Part A (SML):

• Syntax vs. semantics vs. idioms vs. libraries vs. tools

编程语言的组成部分（主要是语法、语义、语用）:
1、语法(Syntax): 如何编写语言的各个部分
2、语义(Semantics): 程序的含义
- 类型检查 Type-checking (before program run)
- 求值 Evaluation (as program run)

3、语用(idioms): 使用语言功能表达计算的常用方法,涉及使用者
4、库(libraries): 提供库支持。
5、工具(tools): 可用于操作语言程序的程序（编译器，解释器等）

• ML basics (bindings, conditionals, records, functions)

1、在 SML 中，所有值都是表达式，但并不是所有值都有值。
2、SML 中没有赋值语句(assignment statement)，它只有变量绑定(variable binding)以及函数绑定(function binding)。
3、对于变量绑定:
1）类型检查表达式和扩展静态环境，即使用先前绑定生成的静态环境按顺序对每个绑定进行类型检查
2）求值表达式并扩展动态环境，使用先前绑定生成的动态环境按顺序求值每个绑定。
4、每个 val 绑定和函数绑定都使用模式匹配。
5、ML 中的每个函数只需要一个参数。

• Recursive functions and recursive types

• Benefits of no mutation

没有变异（赋值语句）是一种有用的编程语言特征，无需跟踪共享/别名。

• Algebraic datatypes, pattern matching

1、3 种构建数据模块的方法
1）“Each of”: A t value contains values of each of t1 t2 ··· tn
2）“One of”: A t value contains values of one of t1 t2 ··· tn
3） “Self reference”: A t value can refer to other t values
2、模式匹配(pattern matching)

• Tail recursion

1、尾递归的好处: ML 在编译器中识别这些尾部调用并以不同方式处理它们,
- 在调用之前弹出调用者，允许被调用者重用相同的堆栈空间 。
- （与其他优化一起），与循环一样高效。

2、可以通过构建辅助函数，在尾部位置（tail position），调用函数，从而实现尾递归。

• Higher-order functions; closures

1、第一类函数: 可以在我们使用值的任何地方使用它们。意味着函数本身也是可计算的值。

可以将多个函数作为参数传递，可以将函数放在数据结构中（元组，列表等），可以将函数作为结果返回，可以编写遍历自己的数据结构的高阶函数。
2、函数闭包：即第一类函数的求值，也是一个函数，函数可以使用函数定义外部的绑定（在定义函数的范围内）。
3、高阶函数(Higher-order functions),或称为算子，是一个操作于另一个或几个函数之上的函数。许多高阶函数都是多态（用于忽略无关的数据的类型）的，因为它们是可重用的，有些类型，“可以是任何东西”。

• Lexical scope

词法作用域与动态作用域

1）词法作用域(lexical scope)，即静态作用域(static scope)，在不修改绑定时，作用域在词法解析阶段既确定了，不会改变。对于函数而言，寻找绑定时，检查函数的定义环境。词法作用域的一般规则有最内嵌套作用域规则。
2）动态作用域(dynamic scope)里，变量的寻找过程发生在程序运行时期。对于函数而言，寻找绑定时，检查函数的执行环境。动态作用域里，函数执行遇到一个符号，会由内向外逐层检查函数的调用链，并打印第一次遇到的那个绑定的值。显然，最外层的绑定即是全局状态下的那个值。
3）两者的区别

简单的说，静态作用域中，变量的值是在变量定义时就已决定了，运行时值保持不变（没有重新赋值的情况下），而在动态作用域中，变量的值与定义无关，由运行时决定。

动态作用域在每次函数求值的时候都会在这唯一的一个环境里查询或更新。

而静态作用域是每次函数求值的时候都创建一个新的环境，包含了函数定义时候的所能访问到的各种绑定。这个新的环境连同那个函数一起，俗称闭包。

• Currying

一般情况下，第一类函数只能有一个参数，因此需要使用柯里化实现高阶函数。

• Syntactic sugar

• Equivalence and effects

等价性主要考虑代码维护，向后兼容性，优化，抽象。

Different definitions for different jobs

- PL Equivalence (341): given same inputs, same outputs and effects
  - Good: Lets us replace bad **`max`** with good **`max`**
  - Bad: Ignores performance in the extreme
- Asymptotic equivalence (332): Ignore constant factors
  - Good: Focus on the algorithm and efficiency for large inputs
  - Bad: Ignores “four times faster”
- Systems equivalence (333): Account for constant overheads, performance tune
  - Good: Faster means different and better
  - Bad: Beware overtuning on “wrong” (e.g., small) inputs; definition does not let you “swap in a different algorithm”

- **Parametric polymorphism and container types**
- **Type inference**

ML 语言是静态隐式类型的。

- **Abstract types and modules**

Part B(Racket):

• Racket basics

1、Racket 通过将所有内容括起来，将程序文本转换为表示程序的树（解析）。
- 原子是叶子
- 序列是具有子元素的节点

2、Racket 有 4 种方式定义局部变量（let 、let*、letrec、define）

3、ML is like a well-defined subset of Racket。One way to describe Racket is that it has "one big datatype"：all values have this type.

4、Racket 不是弱类型的语言。

• Dynamic vs. static typing

1、类型系统的一些基础概念

Program Errors

1）trapped errors：导致程序终止执行，如除 0，Java 中数组越界访问

2）untrapped errors：出错后继续执行，但可能出现任意行为。如 C 里的缓冲区溢出、Jump 到错误地址

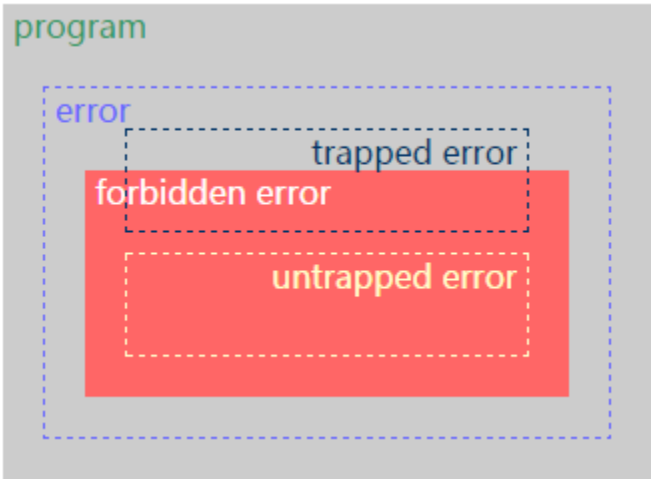
Forbidden Behaviours

语言设计时，可以定义一组 forbidden behaviors. 它必须包括所有 untrapped errors，但可能包含 trapped errors.

Well behaved、ill behaved

1）well behaved:程序执行不可能出现 forbidden behaviors,

2）ill behaved:程序执行可能出现 forbidden behaviors



2、类型系统分类

强、弱类型

1）强类型 strongly typed: 如果一种语言的所有程序都是 well behaved——即不可能出现 forbidden behaviors，则该语言为 strongly typed。即该语言的实现能贯彻一套规则，禁止把任何操作应用到并不准备支持这一操作的对像上。

2）弱类型 weakly typed: 程序执行时可能出现 forbidden behaviors。比如 C 语言的缓冲区溢出，属于 trapped errors，即属于 forbidden behaviors.故 C 是弱类型

动态、静态类型

1)静态类型(statically type): 如果在编译时拒绝 ill behaved 程序，则是 statically typed。即是强类型的，并且所有检查都能在编译时执行.

优点：结构非常规范，便于调试，方便类型安全

缺点：为此需要写更多类型相关代码，不便于阅读、不清晰明了

2)动态类型(dynamically type): 如果在运行时拒绝 ill behaviors，则是 dynamiclly typed。

优点：方便阅读，不需要写非常多的类型相关的代码;

缺点：不方便调试，命名不规范时会造成读不懂，不利于理解等

Correctness: sound 和 complete

1)区别

• sound:永远不会接受含有 forbidden behaviors 的程序

• complete:永远不会拒绝不含有 forbidden behaviors 的程序

2）一般 PL 类型系统是 sound，但不是 complete。

• Laziness, streams, and memorization

1. **调用方式**

传值调用 call-by-value：在调用函数时，表达式先被求值再作为参数。

传名调用 call-by-name：将变量作为结构代入函数体后计算所得到的表达式的值。

传需调用 call-by-need： 类似于传名调用，在需要的时候再计算，通过图归约(graph reduction)的方法计算。

在大多数编程语言中都是第一种方式, 在一些语言中我们可以用一种既巧妙又简单的方式实现第三种方式，即惰性求值(Lazy Evaluation)。

2、可以通过特殊的 lambda 表达式，即 thunking（匿名零参数函数）实现惰性求值，比如流(stream) 的实现。

• Implementing languages, especially higher-order functions

1、解释器与编译器与组合是关于特定的语言实现，而不是语言定义。

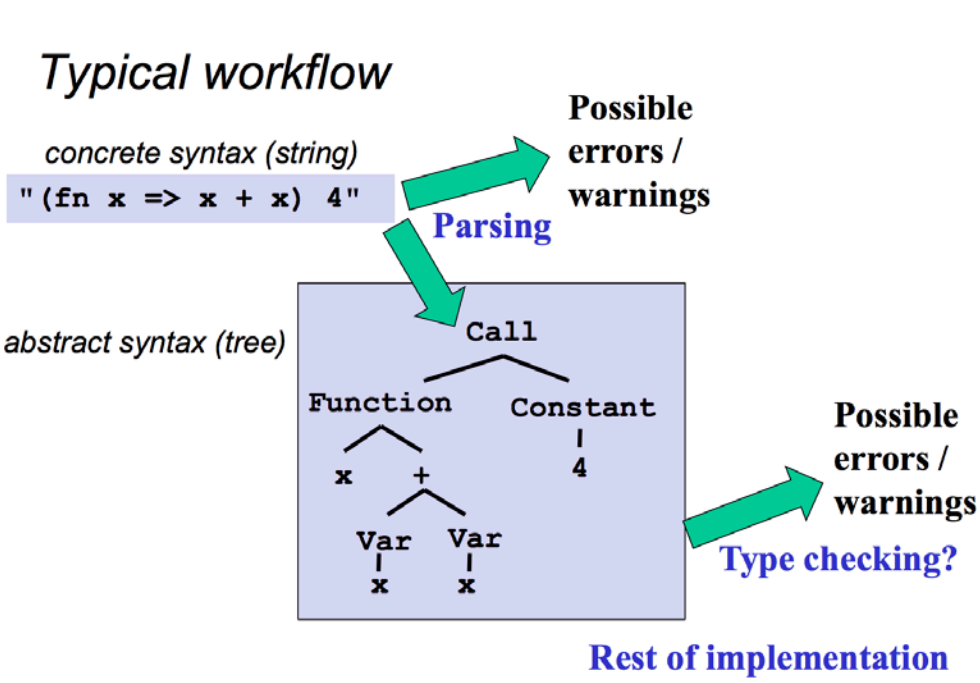
2、Parser 读取程序文本，检查 syntax，如果语法正确则输出 AST（abstract syntax tree）。如果该编程语言有 type checker, 则将 AST 丢给 type checker 检查, 通过 type check 后，就由 interpreter 或 compile 来运行程序并输出结果。

3、在实现编程语言 B 通常有下面两种办法：

1）使用另一种编程语言 A 来实现 interpreter（命名为 evaluator 或 executor 更恰当），输入 B 语言写的代码，输出结果

2）使用另一种编程语言 A 实现 compiler（命名为 translator 更恰当），将 B 翻译成第三种编程语言 C

4、如果基于编程语言 A 来实现编程语言 B，就可以跳过 parsing 阶段：Have B programmers write ASTs directly in PL A。



- Macros

宏定义描述了如何将一些新语法转换为源语言中的不同语法

- Eval

Part C(Ruby):

• Ruby basics

1、Ruby 是一种纯粹的面向对象语言，这意味着语言中的所有值都是对象。
2、Ruby 基于类：每个对象都有一个确定行为的类，属于脚本动态类型语言。
3、Ruby 中，块（Blocks）类似于函数式编程语言中的闭包(closures)，或者说功能上很类似匿名函数，可以传递给一个函数，在函数内部执行，或者结合数组自带的那些方法使用（类似于函数式编程语言中的高阶函数）。块即是在调用方法时，能与参数一起传递的多个处理的集合。

• Object-oriented programming is dynamic dispatch

• Pure object-orientation

• Implementing dynamic dispatch

动态调度
- 也称为后期绑定或虚方法（例如在类 C 中定义的方法 m1 中调用 self.m2（）可以解析为在 C 的子类中定义的方法 m2 ）
- OOP 的最独特特征

• Multiple inheritance, interfaces, and mixins

Multiple dispatch 中在运行时才决定具体去调用哪个同名函数，而 Method overloading 在编译时已经确定了类型。

• OOP vs. functional decomposition and extensibility

• Subtyping for records, functions, and objects

• Class-based subtyping

• Subtyping

– If t3 <: t1 and t2 <: t4, then t1 -> t2 <: t3 -> t4
– Function subtyping contravariant in argument(s) and covariant in results

• Subtyping vs. parametric polymorphism; bounded polymorphism