1 语言介绍

本课设中实现的是一个 C 语言变体的解释器。这个解释器是用 Haskell 语言写成的。由于这个解释器的代码中使用了 Haskell 语言的特性，下文或多或少会提及，因此这里预先说明。

1.1 文法简述

C 语言的文法可以分成表达式、语句、函数等构成，它们的简要定义如下：

$$
\begin{aligned}
\text{exp} ::=\ & \text{constant} \\
|\ & \text{name} \\
|\ & \text{exp op exp} \\
|\ & \text{unaryop exp} \\
|\ & \text{exp unaryop} \\
|\ & \text{name (optional(\&) name, \dots)} \\
\text{stmt} ::=\ & \text{exp;} \\
|\ & \text{def;} \\
|\ & \text{if (exp) stmt;} \\
|\ & \text{if (exp) stmt else stmt;} \\
|\ & \text{while (exp) stmt;} \\
|\ & \text{for ((def | exp) ; exp; exp) stmt;} \\
|\ & \text{break;} \\
|\ & \text{continue;} \\
|\ & \text{return optional(exp);} \\
|\ & \text{\{ many(stmt) \}} \\
\text{function} ::=\ & \text{type name (type optional(\&) name, \dots) \{ many(stmt) \}}
\end{aligned}
$$

1.2 语言功能简述

解释器实现的 C 语言变体有如下功能：

- 高维数组的读写
- 函数的传值和传引用调用，以及递归调用
- 对字符串的支持（字符串不视为数组）
- 内置函数 `print`，`readInt`，`readFloat`，`readChar`，`readString`
- 全部控制流语句

2 词法分析和语法分析

词法分析和语法分析使用了 Haskell 语言专用的工具：alex 和 happy。Alex 相当于 flex，happy 相当于 bison。

2.1 词法分析

Alex 的使用基本与 flex 相同，不同的是 alex 的词法分析器有一个状态机，用户可以自己在自定义的状态之间转移。Alex 的规则如下：

```
<0> if        { tok If }
```

其中 `<0>` 要求状态机处于 `0` 这个状态时才使用这条规则，后面与 flex 相同。使用这个规则能够很方便地处理嵌套注释：

```
<0>       "/*" { nestComment `andBegin` comment }
<0>       "*/" { \_ _ -> alexError "Error: unexpected closing comment" }
<comment> "/*" { nestComment }
<comment> "*/" { unnestComment }
```

`` `andBegin` comment `` 的作用是转移到状态 `comment`，`nestComment` 将全局状态中的计数器（初值 0）增加 1，`unnestComment` 减少 1，减少到 0 时转移到状态 `0`。同时，遇到 EOF 时如果嵌套层数不是 0 也报错。这样就解决了嵌套注释问题。

2.2 语法分析

Happy 的使用也与 bison 类似，但是 happy 可以定义 “函数”，函数的参数是其它非终结符，这样就增强了代码可复用性。

在对语句进行语法分析时，经常遇到 shift/reduce 冲突。在 bison 中，这可以通过指定一条规则的优先级高于另一条来解决，而 happy 中有更好的方法，可以直接使用 `shift` 指定此处进行 shift：

```
stmt :: { Stmt L.Range }
    : if '(' exp ')' stmt %shift
    | if '(' exp ')' stmt else stmt
```

另外注意我们可以在语法分析的时候就区别左值和表达式（右值）。我们注意左值具有如下形式：要么是一个变量名，要么是一个左值加一个数组索引运算符。同时，出现在赋值操作左侧的，以及自增自减运算符的操作数必须是左值。注意到这两点，就可以在语法分析期间就区别这两者了。这为后续的工作节省了许多麻烦。

3 语义分析和中间代码生成

我们的解释器选用的中间代码形式是抽象语法树（AST）。

解释器中，语义分析和中间代码生成是同时完成的。语义分析的任务就是检查语法分析生成的语法树是否合法，而经检查确定合法的语法树就是（或者差不多是）AST。因此，语义分析产生的 AST 正是语法树合法的**证明**。这里反映了一个极其重要的思想，我们接下来还会看到：

定义特定的数据类型来保证数据的合法性。

与常见的语言不同，Haskell 能够定义复杂的数据类型，甚至通过数据类型表达某些对数据的约束的。这一点为解释器的良好性质提供了莫大的帮助。

3.1 类型论的记号

为了方便对语法分析的规则，尤其是其中关于类型的规则进行方便和形式化的叙述，这里将采用类型论的记号。这套记号来自于数理逻辑中的相继式（sequent calculus），在类型论中一般用来表达类型规则，也作为编程语言中表达类型检查或类型推导规则的记号。例如，在函数式编程中有名的 Hindley–Milner 类型系统的类型推导规则使用的就是这种记号，见 [2]。

我们依次介绍这套记号与相关的概念。

类型

类型（type）是编程语言中常见的概念：C 语言中，`int`、`float` 就是类型。熟悉计算机的读者可能会说：“`int` 就是内存中采取补码表示的 4 个字节；`float` 就是采取 IEEE754 浮点数表示的 4 个字节。” 但是这里我们不采取如此底层的视角来看待类型：我们认为，类型是一种集合。[1]

> **定义 3.1.1 (类型).** **类型**是一种集合，满足其中的每个元素不能同时属于两个类型。

这个定义隐含的意思是，当我们把一个 `int` 类型的值当作浮点数来使用时，并不是这个值具有了两种类型，而是有一个 `int` 到 `float` 的函数隐式完成了类型的转换。

特别的，我们认为 C 语言中的 `void` 也是一个类型，虽然它的值既不可以直接构造，也不可以参与运算。然而，这个类型在我们的抽象语法树中有重要的作用。注意到 `void` 类型不是没有值，而是有恰好**一个**值（虽然这个值在运行时不会真正被表示出来），返回类型为 `void` 的函数返回的正是这个值。

如果值 E 属于类型 s 确定的集合，我们就说 E 具有类型 s，记作 $E : s$。例如，$1+2$ 是整数，因此有 $1+2 : \mathrm{int}$。我们用小写希腊字母（如 α）表示泛指的类型。

有一些能够从其他类型中构造的类型，我们列举几个如下：

- 如果 α, β, γ 是类型，那么 $\alpha \to \beta$ 也是类型，表示 α 到 β 的函数，$\alpha \to \beta \to \gamma$ 表示 α, β 到 γ 的函数。[2]
- 如果 α 是 C 语言中的类型，那么 $\alpha[]$ 是 α 类型的元素构成的数组的类型。

C 语言中所有类型的 BNF 文法定义如下（不包括 C 函数）：

$$
\begin{aligned}
\mathrm{Basic} ::=\ & \mathrm{int} && \text{（基本类型）} \\
\mid\ & \mathrm{float} \\
\mid\ & \mathrm{char} \\
\mid\ & \mathrm{string} \\
\mid\ & \mathrm{void} \\
\mathrm{Type} ::=\ & \mathrm{Basic} && \text{（全部类型）} \\
\mid\ & \mathrm{Type}[]
\end{aligned}
$$

上下文

类型论中不采用符号表的概念，取而代之的是上下文（context）。这是因为在进行语法分析时，我们几乎只需要关心进行检查的当前位置有定义的变量，而非全部。

> **定义 3.1.2** (上下文). 在程序某处有定义的全部变量称为程序该处的**上下文**。

我们一般用逗号分隔的列表表示上下文，先定义的变量写在左侧，后定义的变量写在右侧。例如，$a : \mathrm{int}, b : \mathrm{float}$ 就是一个合法的上下文。我们用大写希腊字母（如 Γ）表示泛指的类型。

我们也用逗号表示上下文的扩展。这就是说，如果 Γ 是合法的上下文，则 $\Gamma, a : \alpha$ 是表示在 Γ 之后新定义了类型为 α 的变量 a 的合法上下文。类似的，我们也像这样表示上下文之间的连接。

判据

对一段程序合法性的判断离不开上下文。例如， `a = 1;` 只在 `a` 有定义时是合法的程序。由此我们引入判据（judgement）的概念：

> **定义 3.1.3** (判据). 一个（可能）在上下文中的命题称为一个**判据**。

这里的命题一般是对表达式类型的判断。我们用符号 $\Gamma \vdash \ldots$ 表示判据，给 $\vdash$ 加上角标以表示判据的种类（关于表达式的，关于语句的，等等）。

例如，考虑以下程序：

```
int a, b;
a + b;
```

其中 `a + b` 对应的一个成立的判据就是 $\mathrm{a} : \mathrm{int}, \mathrm{b} : \mathrm{int} \vdash_E a + b : \mathrm{int}$（$_E$ 表示表达式）。

类型规则

判据本身是关于程序合法性的论述，但是判据是命题，同样可能不成立。例如，上例中的判据可以改为 $\mathrm{a} : \mathrm{int}, \mathrm{b} : \mathrm{int} \vdash_E a + b : \mathrm{string}$，但是显然不成立。类型规则（typing rule）就是机械地判断一个判据是否成立的方法。

> **定义 3.1.4 (类型规则).** “如果某些作为条件的判据成立，那么某个作为结论的判据成立”，形如这样的规则称为一个**类型规则**。

类型规则的记号是这样的：条件和结论有分数线隔开，上面是条件，下面是结论。特别的，条件可以为空，这时作为结论的判据恒成立。

类型规则不是像判据那样可以成立可以不成立的；类型规则是给定的**规则**，这些规则确定了语言中所有合法的程序。例如，C 语言的一条类型规则可能是这样的：

$$\frac{\Gamma \vdash_E e_1 : \mathrm{int} \quad \Gamma \vdash_E e_2 : \mathrm{int}}{\Gamma \vdash_E e_1 + e_2 : \mathrm{int}}$$

如果给出了一门语言全部的类型规则，稍加改写，就可以得到这门语言的类型检查算法。因此，把算法用类型规则来给出完全没有问题。事实上，Hindley–Milner 类型系统中的类型检查算法 Algorithm W 就是以这种形式给出的。

作为示例，我们给出上面规则对应的伪代码：

```
type checkExp(Context ctx, Exp exp) {
    if (exp is Add exp1 exp2) {
        assert(checkExp(ctx, exp1) == int);
        assert(checkExp(ctx, exp2) == int);
        return int;
    } else ...
}
```

3.2 语义分析

接下来我们就使用上述的记号给出 C 语言的类型规则。受限于篇幅，这里只给出部分重要的规则，以体现出算法中关键性的思想。

de Bruijn index

至今为止，我们都是用变量的名称来指代上下文中的某个变量的，这就有诸多问题：如果我们使用了不存在的变量名呢？又或者变量名重复了呢？（注意，可能内外两个作用域中有同名的变量，这时并没有变量重定义错误。）为此，我们采用 de Bruijn index [1]（下文简称 DBI）来表示变量。DBI 的类型规则有两条（I 表示 DBI）：

$$\frac{}{\Gamma, a : \alpha \vdash_I \mathrm{var}_0 : \alpha}$$

$$\frac{\Gamma \vdash_I \mathrm{var}_i : \alpha}{\Gamma, b : \beta \vdash_I \mathrm{var}_{i+1} : \alpha}$$

这其实就是说，如果从 0 开始计数的话，var_i 指代的就是所在上下文中的右起第 i 个变量，并且具有那个变量的类型。因此，只要下标 $i \geq 0$ 并且小于上下文的长度，所有用 DBI 表示的变量就被保证具有正确的作用域。

使用 DBI，我们就可以完全使用序号来指代上下文的变量，而变量名从此就不需要了。因此，在下文中，我们统一省略变量名，上下文也因此只写作类型的列表。

注意，这里介绍 DBI 不仅仅是为了记号上的方便。DBI 不仅是下文中将要使用的记号，同样也是解释器的代码中表示变量的方式。在解释器中正是使用 DBI 来代替符号表的。前面已经说过，Haskell 有通过定义数据类型来约束数据的能力，这里 DBI 对应的数据类型就施加了 $0 \leq i < \mathrm{length}(\Gamma)$ 的约束。这里，我们再一次体现了 “定义特定的数据类型来保证数据的合法性” 的思想：DBI 保证了变量具有正确的作用域。除此之外，DBI 在解释运行时也有独有的优势，这一点我们之后就会看到。

编译器是怎么将语法树中的变量名翻译为 DBI 的呢？在语义分析过程中，编译器维护了一个从变量名到变量对应的 DBI 的映射，当：

- 遇到映射中存在的变量时，翻译为对应的 DBI；
- 遇到映射中不存在的变量时，报错；

- 定义新变量时，将其映射到 var_0，原本所有映射到的 var_i 改为 var_{i+1}（因为新的变量加入上下文的右侧了）。

左值

在之前的语法分析中，我们已经区别了左值和表达式（右值），因此这里我们只检查左值是否合法，而不需要判断一个表达式是不是左值。这里给出检查左值的类型规则。

变量都是左值（LV 表示左值）：

$$\frac{\Gamma \vdash_I \text{var}_i : \alpha}{\Gamma \vdash_{LV} \text{var}_i : \alpha}$$

如果 lv 是具有数组类型的左值，exp 是能够转换成 int 类型的表达式，那么 $lv[exp]$ 是左值（cast 的定义见下一节）：

$$\frac{\Gamma \vdash_{LV} lv : \alpha[] \quad \Gamma \vdash_E exp \quad \text{cast}(exp, \text{int})}{\Gamma \vdash_{LV} lv[exp] : \alpha}$$

最后，最终得到的左值必须具有基本类型（因为我们的解释器不允许对数组的直接操作），才是一个合法的左值。这就确保了所有对数组的索引操作层数都与数组的维数相同。

表达式

由于表达式都具有基本类型，因此本节中的类型变量默认都是基本类型。

在给出表达式的类型规则之前，我们先定义两个函数来表达 C 语言中复杂的类型转换规则。

- $\text{cast}(exp, \alpha)$ 尝试通过插入类型转换函数将 exp 转换为 α 类型。虽然得到了不同的表达式，但转换得到的表达式将沿用 exp 的名字。这个函数能够在 float, int, char 之间执行有损转换，以及任何类型到 void 的转换（返回 void 类型唯一的那个值）。

- $\text{unify}(exp_0, exp_1)$ 尝试通过插入类型转换函数将 exp_0 和 exp_1 转换为相同类型，并返回那个类型。同样，转换得到的表达式将沿用之前的名字。这个函数只在 float, int, char 之间执行无损转换。特别的，参数有一方的类型是 string 或 void 时转换必定失败（因为这些类型的值不能参与计算）。

这里插入的类型转换函数都是抽象语法树中的构造，因此在 C 语言中隐式的类型转换，在抽象语法树中就是显式的。

这两个函数会出现在类型规则中条件的位置。如果转换不成功，这两个函数就会报错并停止整个类型检查。它们在 “不报错（真）/报错（假）” 的视角下可以被看作命题。

下面就是表达式的类型规则。首先，常量是表达式（以 int 类型的常量为例）：

$$\frac{n \text{ is an integer}}{\Gamma \vdash_E n : \text{int}} \cdots$$

左值也是表达式：

$$\frac{\Gamma \vdash_{LV} lv : \alpha}{\Gamma \vdash_E lv : \alpha}$$

二元运算，包含 $+, -, *, /$，是表达式（以 $+$ 为例）：

$$\frac{\Gamma \vdash_E e_1 \quad \Gamma \vdash_E e_2 \quad \alpha := \text{unify}(e_1, e_2)}{\Gamma \vdash_E e_1 + e_2 : \alpha} \cdots$$

取模运算 $\%$ 稍有不同，因为它要求运算数是整型：

$$\frac{\Gamma \vdash_E e_1 \quad \Gamma \vdash_E e_2 \quad \text{unify}(e_1, e_2) = \text{int}}{\Gamma \vdash_E e_1 \,\%\, e_2 : \text{int}}$$

比较运算，包含 $=, \neq, <, >, \leq, \geq$，是表达式（以 $<$ 为例）：

$$\frac{\Gamma \vdash_E e_1 \quad \Gamma \vdash_E e_2 \quad \text{unify}(e_1, e_2)}{\Gamma \vdash_E e_1 < e_2 : \text{int}} \cdots$$

最后是赋值运算。抽象语法树中有两种赋值运算符：一个返回赋值后的值（称为 assign），一个返回赋值前的值（称为 retassign）。其中，前者是 C 语言中固有的赋值，后者则是抽象语法树中的内部表示。它们的类型规则是相同的（以 assign 为例）：

$$\frac{\Gamma \vdash_{LV} lv : \alpha \quad \Gamma \vdash_{E} e \quad \text{cast}(e, \alpha)}{\Gamma \vdash_{E} \text{assign}(lv, e) : \alpha} \cdots$$

抽象语法树的表达式中只允许这些运算符。其它的运算符，比如 `a += b`，可以翻译成 $\text{assign}(a, a+b)$，`++a` 翻译成 $\text{assign}(a, a+1)$，`a++` 翻译成 $\text{retassign}(a, a+1)$，`-a` 翻译成 $0-a$。这里虽然有一些类型检查上的细微差异，但是不再赘述了。

表达式中也允许函数调用，这将推迟到函数一节。

语句和语句块

语句和语句块的类型要求相对较少：所有选择和循环的条件表达式可以转换（cast）成 int，所有返回值能够转换成函数的返回值类型，只有这两项。

这里还有一个重要的语义检查：判断 `break` 和 `continue` 是否在循环中。这可以简单地检查完成：

1. 置布尔型变量 inloop 的初始值为 False。
2. 遍历语法树：
   - 进入循环时，置 inloop 为 True。
   - 退出循环时，还原 inloop 的值。
   - 遇到 `break` 或 `continue` 时，如果 inloop 为 False，报错。

这里还有一种方法：通过 CPS 变换将控制流中的 `break` 和 `continue` 完全消除（见 [4]），不过这里没有使用。

前面提到了对于变量定义的处理，这里特别给出一下变量定义的类型规则（$_{SB}$ 表示语句块，语句块的类型是其中所有返回值共同的类型）：

$$\frac{\Gamma, \beta \vdash_{SB} stmts : \alpha}{\Gamma \vdash_{SB} \text{define new var of type } \beta; stmts : \alpha}$$

可以看到定义新变量后上下文确实增大了。如果此时定义的变量有赋予初始值，则在 $stmts$ 前插入一句赋值表达式。这里我们让新变量的作用域持续到当前语句块的末尾，由此，我们获得了在任何地方新建语句块以限制其中变量的作用域的能力。以下就是一个实际例子。

与表达式的情况类似，抽象语法树中的语句结构也比语法树中少。`if` 可以轻松地翻译成 `if ... else`，`for` 循环也可以翻译成 `while`，这里只说明最困难的 `for` 中包含定义的情况：

```
for(int a = 0; a < 10; a++) {...}
```

通过创建语句块限制作用域，翻译为

```
{
    int a = 0;
    while (a < 10) {
        ...
        a++;
    }
}
```

函数

函数的检查和语句块的检查相差无几：将函数参数当作新定义的变量加入上下文，就可以正常进行（$_{F}$ 表示函数）：

$$\frac{\Gamma, \Delta \vdash_{SB} stmts : \alpha}{\Gamma \vdash_{F} stmts : \text{Function}(\Delta, \alpha)}$$

这里 Δ 是函数参数（对应的上下文）的类型，α 是返回值的类型。

本节的重点在于函数调用。我们的解释器允许两种形式的参数：传值和传引用。其中，值只能是表达式，引用只能是 DBI。

我们考虑函数的参数每个以调用函数：

- 如果函数的参数是传值调用的，就把这个参数翻译为定义新变量并赋实参作为初值。
- 如果函数的参数是传引用调用的，就用实参的引用替换掉函数体中所有型参对应的引用。
- 最后，将得到的语句块内联进调用处。

读者可能会注意到，这样内联会导致递归函数产生无限长的抽象语法树。这是正确的，但是不要紧。Haskell 语言有一个称为惰性求值的特性：一个值只要没有用到，它就不会被求出来。对这个概念理解困难的读者，也可以设想这里内联的是一个零元的匿名函数，当之后实际解释运行到这里时才调用这个函数，进一步展开抽象语法树。因此，只要程序没有无限地递归下去，抽象语法树就不会无限地展开，也就不会有问题。如果进入死递归了，因为有垃圾回收器，也不会导致内存溢出。但是，这导致了抽象语法树不能打印，因为打印的抽象语法树仍然会是无限长的，唯一的解决办法是在打印时将内联的语句块丢掉。

最后，我们还要对每个函数进行一个简单的程序流分析，确保返回值不为 void 的函数的每一条可能的运行路径上都有一个 `return` 语句。算法如下：

- 遇到循环语句时，跳过；
- 遇到分支语句时，递归检查两个分支，如果都成功，则成功返回；否则继续；
- 遇到返回语句时，成功。
- 遇到函数结尾时，失败。

为了形式统一，我们在返回值是 void 的函数体最后插入一条 `return` ，这样函数执行时就可以保证会碰到 `return` 语句了。

4 解释运行

所谓解释运行，就是把一段 C 代码映射到一段可以运行的 Haskell 代码。这个映射分为数个部分：

(1) 将 C 的类型映射到 Haskell 的类型；

(2) 将 C 的上下文映射到 Haskell 的程序状态；

(3) 将 C 的表达式映射到 Haskell 中具有由 (1) 确定的对应类型的表达式；

(4) …

不过在此之前，我们需要先定义笛卡尔积的记号。

> **定义 4.0.1 (笛卡尔积).** 对于类型 A 和 B，记类型 $A \times B$ 为它们的**笛卡尔积**，使得对于任意的 $a : A, b : B$，都有 $(a, b) : A \times B$。并且存在映射 $\pi_1 : A \times B \to A, \pi_2 : A \times B \to B$，使得 $\pi_1((a, b)) = a, \pi_2((a, b)) = b$。

定义 C 的基本类型到 Haskell 的类型的映射 M_{basic}：

$$
\begin{aligned}
M_{\text{basic}} : \text{CType} &\to \text{HaskellType} \\
\text{int} &\mapsto \text{Int} \\
\text{float} &\mapsto \text{Float} \\
\text{char} &\mapsto \text{Char} \\
\text{string} &\mapsto \text{Text} \\
\text{void} &\mapsto ()
\end{aligned}
$$

这里第一行表示要定义的映射的类型，下面的行表示具体的映射关系。注意 Haskell 中的类型是以大写开头的。() 是 Haskell 中一个特殊的类型，它只有一个值，也写作 ()。

递归地定义 C 的类型到 Haskell 的类型的映射 M_{type}：

$$\begin{aligned} M_{\text{type}} : \text{CType} &\to \text{HaskellType} \\ \alpha &\mapsto M_{\text{basic}}(\alpha) \quad (\alpha \in \text{Basic}) \\ \alpha[] &\mapsto \text{Vector } M_{\text{type}}(\alpha) \end{aligned}$$

Vector a 是 Haskell 中 a 类型的数组。

递归地定义 C 的上下文到 Haskell 的程序状态的映射 M_{ctx}：

$$\begin{aligned} M_{\text{ctx}} : \text{Context} &\to \text{HaskellType} \\ [] &\mapsto () \\ \Gamma, \alpha &\mapsto M_{\text{ctx}}(\Gamma) \times M_{\text{type}}(\alpha) \end{aligned}$$

我们注意到在上下文中某个位置的变量，经过 M_{ctx} 的映射后仍在相同位置。这时 DBI 的良好性质再一次显现出来了：我们可以非常方便地定义从上下文中得到 DBI 对应变量的值的映射（不考虑数组的复杂情况）。

$$\begin{aligned} M_{\text{dbi}} : (\Gamma \vdash_I \text{var}_i : \alpha) &\to (M_{\text{ctx}}(\Gamma) \to M_{\text{basic}}(\alpha)) \\ \text{var}_0 &\mapsto \pi_2 \\ \text{var}_{i+1} &\mapsto M_{\text{dbi}}(\text{var}_i) \circ \pi_1 \end{aligned}$$

这里的 $\circ$ 是函数复合。读者可以看到，这个函数的类型很复杂，其中甚至包含了别的函数。然而读者同样可以验证，这个函数具有正确的类型，而解释器中真正的代码与此相差无几，这也就是说解释器的良好性质——表达式具有正确的类型，一直被保持到了现在。给变量赋值的映射稍显复杂，有兴趣的读者可以尝试写出。提示：这个映射具有类型 $(\Gamma \vdash_I \text{var}_i : \alpha) \to M_{\text{basic}}(\alpha) \to (M_{\text{ctx}}(\Gamma) \to M_{\text{basic}}(\alpha) \times M_{\text{ctx}}(\Gamma))$。在实际的解释器中，这部分代码通过了 state monad 进行了抽象，关于 monad 的介绍可以见 [3]。

我们还可以定义 C 的表达式到 Haskell 中类型正确的表达式的映射（同样略去赋值部分）：

$$\begin{aligned} M_{\text{exp}} : (\Gamma \vdash_E \mathit{exp} : \alpha) &\to (M_{\text{ctx}}(\Gamma) \to M_{\text{basic}}(\alpha)) \\ \text{var}_i &\mapsto M_{\text{dbi}}(\text{var}_i) \\ \text{Add } e_1 e_2 &\mapsto M_{\text{exp}}(e_1) + M_{\text{exp}}(e_2) \\ \text{FloatToInt } e &\mapsto \text{round } M_{\text{exp}}(e) \\ \text{Run } \mathit{stmts} &\mapsto \text{catchError } M_{\text{stmts}}(\mathit{stmts}) \\ &\cdots \end{aligned}$$

这里 round 是取整函数。注意 Haskell 的函数调用不打括号。Run 是之前提到的内联函数体，这里为什么要捕获异常马上就会讲到。

为了定义语句执行的映射，我们定义类型 $\text{Next} = \{\text{B}, \text{C}, \text{N}\}$，分别表示 `break`，`continue` 和继续执行。

这样我们就能定义最后的语句执行的映射：

$$\begin{aligned} M_{\text{stmts}} : (\Gamma \vdash_{SB} \mathit{stmts} : \alpha) &\to (M_{\text{ctx}}(\Gamma) \to \text{Next}) \text{ throws } M_{\text{basic}}(\alpha) \\ \{\} &\mapsto \text{N} \\ \mathit{stmt}; \mathit{stmts}; &\mapsto \text{let next } := M_{\text{stmt}}(\mathit{stmt}) \text{ in} \\ &\quad\ \text{if next } \neq \text{N} \\ &\quad\ \text{then next} \\ &\quad\ \text{else } M_{\text{stmts}}(\mathit{stmts}) \end{aligned}$$

$$
\begin{aligned}
M_{\text{stmt}} : (\Gamma \vdash_S \mathit{stmt} : \alpha) &\to (M_{\text{ctx}}(\Gamma) \to \text{Next}) \text{ throws } M_{\text{basic}}(\alpha) \\
\text{break} &\mapsto \text{B} \\
\text{continue} &\mapsto \text{C} \\
\text{Exp } e &\mapsto M_{\text{exp}}(e); \text{N} \\
\text{IfElse } \mathit{cond}\ b_1 b_2 &\mapsto \text{if } M_{\text{exp}}(\mathit{cond}) \\
& \quad \text{then } M_{\text{stmts}}(b_1) \\
& \quad \text{else } M_{\text{stmts}}(b_2) \\
\text{While } \mathit{cond}\ \mathit{body} &\mapsto \text{if } M_{\text{exp}}(\mathit{cond}) \text{ and } M_{\text{stmts}}(\mathit{body}) \neq \text{B} \\
& \quad \text{then } M_{\text{stmt}}(\text{While } \mathit{cond}\ \mathit{body}) \\
& \quad \text{else N} \\
\text{return } e &\mapsto \text{throwError } M_{\text{exp}}(e)
\end{aligned}
$$

刚刚提到的捕获异常正是捕获的这里函数的返回值。之前我们已经保证了函数的任何可能的执行路径上都有一条 `return` 语句，因此这个异常是必定会被抛出并被捕获的。

最后，我们要求 main 函数不接受参数，返回 void。这样，$M_{\text{stmts}}(\text{main})$ 的类型就确定为了 $(M_{\text{ctx}}(\text{global}) \to \text{Next}) \text{ throws } ()$。其中 global 是所有全局变量对应的上下文，它们的初始值就构成了 $M_{\text{ctx}}(\text{global})$。我们传入这个参数，并安静地捕获这个异常，一个 C 语言解释器就完成了。

5 总结

在课设过程中我几乎没有遇到困难。由于之前的使用，在词法语法生成工具方面几乎没有遇到问题；而由于长久以来对于类型论和 DBI 的熟悉，在语义分析之后的工作也几乎一帆风顺。

本次的解释器有诸多亮点：

- 使用 DBI 代替符号表，以此约束变量作用域；
- 简单的控制流分析；
- 函数的传值和传引用调用；
- 能检查作用域（未定义）错误，重定义错误，参数数量错误，`break`/`continue` 不在循环内的错误，函数可能的控制流上可能没有返回值的错误，以及各种类型错误；
- 直到最后都保持的类型正确。

脚注

1. 严格来说，这是在讨论类型论的集合模型。
2. $\alpha \to \beta \to \gamma$ 的含义是这样的：$\to$ 是右结合的，因此它是一个接受一个 α，返回 $\beta \to \gamma$ 的函数，而后者又接受 β，返回 γ，因此总共接受了两个参数。由于 C 语言中函数不能返回函数，因此没有采用这种说法，以免引起疑惑，但还是沿用了类型论的这个记号。

参考文献

[1] de Bruijn, Nicolaas Govert (1972). *Lambda Calculus Notation with Nameless Dummies: A Tool for Automatic Formula Manipulation, with Application to the Church-Rosser Theorem*. Indagationes Mathematicae. 34: 381–392. ISSN 0019-3577.

[2] Damas, Luis; Milner, Robin (1982). *Principal type-schemes for functional programs*. 9th Symposium on Principles of programming languages (POPL'82). ACM. pp. 207–212. doi: 10.1145/582153.582176. ISBN 978-0-89791-065-1.

[3] Moggi, Eugenio (1991). *Notions of computation and monads*. Information and Computation. 93

(1): 55–92. CiteSeerX 10.1.1.158.5275. doi:10.1016/0890-5401(91)90052-4.

[4] Appel, Andrew W. (2007). *Compiling with Continuations*. ISBN 978-0521033114.