中的 unit, triv；和类型中的 void。

AST 按语法的不同形式分为不同的类别 (sort)。多个 AST 通过 operator 进行组合。我们通过运算符的元数 (arity) 规定运算符的类别及其参数的数目和类别。变量是某个领域内的未知的对象，用特定对象代换 (substitute) 某个表达中的全部同个变量，则变量成为已知。即，变量是一个未知的对象或者占位符，其含义由代换赋予。

例如，常见的编程语言区分 expression 和 statement，这就是两个不同的 sort，分别记为 s_1 和 s_2。

假设 s_1 中存在运算符 `plus`, `times` 和 `num`，它们的结构分别类似于 `plus(expr1; expr2)`, `times(expr1; expr2)` 和 `num[n]` (其中 $n \in \mathbb{N}$)；s_2 中存在运算符 `if` ，它的结构类似于 `if (expr) then stmt1 else stmt2`。

Question 2 (1 point)**.** `num` 的元数记作 $(\mathbb{N})s_1$。请写出其他 3 个运算符的元数。

Answer: plus 和 times 的 arity 均记作 $(s_1, s_1)s_1$，if 的 arity 记作 $(s_1, s_2, s_2)s_2$

$2 + (3 \times x)$ 可以表达为 `plus(num[2]; times(num[3]; x))`。这里 x 是一个变量。

由于 `num[4]` 也是 s_1 类型的 AST，因此可以用它代换上面 AST 中的 x，得到 `plus(num[2]; times(num[3]; num[4]))`。

Question 3 (1 point)**.** 可以用 `plus(num[1]; num[2])` 代换上面 *AST* 中的 x 吗？为什么？

Answer: 可以，因为 `plus(num[1]; num[2])` 也是 s_1 类型的 AST

我们给出了 x 对 X 是新的 (x is fresh to X) 的定义；从而定义了 X, x。

结构归纳法。我们形式化地给出了 AST 的定义。由于 AST 的树形结构，如果我们希望证明所有 AST a 都具有性质 $P(a)$，那么我们只需要对于所有生成 a 的方式，都证明：「如果其子 AST 都具有该性质，那么生成的 a 也具有该性质」即可。

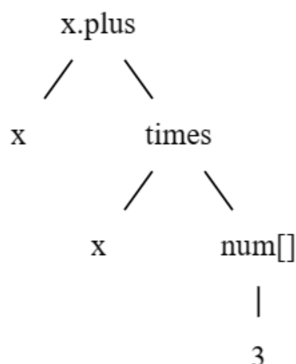
我们进一步形式化地给出了变量代换的定义。

Question 4 (1 point). 请证明：如果 $a \in A[X,x]_s$，那么对于任意 $b \in A[X]$ 都存在唯一的 $c \in A[X]$ 满足 $[b/x]a = c$。

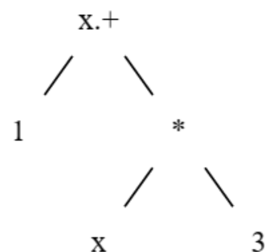
Answer: 对 a 使用结构归纳法。如果 $a = x$，那么根据定义有 $c = b$；。如果 $a = y \neq x$，那么根据定义有 $c = y$；。如果 $a = o(a_1;\ldots;a_n)$，根据归纳假设，有唯一的 $c_1,\ldots,c_n$ 满足 $c_1 = [b/x]a_1,\ldots,c_n = [b/x]a_n$，因此 $c = o(c_1;\ldots;c_n)$。

可以看到，我们在 AST 中引入了 variables 的概念，但是没有说明其含义。事实上，variable 的含义是通过 binding 给定的。我们介绍了绑定的定义，从而将 AST 泛化为了抽象绑定树 (ABT)。我们给出了抽象子、约束和作用域的概念。

举例说明，x.plus(x;times(x;num[3])) 这棵 ABT 里，x 是绑定的。因此根节点中我们用 x.plus 来表示这个绑定的作用域和符号：



在作业中，我们允许大家在不引起误解的前提下自由选择抽象语法或者具体语法，也可以混用。如果用具体语法表示，这棵树可以画成：



绑定使得 operator 的 arity 得到了推广或者说泛化。我们给出了泛化元数和价的概念。

我们通过 fresh renaming 给出了 ABT 的形式化的定义；定义了这种意义下的结构归纳法；定义了 α-等价；定义了 ABT 上的代换。

判断是关于某种类别的一棵或者多棵 ABT 的陈述，表明一棵或多棵 ABT 有某种性质或者彼此之间有某种联系。这些「性质或联系」本身称为判断形式。我们给出了实例、谓词、主词的概念。

我们指出，规则规定一个 judgment 有效的充要条件，因而完全决定了这个 judgment 的含义。我们根据一个规则是否有前提来区分它是公理还是一般规则。

一个 judgment form 的归纳定义由一组 rules 组成。我们强调，「A collection of rules is considered to define the strongest judgment form that is closed under these rules」。这是「推导」和「规则归纳」的理论基础。

Question 5 (1 point). *判断正误：在做规则归纳时，归纳假设是有可能用不到的。*

Answer: 正确。例如补充视频中对引理「如果 succ(a) nat 成立，那么 a nat 成立」的证明。

我们区分了迭代和联立归纳定义，并讨论了用规则定义函数的方法。

在基本判断之上，我们引入了假言判断 (hypothetical judgment)，用来表示一个或多个假设和一个结论之间的蕴含关系。我们介绍了可导性 (derivability) 和可纳性 (admissibility) 两种蕴含概念，并讨论了可导性判断和可纳性判断的一些性质。

我们讨论了一般性判断 (general judgments)。简单来说，相对于 $\Gamma \vdash_R J$ 而言，我们用 $\Gamma \vdash_R^X J$ 表示 Γ 中用到了一些 variable，这些 variable 的集合为 X。

进一步地，我们把 X 分为两个部分 XY，其中 Y 是 Γ 中用到的所有自由变量，也就是说它们可以被任意命名和替换；X 是剩余的变量，$X \cap Y = \emptyset$，即 $\Gamma \vdash_R^{XY} J$。我们记 $Y \mid \Gamma \vdash_R^X J$ 与之等价。

静态语义和动态语义

我们区分了编程语言的静态和动态阶段。

Question 6 (1 point). *代码的编译包括词法分析（将代码分割成若干单词）、语法分析（按照语法，将单词组合成树）、语义分析（表达式中进行类型推断和类型检查等）等步骤。请问其中哪些步骤属于静态阶段？*

Answer: 都属于静态阶段。

课本给出了语言 E 的语法表，给出了 E 的抽象语法和具体语法。

Question 7 (1 point). *语法表共 10 行。这 10 行中，哪些是 operator？*

注意： *回答时，请清晰地区分 num 和 num$[n]$，以及 str 和 str$[s]$。你可以使用行号来回答。*

Answer: 除了 variable 都是。

我们在第 1 节提到，语法 (syntax) 规定了如何将各种 phrases(expr, commands / statements, decl, etc.) 组合成程序。同时，在静态阶段，为了保证一个程序是 well-formed 的，我们需要检查它满足语法和静态语义。这里提到的静态语义 (statics) 由一系列规则组成，这些规则是用来推导定型判断 (typing judgments) 的。所谓 typing judgments，就是用来陈述某个表达式符合某个类型的判断。

容易理解的是，表达式 `plus(x; num[n])` 是否合法，取决于 `x` 的类型是否是 `num`。也就是说，phrases 对于其所处的上下文是敏感的。因此，E 的 statics 由形如 $\vec{x} \mid \Gamma \vdash e : \tau$ 的泛型假言判断组成。其中，$\vec{x} = \mathrm{dom}(\Gamma)$ 是变量的有限集合；Γ 是定型上下文，它对每个 $x \in \vec{x}$ 有一个形如 $x : \tau$ 的假设。

如果 $x \notin \mathrm{dom}(\Gamma)$，那么我们称 x 对于 Γ 是新的。这时，我们可以把 $x : \tau$ 添加到 Γ 中得到 $\Gamma, x : \tau$。

课本通过规则 (4.1) 定义了 E 的静态语义。

Question 8 (1 point)**.** *请简要解释规则 (4.1h) 的含义。*

Answer: 如果定型上下文中有类型为 τ_1 的表达式 e_1，同时引入一个新的、类型为 τ_1 的变量 x 可以得到一个类型为 τ_2 的表达式 e_2，那么 $\mathrm{let}(e_1; x.e_2)$ 在上下文中的类型是 τ_2。

我们讨论了引入模式和消去模式；讨论了 E 语言的几个性质。

动态语义描述程序如何执行。为了定义语言的动态语义，我们定义了转换系统 (transition system)，归纳地指明程序执行的一步步过程。

在转换系统中，我们定义了状态、终结状态、初始状态和转换，以及一些其他定义。

课本通过转换系统给出了 E 语言的结构化动态语义，定义了闭值 (5.3) 和状态转换 (5.4)。我们区分了指令转换和搜索转换；区分了按值解释和按名解释。

Question 9 (1 point)**.** *请简要解释规则 (5.4h) 的含义。请注意规则中含有的方括号，这也是需要解释的部分。*

Answer: 在 let 的按值解释中，当 e_1 为闭值时，$\mathrm{let}(e_1; x.e_2)$ 能够转换到 $[e_1/x]e_2$；在按名解释中，无论 e_1 是否为闭值，$\mathrm{let}(e_1; x.e_2)$ 都能够转换到 $[e_1/x]e_2$。

我们给出了类型安全的定义，即满足保持性和进展性的语言是类型安全的。

Question 10 (2 points)**.** *为了证明 E 的保持性，我们引入 $P(e, e')$ 表示如果 $e : \tau$ 且 $e \mapsto e'$，则 $e' : \tau$；接下来要对 $e \mapsto e'$ 的定义做规则归纳。我们只尝试规则 (5.4g) 的情况。请证明：如果 $P(e_1, e_1')$ 且 $e_1 \mapsto e_1'$，那么 $P(let(e_1; x.e_2), let(e_1'; x.e_2))$。*

Answer: 由于 $\mathrm{let}(e_1; x.e_2) : \tau$
因此根据类型反转 (引理 4.2)，存在 $\tau_1, e_1 : \tau_1$ 使得 $x : \tau_1 \vdash e_2 : \tau_2$
根据归纳假设，有 $e_1' : \tau_1$ 使得 $x : \tau_1 \vdash e_2 : \tau_2$
由 (4.1h)，有 $\mathrm{let}(e_1'; x.e_2) : \tau$，即证。

全函数与 Gödel's System T

我们说，我们可以通过把一个名字绑定到一棵带约束变量的 ABT 上来定义一个函数。这里的约束变量就是函数参数。我们可以用特定的表达式替换约束变量来完成函数的应用。

语言 ED 在语言 E 的基础上扩充了函数定义和函数应用。我们介绍了函数头、定义域、值域等概念。我们给出了 ED 的静态语义。

为了给出 ED 的动态语义，规则 (8.2) 定义了函数代换。

Question 11 (1 point)**.** *阅读规则 (8.2)。请解释：为什么右边 let 的第一个参数是 $[\![x_1.e_2/f]\!]e_1$，而不是 e_1？*

Answer: 因为 f 可能在 e_1 中还有出现。

我们观察到在 ED 中，变量定义和函数定义非常相似；我们试图将它们统一起来。因此我们引入了语言 EF。EF 在语言 E 的基础上扩充了函数类型。我们给出了 EF 的语法。

这样，我们就可以把函数当做任何其他表达式一样使用了，包括作为参数和返回值。因此，我们说在这样的语言中，函数是一等 (first-class) 的；这样的函数被称为是高阶的，而非一阶的。

我们给出了 EF 的静态和动态语义。

我们讨论了动态作用域和静态作用域的区别。

Question 12 (5 points). 看下面一段伪代码，完成后面的问题：

```
x <- 1
f <- function(a) x + a
g <- function(b) {
    x <- b
    f(0)
}
g(2)
```

(1) 请你根据语言 *EF*，按树形结构画出等价于这段伪代码的 *ABT*。**提示：** 如果你不知道语法中函数抽象 $\lambda\{\tau\}(x.e)$ 中的 {} 如何处理，你可以在课本 *1.2* 节末尾找到答案。

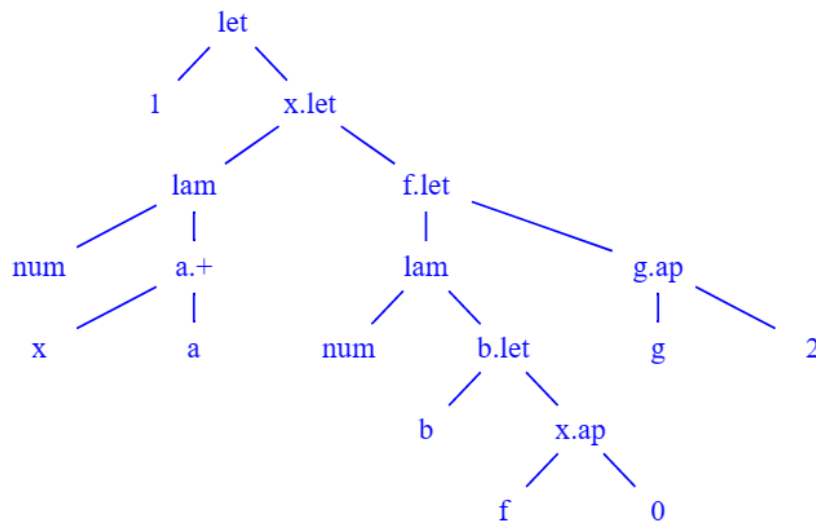
(2) 请你根据语言 *ED* 按值调用解释下的结构化动态语义，写出这段代码的运行过程（你可以按需选用具体语法或抽象语法）。在每一步，你应该注明你使用的规则；可以用规则的名称或者课本中的规则编号注明。**提示：** 两步之间的关系不一定只有 $\mapsto$，还有可能有 $=$。

(3) *ED* 语言是静态作用域的还是动态作用域的？

Answer: 我们希望通过这道题，让大家对「我们学的这些东西和实际的编程语言到底怎么关联起来的」这个问题有一个初步的理解。

对于这个题目来说，核心需要理解的内容是「赋值」如何实现。通过排除法我们可以容易地找到，let 运算符可以实现这个效果。它的具体语法 let x be e_1 in e_2 可以 (不那么形式化地) 理解为「在 e_2 中的 x 是 e_1」，那么如果 e_2 是「下文」对应的 ABT，那么 let x be e_1 in e_2 就是「在下文中，x 是 e_1」。这其实和我们代码中赋值的含义是一致的。由此，我们给出下面的答案：

(1)



(2) 上述代码用 ED 可以表示为

$$\begin{aligned}&\text{let } x \text{ be } 1 \text{ in}\\&\quad \text{fun } f(a:\text{num}):\text{num} = x + a \text{ in}\\&\quad\quad \text{fun } g(b:\text{num}):\text{num} = (\text{let } x \text{ be } b \text{ in } f(0)) \text{ in } g(2)\end{aligned}$$

即 $\text{let}(1;x.\text{fun}\{\text{num};\text{num}\}(a.x+a;f.\text{fun}\{\text{num};\text{num}\}(b.\text{let}(b;x.f(0));g.g(2))))$
因此运行过程为:

$$\begin{aligned}
&\text{let}(1;x.\text{fun}\{\text{num};\text{num}\}(a.x+a;f.\text{fun}\{\text{num};\text{num}\}(b.\text{let}(b;x.f(0));g.g(2)))) \\
\mapsto\ &[1/x](\text{fun}\{\text{num};\text{num}\}(a.x+a;f.\text{fun}\{\text{num};\text{num}\}(b.\text{let}(b;x.f(0));g.g(2)))) && (5.4h)\\
=\ &\text{fun}\{\text{num};\text{num}\}(a.1+a;f.\text{fun}\{\text{num};\text{num}\}(b.\text{let}(b;x.f(0));g.g(2))) && (\text{代换})\\
\mapsto\ &[\![a.1+a/f]\!](\text{fun}\{\text{num};\text{num}\}(b.\text{let}(b;x.f(0));g.g(2))) && (8.3)\\
=\ &\text{fun}\{\text{num};\text{num}\}(b.\text{let}(b;x.[\![a.1+a/f]\!]f(0));g.g(2)) && (\text{函数代换定义})\\
=\ &\text{fun}\{\text{num};\text{num}\}(b.\text{let}(b;x.\text{let}([\![a.1+a/f]\!]0;a.1+a));g.g(2)) && (8.2)\\
=\ &\text{fun}\{\text{num};\text{num}\}(b.\text{let}(b;x.\text{let}(0;a.1+a));g.g(2)) && (\text{函数代换定义})\\
\mapsto\ &[\![b.\text{let}(b;x.\text{let}(0;a.1+a))/g]\!]g(2) && (8.3)\\
=\ &\text{let}([\![b.\text{let}(b;x.\text{let}(0;a.1+a))/g]\!]2;b.\text{let}(b;x.\text{let}(0;a.1+a))) && (8.2)\\
=\ &\text{let}(2;b.\text{let}(b;x.\text{let}(0;a.1+a))) && (\text{函数代换定义})\\
\mapsto\ &[2/b](\text{let}(b;x.\text{let}(0;a.1+a))) && (5.4h)\\
=\ &\text{let}(2;x.\text{let}(0;a.1+a)) && (\text{代换})\\
\mapsto\ &[2/x]\text{let}(0;a.1+a) && (5.4h)\\
=\ &\text{let}(0;a.1+a) && (\text{代换})\\
\mapsto\ &[0/a](1+a) && (5.4h)\\
=\ &1+0 && (\text{代换})\\
\mapsto\ &1 && (5.4a)
\end{aligned}$$

(3) 静态作用域。

在介绍 Gödel's System T 之前，我们补充了一些预备知识。我们讨论了部分函数和全函数，指出了部分函数在计算机科学领域的重要性。我们简单介绍了递归函数论。

我们引入了原始递归运算。设 f 是一个 n 元全函数，g 是 $n+2$ 元全函数，令

$$\begin{aligned}h(x_1,\dots,x_n,0) &= f(x_1,\dots,x_n)\\ h(x_1,\dots,x_n,t+1) &= g(t,h(x_1,\dots,x_n,t),x_1,\dots,x_n)\end{aligned}$$

则称 h 是由 f 和 g 经过原始递归运算得到的。

System T 是函数类型和自然数类型的结合，同时引入了原始递归机制。我们给出了它的语法、静态语义和动态语义。

Question 13 (1 point). *规则 (9.2b) 的前提包含在后继的急切解释中，但不包含在惰性解释中。「急切解释」和「惰性解释」与我们之前讨论的「按值调用」和「按名调用」有何异同？*

Answer: 急切解释和惰性解释是描述后继操作符 s 的，因此不能叫做「调用」。除此之外，其含义对应类似。

可以看到，T 对于函数的处理和 EF 是一致的；对于自然数的定义和 2.2 小节定义的基本一致，只是采用的符号略有不同。我们把 s(...s(z)) 简写为 $\overline{n}$，表示后继被作用到 z 上 $n \geq 0$ 次。

我们下面来讨论 T 语言中的递归操作。递归式 (recursor) 的抽象语法是 $\text{rec}\{e_0; x.y.e_1\}(e)$；具体语法是 $\text{rec}\, e\{\text{z} \hookrightarrow e_0 | \text{s}(x) \text{ with } y \hookrightarrow e_1\}$，或者写作 $\text{rec}\{\text{z} \hookrightarrow e_0 | \text{s}(x) \text{ with } y \hookrightarrow e_1\}(e)$。

Question 14 (1 point). *请简要解释递归式 $rec\{e_0; x.y.e_1\}(e)$ 的含义。*

Answer: 如果 e 满足 z 的形式，则表达式的值为 e_0；否则 e 可以表示为 $\text{s}(e')$ 的形式，此时表达式的值为 e_1，e_1 有绑定变量 x 和 y，e' 被绑定到 x 上，以 e' 为操作数递归，将递归的结果绑定到 y 上。

为了理解递归式的含义，不妨举一个例子。我们在 OCaml 中定义如下的 nat 类型：

```
type nat = Z | S of nat;;
```

我们定义「加倍」函数：

```
let rec double a =
  match a with
  | Z -> Z
  | S x -> S(S(double x));;
```

进一步地，我们将其改写为递归式的形式：

```
let rec double a =
  match a with
  | Z -> Z
  | S x -> let y = double x in S(S y);;
```

因此，我们可以容易地写出 double 的定义：

$$\text{double} \triangleq \lambda(e : \text{nat})\text{rec}\, e\{\text{z} \hookrightarrow \text{z} | \text{s}(x) \text{ with } y \hookrightarrow \text{s}(\text{s}(y))\}$$

或者表示为

$$\text{double} \triangleq \lambda\{\text{nat}\}(e.\text{rec}\{\text{z}; x.y.\text{s}(\text{s}(y))\}(e))$$

Question 15 (4 points). *模仿上述例子，用 T 语言定义两个 nat 的「加法」函数。你可以参考课本 9.3 节阿克曼函数的定义，学习如何处理多个参数的函数。你只需要写出你使用的 OCaml 代码和对应的 T 语言定义。*

Answer: OCaml 代码：

```
let rec plus a b =
    match a with
    | Z -> b
    | S x -> let y = plus x b in S y;;
```

T 语言定义：plus $\triangleq \lambda(a:\text{nat})\lambda(b:\text{nat})\text{rec}\ a\{\text{z}\hookrightarrow b|\text{s}(x)\ \text{with}\ y\hookrightarrow \text{s}(y)\}$

作为示例，我们展示 $1+3$ 的计算过程。为了简便，我们记 z 为 0，s(z) 为 1，s(s(s(z))) 为 3，s(3) 为 4：

$$
\begin{aligned}
& \text{ap}(\text{ap}(\text{plus};1);3) & \\
\mapsto\ & \text{ap}([1/a](\lambda(b:\text{nat})\text{rec}\ a\{\text{z}\hookrightarrow b|\text{s}(x)\ \text{with}\ y\hookrightarrow \text{s}(y)\});3) & (9.3\text{d})\\
=\ & \text{ap}(\lambda(b:\text{nat})\text{rec}\ 1\{\text{z}\hookrightarrow b|\text{s}(x)\ \text{with}\ y\hookrightarrow \text{s}(y)\};3) & (\text{代换})\\
\mapsto\ & [3/b]\text{rec}\ 1\{\text{z}\hookrightarrow b|\text{s}(x)\ \text{with}\ y\hookrightarrow \text{s}(y)\} & (9.3\text{d})\\
=\ & \text{rec}\ 1\{\text{z}\hookrightarrow 3|\text{s}(x)\ \text{with}\ y\hookrightarrow \text{s}(y)\} & (\text{代换})\\
\mapsto\ & [0,\text{rec}\ 0\{\text{z}\hookrightarrow 3|\text{s}(x)\ \text{with}\ y\hookrightarrow \text{s}(y)\}/x,y]\text{s}(y) & (9.3\text{g})\\
=\ & \text{s}(\text{rec}\ 0\{\text{z}\hookrightarrow 3|\text{s}(x)\ \text{with}\ y\hookrightarrow \text{s}(y)\}) & (\text{代换})\\
\mapsto\ & \text{s}(3) & (9.3\text{f})\\
=\ & 4 &
\end{aligned}
$$

有限数据类型

我们给出了「类型即集合」的想法，据此解释了积类型的取值范围即为各类型的笛卡尔积，而和类型的取值范围即为各类型的并。

我们给出了二元积的定义。两种类型 τ_1 和 τ_2 的二元积 (binary product) $\tau_1\times\tau_2$ 是有序对 (ordered pair)（或称为元组 (tuple)）$\langle e_1,e_2\rangle$（或记为 (e_1,e_2)）的类型，其中 $e_1:\tau_1, e_2:\tau_2$。

更一般地，有限积 (finite product) $\langle\tau_i\rangle_{i\in I}$ 的每个元素都是一个 tuple，每个 tuple 的第 i 个元素的类型为 τ_i。I 是索引的有限集；上述 tuple 称为 I-indexed tuple。除了常见的 $I=\{0,\cdots,n-1\}$ 索引的 n-tuples 以外，也有用有限的符号集索引的 labeled-tuples，也称为 records。

空积 (nullary product) (a.k.a unit) 类型仅由唯一的不含任何值的空 tuple 构成。

我们给出了二元积和有限积的静态和动态语义。

Question 16 (1 point). 请分别指出二元积语法中各类型（即空积和二元积）的引入模式和消去模式。请使用课本中规则的序号来回答。

Answer: (10.1b) 是二元积的引入形式，(10.1c) 和 (10.1d) 是其消去形式；(10.1a) 是空积的引入形式，没有消去形式。

我们指出，有了 tuple，我们就可以用迭代式代替递归式了。

Question 17 (2 points)**.** 如何完成代替？请补全下面式子右边未完成的部分，并简要解释其含义：

$$rec\ e\{z \hookrightarrow e_0 | s(x)\ with\ y \hookrightarrow e_1\} \triangleq iter\ \ [\textbf{your answer here}]$$

Answer:

$\mathrm{rec}\ e\{\mathrm{z} \hookrightarrow e_0 | \mathrm{s}(x)\ \mathrm{with}\ y \hookrightarrow e_1\} \triangleq \mathrm{iter}\ e\{\mathrm{z} \hookrightarrow \langle \mathrm{z}; e_0\rangle | \mathrm{s}(x_1) \hookrightarrow \langle \mathrm{s}(x_1 \cdot \mathrm{l}), [x_1 \cdot \mathrm{l}, x_1 \cdot \mathrm{r}/x, y]e_1\rangle\} \cdot \mathrm{r}$

简单来说，我们将 rec 中的两个绑定变量 x 和 y 用一个新的积类型代替了，这个积类型变量的第一个元素是前驱，而第二个元素是递归调用的结果。

我们引入了原始互递归，即通过原始递归同时定义两个函数。

Question 18 (2 points)**.** 考虑这样的函数：

$$f(0) = 1$$
$$g(0) = 0$$
$$h(0) = 0$$
$$f(s(x)) = h(x)$$
$$g(s(x)) = g(x)$$
$$h(s(x)) = f(x)$$

这三个函数表示什么含义？请给出这三个函数的定义。

注意：给出的定义应当和课本 *10.3* 节类似，基于 *System T* 与和类型给出形式化的定义。

Answer: (这个题写错了，本来是想写 $f(\mathrm{s}(x)) = h(x)$，$g(\mathrm{s}(x)) = f(x), h(\mathrm{s}(x)) = g(x)$ 的)

我们给出了和类型的语法和语义。和类型 $\tau_1 + \tau_2$ 的一个值包含 τ_1 类型的一个值或者 τ_2 类型的一个值，同时包含了一种能够确定该值属于哪个类型的机制。这种机制是为了满足语法中 case 运算符的需求。

我们指出了积类型和和类型在编程语言中的重要意义。

我们区分了 void 和 unit 的区别，分析了布尔类型和 option 如何用和类型实现。

Question 19 (1 point)**.** *C* 语言的 *union* 是否符合我们对和类型的定义？为什么？

Answer: 不符合。和类型保存有「选取了哪个分支」的机制，但是 union 并没有。同时，union 类型不安全，即可以从其他元素的视角解释其中的值。

Question 20 (1 point)**.** *T* 语言以及积类型、和类型的语义中都出现了符号 $\hookrightarrow$。你如何理解其含义？

Answer: Cheat Sheet for PFPL 中给出的名字是「选择」。可以理解为在某种情况下选取某个分支。(言之成理即可。)

Question 21 (2 points)**.** 有的函数能够传入多个参数，或返回多个返回值。请用所学知识简述其实现。

Answer: 传入可以用柯里化，传入和返回都可以用积类型。

无限数据类型

我们讨论了泛型编程。我们引入了类型算子 $t.\tau$，其中 t 是一个类型变量，τ 是一个类型，且 t type $\vdash \tau$ type。C++ 的模板类 `template<typename T> class Foo;` 就可以类似地表示成类型算子 T.Foo。

我们指出，泛型编程的能力取决于它所允许使用的类型算子。我们介绍了多项式类型算子，它只由类型变量、unit、void、积类型、和类型构成。

我们给出了多项式类型算子的泛型扩展操作 map，并给出了其静态语义和动态语义。泛型扩展表达式形如 $\mathrm{map}\{t.\tau\}(x.e')(e)$，这样的操作让我们能够对 $e:\tau$ 中的每一个组分（即那些被积类型和和类型构造起来的 void 和 unit）经过 e' 转换为新的类型。

Question 22 (3 points)**.** 我们给出这样的泛型扩展表达式：

$$map\{t.t\times t\}(x.case\ x\{l\cdot_\hookrightarrow r\cdot\langle\rangle\mid r\cdot_\hookrightarrow l\cdot\langle\rangle\})(e)$$

这个表达式完成什么样的转换？

作为例子，请计算当 e 是 $\langle r\cdot\langle\rangle, l\cdot\langle\rangle\rangle$ 时表达式的值。在每一步，你应该注明你使用的规则；可以用规则的名称或者课本中的规则编号注明。

Answer: 这个表达式将 $r\cdot_$ 转为 $l\cdot\langle\rangle$，将 $l\cdot_$ 转为 $r\cdot\langle\rangle$。结合布尔类型的相关讨论，可以完成逻辑取反运算。

记 $e'\triangleq \mathrm{case}\ x\{l\cdot_\hookrightarrow r\cdot\langle\rangle\mid r\cdot_\hookrightarrow l\cdot\langle\rangle\}$，计算过程如下：

$$\begin{aligned}
&\mathrm{map}\{t.t\times t\}(x.e')\langle r\cdot\langle\rangle, l\cdot\langle\rangle\rangle \\
\mapsto\ &\langle \mathrm{map}\{t.t\}(x.e')(r\cdot\langle\rangle), \mathrm{map}\{t.t\}(x.e')(l\cdot\langle\rangle)\rangle && (14.3c, 10.2g, 10.2h)\\
\mapsto\ &\langle [r\cdot\langle\rangle/x]e', [l\cdot\langle\rangle/x]e'\rangle && (14.3a)\\
=\ &\langle \mathrm{case}\ r\cdot\langle\rangle\{l\cdot_\hookrightarrow r\cdot\langle\rangle\mid r\cdot_\hookrightarrow l\cdot\langle\rangle\}, \mathrm{case}\ r\cdot\langle\rangle\{l\cdot_\hookrightarrow l\cdot\langle\rangle\mid r\cdot_\hookrightarrow l\cdot\langle\rangle\}\rangle && (\text{代换})\\
\mapsto\ &\langle [r\cdot\langle\rangle/x_2]l\cdot\langle\rangle, [l\cdot\langle\rangle/x_1]r\cdot\langle\rangle\rangle && (11.2g, 11.2h)\\
=\ &\langle l\cdot\langle\rangle, r\cdot\langle\rangle\rangle && (\text{代换})
\end{aligned}$$

我们简单介绍了正类型算子的定义。

我们看到了 map 能够完成的转换。但是，我们为什么不直接编写完成对应转换的代码，而是要引入 map 呢？其重要意义在于，当我们在语言中加入某个新的类型，map 能够自动完成相关的转换，而不需要更改语言的动态语义。

泛型编程的这种能力使我们能够创建递归类型。我们通过列表的例子讨论了归纳类型和余归纳类型。

Question 23 (2 points)**.** 课本 15.1 节开头给出了用归纳类型定义 nat 的方式，即 $nat\triangleq\mu(t.unit+t)$，用 $fold_{nat}(l\cdot\langle\rangle)$ 表示 z，用 $fold_{nat}(r\cdot e)$ 表示 $s(e)$。

我们定义自然数的列表类型 $list\triangleq\mu(t.unit+(nat\times t))$，给定 $list$ 的部分定义：

$$\frac{}{\Gamma\vdash nil:list}\qquad \frac{\Gamma\vdash e_1:nat\qquad \Gamma\vdash e_2:list}{\Gamma\vdash cons(e_1;e_2):list}$$

请在理解 *nat* 的基础上，尝试理解 *list* 的定义。你的任务是：用归纳类型表示 *nil* 和 *cons*。

Answer: $\mathtt{nil} \triangleq \mathtt{fold}_{\mathtt{list}}(\mathtt{l} \cdot \langle\rangle)$, $\mathtt{cons}(e_1;e_2) \triangleq \mathtt{fold}_{\mathtt{list}}(\mathtt{r} \cdot \langle e_1, e_2\rangle)$

PCF

我们之前讨论过全函数和部分函数的区别。在这之前讨论到的 E, ED, EF, T 等语言只能用来表示那些能够终止并产生一个值的计算，也就是说，它们是全 (total) 计算的。在此基础上，我们引入了一种部分 (partial) 的编程语言 PCF，这也是这门课程中遇到的第一门图灵完备 (Turing-complete) 的语言。我们引入了一般递归。

在作业中，我们也进行了关于 PCF 的练习。这里再用一道题目帮助大家回顾。

Question 24 (3 points)**.** 我们学过杨辉三角，可以由此计算二项式系数：

$$P(n,0) = 1$$
$$P(0,k+1) = 0$$
$$P(n+1,k+1) = P(n,k) + P(n,k+1)$$

请你用 *PCF* 语言定义 $coef : nat \to nat \to nat$，使得对于任意 $n,k \in \mathbb{N}$，有 $coef\ \overline{n}\ \overline{k} \mapsto^* \overline{P(n,k)}$。在定义中，你可以直接使用 $plus : nat \to nat \to nat$ 计算两个 *nat* 之和。

注意：你可以选择使用课件或者课本上的 *PCF* 语法定义函数，但你要保证符合其中一种语法，且定义出来的是函数，而不是函数调用表达式。

提示：如果你不记得 $\overline{x}(x \in \mathbb{N})$ 表示什么，请你回顾 *System T* 相关章节。

Answer: $\text{coef} \triangleq \lambda n : \text{nat}.\lambda k : \text{nat}.\text{ifz}\ k\{\text{s z}; k'.\text{ifz}\ n\{\text{z}; n'.\text{plus}\ (\text{coef}\ n'\ k')(\text{coef}\ n'\ k)\}\}$

我们另外还讨论了多态、变量类型、并行、协程等话题。这些话题中，我们并没有给出很多形式化的表述，因此请大家自行回顾复习。