基于数值型输入输出样例的程序合成器

摘要

自从第一台电子计算机诞生至今，计算机程序帮助人们完成了各种各样繁重而复杂的任务，给人们的生活带来了极大的便利。但程序的编写一直极其让人耗费心力。人们希望能借助计算机自动合成程序，将人们从枯燥的编程工作中解放出来。程序合成，或许可以帮我们实现这个愿望。它也可以在新算法的发现、自动解题、智能教学等方面大显身手。而基于输入输出样例的方式是程序合成中表达用户的意图的最自然、最简单的形式。我使用 C++语言实现了一个基于数值型输入输出样例的程序合成器的简易版本。它包含两个引擎，一个运用程序合成技术，能够发现输入输出之间的规律的函数合成引擎，和一个专门用于计算多项式函数的通项公式的通项引擎。其中前者是我的工作的重点，而后者只是用来做参照对比的。我将合成引擎应用在公务员考试的数字推理题中，能够快速有效地解决其中绝大部分问题。本论文将详细叙述基于数值型输入输出样例的程序合成器的实现过程、相关技术以及遇到的问题和解决方法。

关键词：程序合成，自动编程，函数片段，输入输出样例，规律发现

A PROGRAM SYNTHESIZER BASED ON NUMERICAL INPUT-OUTPUT EXAMPLES

ABSTRACT

Since the birth of the first electronic computer, computer programs have helped us accomplished all kinds of hard and difficult tasks, leaving us only convenience. Nevertheless, the coding work always puzzles many programmers. We hope that computers could help us generate programs automatically, setting us free from the boring coding work. Program synthesis may help us realize that dream. It also does very good job in the discovery of new algorithm, auto-solving, intelligent torturing and so on. Moreover basing on the input-output examples is the most natural and simplest form to present user intents. I made a simple PROGRAM SYNTHESIZER BASED ON NUMERICAL INPUT-OUTPUT EXAMPLES by the language of C++. It contains two engines, one of which is a synthesis engine using the program synthesis technique, which can discover the regular pattern between the terms; the other is especially used for calculate the General Term of the polynomial expression. The former is the focus of my work, and the latter is only for reference and comparison. When applied to the numerical reasoning problems of the Civil Service Examinations, it works quite well and is able to solve most of them. This paper illustrates the details of my realization of the PROGRAM SYNTHESIZER BASED ON NUMERICAL INPUT-OUTPUT EXAMPLES, the related techniques, the problems encountered and solutions to them.

Key words: program synthesis, automatic programming, function fragment, input-output examples, pattern discovery

目 录

第一章 绪论

面对日益增长的数据信息，人们越来越渴望能够使用自动化的方法，将数据中有用的信息提炼成易于展示和理解的形式，来帮助人们发现数据中的规律，获得新的知识，自动化完成繁重却乏味的作业。成千上万的终端用户每天面对着海量的数据，进行着重复而乏味的整理工作，耗费了大量的劳力。采用自动化的方法，可以使这些终端用户从无趣的重复劳动中解放出来，利用计算机强大的计算优势，自动地生成用户想要的处理程序来处理他们的工作。因此，最近程序合成成了人们关注的一个新兴领域。

程序合成，原来的相关工作一般被归为自动编程。用户输入一定的约束说明或规约，由计算机通过搜索组合等方式自动生成满足规约的程序。其实，自动编程的概念并不是十分新奇的，不过较早的研究大都受限于计算机的计算能力、搜索等方面的技术成熟度等等，最终成果并不显著。近期，该领域又开始重新被人们重视起来。微软的一个研究组在研究中取得了一定的成果，并初步总结了相关问题的方法论。不过，目前这个领域的发展还处于初级阶段，相关的研究还有很长的路要走。

程序合成的任务就是由某些约束形式的用户意图来生成可执行程序。它做的事情与编译器相比较的话，编译器是将输入程序写成有结构性的语言，通常执行表达式导向的翻译；而合成器能够接受多样的混合形式的约束（比如，输入输出样例，示范，输入输出间的逻辑关系，自然语言，程序片段或低效的程序等等），通常执行某些程序的空间下的某种搜索。[1] 简言之，给定高级的约束说明和低级的编程语言之后，我们的目标就是自动地合成符合约束条件的有效的程序。[3]

我的工作致力于实现一个简单的数值型函数的合成器。它基于用户的输入输出样例，也就是试图利用用户提供的输入输出数值对，找出输入输出，及各个输出之间的内在运算规律和逻辑关系，推测用户的意图，生成最能满足用户需求的小函数。

我将根据已有的程序合成的基本方法，结合相关领域的较成熟的各项技术，如人工智能的相关知识等，搜索技术中，如穷举搜索，版本空间代数，逻辑推理技术等等，完成这个简单的程序合成器[2][12][13]。我的程序合成器将能够根据用户给定的数列的部分项，推测出其中的规律，生成足以表达完整数列的程序。并通过用户的反馈进行调整，或者进行重新生成。

程序合成，很有潜力开启编程语言的一个新纪元，把我们从比较低级的 how（怎么做）语言带入到更高级的 what（是什么）语言的时代。[1]我所进行的研究，应用前景也是十分广泛的。对于一维的数值规律发现，可以应用于教育领域，如 IQ 测试的解答数组找规律的问题等等；作为引擎，可以为编程人员提供快速生成代码的接口；数据中的模式发现，在对数据进行分析预测中发挥着相当重要的作用；如果扩展到二维，很可能应用到对图像和声音等多媒体信息的理解，缺失信息的修复，信息和数据的压缩等等领域中。

第二章　程序合成

2.1　基于输入输出样例的程序合成

从输入和输出样例来合成程序，是具有相当的难度的。

我把程序运行的过程分为“三个元素”，即程序的输入，程序的算法，程序的输出。通常，程序本身就是算法的体现。把实现好的“算法”（程序或函数）以及“输入”交给计算机，计算机总是给出确定的“输出”。如果将这种流程变换一下，给定输入和输出，让计算机生成由输入到输出的内在联系（算法）就成了我要研究的问题。

这个问题具有相当的难度，是因为，输入、算法和输出，这三个元素并不是给定任何两者都能推出第三者的。举个例子来说，比如一些不可逆的加密算法（如非对称加密 RSA 等[8]），已知了算法本身，从输出不可能有效地转换回输入。这里的有效是指，人类还没有发现比“穷举所有可能的输入，看是否匹配输出”更高效的方法。而软件合成，仅仅从输入和输出就推出算法，其难度又更进一步。

从宏观概念上来看，三个元素已知两者来生成第三者的问题的难度序列如下：

- 输入+程序→输出：这是最简单的，其难度等同于面对实际问题，编程序解决；
- 输出+程序→输入：这是中等难度，程序合成的研究中也含盖了已知程序的逆程序；
- 输入+输出→程序：这是最高难度，该问题具有很高的歧义性，更偏向于智能领域。

再与程序验证来对比，程序验证是在源代码已经提供了的条件下做的，其难度之高也已被公认；而程序合成是在没有程序代码的条件下，要生成这个程序，更加提升了其难度。

当然，程序合成的概念是更加宽泛的，它不仅包括根据用户的输入输出来生成程序，还包括各种各样的用户意图的描述方式。下一小结中将简要介绍程序合成的总体研究方面。

2.2　三个维度

程序合成中主要有以下三个维度[1]：

2.2.1　用户意图

用户意图即约束的种类。程序合成中的最重要的一个维度，就是从用户的角度，如何表示用户意图，这种机制描述方法。

描述用户意图的机制有多种多样。比如：

用户提供输入与输出间的逻辑规约，这适合于解决位向量的逻辑问题；

再如：用户用自然语言（如英语）描述如何访问数据库，要完成怎样的操作，程序合成器通过自然语言处理的方式理解了用户的意图后，合成出相应的 SQL 语句，以此协助用户完成数据库的操作；

再如：用户给定输入输出样例，即示范输入和其对应输出的示例，合成器推测出用户的意图，以合成出相应的小程序，来完成更多的输入到输出的自动化转换；

还有，程序合成器可以根据已有的部分程序片段，或低效率的程序等等，来合成出

更完整，更高效的有实用价值程序等等。

这些程序合成的例子在实际应用中都会帮助用户极大地提高工作效率，将人从众多繁琐，重复的劳动中解放出来，将很大程度地节省人的劳动。

2.2.2 搜索空间

一般的程序合成的过程中，首先要根据解决问题的领域的特殊性，设计领域相关的目标语言，由该语言完成程序合成过程中搜索空间的定义。程序合成的搜索空间取决于程序、语法以及逻辑等方面的空间。

程序的空间，在操作符上可以包括比较操作符、算术运算符、逻辑和位操作符、给定库的接口（API），已知算法的组合等等；在控制结构上可以包含给定循环模版的，给定 API 调用顺序集的，或者更简单的无循环的线性控制结构。

语法的空间，可以是命令式或函数式程序（需要缩减控制结构，限制操作符集合），上下文无关语法/正则表达式，简洁的逻辑表达式等等。

逻辑的空间，比如作用在树、图、字符串等这样的有组织的数据结构上的算法（PTIME[7]），由于其简洁性，将这些逻辑表达用作目标语言对于一些特定问题也十分合适。

2.2.3 搜索技术

可以基于穷举搜索，版本空间代数，机器学习技术，逻辑推理技术等等。许多搜索也会基于 SMT solver 来进行，如微软的 Z3 [14]。

2.3 我的工作范围

毕业设计阶段，我用 C++语言实现了一个基于数值型输入输出样例的程序合成器的简易版本。

对于初步研究，为了化简问题，我决定在以下方面对我的项目做一些限制：计算机可执行的程序，一般都可以抽象成为函数，而函数的输入和输出可以有多种复杂结构的数据类型。此外，有些函数还附带有一些副作用，如申请和释放内存，读写硬盘和外围设备的 IO 操作，绘制屏幕，锁和信号量等的操作等等。我只针对数值型输入输出，不包含副作用的简单函数进行合成。

在数值型的函数中，又有很多种输入输出形式，如整数型，定点小数型，单精度、双精度的浮点型，用一定数据结构表示成的大数类型，还有它们组成的数组或者向量型等等。从现实情况考虑，为了使我的工作量不致过大，课题的领域不要太泛，我做了进一步的限制：

- 对于函数的类型的限制

输入只有一个参数，其数据类型为机器内置的有符号整型（int）；输出的类型为 IEEE754 规定[9]的 64 位双精度浮点型，即机器内置的 double 类型。输入类型定为 int 首先是因为 int 是最简单最常用的数据类型，各个编程语言中都会包含这样的基本数据类型；并且，这样的函数可以等价为数列，输入的数值可以表示数列的第几项，输出的数值可以对应于数列的项的值。其实对于离散的计算机世界，整数也已经具有足够的能力处理全部逻辑和运算问题了。输出的类型定位 double，一是为了使做除法时不会被“截尾”，乘法可以有它对应的逆运算除法了；二是对那些输出随着输入增长飞快的函数（如公比大于 1 的等比数列），double 的输出就不会像整型那样很快就溢出了；使用 double 类型作为输出还带来了另一个好处，IEEE754 中定义的 double 类型中包含了 NaN，+inf，-inf 等，我可以不必过多操心 0 做除数的情况。

- 对于函数逻辑的复杂度的限制

目前，限于搜索的代价，程序合成技术只能生成那些逻辑简单的短小的程序。函数

逻辑的复杂度以及搜索的空间由定义目标语言来限制。我为我的程序合成器编写了两个引擎，它们通过使用不同的技术对同一函数合成问题给出不同方面的答案。通项引擎的相关细节请参看第五章的内容，它用到的更多是数学的知识（待定系数法），用矩阵的运算来解方程，最终得到多项式函数的通项公式。这种合成方法并没有对目标语言的空间进行搜索，而是单纯靠单一模式的运算来得到最终答案，它将所有的输入输出看作是多项式函数产生的，因此结果会比较机械死板，缺乏智能。做这部分只是为了对比参照。合成引擎是我工作的重点部分，它的相关细节请参看第三章和第四章的内容。不同于通项引擎，合成引擎能够探查出函数输出值之间的内在规律，合成出更多样形式的，更加符合用户意图的程序，其缺点是对于稍复杂些的规律，由于生成的表达式的表现形式之繁琐，可能会更难以让人理解，由于是用递推公式形式描述的，具体求某一项的函数值时执行效率可能会稍低。

2.4 程序的使用

合成器通过分别调用通项引擎和合成引擎两个模块尝试解决同一个问题。用户只需给定想要合成的函数的几组输入和输出，即可得到他想要的合成后的函数，以及范围更大的输入输出的结果，用于验证函数合成的正确性。

程序提供了两种输入模式：文件输入和交互式输入。文件输入模式可以将一系列合成问题写入文件，格式请见附录 2，合成器可以批量处理文件中的这些问题，并将结果输出到屏幕以及输出文件中。交互式输入模式下，用户可以即时地输入问题（给出输入输出样例），然后交给合成器去处理，合成器将会把结果返回在屏幕上。当然，程序中的两个引擎的封装性比较好，将其直接编译成库，作为中间件，供编程人员调用也是可以的。

我编写的程序合成器对于解决数字推理类型的题目已经十分有效，基本上可以在很短暂的时间内，找出固有的规律。甚至在人直观上很难找出规律的情况下，它能够提供出一种合理的规律，并能准确预测其他项的值。有时候，它还会探测出新的规律，给出的结果可能与人看出的规律不同，或许会更加简单合理，着实能让人眼前一亮。

2.5 本章小结

程序合成，就目前国内外的发展状况而言，都处于较初级的阶段，可以研究的空间还很大。就程序合成的三个维度来讲，我实现的函数合成器：

（1）采用输入输出样例来表达用户意图；

（2）规定了一套形式化的目标语言来确定合成的搜索空间；

（3）在搜索技术方面采用一定的启发式，并在剪枝后的逻辑树内进行逐步加深的深度优先搜索。

由于是初步探究，为了简化问题，我对课题做出了一系列限制，包括对合成的函数类型的限制，和函数逻辑复杂度的限制。在这种研究方法的指导下，已取得了阶段性成果。

微软研究院 Redmond 实验室的 Sumit Gulwani 博士带领他的小组在程序合成方面做出了许多卓越贡献。他们在程序合成的方法论上奠定了理论基础，并在自动编程、教育等领域利用程序合成技术完成了一些小的样例程序。在基于输入输出样例的合成中，他们做的工作主要在于实际应用性比较强的位向量算法、电子表格数据操作、智能辅导系统等[10]。而我的工作更加偏向理论基础，在更底层上实现数值型函数的合成器。

目前，将我完成的函数合成器应用到公务员考试中的数字推理题中，已经可以较好地解决大部分题目。见附录 2。

第三章　合成引擎目标语言的定义

3.1　目标语言演化史

最初，我对于合成数值型函数毫无思路，只考虑将各种片段随机组合，通过依次验证是否满足所给定的输入输出条件，来确定是否是所需要的合成结果。所以，在最初版本的目标语言定义中，我采用了树的结构表示最终的表达式，将二元运算符号作为节点之间的连接，将最简单的最常用的常数以及变量作为叶子节点。

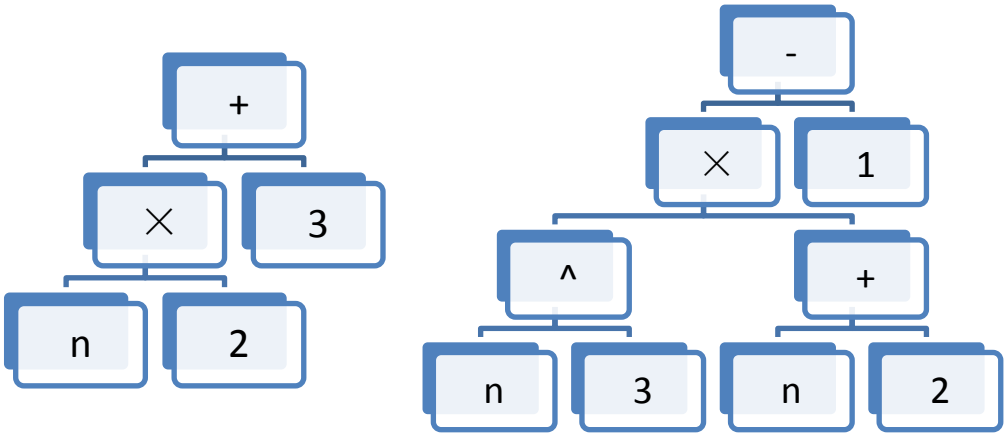

图 1

如图 1 左，代表$f(n) = n \times 2 + 3$；图 1 右，代表$f(n) = n^3 \times (n + 2) - 1$。

这样定义的语言，几乎可以将所有只涉及加减乘除和乘方运算的通项公式表示出来。但其缺点也很明显，叶子节点除了变量 n 之外，需要枚举许多常数（包括正负的整数，有理数，以及无理数），层次结构过于自由，在如此广阔的搜索空间中搜索的时候，就像在大海中捞针，而且搜索的路径中没有什么好的办法判断现在对于最终目标的接近程度，很难进行启发式搜索。而对这样的树进行穷举搜索，树的深度每加深一层，就相当于增加了一个维度，对于搜索来讲将是爆炸级的空间范围。而且，对于像斐波那契数列这种，需要根据前两项的运算来确定函数值的，显得就有些吃力了。

斐波那契函数：

$$f(n) = f(n-1) + f(n-2), \qquad f(0) = 0, f(1) = 1$$

其通项公式由法国数学家比内（Binet）求出，为：[11]

$$f(n) = \frac{1}{\sqrt{5}}[(\frac{1+\sqrt{5}}{2})^{n+1} - (\frac{1-\sqrt{5}}{2})^{n+1}]$$

试想用穷举搜索法要想搜出这样的通项公式，需要多大的代价。而其递推公式却十分简洁，每项等于其前两项的和。这给了我一个重要的启发，用通项公式不如用递推关系的描述更简洁、更直观。

于是，我的目标语言定义有了第二个版本，使用到了各项之间的递推关系。等差数列，各项之差为常数；等比数列，各项之比为常数；还有无数更复杂规律的数列，都可以用通项公式表示。

比如：

f(0) = 4

f(1) = 5

f(2) = 7

f(3) = 11

f(4) = 19

这样的数列，其递推关系是：

$$f(n) = f(n-1) + 2^{n-1}, f(0) = 4$$

即每一项等于上一项加上一个以 2 为公比的等比数列。然而，以 2 为公比的数列的这个部分，还是用通项公式来表示的。当然，我们可以用分而治之的方式把它完全用递推公式表示出来：

$$f(n) = f(n-1) + g(n), \quad f(0) = 4$$
$$g(n) = g(n-1) \times 2, \quad g(1) = 1$$

这样，上述通项公式中的叶子节点 n 就完全不需要了，可以用递推公式替换掉，而且，所需的常数也完全可从用户给定的输入输出值中计算得到。如：$f(0) = 4$中的 4 可直接从输入中得到；$g(1) = 1$中的 1 可由$f(1) - f(0)$得到；而 $g(n)$的公比 2 可由$(f(2) - f(1))/(f(1) - f(0))$得到。这样，所需的常数可以从输入中获取，而不是像上一个版本那样，只能靠盲目地猜测。此时，表示指数函数也只需要乘法即可，乘方的符号也被我去掉了（还因为后面需要每种运算的逆运算，为了避免开方和求对数，所以去掉乘法操作符省去了不少麻烦）。

目标语言最终的版本中，将函数用递归形式化地定义了一下，包含了最多与前两项有关的递推关系，采取了加减乘除四种运算，并考虑到二元运算符两个参数的顺序，将没有交换律的减法和除法又定义了逆向的运算，共六种操作符，详见下一节。而常数被我定义为函数的一种，称为常数函数（constFunction），该函数的每一项都是同一常数。这样统一了规定，为后面的工作提供了便利。

3.2 操作符的定义

首先，为了进行加减乘除的运算我定义了 6 个操作符（Operation）：PLUS，MINUS，R_MINUS，MUL，DIV，R_DIV，其中 R_MINUS，R_DIV 是第二操作数减去/除以第一操作数（即逆向运算）。后文将用符号*表示这六种操作符之中任意一个。考虑到要将其转化成 string 类型的输出形式，操作符号根据运算优先级进行分类，这样在为子表达式加括号的时候便有了依据。具体将函数字符串化的内容请参看后面 3.6 节的内容。

3.3 基本变换与变换

一个基本变换（BasicTrans）是由一个操作符和一个递推函数（Recursion，代码中的类命名为 Function）构成的结构体。这里的递推函数包含多种形式（常数函数、只和 n-1 项有关的递推函数、只和 n-2 项有关的递推函数、和 n-1、n-2 项都有关的递推函数等），具体将在下一节中介绍。

变换（Transform）就是一系列基本变换的依次作用，变换将简记为[ ]符号。

$$[x] := x * g(n) * h(n) \ldots$$

其中*代表上一节中提到的六个运算操作符中任意一种，而 g 和 h 均为递推函数，它们的定义请参看下一节。当然，最简单的变换包含 0 个基本变换，在它的作用下，原数值保持不变。

要将最终结果转化成可打印的字符串的输出形式，由于要考虑加括号的情况，故基本变换和变换都需要区分优先级。具体请参看 3.6 节。

用户给定后面的项的输入输出用例，可能需要推出前面的项的值。故还需要根据f(n)与f(n-1)，f(n-2)之间的规律推出f(n)与f(n+1)，f(n+2)之间的关系。所以，基本变换和变换还需要提供求其逆运算的接口。具体请参看3.5节。

3.4 递推函数的定义

列举跟前一项，前两项的递推关系，递推函数定义为：

$f(n) := C_0$

或 $f(n) := [f(n-1)],\ f(n_0) := C_0$

或 $f(n) := [f(n-2)],\ f(n_0) := C_0,\ f(n_0+1) := C_1$

或 $f(n) := \left[[f(n-2)] * [f(n-1)]\right],\ f(n_0) := C_0,\ f(n_0+1) := C_1$

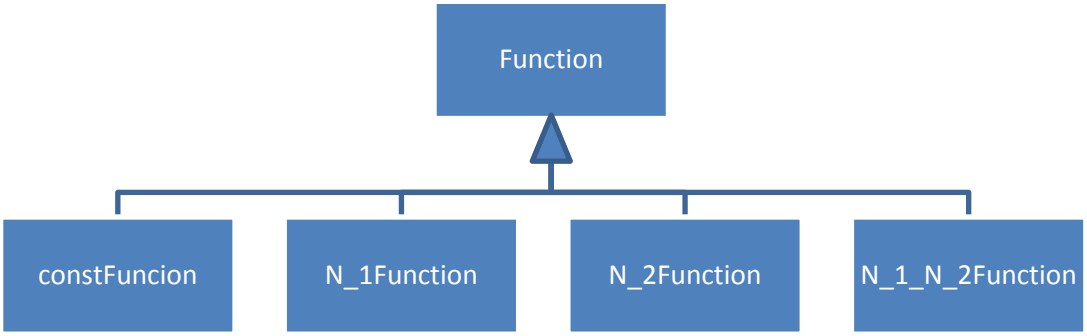

图 2

图2为面向对象的类之间的继承关系，递推函数Function为抽象基类，它具有的多态性，有四种形式：constFuncion（常数函数），N_1Function（只和n-1项有关的递推函数），N_2Function（只和n-2项有关的递推函数），N_1_N_2Function（和n-1、n-2项都有关的递推函数）。这四类递推函数都可以与一个操作符一起构成上一节介绍的基本变换的结构体。

请注意，递推函数的定义，除了给出当前项与前一到两项直接的运算关系以外，一定还要给出边界特殊值，即递归的终止之处（常数函数不需要）。只与前一项有关的N_1Function只需要一对输入输出作为给定特殊值，即上式中的（$n_0 \rightarrow C_0$）。与前两项有关的，则需给定两组特殊值，它们的输入要求是连续的。故N_2Function和N_1_N_2Function的定义中需要给定（$n_0 \rightarrow C_0$）和（$n_0+1 \rightarrow C_1$）。

这样定义之后，可以涵盖几乎全部形式的最多包含前两项的递推关系。比如，以5为公比的等比数列可以表示为：

f(n) = f(n-1)×g(n), f(0) = 1

g(n) = 5

其中f为N_1Function，它用到的变换包含乘号和g，g为constFuncion。再比如，二次函数$f(n) = n^2 - 1$可以表示为：

f(n) = f(n-1) + g(n), f(1) = 0

g(n) = g(n-1) + h(n), g(1) = 1

h(n) = 2

其中f，g均为N_1Function，它们的变换为加号和g，以及加号和h，h为constFuncion。如果递推关系涉及到前两项，如斐波那契函数$f(n) = f(n-1) + f(n-2),\ f(0) = 0, f(1) = 1$，则可以直接表示为：

f(n) = f(n-2) + f(n-1), f(0) = 0, f(1) = 1

其中 f 为 N_1_N_2Function。

Function，constFuncion，N_1Function，N_2Function，N_1_N_2Function 各个类的声明见附录 1。其中抽象基类 Function 中规定了一些纯虚函数，如计算函数值、字符串化等等，由各个实例类分别实现各自的版本。

3.5 函数值的计算

语言规定好函数的形式之后，下面要做的是给定一个参数，能计算出函数的值。

如果像 2.1 节中介绍的最初的版本定义的语言（如图 1 的树状结构表达式），那么求函数值只需将叶子节点中的变量 n 替换为参数给出的具体值，然后再按照后序遍历整棵树即可。

然而，最终版本由递归定义的函数，只能用递归的方式，迭代地求出每一项的函数值。这是有递归的定义决定的，后项就是会依赖到前一到两项的函数值。

从底层的操作符，到中间的变换与基本变换，再到最终的递归函数，它们都有相应的计算数值的方法。对于不同类型的递归函数，根据自身的定义，也有各自计算数值方法的实现。下面依次简述各层次计算数值的方法：

- 操作符

给定两个操作数 a，b，返回运算结果

PLUS：a+b

MINUS：a-b

R_MINUS：b-a

MUL：a×b

DIV：a/b

R_DIV：b/a

- 基本变换

将输入的参数作为第一操作数，将变换的递归函数的求出的结果值作为第二操作数，调用操作符的求值的方法进行求值。

- 变换

如果变换中的基本变换数为 0，则直接将输入的参数作为返回值返回；如果含有一列基本变换，则迭代各个基本变换。

- 递归函数

constFuncion：不论输入任何参数，直接返回常量。

N_1Function：类的定义中包含了一对输入输出的特殊值，以及与 n-1 项的递推变换。如果参数等于给定的输入特殊值，则直接返回输出值；否则，如果参数大于给定的输入特殊值，根据定义，将当前项的前一项经过变换，即得到当前项。

N_2Function：类的定义中包含了两对输入输出的特殊值，以及与 n-2 项的递推变换。如果参数等于某个给定的输入特殊值，则直接返回其对应的输出值；否则，如果参数大于所有给定的输入特殊值，根据定义，将当前项的前两项经过变换，即得到当前项。

N_1_N_2Function：类的定义中包含了两对输入输出的特殊值，以及与 n-2 项的递推变换。如果参数等于某个给定的输入特殊值，则直接返回其对应的输出值；否则，如果参数大于所有给定的输入特殊值，根据定义，将当前项的前一项和前两项分别经过变换，运算符操作，以及再次变换，即得到当前项。

正如 3.3 节最后提到的，如果作为递归终止点的特殊值并不是最小的，即如果要计算给定的特殊值之前的项，那么仅仅有 f(n)和 f(n-1)的关系是不够的。我们还需要利用

数学的方法，根据 f(n)与 f(n-1)，f(n-2)之间的规律，逆推出 f(n)与 f(n+1)，f(n+2)之间的关系。

要计算比给定的 n0 小的输入的函数值，需要将函数的定义转换成如下形式：

$f(n) := C_0$

或 $f(n) := \{f(n+1)\},\ f(n_0) := C_0$

或 $f(n) := \{f(n+2)\},\ f(n_0) := C_0,\ f(n_0+1) := C_1$

或 $f(n) := \{\{f(n+2)\} ** [f(n+1)]\},\ f(n_0) := C_0,\ f(n_0+1) := C_1$

其中{ } 为对应[ ]的逆运算，**为*对应的逆运算。

符号的逆运算对于关系为：

c = a + b → a = c - b

c = a - b → a = c + b

c = b – a → a = b - c

c = a×b → a = c / b

c = a / b → a = c×b

c = b / a → a = b / c

而变换的求逆，需要将每个基本变换求逆，再逆序依次计算。

例如：

若[f(n)]表示 f(n)×3 + 1，则{ f(n+1) }表示(f(n+1) - 1) / 3；

若 f(n) = f(n-2) * f(n-1)，则 f(n-2) = f(n) ** f(n-1)，将 n+2 替换 n，则有 f(n+2) = f(n) * f(n+1)，即 f(n) = f(n+2) ** f(n+1)。

这样，利用[ ]和*运算对应的逆运算{ }和**，以及 f(n+2)，f(n+1)就可以进行逆向推算了。

如此做法的目的是，若用户给定 f(1)=1, f(2)=2, f(3)=3, f(4)=4,合成器经过合成，给出结果函数定义如下：

f(n) = f(n-1)+g(n), f(1) = 1

g(n) = 1

那么如果用户也需要计算 f(0), f(-1), f(-2)的值，这样的函数值的计算方法可以调用逆向推导的方法，求出 f(n)与 f(n+1)的关系，进而满足用户的要求。因此，合成出的函数的定义域是整个有符号整数的范围，当然，最常用的区间还是靠近用户给出的输入值附近的。由于递归式的迭代运算方式，靠近用户给定的输入的这部分区间的函数值的运算效率也更高。

3.6 合成的结果的输出——字符串化·优先级

函数合成后，虽然可以输出任意参数对应的数值输出结果。但用户总还是想知道合成的函数的形式化定义是怎样的。如果合成引擎用于协助终端用户编程，那么也需要一种将合成好的函数的数据结构用文字表达出来的机制。

3.6.1 递推函数的命名

Function 基类有一个函数名字属性，它允许用户给生成的函数起函数名。如 f(n)，g(n)中 f 和 g 就是这个名字。但是，在生成的过程中，会有大量的中间函数，不可能让用户给每个函数都手动起一个名字。我便将无关紧要的中间函数自动命名。命名的方式是字符串 temp 加上一个数字。数字的作用是为了避免重复而产生冲突。这个数字是 Function 类的静态属性，由静态方法 getGlobal()每次将其递增 1。而 Function 的构造函数里会调用 getGlobal()静态方法，故每个 Function 的子类实例化的时候，都会得到一个新的函数名。

3.6.2 字符串化

我设计的 Function 基类规定了一个抽象方法 toString，它的所有可实例化的子类必须覆写这个方法。

要想将每种合成的递推函数字符串化，又需要将函数的每个组成部分的字符串化实现。故从操作符到基本变换和变换的类都要实现 toString 方法，然后一层一层地向底层调用。

Operation 的字符串化只需将它的两边操作数的 toString 结果用加减乘除的符号连接起来。当然，R_MINUS 和 R_DIV 要将两个操作数的字符串位置调换。

定义好 Operation 的字符串化函数后，BasicTrans 的字符串化只需调用操作符的 toString 方法。将需要变换的数（BasicTrans::toString()的参数）作为第一操作数，将 BasicTrans 中的函数名字符串化之后，作为第二参数，传递给操作符的字符串化函数。

而 Transform 的字符串化则将构成它的各个 BasicTrans 迭代地字符串化，即将 Transform 的参数传递给第一个 BasicTrans，然后，前一个 BasicTrans 的 toString 结果作为参数传递给下一个 BasicTrans，返回最后的结果即可。

函数名的字符串化，只是将函数名与“(n)”连在一起。其中的 n 可以替换为 n-1，n-2，由传入的参数决定。

具体种类的递推函数的字符串化（假设函数名为 f）：

constFunction：f(n)，连接上等号，再连接上常数值；

N_1Function：f(n)，连接上等号，再连接上 f(n-1)经过变换的字符串化，一个特殊值的定义，再换行将变换中各个基本变换使用到的函数依次字符串化（每行一个函数）；

N_2Function：f(n)，连接上等号，再连接上 f(n-2)经过变换的字符串化，两个特殊值的定义，再换行将变换中各个基本变换使用到的函数依次字符串化（每行一个函数）；

N_1_N_2Function：f(n)，连接上等号，再分别将 f(n-1)和 f(n-2)经过各自的变换，用操作符连接，再将整体进行变换，然后是两个特殊值的定义，再换行将变换中各个基本变换使用到的函数依次字符串化（每行一个函数）。

3.6.3 优先级·加括号。

要想尽量提高输出的结果的可读性，要贴近人类书写算式的习惯，根据优先级为子式加上适当的括号。这一过程是在字符串化函数中实现的，上一节介绍字符串化时略去了这一点。

我根据不同的运算符，定义了三个优先级：PLUS_MUNUS 对应于加法减法，优先级最低；MUL_DIV 对应于乘除法，优先级较高；SINGLE 是优先级最高的，像“f(n-1)”等独立的部分，就属于这个优先级，它在任何时候都不需要再加一层括号。

不光 Operation 有优先级的属性，BasicTrans 和 Transform 都有优先级的属性。它们的优先级属性取决于他们做的最后一次变换的操作符。这样定义之后，将操作数连同操作数的优先级一同交给 toString 函数，在 toString 函数中，如果左操作数的优先级低于操作符的优先级，那么就需要给左操作数加括号；如果右操作数的优先级低于或等于操作符的优先级，那么就需要给右操作数加括号。

3.7 本章小结

目标函数的定义是软件合成的至关重要的过程。目标语言的定义就像建筑的地基，如果地基打得稳定，建筑自然可以直插云霄；反之，后面付出再大的心血也只能盖成危楼一座，摇摇欲坠。目标语言定义得好，可以使搜索效率大大提升，用户会更快地得到他们想要的结果，最终合成的函数也会简洁明了。本章主要从数据结构方面介绍了基于输入输出样例的程序合成器的领域相关目标语言的设计思路与过程中各细节以及需要

注意的问题。

其中涉及的类有 Operation，BasicTrans，Transform，Function，constFunction，N_1Function，N_2Function，N_1_N_2Function。介绍了他们的继承关系以及调用关系。他们都包含求值的方法，字符串化的方法。函数值的计算用递推的方式完成。为了计算给定的特殊值之前的项的值，则需要定义逆运算，反向推出 f(n)与 f(n+1)和 f(n+2)的关系。字符串化时需要考虑优先级与加括号的问题。

第四章　自信度和搜索策略

4.1　规律·自信度

对于一个数列，计算机看到的是一个一个单独的数值，只知道它们之间能够比较和运算，不会看到它们之间的任何联系和规律。那么我就要用写程序的方式“教给”计算机什么样的形式算是人们眼中的“规律”，还要告诉计算机什么样的规律比较强，什么样的规律比较弱。自信度，就是量化地衡量规律的强弱的量度。下面将由“规律”的概念衍生出自信度的定义。

我“教给”计算机所谓的“规律”，可以简单地描述如下：

- 如果一个输入输出集合符合常数函数（即任何输入所对应的输出都是常数），那么这个输入输出集合是有规律的（可以将它认定为常数函数）；
- 如果一个输入输出集合的每一项都是它的前一项或前两项经过某变换得来的，若这个变换利用的，是有规律的输入输出集合所推出的递推函数，那么原来的输入输出集合也是有规律的(可以将它认定为与前一项或前两项有关的递推函数)。

显然，上面的定义是递归式的，找规律的过程自然也是由原始给定的输入输出集合出发，不断地将其做变换，产生新的集合，直到集合变为一个常数集合为止，这样递归下去，规律就被发现了。

将用户给定的输入输出集合表示为 S，S 中输入输出对的表示形式为$(\alpha \to \beta)$，输入输出集 S 是有规律的表示为$S \in S_f$，f为描述规律的递推函数（见 3.4 节），$[S]_f$表示将 3.3 节描述的变换作用在 S 集合的所以输出之后形成的新的集合，其中元素为$(\alpha \to [\beta])$，下标f表示该变换用到的递推函数。那么，上述规定可以形式化地表示为：

$$S \neq \emptyset \bigwedge S \subset \{(\alpha \to \beta) | \exists C \in \mathbb{R}\ \forall \alpha : \beta = C\} \implies S \in S_f$$

$$S' \in S_{f'} \bigwedge [S]_{f'} = S' \Rightarrow S \in S_f$$

像等差和等比数列就是后一项与前一项之间经过一次变换就转变为了常数数列，规律很快就会被发现；斐波那契数列也就是后一项减去前两项的和为一个常数数列（每一项都是 0）；而像等差和等比混合的数列等等更为复杂的规律，则需要转换多次，但最终也会变为常数数列。

举个简单的例子，平方函数$f(n) = n^2$，用户给出的输入输出样例为：

f(0) = 0
f(1) = 1
f(2) = 4
f(3) = 9
f(4) = 16

如果用后一项等于前一项加上某函数的变换，那么这个“某函数”的各项变换为：

g(1) = 1
g(2) = 3

g(3) = 5

g(4) = 7

再做一次这样的变换，新数列的各项将变为：

h(2) = 2

h(3) = 2

h(4) = 2

此时，“规律”（常数函数）出现！计算机即可将结果表示成：

f(n) = f(n-1) + g(n), f(0) = 0

g(n) = g(n-1) + h(n), g(1) = 1

h(n) = 2

这就是规律推导的简要过程。

如果和前一项做变换，输入输出集合中的元素就会少一个；和前两项做变换，输入输出集合中的元素就会少两个。那么，这样不断做下去，如果始终找不到规律，最终集合中的元素会变为 1 个，那也就一定能变为常数函数，找到规律。可是，经过不同的变换，找到的规律有好的，也有不好的。

通常的评判标准会认为：（1）经过变换越少，规律性越强（简单原则）；（2）推导规律的计算过程中并没有实际用到的输入输出对(但是推出的结果却能满足它们)越多，推出的规律性越强（由于经过了更多的额外验证）。

我用一种简单的方式定义了自信度，满足了上述评判标准：当寻找规律的过程进行到最终发现了常数数列时，将常数数列中的样例数目作为自信度。这样定义之后，自信度将具有这样的性质：发现规律的过程中，某函数的自信度等于它包含的变换用到的函数的自信度。对于同一个规律，给出的符合条件样例增多，会提升自信度。并且，要想维持同样的自信度等级，规律越复杂，经过的变换越多，则需要用户给出更多的样例。

4.2 搜索策略

在本问题中，用暴力的穷举搜索方式是不可行的。首先，递推函数的定义是递归式的，可以无穷下去，每种又有操作符和函数种类的分支变化，就算推到底，到达常数函数处，常数的个数也是无穷的，所以，搜索空间是无穷大的。其次，穷举搜索没有特殊规定搜索顺序，不会保证优先搜索用户最想要的答案，耗费了极大的代价之后，搜索到某个满足条件的结果后，还可能不是用户所想要的（不是自信度最高的一个）。

我实现的搜索算法利用到了用户给定的输入输出样例的启发性信息，采用了剪枝技术，以最大自信度为搜索目标，实现了逐步加深的深度优先搜索。

对于本问题，普通的深度优先搜索，搜索的最大深度会随着用户给定样例的规模增加，效率则以指数型下降。而这恰好与“对于同样问题，用户给出越多样例，越有助于问题的解决”的实际相矛盾。于是，我采取不论用户给的输入多么多，我都从前三个连续的输入开始做起（如果少于三个，那就取全部），由前 n 个（初始为 3）样例推出中间结论，由中间结论验证后面没有采取的样例；如果通过验证，则可以立即结束；如果验证失败，则将 n 递增 1，继续前面的步骤，直到采取了全部的输入。由于验证的效率远远高于推导的效率，对于一些规律比较简单，用户给定的样例比较多的问题，可以迅速得到答案。完成了深度逐步加深的深度优先搜索。同时，在这种模式下，还可以设定一个搜索深度的上限，当规律过于复杂时，就不在继续无尽的运算了，而是直接返回一个次优的结果。

4.3 搜索效率的进一步优化

剪枝法，搜索的过程中，推导出的中间结果如果已经达到自信度 c，继续推导是否有更优的结果时，如果发现搜索下去得到结果的自信度不可能大于 c，那么就可以直接终止这一分支的向下搜索。由于搜索的顺序是按照从简单的规律到复杂的规律，从加法到减法，再到乘法，所以，对于同样自信度的结果，先搜到的应该规律更简单，故按照前述方法中断搜索的时候仍能保证结果最优。

进一步的剪枝，将操作符中的 MINUS，DIV 和 R_DIV 在搜索中去掉了。其中 MINUS 操作符完全可以由 PLUS 替换，只要将 PLUS 后面的操作数去相反数即可（但是 R_MINUS 不可替代，很多规律中用到了 R_MINUS）；而除法在实践中起到的作用很小，也将它去掉了。这样，相当于搜索树的每一层的叶子节点数减小到最多为 3，大大降低了搜索时间。

4.4 规律找不到

规律过于复杂的情形是时有发生的，这或许是由于给出的样例就是杂乱无章的，或许是由于描述样例的规律超出我函数的定义范围。面对这种情形，我的做法是，当搜索深度到达 10 的时候（经实验，如果此时还未得到结果，那么往后在有限时间内很难返回出合理结果，还很可能因为堆栈空间不足而导致程序意外终止），直接返回一个多项式函数的递推公式。多项式函数即每次都用后一项减去前一项，构成新的数列继续用后一项减去前一项，直到得到常数数列。该方法即将搜索树退化为一条链表，效率很高。这样就保证了总能在有限时间内给出结果，而这个结果在绝大多数情况下（给定的数列本身有规律）能得到正确的结果，极少数找不到规律的情况下，仍可将其假设为多项式函数，给出次优结果。

4.5 本章小结

本章内容是我工作的核心内容以及重点和难点。首先由“规律”的定义引出了“自信度”的概念，它是衡量合成结果的好坏的度量。然后阐述了如何以寻找最大自信度的结果为目标的搜索。采用一定的启发式，并在剪枝后的逻辑树内进行逐步加深的深度优先搜索。最后阐述了如何对于实际问题做进一步优化，去掉冗余和不太可能出现的分支，并且阐述了对于及其复杂的问题，当规律找不到的时候，我的妥协做法。

第五章　多项式函数通项公式的计算

多项式函数的基本形式可以表示为：

$$y = c_0 + c_1 x + c_2 x^2 + \cdots + c_n x^n \quad (1)$$

其中，c_0到c_n为多项式系数。

5.1　待定系数法

用户给定的 m 个 x→y 这样的输入输出对儿，我们可以假设它们满足 n 次多项式函数（1），将x_0，y_0到x_m，y_m依次代入，可得方程组：

$$\begin{aligned} y_0 &= c_0 + c_1 x_0 + c_2 x_0^2 \ldots + c_n x_0^n \\ y_1 &= c_0 + c_1 x_1 + c_2 x_1^2 \ldots + c_n x_1^n \\ y_2 &= c_0 + c_1 x_2 + c_2 x_2^2 \ldots + c_n x_2^n \\ &\cdots\cdots \\ y_m &= c_0 + c_1 x_m + c_2 x_m^2 \ldots + c_n x_m^n \end{aligned}$$

这样，如果将c_0到c_n这 n+1 个系数当作未知数，那么，由高中代数知识可知，需要 n+1 个方程联立成方程组，即可得到唯一一组解。故 m=n+1，当用户给出的输入输出组数为 m 时，可先假设多项式函数为 n=m-1 次函数，如果解出最高的几次项系数为 0，那么可以相应地降低最终结果函数的次数，并增加自信度（自信度的概念请参看 4.1 节）。

使用线性代数的知识可以方便地用矩阵解出上述方程组的解。

下面，我们将各次项系数构成向量$[c_0 \quad c_1 \quad \cdots \quad c_n]$，将 x0 的各次方组成向量$\begin{bmatrix} 1 \\ x_0 \\ \vdots \\ x_0^n \end{bmatrix}$，

其中第一个 1 相当于x_0的零次方，代表常数项。如果将这两个向量做内积（相当于 1×n 的矩阵与 n×1 的矩阵做叉乘运算），我们将得到$c_0 + c_1 x_0 + c_2 x_0^2 \ldots + c_n x_0^n$，这正是$y_0$的值。那么往下，我们把$x_0$到$x_n$按照上面的方式组成的向量都并排写在一起，将构成一个矩阵：

$$\begin{bmatrix} 1 & 1 & \cdots & 1 \\ x_0 & x_1 & \cdots & x_n \\ \vdots & \vdots & \ddots & \vdots \\ x_0^n & x_1^n & \cdots & x_n^n \end{bmatrix}$$

这样，将系数向量与 X 矩阵相乘，则得到 Y 向量：

$$[c_0 \quad c_1 \quad \cdots \quad c_n] \times \begin{bmatrix} 1 & 1 & \cdots & 1 \\ x_0 & x_1 & \cdots & x_n \\ \vdots & \vdots & \ddots & \vdots \\ x_0^n & x_1^n & \cdots & x_n^n \end{bmatrix} = [y_0 \quad y_1 \quad \cdots \quad y_n]$$

简记为：$C \times X = Y$。那么，根据用户的输入 X→Y，矩阵 X 和向量 Y 是可以很容易构造出来的。如果能找到向量 C 满足方程$C \times X = Y$，即可解出多项式函数的通项公式的各次项系数，我们这一阶段的任务就完成了。

由线性代数的知识可知，如果 X 可逆，那么满足$C \times X = Y$方程的 C 应该为

$Y(X^TX)^{-1}X^T$。

简单证明如下：

$$CX = (Y(X^TX)^{-1}X^T)X = Y((X^TX)^{-1}(X^TX)) = YE = Y$$

因为此通解中 X 未必为方阵，故将其与自身转至相乘，得到方阵以便求逆矩阵。在上面的例子中，我们已经构造了方阵 X，故 C 可简单变为YX^{-1}，证明略。由于限制了x_0到x_n各项互不相等，X 各行的次方数也不同，故 X 各行一定线性无关，矩阵 X 的秩等于 n+1，故 X 一定可逆。下一节将讲述如何求矩阵的逆。

5.2 “全选主元高斯－约当(Gauss-Jordan)消元法”

使用“全选主元高斯－约当(Gauss-Jordan)消元法”计算矩阵的逆步骤如下：

首先，对于 k 从 0 到 n - 1 作如下几步：

从第 k 行、第 k 列开始的右下角子阵中选取绝对值最大的元素，并记住次元素所在的行号和列号，在通过行交换和列交换将它交换到主元素位置上。这一步称为全选主元。

m(k, k) = 1 / m(k, k)

m(k, j) = m(k, j) * m(k, k)，j = 0, 1, ..., n-1；j != k

m(i, j) = m(i, j) - m(i, k) * m(k, j)，i, j = 0, 1, ..., n-1；i, j != k

m(i, k) = -m(i, k) * m(k, k)，i = 0, 1, ..., n-1；i != k

最后，根据在全选主元过程中所记录的行、列交换的信息进行恢复，恢复的原则如下：在全选主元过程中，先交换的行（列）后进行恢复；原来的行（列）交换用列（行）交换来恢复。

5.3 通项引擎存在的问题

该方法能够计算出多项式函数的通项公式，表达简洁易懂。在所给输入数据的规模不是很大的情况下，能够快速得到结果。并且在得到函数后，计算函数某项的值的时候，不需要相邻的项的递归，而是在常数时间内直接得到结果，效率较高，在这点上比前面介绍的合成引擎占优势。但在其他方面也存在许多不足，说明如下。

对于各式各样的用户输入，虽然将它们全归结为多项式函数一定能得到解，但这样做并不能做到很符合用户的意图。如果用户想要合成 f(n) = (n+1)3-1 这样的函数，那么他输入：

f(0) = 0

f(1) = 7

f(2) = 26

f(3) = 63

f(4) = 124

通项引擎能够计算出结果 f(n) = 3n + 3n2 + n3。这是正确的并且符合要求的。可是假如用户想的并非是一个多项式函数，而是 f(n) = 2n + 3 这样的函数，他将输入：

f(0) = 4

f(1) = 5

f(2) = 7

f(3) = 11

f(4) = 19

此时，通项引擎接收到 5 组输入输出，即会将其视为 4 次函数，进而合成出

f(n) = 4 + 0.583333n + 0.458333n2 + (-0.0833333)n3 + 0.0416667n4 这样怪异的函数

来，虽然 0 到 4 项符合要求，但是后面几项推出的结果，f(5) = 34，f(6) = 60，f(7) = 102 这显然不会符合用户的意图。而合成引擎给出的答案为：

f(n) = f(n-1) + temp3760(n), f(0) = 4

temp3760(n) = temp3760(n-1) * temp3761(n), temp3760(1) = 1

temp3761(n) = 2

即 f(n)等于前一项 f(n-1)加上一个以 2 为公比的等比数列 temp3760(n)。虽然表现形式和 f(n) = 2n + 3 有差别，可是它们的本质是等价的，后几项 f(5) = 35，f(6) = 67，f(7) = 131 都是符合规律的。

由于使用浮点数进行运算，并且需要反复计算乘除法，数值精确度难免会有损失，往往会出现如将 3 显示为 2.99999999，2.5 显示为 2.499999999 等等比较接近的小数的情况。因此，我在运算后的结果的基础上，将其四舍五入取近似值，精确到 10 的-10 次方。这样可以部分地解决精度造成的误差的问题。

当给出的样例数目较多时会有很多无用的计算，如用户输入：

f(0) = 0

f(1) = 1

f(2) = 2

f(3) = 3

f(4) = 4

f(5) = 5

f(6) = 6

f(7) = 7

f(8) = 8

f(9) = 9

这样 10 对输入输出样例，则通项引擎会将其假设为 9 次函数，经过一次 10×10 的矩阵求逆和一次矩阵乘法后，推出n^2到n^9项系数都为 0，得出 f(n) = n 的结论。而合成引擎只会根据前 3 组输入输出推出规律后，拿后面的去验证，大大简化了运算。

合成引擎结果：

f(n) = f(n-1) + temp54(n), f(0) = 0

temp54(n) = 1

即每一项等于前一项加上一个常数函数 temp54(n) = 1。

由于存在以上所述各种问题，所以通项引擎只作为与合成引擎进行比较参考之用。

5.4 本章小结

通项公式的计算只能在假设目标函数为多项式函数的基础上，因此带来了很大的限制。但在此假设的基础上，目标函数的形式已经确定，可以使用待定系数法，利用线性代数的知识，通过矩阵来解线性方程，计算出每一项的系数。利用求通项的方法合成出的函数更易被人理解，并且在计算函数值时效率很高，但是它除了严重限制了函数的形式，影响了用户意图的理解之外，还由于会损失精度，并且在给出的输入输出样例数较多时做了许多无用的运算。由于这些缺陷和限制，通项引擎只是作为合成引擎的辅助，用来参考对比。

第六章　结论

6.1　工作总结

在了解了程序合成的研究现状和基本方法步骤之后，我尝试寻找自己的课题进行设计开发。经过反复斟酌，衡量工作意义、工作量、难度等因素后，最终确定了该课题方向。开题初步，进行了各种学习研究、论文的阅读以及相关工具摸索，其中微软研发的Z3[14]，pex4fun[15]都是软件工程与人工智能相结合的经典。

当着手开始工作时，首先要解决的一个巨大问题是目标语言的设计任务。目标语言定义得好，可以使搜索效率大大提升，用户会更快地得到他们想要的结果，最终合成的函数也会简洁明了。在确定了目标语言之后，还要有一个很好的衡量标准，这个衡量标准也将作为策略的关键目标。这个衡量标准就是函数的自信度。最后，在万事俱备的条件下，完成高效率的搜索是任务成败的关键。算法的设计与数据结构要互相配合，不断相互调整，最终才能完成现在的合成器版本。最后的阶段便是不断的试验和调优，使得合成器的效率进一步加快，解决问题的范围尽可能增广。

6.2　存在的问题

这个合成器只是个简易的实现，由 2.3 节可知，我对合成器做了很多限制，包括对合成的函数类型的限制，和函数逻辑复杂度的限制。这些限制也并不是那么轻松就能够打破的。搜索的空间可以在运算符的种类（乘方，取模，三角函数，微积分等等），相关项的个数（不只是最多前两项相关），输入参数的类型与个数等等维度上进行扩充。当搜索空间维度扩充了之后，搜索的时间将会是难以形容的爆炸式增长（比指数增长还要高）。

起初还想在合成器中加入自定义函数的规则。这将会更加扩广其适用度。比如，质数的数列，是没法用代数式表示其每一项的，它们之间也不存在直观的递推关系。想要合成与质数数列相关的函数，只能是提供用户自定义函数。但由于工程进度的原因，这个特性目前还没有加入。

6.3　未来展望

程序合成的为来发展空间是十分广阔的。目前还有许多困难问题亟待人们去解决，比如，循环体的合成等等。当然，真正实用的程序应包含各种复杂的流程控制：循环，跳转，条件判断等等。在小函数的合成中，当然也可以扩展它的复杂性，像输入输出的类型，函数的副作用（文件操作，数据库操作，绘制屏幕……）等等

我的合成器也有许多需要改进的方面。

在搜索技术上，可以考虑结合机器学习，将常用的程序片段（如变换、基本变换、操作符和常数以及他们的组合顺序）进行一个快速缓存，并按照其使用热度排序，保证每次最优先搜索最可能的片段。还可以提供用户的反馈机制，让机器能够智能学习，同一个问题在第二次提出，应该输出结果的准确性以及合成速度都会有所提升。

还有，从另一个方面着手，类似于曲线拟合，不必要将用户给出的每一个点都刻意地通过，可以取一个这种，将曲线尽量接近所有的点。智能地排除掉一些用户输错的噪

音点。这就更类似于模式发现，在海量的实际数据中，探寻出规律，将会更有实际价值。

如果将基于用例的合成器作为编程语言的一个部分，将会成为一个十分有趣的研究方向。终端用户想要完成某部分函数，无需关心函数具体如何实现，只需提供函数对应的输入输出样例，函数即可自动被合成出来，完成其功能，这样的更高级的编程语言势必会有广阔的发展前景。

参考文献

[1] Sumit Gulwani. Dimensions in program synthesis. In ACM Symposium on PPDP, 2010.
[2] D E Goldberg. Genetic Algorithms in Search, Optimization, and Machine Learning. Publisher: Addison-Wesley, 1989.
[3] Shachar Itzhaky, Sumit Gulwani and Neil Immerman. A Simple Inductive Synthesis Methodology and its Applications. In Proceedings of the ACM international conference, on OOPSLA 2010.
[4] Saurabh Srivastava, Sumit Gulwani, Swarat Chaudhuri and Jeffrey S. Foster. Path-based Inductive Synthesis for Program Inversion. In Proceedings of the 32nd ACM SIGPLAN conference, on PLDI 2011.
[5] Shachar Itzhaky, Sumit Gulwani, Neil Immerman and Mooly Sagiv. A Simple Inductive Synthesis Methodology and its Applications. In Proceedings of the ACM international conference on OOPSLA 2010
[6] Pieter Koopman and Rinus Plasmeijer. Synthesis of Functions Using Generic Programming. Lecture Notes in Computer Science, 2010, Volume 5812/2010, 25-49, DOI: 10.1007/978-3-642-11931-6_2
[7] N. Immerman. Upper and lower bounds for first order expressibility. J. Comput. Syst. Sci., 25(1):76–98, 1982
[8] 现代密码算法研究 郑世慧 北京邮电大学网络与交换技术国家重点实验室信息安全中心 北京 100876 《中兴通讯技术》2007 年 第 5 期
[9] IEEE Standard for Floating-Point Arithmetic (IEEE 754)
[10] Sumit Gulwani. Synthesis from Examples. In WAMBSE (Workshop on Advances in Model-Based Software Engineering) 2012
[11] 《费波那契数》 孙智宏 http://www.hytc.cn/xsjl/szh/lec5.pdf
[12] 《人工智能》 刘凤岐 2011 年 7 月第一版 机械工业出版社
[13] 《人工智能原理及应用》 罗兵 李华嵩 李敬民 2011 年 8 月第一版 机械工业出版社
[14] Leonardo de Moura and Nikolaj Bjørner. Z3 一个有效的 smt solver. http://research.microsoft.com/en-us/um/redmond/projects/z3/.
[15] Pex4fun. http://www.pex4fun.com/.
[16] S. Gulwani. Automating string processing in spreadsheets using input-output examples. In POPL, 2011.
[17] S. Gulwani, S. Jha, A. Tiwari, and R. Venkatesan. Synthesis of loop-free programs. In PLDI, 2011.
[18] S. Gulwani, V. A. Korthikanti, and A. Tiwari. Synthesizing geometry constructions. In PLDI, pages 50-61, 2011.
[19] S. Gulwani, S. Srivastava, and R. Venkatesan. Program analysis as constraint solving. In PLDI, pages 281–292, 2008.
[20] S. Srivastava, S. Gulwani, and J. S. Foster. From program verification to program synthesis. In POPL, pages 313–326, 2010.

附录 1 关键类的声明头文件

```
// Operation.h

enum OP
{
    NONE,
    PLUS,
    MINUS,
    R_MINUS,
    MUL,
    DIV,
    R_DIV,
};

enum PRIORITY
{
    PLUS_MUNUS = 1,
    MUL_DIV = 2,
    SINGLE = 3,
};

class Operation
{
public:
    Operation();
    Operation(OP op);
    // TODO: this left public??
    double calculate(double num1, double num2) const;
    double inverse1(double result, double num2) const; // known result and num2, return num1
    double inverse2(double result, double num1) const; // known result and num1, return num2
    PRIORITY getPriority() const;
    OP getOp() const;
    string toString(const string &num1, PRIORITY p1, const string &num2, PRIORITY p2) const;

private:
    PRIORITY p;
    OP op;
};
```

```
// GeneralTerm.h

#include "Function.h"

typedef vector<double> coef_t;

class GeneralTerm : public Function
{
public:
    GeneralTerm(const coef_t& coefficients);
    virtual ~GeneralTerm();
    virtual GeneralTerm* newInstance() const;
    virtual double funcVal(int n) const;
    virtual string toString() const;
    virtual string toString(const string &name);

private:
    coef_t coefficients; // (a, b, c, d, ...) in a + bn + cn^2 + dn^3 + ...
};

// Function.h

#include <vector>
#include "Operation.h"

class Function;

// [ f ] := f * g * h ...
class Transform
{
    friend class N_1Function;
    friend class N_2Function;
    friend class N_1_N_2Function;

public:
    Transform();
    void addBasicTrans(OP operation, const Function& function);
    double calcuate(double val, int n) const;
    double inverse(double val, int n) const;
    PRIORITY getPriority() const;
    string toString(const string &val, PRIORITY p, const string &n) const;

private:
    PRIORITY p;
```

```
    class BasicTrans
    {
        friend class Transform;
        friend class N_1Function;
        friend class N_2Function;
        friend class N_1_N_2Function;

    public:
        BasicTrans(OP op, const Function& function);
        BasicTrans(const BasicTrans &bt);
        ~BasicTrans();
        double calculate(double val, int n) const;
        double inverse(double val, int n) const;
        string toString(const string &val, PRIORITY p, const string &n) const;

    private:
        Operation operation;
        Function* function;
    };

    typedef vector<BasicTrans> transVector_t;

    transVector_t transforms;
};

////////////////////////////////////////////////////////////////////////

// abstract base class
class Function
{
public:
    Function();
    virtual ~Function();
    virtual Function* newInstance() const = 0;
    virtual double funcVal(int n) const = 0;
    virtual string nameToString(const string &n) const;
    virtual string toString() const = 0;
    virtual string toString(const string &name) = 0;

protected:
    string functionName;
private:
    static int number;
    static int getGlobal(); // number added one each time this is called
```

```
};

// f(n) := constVal
class constFunction : public Function
{
public:
    constFunction(double constVal);
    virtual ~constFunction();
    virtual constFunction* newInstance() const;
    virtual double funcVal(int n) const;
    virtual string toString() const;
    virtual string toString(const string &name);

private:
    double constVal;
};

// f(n) := [ f(n-1) ]
// f(input) := f(output)
class N_1Function : public Function
{
public:
    N_1Function(const Transform& trans_1, int input, double output);
    virtual ~N_1Function();
    virtual N_1Function* newInstance() const;
    virtual double funcVal(int n) const;
    virtual string toString() const;
    virtual string toString(const string &name);

private:
    int input;
    double output;
    Transform trans_1;
};

// f(n) := [ f(n-2) ]
// f(input) := f(output)
// f(input+1) := f(nextOutput)
class N_2Function : public Function
{
public:
    N_2Function(const Transform& trans_2, int input, double output, double nextOutput);
    virtual ~N_2Function();
    virtual N_2Function* newInstance() const;
```

```
    virtual double funcVal(int n) const;
    virtual string toString() const;
    virtual string toString(const string &name);

private:
    int input;
    double output;
    double nextOutput;
    Transform trans_2;
};

// f(n) := [ [ f(n-2) ] * [ f(n-1) ] ]
// f(input) := f(output)
// f(input+1) := f(nextOutput)
class N_1_N_2Function : public Function
{
public:
    N_1_N_2Function(const Transform& trans_1, const Transform& trans_2, const Transform&
trans_other, int input, double output, double nextOutput, OP op);
    virtual ~N_1_N_2Function();
    virtual N_1_N_2Function* newInstance() const;
    virtual double funcVal(int n) const;
    virtual string toString() const;
    virtual string toString(const string &name);

private:
    int input;
    double output;
    double nextOutput;
    Transform trans_1;
    Transform trans_2;
    Transform trans_other;
    Operation operation;
};

// Conjecture.h

#include <map>
#include "GeneralTerm.h"

typedef map<int, double> ioPair_t;
typedef vector<Transform> TransformVector_t;
typedef vector<OP> OPVector_t;
```

```
const double zero = 0;

class Conjecture
{
public:
    Conjecture();
    Function* generalTerm(const ioPair_t &ioPair, int &cd);
    Function* synthesise(const ioPair_t &ioPair, int &cd);

private:
    bool allSame(const ioPair_t &ioPair);
    ioPair_t diffMapN_1(const ioPair_t &ioPair, const Operation &op);
    ioPair_t diffMapN_2(const ioPair_t &ioPair, const Operation &op);
    ioPair_t diffMapN_1_N_2(const ioPair_t &ioPair, const Transform &trans_1, const
Transform &trans_2, const Operation &op, const Operation &newOp);

    ioPair_t extract(const ioPair_t &ioPair, unsigned &group_num);
    bool check(const ioPair_t &ioPair, Function* f);

    Function* synthesiseN_1(const ioPair_t &ioPair, const Operation &op, int &cd);
    Function* synthesiseN_2(const ioPair_t &ioPair, const Operation &op, int &cd);
    Function* synthesiseN_1_N_2(const ioPair_t &ioPair, Transform trans_1, Transform
trans_2, const Operation &op, const Operation &newOp, int &cd);

    Function* processN_1(const ioPair_t &ioPair, int &cd);
    Function* processN_2(const ioPair_t &ioPair, int &cd);
    Function* processN_1_N_2(const ioPair_t &ioPair, int &cd);

    Function* nonspeedup_synthesise(bool enableN_1_N_2, const ioPair_t &ioPair, int &cd);


    OPVector_t OPPri;
    TransformVector_t TransPri;
};
```

附录 2 数字推理实际题目举例

输入输出样例文件

Sample.csv

```
odd
        0         12
        1         34
        2         56
        3         78

quadratics
        0          2
        1          5
        2         10
        3         17
        4         26
        6         50

mixed
        1          1
        2          3
        3          3
        4          5
        5          7
        6          9
        7         13
        8         15

secondary
        1          4
        2          6
        3         10
        4         18
        5         34

ambiguous
        1          8
        2         12
        3         24
```

```
        4          60
        5          180

loop2
        0          2
        1          4
        2          2
        3          4
        4          2
        5          4

loop3
        0          1
        1          2
        2          3
        3          1
        4          2
        5          3

loop4
        0          2
        1          4
        2          3
        3          3
        4          2
        5          4
        6          3
        7          3

plus
        1          2
        2          4
        3          6
        4          10
        5          16

minus
        1          25
        2          16
        3          9
        4          7
        6          5

mul
        0          1
```

1	0
2	0
3	0
4	0

what

0	-1
1	0
2	7
3	26
4	63
5	124

how

0	2
1	1
2	5
3	15
4	53
5	187

miss

1	1
3	3
5	5
7	7
9	9

wrong

0	1000
1	1
2	4
3	9
4	16
5	25
6	36
7	49

通项引擎输出文件

GeneralTerm.csv

```
-- Confidence Degree: 3 --

The define of odd(n):
odd(n) = 12 + 22n

odd(0) = 12
odd(1) = 34
odd(2) = 56
odd(3) = 78
odd(4) = 100
odd(5) = 122
odd(6) = 144
odd(7) = 166
odd(8) = 188
odd(9) = 210
odd(10) = 232
odd(11) = 254

================================================================================

-- Confidence Degree: 4 --

The define of quadratics(n):
quadratics(n) = 2 + 2n + n^2

quadratics(0) = 2
quadratics(1) = 5
quadratics(2) = 10
quadratics(3) = 17
quadratics(4) = 26
quadratics(5) = 37
quadratics(6) = 50
quadratics(7) = 65
quadratics(8) = 82
quadratics(9) = 101
quadratics(10) = 122
quadratics(11) = 145
quadratics(12) = 170
quadratics(13) = 197
quadratics(14) = 226

================================================================================
```

```
-- Confidence Degree: 2 --

The define of mixed(n):
mixed(n) = -29 + 62.7n + (-46.7944)n^2 + 17n^3 + (-3.19444)n^4 + 0.3n^5 +
(-0.0111111)n^6

mixed(1) = 1
mixed(2) = 3
mixed(3) = 3
mixed(4) = 5
mixed(5) = 7
mixed(6) = 9
mixed(7) = 13
mixed(8) = 15
mixed(9) = -11
mixed(10) = -137
mixed(11) = -513
mixed(12) = -1407
mixed(13) = -3253
mixed(14) = -6707
mixed(15) = -12711
mixed(16) = -22565

================================================================================

-- Confidence Degree: 1 --

The define of secondary(n):
secondary(n) = 4 + (-1.5)n + 1.91667n^2 + (-0.5)n^3 + 0.0833333n^4

secondary(1) = 4
secondary(2) = 6
secondary(3) = 10
secondary(4) = 18
secondary(5) = 34
secondary(6) = 64
secondary(7) = 116
secondary(8) = 200
secondary(9) = 328
secondary(10) = 514
secondary(11) = 774
secondary(12) = 1126
secondary(13) = 1590
```

```
================================================================================

-- Confidence Degree: 1 --

The define of ambiguous(n):
ambiguous(n) = 40 + (-70.3333)n + 52.1667n^2 + (-15.6667)n^3 + 1.83333n^4

ambiguous(1) = 8
ambiguous(2) = 12
ambiguous(3) = 24
ambiguous(4) = 60
ambiguous(5) = 180
ambiguous(6) = 488
ambiguous(7) = 1132
ambiguous(8) = 2304
ambiguous(9) = 4240
ambiguous(10) = 7220
ambiguous(11) = 11568
ambiguous(12) = 17652
ambiguous(13) = 25884

================================================================================

-- Confidence Degree: 1 --

The define of loop2(n):
loop2(n) = 2 + 17.0667n + (-26.6667)n^2 + 14.6667n^3 + (-3.33333)n^4 + 0.266667n^5

loop2(0) = 2
loop2(1) = 4
loop2(2) = 2
loop2(3) = 4
loop2(4) = 2
loop2(5) = 4
loop2(6) = 66
loop2(7) = 324
loop2(8) = 1026
loop2(9) = 2564
loop2(10) = 5506
loop2(11) = 10628
loop2(12) = 18946
loop2(13) = 31748

================================================================================
```

```
-- Confidence Degree: 1 --

The define of loop3(n):
loop3(n) = 1 + (-5.85)n + 13.125n^2 + (-8)n^3 + 1.875n^4 + (-0.15)n^5

loop3(0) = 1
loop3(1) = 2
loop3(2) = 3
loop3(3) = 1
loop3(4) = 2
loop3(5) = 3
loop3(6) = -26
loop3(7) = -160
loop3(8) = -537
loop3(9) = -1376
loop3(10) = -2995
loop3(11) = -5829
loop3(12) = -10448
loop3(13) = -17575

================================================================================

-- Confidence Degree: 1 --

The define of loop4(n):
loop4(n) = 2 + 22.5429n + (-44.3056)n^2 + 35.0389n^3 + (-13.8889)n^4 + 2.90556n^5
+ (-0.305556)n^6 + 0.0126984n^7

loop4(0) = 2
loop4(1) = 4
loop4(2) = 3
loop4(3) = 3
loop4(4) = 2
loop4(5) = 4
loop4(6) = 3
loop4(7) = 3
loop4(8) = 138
loop4(9) = 956
loop4(10) = 3931
loop4(11) = 12267
loop4(12) = 32058
loop4(13) = 73868
loop4(14) = 154795
loop4(15) = 301083
```

```
================================================================================

-- Confidence Degree: 1 --

The define of plus(n):
plus(n) = -4 + 9.83333n + (-4.91667)n^2 + 1.16667n^3 + (-0.0833333)n^4

plus(1) = 2
plus(2) = 4
plus(3) = 6
plus(4) = 10
plus(5) = 16
plus(6) = 22
plus(7) = 24
plus(8) = 16
plus(9) = -10
plus(10) = -64
plus(11) = -158
plus(12) = -306
plus(13) = -524

================================================================================

-- Confidence Degree: 1 --

The define of minus(n):
minus(n) = 28 + 3.91667n + (-9.29167)n^2 + 2.58333n^3 + (-0.208333)n^4

minus(1) = 25
minus(2) = 16
minus(3) = 9
minus(4) = 7
minus(5) = 8
minus(6) = 5
minus(7) = -14
minus(8) = -66
minus(9) = -173
minus(10) = -362
minus(11) = -665
minus(12) = -1119
minus(13) = -1766
minus(14) = -2653

================================================================================
```

```
-- Confidence Degree: 1 --

The define of mul(n):
mul(n) = 1 + (-2.08333)n + 1.45833n^2 + (-0.416667)n^3 + 0.0416667n^4

mul(0) = 1
mul(1) = 0
mul(2) = 2e-010
mul(3) = 1.6e-009
mul(4) = 6e-009
mul(5) = 1
mul(6) = 5
mul(7) = 15
mul(8) = 35
mul(9) = 70
mul(10) = 126
mul(11) = 210
mul(12) = 330

================================================================================

-- Confidence Degree: 3 --

The define of what(n):
what(n) = -1 + n^3

what(0) = -1
what(1) = 0
what(2) = 7
what(3) = 26
what(4) = 63
what(5) = 124
what(6) = 215
what(7) = 342
what(8) = 511
what(9) = 728
what(10) = 999
what(11) = 1330
what(12) = 1727
what(13) = 2196

================================================================================

-- Confidence Degree: 1 --
```

```
The define of how(n):
how(n) = 2 + (-3.41667)n + 1.20833n^2 + 2.20833n^3 + (-1.20833)n^4 + 0.208333n^5

how(0) = 2
how(1) = 1
how(2) = 5
how(3) = 15
how(4) = 53
how(5) = 187
how(6) = 556
how(7) = 1395
how(8) = 3060
how(9) = 6053
how(10) = 11047
how(11) = 18911
how(12) = 30735
how(13) = 47855

================================================================================

-- Confidence Degree: 4 --

The define of miss(n):
miss(n) = n

miss(1) = 1
miss(2) = 2
miss(3) = 3
miss(4) = 4
miss(5) = 5
miss(6) = 6
miss(7) = 7
miss(8) = 8
miss(9) = 9
miss(10) = 10
miss(11) = 11
miss(12) = 12
miss(13) = 13
miss(14) = 14
miss(15) = 15
miss(16) = 16
miss(17) = 17

================================================================================
```

```
-- Confidence Degree: 1 --

The define of wrong(n):
wrong(n) = 1000 + (-2592.86)n + 2606.56n^2 + (-1343.06)n^3 + 388.889n^4 +
(-63.8889)n^5 + 5.55556n^6 + (-0.198413)n^7

wrong(0) = 1000
wrong(1) = 1
wrong(2) = 4
wrong(3) = 9
wrong(4) = 16
wrong(5) = 25
wrong(6) = 36
wrong(7) = 49
wrong(8) = -936
wrong(9) = -7919
wrong(10) = -35900
wrong(11) = -119879
wrong(12) = -329856
wrong(13) = -791831
wrong(14) = -1.7158e+006
wrong(15) = -3.43177e+006

================================================================================
```

合成引擎输出文件

Synthesis.csv

```
-- Confidence Degree: 3 --

The define of odd(n):
odd(n) = odd(n-1) + temp59(n)                    odd(0) = 12
temp59(n) = 22

odd(0) = 12
odd(1) = 34
odd(2) = 56
odd(3) = 78
odd(4) = 100
odd(5) = 122
odd(6) = 144
odd(7) = 166
odd(8) = 188
odd(9) = 210
odd(10) = 232
odd(11) = 254

================================================================================

-- Confidence Degree: 4 --

The define of quadratics(n):
quadratics(n) = quadratics(n-1) + temp7193(n)    quadratics(0) = 2
temp7193(n) = temp7193(n-1) + temp7194(n)        temp7193(1) = 3
temp7194(n) = 2

quadratics(0) = 2
quadratics(1) = 5
quadratics(2) = 10
quadratics(3) = 17
quadratics(4) = 26
quadratics(5) = 37
quadratics(6) = 50
quadratics(7) = 65
quadratics(8) = 82
quadratics(9) = 101
quadratics(10) = 122
quadratics(11) = 145
quadratics(12) = 170
quadratics(13) = 197
```

quadratics(14) = 226

==========

-- Confidence Degree: 4 --

The define of mixed(n):
mixed(n) = mixed(n-2) + temp147516(n)　　mixed(1) = 1　mixed(2) = 3

temp147516(n) = temp147516(n-2) + temp147517(n)　　temp147516(3) = 2　temp147516(4) = 2

temp147517(n) = 2

mixed(1) = 1
mixed(2) = 3
mixed(3) = 3
mixed(4) = 5
mixed(5) = 7
mixed(6) = 9
mixed(7) = 13
mixed(8) = 15
mixed(9) = 21
mixed(10) = 23
mixed(11) = 31
mixed(12) = 33
mixed(13) = 43
mixed(14) = 45
mixed(15) = 57
mixed(16) = 59

==========

-- Confidence Degree: 3 --

The define of secondary(n):
secondary(n) = secondary(n-1) + temp586905(n)　　secondary(1) = 4
temp586905(n) = temp586905(n-1) * temp586906(n)　　temp586905(2) = 2
temp586906(n) = 2

secondary(1) = 4
secondary(2) = 6
secondary(3) = 10
secondary(4) = 18
secondary(5) = 34
secondary(6) = 66

secondary(7) = 130
secondary(8) = 258
secondary(9) = 514
secondary(10) = 1026
secondary(11) = 2050
secondary(12) = 4098
secondary(13) = 8194

==========

-- Confidence Degree: 3 --

The define of ambiguous(n):
ambiguous(n) = ambiguous(n-1) * temp610995(n)　　ambiguous(1) = 8
temp610995(n) = temp610995(n-1) + temp610996(n)　　temp610995(2) = 1.5
temp610996(n) = 0.5

ambiguous(1) = 8
ambiguous(2) = 12
ambiguous(3) = 24
ambiguous(4) = 60
ambiguous(5) = 180
ambiguous(6) = 630
ambiguous(7) = 2520
ambiguous(8) = 11340
ambiguous(9) = 56700
ambiguous(10) = 311850
ambiguous(11) = 1.8711e+006
ambiguous(12) = 1.21622e+007
ambiguous(13) = 8.51351e+007

==========

-- Confidence Degree: 5 --

The define of loop2(n):
loop2(n) = temp644961(n) - loop2(n-1)　　loop2(0) = 2
temp644961(n) = 6

loop2(0) = 2
loop2(1) = 4
loop2(2) = 2
loop2(3) = 4
loop2(4) = 2
loop2(5) = 4

loop2(6) = 2
loop2(7) = 4
loop2(8) = 2
loop2(9) = 4
loop2(10) = 2
loop2(11) = 4
loop2(12) = 2
loop2(13) = 4

==========

-- Confidence Degree: 4 --

The define of loop3(n):
loop3(n) = temp652503(n) - (loop3(n-2) + loop3(n-1)) loop3(0) = 1 loop3(1) = 2
temp652503(n) = 6

loop3(0) = 1
loop3(1) = 2
loop3(2) = 3
loop3(3) = 1
loop3(4) = 2
loop3(5) = 3
loop3(6) = 1
loop3(7) = 2
loop3(8) = 3
loop3(9) = 1
loop3(10) = 2
loop3(11) = 3
loop3(12) = 1
loop3(13) = 2

==========

-- Confidence Degree: 5 --

The define of loop4(n):
loop4(n) = temp668576(n) - loop4(n-2) loop4(0) = 2 loop4(1) = 4
temp668576(n) = temp668577(n) - temp668576(n-1) temp668576(2) = 5
temp668577(n) = 12

loop4(0) = 2
loop4(1) = 4
loop4(2) = 3
loop4(3) = 3

```
loop4(4) = 2
loop4(5) = 4
loop4(6) = 3
loop4(7) = 3
loop4(8) = 2
loop4(9) = 4
loop4(10) = 3
loop4(11) = 3
loop4(12) = 2
loop4(13) = 4
loop4(14) = 3
loop4(15) = 3

================================================================================

-- Confidence Degree: 3 --

The define of plus(n):
plus(n) = plus(n-2) + plus(n-1)                    plus(1) = 2    plus(2) = 4

plus(1) = 2
plus(2) = 4
plus(3) = 6
plus(4) = 10
plus(5) = 16
plus(6) = 26
plus(7) = 42
plus(8) = 68
plus(9) = 110
plus(10) = 178
plus(11) = 288
plus(12) = 466
plus(13) = 754

================================================================================

-- Confidence Degree: 3 --

The define of minus(n):
minus(n) = minus(n-2) - minus(n-1)                 minus(1) = 25   minus(2) = 16

minus(1) = 25
minus(2) = 16
minus(3) = 9
minus(4) = 7
```

```
minus(5) = 2
minus(6) = 5
minus(7) = -3
minus(8) = 8
minus(9) = -11
minus(10) = 19
minus(11) = -30
minus(12) = 49
minus(13) = -79
minus(14) = 128

================================================================================

-- Confidence Degree: 3 --

The define of mul(n):
mul(n) = mul(n-2) * mul(n-1)                              mul(0) = 1     mul(1) = 0

mul(0) = 1
mul(1) = 0
mul(2) = 0
mul(3) = 0
mul(4) = 0
mul(5) = 0
mul(6) = 0
mul(7) = 0
mul(8) = 0
mul(9) = 0
mul(10) = 0
mul(11) = 0
mul(12) = 0

================================================================================

-- Confidence Degree: 3 --

The define of what(n):
what(n) = what(n-1) + temp795098(n)                       what(0) = -1
temp795098(n) = temp795098(n-1) + temp795099(n)           temp795098(1) = 1
temp795099(n) = temp795099(n-1) + temp795100(n)           temp795099(2) = 6
temp795100(n) = 6

what(0) = -1
what(1) = 0
what(2) = 7
```

what(3) = 26
what(4) = 63
what(5) = 124
what(6) = 215
what(7) = 342
what(8) = 511
what(9) = 728
what(10) = 999
what(11) = 1330
what(12) = 1727
what(13) = 2196

==========

-- Confidence Degree: 4 --

The define of how(n):
how(n) = how(n-2) * temp829774(n) + how(n-1) * temp829773(n) + temp829775(n)
how(0) = 2 how(1) = 1
temp829774(n) = 2
temp829773(n) = 3
temp829775(n) = -2

how(0) = 2
how(1) = 1
how(2) = 5
how(3) = 15
how(4) = 53
how(5) = 187
how(6) = 665
how(7) = 2367
how(8) = 8429
how(9) = 30019
how(10) = 106913
how(11) = 380775
how(12) = 1.35615e+006
how(13) = 4.83e+006

==========

-- Confidence Degree: 0 --

No result!

==========

```
-- Confidence Degree: 2 --

The define of wrong(n):
wrong(n) = wrong(n-2) * (wrong(n-1) - temp2406232(n))   wrong(0) =
+ temp2406233(n)                                       1000          wrong(1) = 1
temp2406232(n) = 1
temp2406233(n) = temp2406233(n-1) + temp2406234(n)     temp2406233(2) = 4
temp2406234(n) = temp2406234(n-1) + temp2406235(n)     temp2406234(3) = 2
temp2406235(n) = temp2406235(n-1) + temp2406236(n)     temp2406235(4) = -24
temp2406236(n) = temp2406236(n-1) + temp2406237(n)     temp2406236(5) = -48
temp2406237(n) = -24

wrong(0) = 1000
wrong(1) = 1
wrong(2) = 4
wrong(3) = 9
wrong(4) = 16
wrong(5) = 25
wrong(6) = 36
wrong(7) = 49
wrong(8) = 64
wrong(9) = 81
wrong(10) = 100
wrong(11) = 121
wrong(12) = 144
wrong(13) = 169
wrong(14) = 196
wrong(15) = 225

================================================================================
```

谢辞

从初入校门到即将离开，在这短暂的四年中我度过了人生最值得回味，最有意义的时光。大学中，我成长了许多，这一路上我受到了许多无私的帮助，热情的鼓励，坚定的支持。我想对所有帮助过我的老师、同学、亲人道一声谢。

首先，我要感谢我的导师赵建军教授。他为人和善，面对同学们的时候他总是笑容可掬，让我感到十分亲切。在治学上，把持着严谨的科研态度和一丝不苟的作风的他，让我深深地感动。赵老师并不过分地限制我们的兴趣方向，为我们建立了自由轻松科研的氛围，他坚持“要做自己感觉有趣的事”的原则，让我们喜欢上自己的工作。在赵老师主张开展的软件理论与实践课后讨论班中，我学习到了领域前沿的技术，也更快地了解学长们所研究的方向。从毕业设计题目的选择，到中期答辩，再到论文的撰写与修改，整个毕业设计的工程项目都是在赵老师悉心指导下完成的。在此我要对赵老师表示深深的谢意并致以我最崇高的敬意！

我还要感谢我的同学和伙伴们，他们在我的论文撰写中对我给予了许多帮助。感谢王彭帮我审阅修改论文中的错别字，提出了许多宝贵意见。他也是我工程做好后的自愿试用者。作为室友的章海峰、梅诚和金辉，感谢你们大学四年的日夜相伴，和你们相处让我学到了很多。感谢同一实验室的学长章程、胡翔和孙强，在开题阶段提供了一些有趣的方向供我参考。我还要感谢我的挚友谢涛、许海南和周伶俐，感谢你们在我的毕设中提出的宝贵意见以及对我的督促。

我还要感谢我亲爱的父母和亲友们，你们在我背后默默的支持是我最大的动力。在此我真诚地感谢你们！

A PROGRAM SYNTHESIZER BASED ON NUMERICAL INPUT-OUTPUT EXAMPLES

Program synthesis, a brand new field of software, which was classified into automatic programing, nowadays has again been paid great attention to. The idea is to present human-understandable task specifications to a computer system and receive in return a program that is known to meet the specifications submitted.

Although it is very, very hard to fully appreciate such an idea, it still draws great attention of the software researches. This problem is usually associated with mathematical reasoning, verification and so on. Moreover program synthesis is even harder because of no ready code to process, but to produce it.

According to Doctor Sumit Gulwani from the Redmond Lab of Microsoft Research Group, a program synthesizer is typically characterized by three key dimensions: the kind of constraints that it accepts as expression of user intent, the space of programs over which it searches, and the search technique it employs. (i) The user intent can be expressed in the form of logical relations between inputs and outputs, input-output examples, demonstrations, natural language, and inefficient or related programs. (ii) The search space can be over imperative or functional programs (with possible restrictions on the control structure or the operator set), or over restricted models of computations such as regular/context-free grammars/transducers, or succinct logical representations. (iii) The search technique can be based on exhaustive search, version space algebras, machine learning techniques (such as belief propagation or genetic programming), or logical reasoning techniques.

The work I did is a simple PROGRAM SYNTHESIZER BASED ON NUMERICAL INPUT-OUTPUT EXAMPLES by the language of C++. It contains two engines, one of which is a synthesis engine using the program synthesis technique, which can discover the regular pattern between the terms; the other is especially used for calculate the General Term of the polynomial expression. The former is the focus of my work, and the latter is only for reference and comparison. For the first dimension, it belongs to input-output examples; for the second dimension, I produced a rule of the target language, which constrains the complexity and the search space; for the third dimension, a kind of heuristic search technique was used, as well as an algorithm of gradually deeper depth-first search. When applied to the numerical reasoning problems of the Civil Service Examinations, my synthesizer works quite well and is able to solve most of them.

Known how the situation of the research of this field and the primary methods to solve this kind of problems, I tried very hard to find a proper issue to deal with, which will not be too hard nor too useless. Considering the importance of the work, the workload and the difficulty, I chose to put my back into it, heart and soul.

After read a lot about the program synthesis and the synthesizer based on input-output examples, I still tried a lot of tools that might help. Afterwards, the most challenging task when I

started the design work is to define a well-defined target language. This is a puzzling job, which decides success of failure. A good target language means a smaller and easier search procedure, and a friendly, easily understood synthesis result. A worse target language means however great effort to take afterwards only turning into rubbish. I used the recursive define to describe the arithmetic relations between the current term and the one or two terms before. For the reason of consistency, I regard all the constant value as a function as well. This helps a lot to the heuristic search process.

Like any other philosophy problems, the solving procedure will never ends at a best answer. So, how to evaluate the result given by the synthesizer? It seems that the question is asked after the synthesizer has finished its work. However, in fact, I have no way to accomplish a synthesizer until I figure this question out. The evaluation system determines how to find the right result, and which is more welcomed by the users. Without the directing of the evaluation system, the search procedure has no goal to run towards, and the possibility of optimizing the search procedure is no more than zero.

Thanks to the definition of the target language, I found a simple but reasonable method to value the strength of the regular patterns. I name it "confidence degree". It is the measurement of the quality of each synthesis result. The confidence degree is higher (which means the synthesizer is more confident to the result is provides) if there are more input-output examples (exclude the ones practically used for synthesis procedure) coinciding with the result. The main work, search procedure, is simply an algorithm to find a result with the highest confidence degree, which is combined from the defined target language.

What is a regulation? It's hard to answer for us humans when suddenly asked. For the computer, an array is just a series of numbers, which can calculate and compare with each other. So how to make the computer know the regulations and patterns is what I should do. We human see the regulation of an array only from the appearance. The arithmetic progression seems to increase step by step. Nevertheless the computer can only do a sort of transforms to the array, and turn to find the differences between the neighboring terms are all equal.

In order to let the computer know what is the regulation. I make the rule to define what a regular pattern is.

When each of the elements in an array equals, the array is in order (it is a constant array).

When the very terms transforms into its neighboring terms, and the transform uses the array in order, the array of the very terms is in order.

Apparently, the definition of the regular pattern is recursive. Therefore the procedure to find a regular pattern in an array for the computer is recursive as well. I make the computer do transforms onto the terms, producing a new array, then find patterns from the new array. This is repeated until a constant array shows up and congratulations, the pattern is discovered! Of course the pattern is always discovered, because each time the transform done onto the neighboring terms, the new array's size is one or two less than the original array, and the array with only one element is trivially a constant array, which means the regular pattern is discovered. But in that case, the confidence degree may be very low. Actually, the degree is defined to be equal to the final constant array size.

The final search strategy is formed during the testing and optimizing process. Firstly brute-force search was used. Within the expected, when trying to solve a little bit complex problems, the speed was rather unacceptable. And with the growth of the scale of the input-output

examples given, the running time expends exponentially. With six or more input-output examples given, the synthesizer didn't respond at all. Thus I was forced to do the optimization. I reduced the total number of the operations used, and also the complexity of the transforms. After pruning the searching tree, the speed raises a lot. In order to solve the absurd problem of that the synthesizer slows down with more provided examples, I choose the algorithm of gradually deeper depth-first search. At first, set n=3, and then pass the first n examples to the synthesizer, and verify the other example, if pass, the best result is found; if not pass, increase n and repeat the procedure. This method is quite good for the simple pattern with many examples given. Moreover, when the pattern is too difficult to discover, the deepest depth can be constrained.

The general term engine is especially for deducing the general term of the polynomial expression. Since the final form of the result is very explicit, the only work is to solve the system of linear homogeneous equations to figure out the undetermined coefficients. The different power of the inputs makes a matrix X, the outputs maks a vector Y and the coefficients makes the vector A, then the matrix equation AX=Y is established. The solution of A contains the coefficients that we want for the final result. Actually A is just YX^{-1} if X is reversible, and so the fact is.

Still, there are a lot of unsolved problems in my project. The constraints I made is a bit strong and they are not so easy to be removed. And in the future, program synthesis may be used in practise more and more when the functions grow more powerful. If my synthesizer is used as a part of a new high-level language, it will open a brand new door of this field. It let the end-users care for no algorithm and the low-level grammars. They just give the input-output examples and easily get the synthesized function they want.

Since the birth of the first electronic computer, computer programs have helped us accomplished all kinds of hard and difficult tasks, leaving us only convenience. Nevertheless, the coding work always puzzles many programmers. We hope that computers could help us generate programs automatically, setting us free from the boring coding work. Program synthesis may probably help us realize that dream. It also does very good job in the discovery of new algorithm, auto-solving, intelligent torturing and so on. Moreover basing on the input-output examples is the most natural and simplest form to present user intents. I hope my work of making the synthesizer may fundamentally contribute a little bit to this field of research.