* 画 deterministic PDA，用PDA去完成判断 string 是否 valid 的过程
* Pumping Lemma for CFG
* left factoring / left recursive
* 判断可不可以 RDP

  ↙

  Follow First

* Simple assignment statement
* Specification

  ↓ 找P ↘ 找I ↘ 先找I再找P

Pushdown Automata (PDA) → 借由读取一个容量无限的 stack，扩充一个能做 ε - 转移的非确定有限状态机

→ Machine Model of language recognition that allows a similar correspondence to be established with the class of all context - free language. → 依据 Context - free language 建立一个 similar correspondence

→ 和 Finite State machine (described by State - transition diagram 很像)

→ 有限状态自动机：表示有限个状态以及在这些状态之间的转移和动作等行为的数学计算模型

区别：PDA 有 "Push down store" or stack of unbounded depth

→ Notation: $c, d \mapsto w$: Current input string token is c and the token at the top of the stack is d

c: vocabulary token

d: stack token, be poped

w: String of stack tokens

上面是目前的状态，下面是作用

Read the input token and replace the top of the stack by the components of w, with the left most component of w becoming the new top of the stack

→ c 可以 = ε : state change without reading an input token

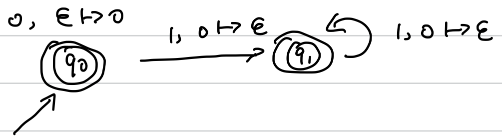
d 可以 = ε : state chang, a stack token string is pushed into the stack, but without poping out anything

→ 甚至，可以在没有东西被 pop 掉的情况下添加东西，初始 stack 可以为空

→ An input string is accepted if there is a directed path from the starting state to an accepting state whose sequence of input tokens spells out the string and the stack

& Acceptance at the end of input, stack and input is empty

Example: $\{0^i 1^i \mid i \geq 0\}$ over $\{0, 1\}$:



0, ε ↦ 0

指的是如果 input 是 0，那么把 ε 从 stack pop 掉（即什么都不 pop），然后把 0 加入 stack，这里的加入，指的 replace!!! 不是 pop! 比如，c, d ↦ w，并不是[先把 d 给 pop 掉，再把 w 给 push 进去]，而是 [把 d 给换成 w]，区别在于，stack 是 FIFO，如果是第一种方法的话，w 会被放在 stack 的最后

Example string: 000 111 → 根据具体的图以及上面的 design

State	input	Read	stack
0	000 111	/	ε
1	00 111	0	0
2	0 111	0	00
3	111	0	000
4	11	1	00
5	1	1	0
6	ε	1	ε

→ Acceptance

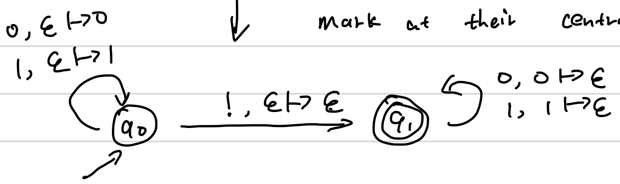
如果到最后 input 和 stack 均只剩下 ε，则是 accepted

注意！上课笔记中 只有三个 column：State, Remain input, Stack

Example 2

$\{w!w^R \mid w \in \{0,1\}^*\}$

→ language of centered palindromes; these palindromes have an exclamation mark at their central position



// 确保！两边完全 symmetry

Example string: 110!011

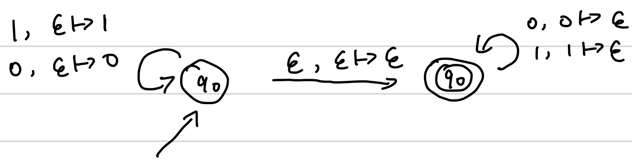
State	input	Read	stack
0	110!011	1	ε
1	10!011	1	1
2	0!011	1	11
3	!011	0	011
4	011	!	011
5	11	0	11
6	1	1	1
7	ε	1	ε

→ Acceptance

Example 3

$\{ww^R \mid w \in \{0,1\}^*\}$

→ All even length palindromes over $\{0,1\}$



Example String: 110011

State	Remain input	stack
0	110011	ε
1	10011	1
2	0011	11
3	011	011
4	11	11
5	1	1
6	ε	ε

→ Accepted

PDA & Context free language

* For any context-free language, we may construct a PDA that recognizes the language.

↘ 首先，对于一个 context free grammar 来说：

打个比方：

$\{a^{3i} b^k c^{2i+3} \mid i \geq 1, \ k \geq 1\}$

nonterminal symbol ← $S \rightarrow \{a^3 S c^2 \mid a^3 X c^2 c^3\}$ → production

$X \rightarrow \{bx \mid b\}$ → terminal symbol

↘

CFG ↓ PDA ↓ Non deterministic

在画 according 的 PDA 时，只有两个 state：q_0 (initial state) 和 q_1 (accepting state)

$\rightarrow q_0 \xrightarrow{\epsilon, \epsilon \mapsto S} q_1$ → 首先肯定得有个这个 transition，才让接下来 PDA 可以 recognize

↓

然后，需要有 for each possible input token (terminal symble) c, there is to be a transition from q_1 to itself : $c, c \mapsto \epsilon$

$\rightarrow q_0 \xrightarrow{\epsilon, \epsilon \mapsto S} q_1$ ↺ $a, a \mapsto \epsilon$; $b, b \mapsto \epsilon$; $c, c \mapsto \epsilon$

↓

最后，For each production N, there is to be a transition from q_1 to itself of the form

$\epsilon, N \mapsto$ the production

↓

$\rightarrow q_0 \xrightarrow{\epsilon, \epsilon \mapsto S} q_1$ ↺ $a, a \mapsto \epsilon$; $b, b \mapsto \epsilon$; $c, c \mapsto \epsilon$

↺ $\epsilon, S \mapsto a^3 S c^2$

$\epsilon, S \mapsto a^3 x c^2 c^3$

$\epsilon, x \mapsto b x$

$\epsilon, x \mapsto b$

Example string: $a^3 b c^5$ = a aa b ccccc

State	Remain input	Stack
0	aaa b ccccc	ϵ
1	aaa bccccc	S
2、	aaa b ccccc	aaa X cc ccc
3、	aaa b ccccc	aaa x ccc cc
⋮	#消 a	
4、	b ccc cc	X ccc cc
5、	b ccc cc	b ccc cc
⋮	#消 b, c	
6、	ϵ	ϵ → accept

例题:

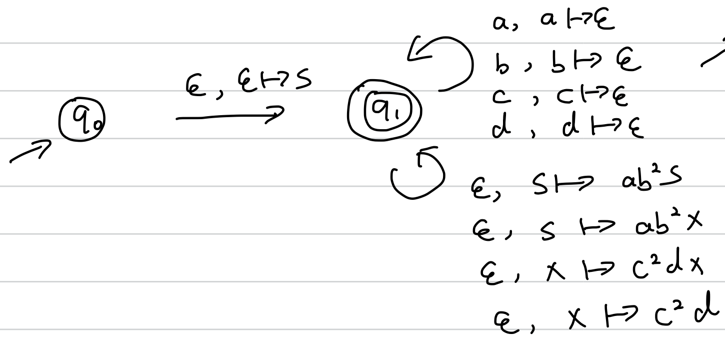
Assignment 5 2019

Design a deterministic pushdown automaton that recognizes the language

$L = \{a^k b^{2k} c^{2i} d^i \mid i \geq 1,\ k \geq 1\}$

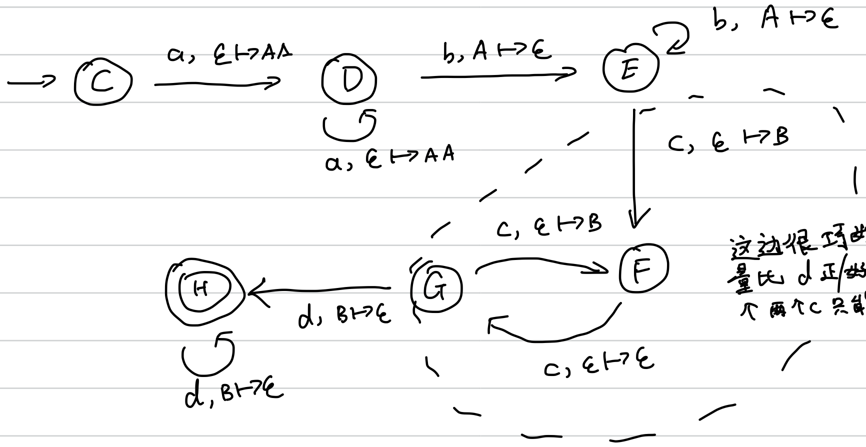
$S \rightarrow ab^2S \mid ab^2X$

$X \rightarrow c^2dX \mid c^2d$



→ Non deterministic

Deterministic:



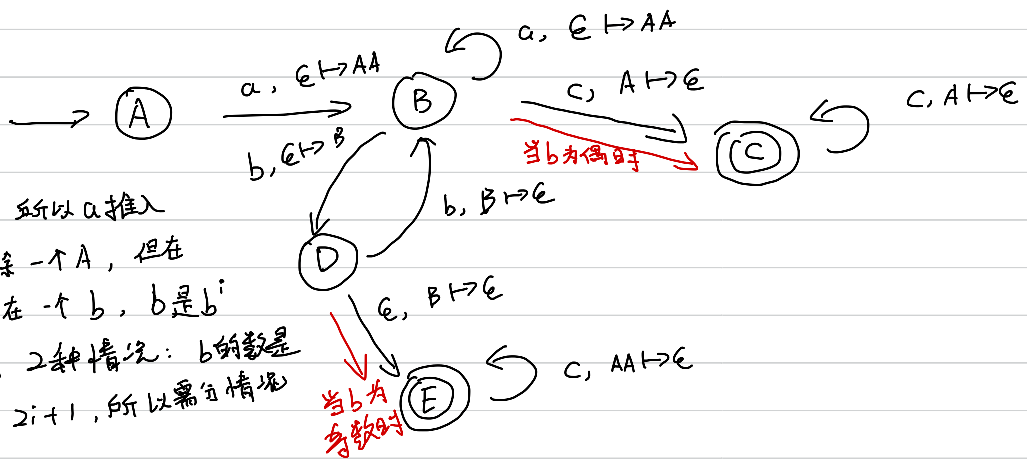
比较关键的就是：需确保到最后都被消掉

这边很巧妙，因为c的数量比d正好多一倍，所以~~一个~~两个c只能插一个B

Assignment 5, 2020

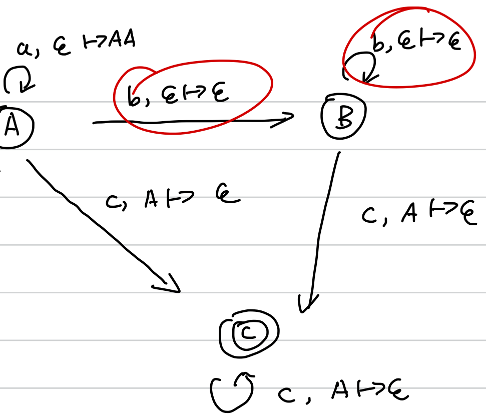
Design a deterministic pushdown automaton that recognizes the language

$L = \{a^k b^i c^{2k} \mid i \geq 0,\ k \geq 1\}$



因为 $c = 2a$，所以 a 推入 AA，每个 c 消除一个 A，但在它们中间，存在一个 b，b 是 b^i，b 的数量有 2 种情况：b 的数量为 $2i$ 或 $2i+1$，所以需分情况讨论

Solution :



... 答案直接为 b, $\epsilon \mapsto \epsilon$ ...

Pumping Lemma For Context-Free Languages CFG

→ Suppose that a language L is CFG.

There exist a constant p such that any string $s \in L$ of length at least p

s can be written as $s = uvwxy$

where:

- p_1) $v \neq \epsilon$ or $x \neq \epsilon$
- p_2) $|vwx| \leq p$
- p_3) $uv^iwx^iy \in L$ for all $i \geq 0$

Example: Prove $A = \{a^i b^i c^i \mid i \geq 0\}$ is not context free

Assume that A is context free

$\therefore$ Let p be the constant

Let s be $s = a^p b^p c^p$

$\because |s| > p$

$\therefore$ By Pumping Lemma

s can be written as $s = uvwxy$

$\because$ By Pumping Lemma

$uv^2wx^2y \in L$

However:

if v, x contain more than one variable: number of a, b, c is inequal

$uv^2wx^2y \notin L$

if v, x contain only one variable: number of a, b, c is inequal

$uv^2wx^2y \notin L$

$\therefore uv^2wx^2y \notin L$

$\therefore$ P.L for L doesn't hold

$\therefore L$ is not context free

例题:

Are the following languages context free or non-context free?

(a) $A = \{a^i b^k c^k \mid i \geq k \geq 1\}$

Assume A is context free

Let p be the constant

Let $s = a^p b^p c^p$ be the string

$\because |abc| > p$

$\therefore s = a^p b^p c^p$ can be written as $uvwxy$

$\because |vwx| < p$

$\therefore vwx$ can only be b or ab

Consider uv^2wx^2y

when vwx is ab, the number of $b \neq |p|$

$\therefore uv^2wx^2y \notin L$

when vwx is b, the number of $b \neq |p|$

$\therefore uv^2wx^2y \notin L$ #或者可以说如果泵的话 $|b| > |a|$

$\therefore uv^2wx^2y \notin L$

$\therefore L$ is not CFG

2. (a) We show that A is not context-free. Assume that A is context-free and let p be the constant given by the pumping lemma. We consider the string $s = a^p b^b c^p \in A$.

Since $|s| \geq p$, we can write $s = uvwxy$ where the parts u, v, w, x, y satisfy the conditions of the pumping lemma.

i. If v or x contains occurrences of two or more distinct symbols, then uv^2wx^2y contains symbols in "incorrect order" and is not in $a^*b^*c^*$, and hence uv^2wx^2y

第一个先只数量上的contradict

is not in A.

In the following cases, we can then assume that v and x each contains occurrences of at most one symbol.

ii. If v and x do not contain any occurrences of symbol a, then uv^2wx^2y contains more b's than a's or more c's than a's. This means that uv^2wx^2y is not in A.

iii. The remaining possibility is that v or x (or both) contains occurrences of the symbol a. Since v and x each contain only one type of symbol, this means that vx has no b's or vx has no c's. Thus $uv^0wx^0y = uwy$ has fewer a's than b's or fewer a's than c's. Thus, uv^0wx^0y is not in A.

只定义上的 contradict

We have shown that all cases lead to a contradiction. This implies that A is not context-free.

(b) $B = \{a^{2i} b^{k+1} c^{3i} d^{2m} \mid i \geq 1,\ k \geq 1,\ m \geq 1\}$

$S \to X \mid Y$

$X \to a^2 X c^3 \mid a^2 Z c^3$

$Z \to bbZ \mid b$

$Y \to d^2 Y \mid d^2$

Assignment 5, 2020

(a) $A = \{a^i b^k c^i d^i \mid i \geq 1, k \geq 1\}$

Assume A is CFG

Let p be constant

Let $s = a^p b^k c^p d^p$

$\because |s| > p$

$\therefore s$ can be written as $uvwxy$

Consider uv^iwx^iy:

if v, x contain more than one elem

then uv^2wx^2y is not in A

So v, x can only contain at most one elem

Suppose v, x contain b, then $uv^0wx^0y = uwy$ which don't have b $\Rightarrow uv^0wx^0y \notin A$ #用b的数量因为定义时，b的指数与a, c, d不一样

Suppose v, x each contain at most one of a, c, d,

then uv^2wx^2y makes the occurences of a, c, d not equal

$\therefore uv^2wx^2y$ not CFG

$\therefore A$ is not CFG

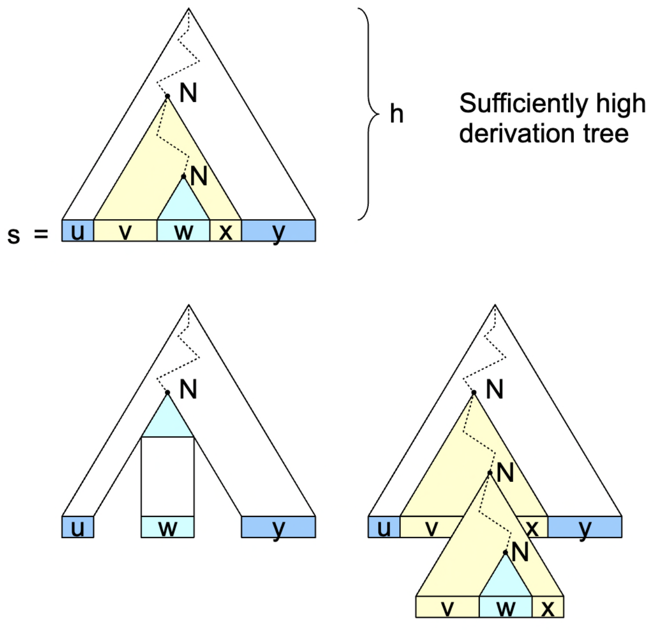
(2) $B = \{ a^{2i} b^{k} c^{3k} d^{i} \mid i \geq 0, k \geq 1 \}$

$S \to a^2 S d^3 \mid a^2 X d^3$

$X \to b c^3 X \mid b c^3$

↳ Pumping Lemma for CFL

↳ 对于一个符合某CFL的字符，可以按照规则重复pump其中的若干部分，得到的新字符串依然属于这个字符



也就是说，在一个足够长的字符串里面，根据鸽子洞原理，肯定会有non terminal是重复的，那么如果向下继续延展这些non terminal，得出的字符串依然属于这个cfl中

Generating uv wx y^0　　Generating uv wx y^2

一些证明的例子：

- $L = \{a^n b^n c^n | n \geq 1\}$ 或 $L = \{a^i b^i c^i | i \geq 0\}$ 或 $L = \{a^i b^i c^i | i \geq 2\}$，换句话说，*L* 就是包含 $a^* b^* c^*$ 所有字串且 *a*、*b*、*c* 三者数目相同的语言。
  - 令 *n* 为**泵引理**常数，$w = a^n b^n c^n$ 属于 *L*，*w* = *uvxyz*，而 |*vxy*| ⩽ *n*，|*vy*| ⩾ 1，则 *vxy* 不可能同时包含 *a* 与 *c*。
    1. 当 *vxy* 不包含 *a* 时，*vy* 只可能包含 *b* 或 *c*，则 *uxz* 包含 *n* 个 *a* 及不到 *n* 个的 *b* 或 *c*，使得 *uxz* 不属于 *L*。
    2. 当 *vxy* 不包含 *c* 时，*uxz* 会包含 *n* 个 *c* 及不到 *n* 个的 *a* 或 *b*，使得 *uxz* 不属于 *L*。
  - 因此，无论是上述何种状况，*L* 都不会是上下文无关语言。
- $L = \{a^i b^j | j = i^2\}$

  $uv^i xy^i z$
  - 令 *n* 为**泵引理**常数，$w = a^n b^{n^2}$，*w* = *uvxyz*，而 |*vxy*| ⩽ *n*，|*vy*| ⩾ 1
    1. 若 *vxy* 只包含 *a*，则 *uxz* 会包含不到 *n* 个 *a* 及 n^2 个 *b*，不属于 *L*；
    2. 若 *vxy* 只包含 *b*，则 *uxz* 会包含 *n* 个 *a* 及不到 n^2 个 *b*，不属于 *L*；
    3. 若 *vxy* 里有 *a* 也有 *b*，
       1. 若 *v* 或 *y* 包含 *a* 与 *b*，uv^2xy^2z 不在 $\{a^i b^j\}$ 里；
       2. 若 *v* 只包含 *l* 个 *a*，且 *y* 只包含 *m* 个 *b*，$uv^{1+k}xy^{1+k}z$ 会包含 *n* + *lk* 个 *a* 与 $n^2 + mk$ 个 *b*，由于两者都是线性成长，不可能永远满足 $\{a^i b^j | j = i^2\}$ 的条件，不属于 *L*。
  - 因此，无论是上述何种状况，*L* 都不会是上下文无关语言。
- $L = \{ww | w \in \{0,1\}^*\}$
  - 令 *n* 为**泵引理**常数，$w = 0^n 1^n 0^n 1^n$ 属于 *L*，*w* = *uvxyz*，而 |*vxy*| ⩽ *n*，则 *vxy* 必然为 $0^i 1^j$ 或$1^j 0^i$形式（此处有$i, j \in \mathbb{N}, i + j \neq 0$）。即 *vxy*无法同时包含前后两组0，也无法同时包含前后两组1。将*uvxyz*转变成*uxz*必然导致前后两组0或两组1的数目产生差异。使得*uxz*不再满足*ww*形式。亦即*uxz*不属于*L*。
  - 因此，*L* 都不会是上下文无关语言。
- $L = \{x^i y^j z^k | i \neq j \; and \; j \neq k\}$
- $L = \{b^n a^{2n} b^n | n \geq 0\}$
- $L = \{a^n b^m c^m | n, m \geq 0\}$

Parsing

→ Parsing is the process of determing whether a string of tokens can be generated by a grammar

- → Brute Force：就是系统地生成这个grammar中能生成的所有string，然后和这个string去比对
- → 非考点：用dynammic programming的一个算法可以完成Parsing in $O(n^3)$
- → Deterministic context-free language can be parsed in linear time - sophisticated parsing table construction
- → Recursive Descent (Simple, relatively efficient approach)
  - ↙ Basic idea：

并不要求知道具体如何定义

- * A recognizing function is coded for each nonterminal symbol in the grammar
  每一个grammar的nonterminal都有一个function
- * The current input token is used to decide which of several possible productions is the appropriate one to use

Example： set of balanced strings $\{0^i 1^i \mid i \geq 0\}$

grammar： <balanced> → 0 <balanced> 1 | ε

idea： 就是该如何定义function：

if next token is 0：

use <balanced> → 0 <balanced> 1 # 有多少个0就有多少个1

if next token is 1 or End of string (EOS)

use <balanced> → ε # 所以在读到1时直接ε

Program 11.1

```
typedef enum { ZERO, ONE, EOS } vocab;

vocab gettoken(void) { ... }

vocab t;

void MustBe(vocab ThisToken)
{ ASSERT( ThisToken != EOS )
  /* verifies and then updates current token t */
  if (t != ThisToken)
  { printf("String not accepted.\n"); exit(0); }
  t = gettoken();
}

void Balanced(void)
{ switch (t)
  { case ONE:
    case EOS: /* <empty> */
      break;
    default: /* 0 <balanced> 1 */
      MustBe(ZERO);
      Balanced();
      MustBe(ONE);
  }
}

int main(void)
{ t = gettoken();
  Balanced();
  if (t != EOS) printf("String not accepted.\n");
  return 0;
}
```

RDP (Recursive descent Parsing) 存在的问题：

① 因为是根据token选择 production，如果一个token实际指向了多个production，Parser不知道该选哪个

⇒ token 0 both begins a production and appears after <PAL>, then Parser can't know when to use ε-production

⇒ 怎么解决这个问题呢？：使用 Concatenation 的 left distributivity: $\underset{\sim}{R}S + \underset{\sim}{R}T = R(S+T)$

<sequence> → <statement> | <statement> ; <sequence>

|| equivalent to

<sequence> → <statement> <sequence Tail>

<sequence Tail> → <empty> | ; <sequence>

这样就叫 left factoring

或是另外一个差不多的例子：

$$A \to \alpha A_1 \mid \alpha A_2$$

$$\Downarrow$$

$$A \to \alpha A'$$

$$A' \to A_1 \mid A_2$$

Example:

$$S \to absa \mid abcSc \mid abdc \mid bcc$$

没注意看是 ab c Sc 而不是 abSc

$$\Downarrow$$

$S \to abS' \mid bcc$ $S \to abs' \mid bcc$

$S' \to SS'' \mid dc$ or $s' \to sa \mid cSc \mid dc$

$S'' \to a \mid c$ ✗

* 书上 Pg 233-234 展示了一个用 left factoring 会令 grammar ambiguous 的结果

Palindromes (RDP 无法解决的问题)

↳ 这个 palindrome 是这样定义的：

<palindrome> ::= <empty> | 0 | 1 | 0 <palindrome> 0 | 1 <palindrome> 1

<empty> ::=

两个问题：

1. <empty>是由<palindrome>生成的，但是，0，1，EOS 都是紧跟着<palindrome>的，所以当process 到<palindrome>时，RDP不知道该不该选择<empty>

2. Two productions both have 0 as their first tokens and two other productions both have 1 as their first tokens. RDP 不知道选哪个

∴ Left factoring not improve the situation

∴ RDP 根本用不了

Second typical problem: Left-recursive productions * 真的不懂

Determining whether a grammar allow RDP:

RDP1:

For each nonterminal $\langle N \rangle$, let $\langle N \rangle ::= \alpha_1 | \alpha_2 | \cdots | \alpha_n$ be all the productions for $\langle N \rangle$

RDP1 required that

$first(\alpha_i) \cap first(\alpha_j) = \emptyset$ 要把这个production的全走一遍

whenever $i \neq j$, where for any string α, $first(\alpha)$ is the set of all initial tokens in non empty strings derivable from α

RDP2:

If the empty string ϵ is derivable from a nonterminal $\langle N \rangle$

RDP2 required that

$follow(N) \cap first(N) = \emptyset$

where for any nonterminal $\langle N \rangle$, follow $\langle N \rangle$ is the set of all initial tokens in nonempty strings that may follow $\langle N \rangle$ in any derivation, possibly including the EOS

follow (N):

$S \to \cdots \to \cdots \to \alpha_k N b \alpha_{k'}$ — $b \in follow(N)$

$S \to \cdots \to \cdots \to B_k N$ — $EOS \in follow(N)$

Assignment 6. 2019.

1. $S \to AB \mid a$

$A \to CD$

$B \to bAB \mid \epsilon$

$C \to dSd \mid c$

$D \to cCD \mid \epsilon$

(a) FIRST(AB) = {d, c, ε} ✗ {d, c, ε}

FIRST (CD) = {d, c, ε} {d, c, ε}

FOLLOW (S) = {d} ✗ {d, EOS} → 所以只要是能derive到一个是跟在N后面的就都在follow中，都要从S开始!!

FOLLOW (A) = {EOS} {b, d, EOS} →

$S \to AB \to AbAB$

$S \to AB \to CDB \to \cdots dAdDB$

$S \to AB \to A$

FOLLOW (B) = {d, EOS}

FOLLOW (C) = {b, c, d, EOS}

FOLLOW (D) = {b, d, EOS}

Follow (B):

$S \to AB \to CDB \to dSdDB \to dAB\underline{d}DB$

Follow (D) =

$S \to AB \to CDB \to CD\underline{b}AB$

$S \to AB \to CDB \to dSdB \to dABdB \to dCD\underline{d}B$

Follow (C) =

$S \to AB \to CDB \to C\underline{c}CDB$

$S \to AB \to CDB \to CB \to C\underline{b}AB$

$S \to AB \to CDB \to dSdDB \to dABdDB \to dCDdDB \to dC\underline{d}DB$

(b) Does the grammar allow the use of RDP?

RDP 1: ✓

RDP 2:

$FOLLOW(B) \cap FIRST(B) = \phi$

$FOLLOW(D) \cap FIRST(D) = \phi$

这里只用对比那些可以 derive 到 ε 的 production 就可以

Left factoring / Left recursive:

(a) $S \to abSa \mid abcSc \mid abdc \mid bcc$

$\Downarrow$

$S \to abS' \mid bcc$

$S' \to Sa \mid cSc \mid dc$

(b) $S \to bSa \mid ccaSb \mid ccbSa \mid abc$

$\Downarrow$

$S \to ccS' \mid bSa \mid abc$

$S' \to aSb \mid bSa$

(c) $S \to Sa \mid Sbc \mid cc \mid \epsilon$

$\Downarrow$

$S \to ccS' \mid S'$

$S' \to aS' \mid bcS' \mid \epsilon$

这样的 cc 本来在 S 中，但用 ccS' 来替代
保持在 S 中

* 把 Sbc → bcS' 并放在 S' 中

ε 保持在 S' 中

这里的 S' 实际上是 εS'，所以如果 S 中
没 ε 的话就不需要 S'，看 √

(d) $S \to SB \mid \epsilon$

$B \to Bb \mid a$

$\Rightarrow$

$S \to S'$

$S' \to BS' \mid \epsilon$ ✓

$B \to B'$ ✗ $B \to aB'$

$B' \to bB' \mid a$ ✗ $\to \epsilon$

(e) $S \to abcSd \mid abdds \mid cdabs \mid cdbb$

$\Downarrow$

$S \to abS' \mid cdS'$

$S' \to cSd \mid dds \mid abs \mid bb$

or

$S \to abS' \mid cdS''$

$S' \to cSd \mid dds$

$S'' \to abs \mid bb$

两个一样的 prefix，它们
的 分开放

(f) $S \to Sab \mid Sac \mid b \mid c$

$\Downarrow$ → left recursion

$S \to S' \mid bS' \mid cS'$

$S' \to abS' \mid acS' \mid \epsilon$

$\Downarrow$

✗ 不需要

$S \to S' \mid bS' \mid cS'$

$S' \to aS'' \mid \epsilon$

$S'' \to bS' \mid cS'$

Assignment 5. 2020

(a) $S \rightarrow acSa \mid acbSb \mid acdb \mid cbb$

$\Downarrow$

$S \rightarrow acS' \mid cbb$

$S' \rightarrow Sa \mid bSb \mid db$

(b) $S \rightarrow bSa \mid ccaSb \mid ccbSa \mid abc$

$\Downarrow$

$S \rightarrow bSa \mid ccS' \mid abc$

$S' \rightarrow aSb \mid bSa$

(c) $S \rightarrow Sa \mid Sbc \mid cc \mid \epsilon$

$\Downarrow$

$S \rightarrow ccS' \mid S'$

$S' \rightarrow aS' \mid bcS' \mid \epsilon$

(d) $S \rightarrow SA \mid \epsilon$ $\Rightarrow$ $S \rightarrow S'$

$A \rightarrow Ab \mid a$ $S' \rightarrow AS' \mid \epsilon$

$A \rightarrow aA'$

$A' \rightarrow bA' \mid \epsilon$

Assignment 6. 2020

$S \rightarrow cAa \mid aAb \mid bB$

$A \rightarrow dAb \mid cB \mid \epsilon$

$B \rightarrow bB \mid cBa \mid \epsilon$

Follow (S) : $\{EOS\}$

Follow (A) : $S \rightarrow cAa \rightarrow \{a\}$
$S \rightarrow aAb \rightarrow \{b\}$ $\{a, b\}$

Follow (B) : $S \rightarrow CAa \rightarrow cdAb \rightarrow cdcB\underline{b} \rightarrow \{b\}$
$S \rightarrow bB \rightarrow bbB \rightarrow bbcB\underline{a} \rightarrow \{a\}$ $\{a, b, EOS\}$

(b) Follow (B) $\cap$ FIRST(B) = $\{b\}$ $\rightarrow$ break R of 2 $\rightarrow$ Can't RDP

Intro to Specification:

* 在设计一个程序 / 算法时，往往需要考虑很多问题 / 需求，所以创建一个程序的过程分为以下几步：

(requirements) → Specification → implementation

↓ Verification { testing / informal reasoning / rigorous reasoning

Specification is a contract:

Total Correctness: Program started in a state satisfying the pre-condition && terminates in a state that satisfying the post-condition 既满足 pre condition，又以满足 post condition 的方式结束

Partial Correctness: If the program terminates, then the variable at end satisfying the post-condition

总而言之，Specification 就是来判断 Program 应该做什么的，是非常必要的，比如一个 Array Search 简单地 program，就需要非常充分的 Specification，像什么：input 是啥，我们搜搜的有哪些范围，……

部分正确性：要求算法在返回结果时这一结果是正确的

完全正确性：在部分正确性的基础上还要求算法必须能结束

* Specification 可以分为 Static 或是 Dynamic 的 → 动态的 dynamic 的 assertion

↓ 一般指的是 identifier 的 type，identifier 被定义为什么类别

↓ 比如刚刚 Array Search 的例子：

const 代表不会变

```
const int max;        /* maximum number of entries */
typename Entry;       /* type of entries, use == for equality */
const int n;          /* number of entries */
const Entry x;        /* search target */
Entry A[max];         /* A[0:n-1] are the entries to search */
bool present;         /* search result */
```

→ Declarative interface

那么像 dynamic Specification 则是诸如：

- The requirements on the value of present after the search
- The assumptions on the value of n before the search
- The requirement that the Array segment A[0:n-1] not be changed

诸如这些都是用 Assertion 来写

↳ 用 logical formula 表写的

↳ &&, ||, ! (not), ==, implies, iff

∃, ∀

Implies 详: if P implies Q holds, then P is said to be Stronger than Q and Q is said to be weaker than P

and Q do not imply P, then P is strictly stronger than Q

else (Q implies P): P iff Q

Bound: ForAll (I) P ⟷ ∀I, P } I is set to be bound
Exists (I) P ⟷ ∃I, P

Correctness Statement:

ASSERT(P) /* 代码满足 pre condition
C → code Segment } P{C}Q
ASSERT(Q) /* 在运行 C 结束之后，代码满足 post condition

注意，P{C}Q 是一个 correctness statement，但它不是 assertion，a correctness statement is either true (valid) or false (invalid), independent of any particular state of a computation. 但是，an assertion can be true / false given different state

{ a correctness statement is a statement about code satisfying a pre- and post- condition
{ an assertion asserts a property of computational states

Simple assignment statement

↳ 所以在这里面做到的题都是 correctness statement $P\ \{c\}\ Q$

$P\ \{c\}\ Q$

↓ 一个例子

$n == n_0\ \{n = n-1;\}\ n == n_0 - 1$

↓ 可以写成更 generalize 的形式

Axiom Scheme: $V == I\ \{V = E;\}\ V == [E]\{V \to I\}$

↳ Expression E with occurrences of V substituted by I

Ex1: $x == y\ \{x = 5x + 2\}\ x == 5y + 2$

$x \to y$

Ex 2: $x == y\ \{x = 3*x + 2;\}\ x == 3*y + 2$

$x \to y$

但是 Axiom Scheme 是有限制的，那就是只有当 pre condition 是 $V == I$ 时才可以用，在这种基础上，提出

Hoare's axiom scheme: $[Q](V \mapsto E)\ \{V = E;\}\ Q$

Assertion Q ← 中把 occurrences of V 换成 E

Lecture 上给的概念：Given post condition, determine most general pre condition that guarantee post condition

Ex: $n-1 >= 0\ \{n = n-1;\}\ n >= 0$

$n = n-1$

Substitution;

在做 Substitution 的时候：

如果被替换的是一个 unbound/free 的 variable，那没关系直接换

如果被替换的是一个 bound 的 variable，那么换不成

如果 a 是不 free 的，要替换 x，但 $x = a$，那要先给被 bound 的 a 换名字，比如 t，再正常替换

书上练习:

(i) $P\ \{x = 0;\}\ x == 0$ P: $0 == 0$ ✓

(ii) $P\ \{x = 0;\}\ x >= 0$ P: $0 >= 0$ ✓

(iii) $P\ \{x = 0;\}\ x > 0$ P: $0 > 0$ ✗

(iv) $P\ \{x = 0;\}\ y > 0$ P: $y > 0$

(v) $P\ \{x = x+1;\}\ x == 1$ P: $x + 1 == 1$

(vi) $P\ \{x = x+1;\}\ x > 0$ P: $x + 1 > 0$

(vii) $P\ \{x = x+1;\}\ x == y$ P: $x + 1 == y$

(viii) $P\ \{x = x - 1;\}\ x == y-1$ P: $x - 1 = y - 1$

Assignment 7 2019

(a) P { x = 2; } x == 1
(b) P { x = 2; } x > 2
(c) P { x = y + z; } x < y + z
(d) P { x = x + y + z; } z > x*x + 2
(e) P { x = x + y + z; } y*y > z + 5
(f) P { x = y + z } Exists(w = 0; w < 10) x + w == 50
(g) P { x = y + z; } ForAll(z = 1; z < 100) x + 2*z > w + 2
(h) P { x = y + z; } ForAll(x = 1; x < z) x + y + 2 < 100
(i) P { x = y + z; } ForAll(y = 1; y < n) Exists(z = 1; z < n) x*y <= 3*z+w
(j) P { x = y + z; } ForAll(y = 1; y < n) Exists(x = 1; x < n) x*y <= 3*z+w

(a) P: 2==1 ✗
(b) P: 2 > 2 ✗
(c) y+z < y+z
(d) z > (x+y+z) * (x+y+z) +2
(e) y*y > z+5
(f) Exists (w= 0 ; w< 10) y+z + w == 50
(g) For All (t=1 ; t< 100) y+z +2*t > w+2 → 先换名字
(h) For All (x=1 ; x < z) x+y +z < 100
(i) For All (t=1 ; t<n) Exists (h=1 ; h<n) (y+z)* t < 3*h + w
(j) For All (y=1 ; y<n) Exists (x=1 ; x<n) x*y <= 3*z +w

Assignment 6 2020

(a) P { x = 1; } x == 1
(b) P { x = 1; } x == 2
(c) P { x = y + z; } 0 < x + y + z
(d) P { x = x*z + 3; } x*x > y + 2
(e) P { x = x*y*z + 1; } y*y > x + 5
(f) P { z = y + 1; } Exists(y = 0; y < 10) z + y == 50
(g) P { x = y + 1; } ForAll(z = 1; z < 100) x + 2*z > w + 2
(h) P { x = z + 1; } ForAll(z = 1; z < x) x + y + z < 100
(i) P { x = x + y; } Exists(x = 0; x < 10) x*x + z == 10
(j) P { x = x + y; } Exists(y = 0; y < 100)(x + y == 15 || z*x + y < 100)

Using Mathematical Facts

*首先，了解一下 mathematical facts 这个概念是怎么来的

↳ ASSERT(n > 0)

// n > 0 implies n-1 >= 0 → 这一行就是 mathematical fact，这一行 is true in every state

HA
ASSERT (n-1 >= 0)
n = n-1 ;
ASSERT(n >= 0)

↳像这样子的，用 mathematical fact to strengthen a pre condition is formalized by

pre condition strengthening :

$$\frac{P'\{C\}Q \qquad P \text{ implies } P'}{P\{C\}Q}$$

或者是，从 post condition 出发，弱化得到 mathematical fact，可以用

Post condition weakening :

$$\frac{P\{C\}Q \qquad Q \text{ implies } Q'}{P\{C\}Q'}$$

以上均是 inference rule : 在 inference rule 中，只要上面的被证明了，下面的就是对的

More inference rule :

Sequencing :

$$\frac{P\{C_0\}Q \qquad Q\{C_1\}R}{P\{C_0C_1\}R}$$

这两个 inference rule 基本上是对等的

$$\frac{P\{C_0\}Q \qquad Q \text{ implies } Q' \qquad Q'\{C_1\}R}{P\{C_0C_1\}R}$$

*然后书里面提到了为什么用 formal proof 去麻烦，formal proof 是证明的一种形式，但用 formal proof is difficult to read

所以改用 proof tableaux

- short hand notation for a formed proof
- contains: pre-condition, post-condition

↳ 就像：

ASSERT (n > 1)
ASSERT (n-1 > 0)
n = n-1
ASSERT (n > 0)
ASSERT (n >= 1)

Example:

Are the following correctness valid? (书[?] P43)

```
ASSERT (n<0)
// Mathematical fact, n<0 implies n*n >0
ASSERT( n*n >0)        \
n= n*n                  } H.A
ASSERT(n >0)           /
```

Sequencing 的更深层例子:

```
ASSERT(x== x0  && y==y0)
ASSERT( x==x0  && y== y0)      \
z=x                             } H.A
ASSERT( z== x0  && y==y0)      /  \
x= y                               } HA
ASSERT( z== x0  && x==y0)      \  /
y=z                             } H.A
ASSERT (y== x0  && x== y0)     /
```

Each intermediate assertion is both a post and a pre condition, recall the sequencing inference rule

Sequencing:

$$\frac{P\{C_0\}Q \qquad Q\{C_1\}R}{P\{C_0C_1\}R}$$

$\Downarrow$

$$\frac{P\{C_0\}Q \qquad Q \text{ implies } Q' \qquad Q'\{C_1\}R}{P\{C_0C_1\}R}$$

Example: Verify the validity:

```
ASSERT( i>=0 && y== power(x, i))
# This is valid since i>=0 implies i++ >=0 and y== x^i implies y*x == x^(i+1)
ASSERT( i++ >=0 && y*x == power (x, i++))   \
y= y*x                                        } HA
ASSERT( i++ >=0 && y== power(x, i++))       /  \
i++;                                             } HA
ASSERT( i >=0 && y== power(x,i))                /
```

Assignment 8, 2019

1. Verify following statement: valid!

```
ASSERT ( y==0 || z >-1 )
// if y==0 , then z==z-2y hold, if z>-1 then z >=0 hold
ASSERT( z >=0 || z == z-2y)
ASSERT( z-y >= -y || z-y+y == z-y-y )    \
x= z-y;                                    } H.A
ASSERT( x>= -y || x+y== x-y )            /
ASSERT( x >= x-(x+y) || x+y == x+x-(x+y))  \
z= x+y;                                      } HA      --> 先替换成 z==x
ASSERT( x >= x-z || z== x+ x-z )           /  \
y= x-z;                                         } H.A  --> 先替换成 z >=0
ASSERT ( x >=y || z == x+y)                    /
```

每一步都力求把 ASSERT 里面的东西都化得最简

Solution:

```
ASSERT( y == 0 || z > -1 ) // z > -1 implies z>=0 when z is integer
ASSERT( z-y+y >= 0 || y == 0 )
x = z-y;
ASSERT( x+y >= 0 || x+y == x )
z = x+y;
ASSERT( z >= 0 || z == x ) //arithmetic simlification
ASSERT( x >= x-z || z == x+x-z )
y = x-z;
ASSERT( x >= y || z == x+y )
```

化得最简

Example 2: Verify the validity of

ASSERT(k>0 && f==Fib(k) && g == Fib(k-1))

是可以的!!
f=Fib(k) 说明 k>=0

// ↓ ?? 不知道怎么imply

// k > 0 Can't imply k>=0, so this is invalid ✗

HA: ASSERT(k>=0 && f+g == Fib(k+1) && f== Fib(k))
t= f+g
HA: ASSERT(k>=0 && t== Fib (k+1) && f == Fib (k))
g= f
HA: ASSERT(k>=0 && t== Fib(k+1) && g == Fib(k))
f= t
ASSERT(k>=0 && f== Fib(k+1) && g == Fib(k))

应该是

HA: ASSERT(k>=0 && f+g == Fib(k+1) && f== Fib(k)) ✓ 这是个可行的 assertion
t= f+g
HA: ASSERT(k>=0 && t== Fib (k+1) && f == Fib (k))
g= f
HA: ASSERT(k>=0 && t== Fib(k+1) && g == Fib(k))
f= t
ASSERT(k>=0 && f== Fib(k+1) && g == Fib(k))

If- Statement

↳ 直接上：P { if (B) C0 else C1 } Q　有 else

没 else：P { if(B) C0 } Q

inference rule:

$$\frac{P \;\&\&\; B \;\{C_1\}\; Q \qquad P \;\&\&\; !B \;\{C_2\}\; Q}{P\{\text{if } (B)\; C_0 \text{ else } C_1\}\; Q}$$

↳ 上面这个 inference rule 给出了下面这个 proof tableau scheme:

```
ASSERT(P)
if (B)
    ASSERT(P && B)
    C0
    ASSERT(Q)
else
    ASSERT(P && !B)
    C1
    ASSERT(Q)
ASSERT(Q)
```

Example:

```
ASSERT(z <= y)
if (w > z || y > x)
    { w = z-1;   x = y; }
ASSERT(w <= z <= y <= x)
```

corresponding proof tableau

```
ASSERT(z <= y)
if (w > z || y > x)
    ASSERT(z <= y && (w > z || y > x))
    ASSERT( z-1 <= z <= x <= x)
HA  w = z-1
    ASSERT(w <= z <= x <= x)
HA  x = y
    ASSERT(w <= z <= y <= x)
ASSERT(w <= z <= y <= x)
```

X =>

```
ASSERT(z <= y)
if (w > z || y > x)
    ASSERT(z <= y && (w > z || y > x))
    ASSERT( z-1 <= z <= y <= y)
HA  w = z-1
    ASSERT(w <= z <= y <= y)
HA  x = y
    ASSERT(w <= z <= y <= x)
```

即使在 else 中没有任何东西，也一定要验证一部分

```
else
    ASSERT(z <= y && !(w > z || y > x))
    // !(w > z || y > x) require w <= z and y <= x
    ASSERT(w <= z <= y <= x)
ASSERT(w <= z <= y <= x)
```

Example 2 :

```
ASSERT (y==y0 && z == z0)
if (y <= z)
    { y = z+1 ; z = z + 2; }
else
    { z = y+2 ; y = y+1; }
ASSERT (max(y0, z0) < y < z)
```

↳ Corresponde proof tableau

```
ASSERT (y== y0 && z == z0)

if (y <= z)
    ASSERT (y==y0 && z==z0 && y<=z)
    // y <= z implies the following mathmatical fact
    ASSERT ( max(y0, z0) < z+1 < z+2)
H.A
    y = z+1
    ASSERT( max (y0, z0) < y < z+2)
H.A
    z = z+2
    ASSERT (max (y0, z0) < y < z)
else
    ASSERT (y==y0 && z==z0 && !(y <= z))
    // ∵ !(y<=z) implies y > z, so the following holds
    ASSERT(max (y0, z0) < y+1 < y+2)
H.A
    z = y+2
    ASSERT( max (y0, z0) < y+1 < z )
H.A
    y = y+1
    ASSERT (max (y0, z0) < y < z)
ASSERT (max (y0, z0) < y < z)
```

⇒ y == y0 && z==z0 && y<=z implies y0 < z+1 < z+2, which implies max (y0, z0) < z+1 < z+2

↳ 然后书介绍了只有 if 的情况，和之前笔记里介绍的一样，即使没有定义 else part，也要在 proof tableau 中有 else 的部分

↳ 一道可以思考的题目： why (B implies P0) && (!B implies P1) is valid as a pre-condition P.

```
ASSERT (P)
if (B)
    ASSERT(P0)
    C0
    ASSERT (Q)
else
    ASSERT (P1)
    C1
    ASSERT (Q)
ASSERT (Q)
```

While statement

↳ 提到了 loop，就不得不提一个 loop Invariant: A single well chosen assertion

Loop Invariant

*1. Formalization of intuition

↳ Incremental (True at every iteration) → 每个循环都是 True 的

*2. In a list, invariants usually say something about the "part of list"

↳ often talks about mathematical relationship / size bound → 至少这个在223中最常见

A[0 ... i-1]

A[j ... i]

*3. Invariants have no concept of time

↳ can tell you: the statement about variable at some moment is true

*4. Loop Variant ≠ loop condition

*书上: invariant may follow directly from some pre condition / some code like initializing assignment may be needed immediately before the loop

循环不变式：循环不变式是一种条件式，对循环而言是保持不变的，无论循环执行了多少次。循环语句每执行一次，就要求中间的结果必须符合不变式的要求

(1) 进入循环语句时，不变式必须成立

(2) 循环语句的循环体不能破坏循环不变式，也就是说，循环体开始循环时不变式成立，结束时也必须成立

(3) 如果循环语句终止时不变式依旧成立，那么至少说明，循环在保持循环不变式上没有犯错

→ 是否可以说明，当循环破坏了不变式的话，循环必定错，当循环满足不变式时，循环也不一定对

While:

Inference rule

$$\frac{I \ \&\& \ B \{c\} \ I}{I \ \{\text{while } (B) \ c\} \ I \ \&\& \ !B}$$

⇓

ASSERT (I) → 可以看到 ASSERT(I) 也是 loop 的 pre condition

while (B)

    ASSERT (I && B)

    c → 在运行 c 的时候，I 有可能会 False 一下下，但出来的时候，I 必须为 true

    ASSERT (I)

ASSERT (I && !B)

Example:

```
while (i != n)
    { y = y * n;
      i++;
    }
```

$\Longrightarrow$

书上：

选什么当 I 呢？ $\Longrightarrow$ I >= 0

we may prove following use assignment axiom scheme and sequencing rule:

```
ASSERT (I)
while (i != n)
    ASSERT (I && i != n)
                                    ASSERT ( i+1 >= 0 && y * x == power (x, i+1))
                              H.A   ASSERT( i++ >= 0 && y * x = power ( x, i+1 ) )
        y = y * x                   y = y * x
                                    ASSERT( i++ >= 0 && y == power ( x, i+1 ) )
                              H.A   i++
        i++                         ASSERT ( i >= 0 && y == power (x, i))
    ASSERT (I)
ASSERT (I && !B)
```

P'{C}Q

+

Then we find following mathematic fact:

P implies P'

- y == power (x, i) implies y * x == (x, i+1)
- i >= 0 implies i+1 >= 0

Then use pre conditioning rule to derive

```
ASSERT ( y == power (x, i) && i >= 0 && i != n)
    y = y * x
    i++;
    ASSERT (i >= 0 && y == power (x, i))
```

P{C}Q

无法检测 termination

看 3rd example

ASSERT (y == power (x, i) && i >= 0) 是 invariant

```
ASSERT ( true)
      ASSERT ( 1 == power ( x, 0) && 0 >= 0)
H.A   i = 0;                              // 新增的两个 assignment
      ASSERT ( 1 == power (x, i ) && i >= 0 )
H.A   y = 1;                              //
ASSERT ( y == power (x, i) && i >= 0)     // I在进入循环前也应该是正确的
while (i != n)
    ASSERT (y == power (x, i) && i >= 0 && i != n)
    // i >= 0 implies i+1 >= 0 , y == power (x, i) implies y * x == power (x, i+1)
      ASSERT ( y * x == power( x, i+1) && i+1 >= 0 )
H.A   y = y * x
      ASSERT (y == power (x, i+1) && i+1 >= 0 )
H.A   i++
    ASSERT (y == power (x, i) && i >= 0)
ASSERT (y == power (x, i) && i >= 0 && i == n)
// y == power (x, i) && i == n implies y == power (x, n)
ASSERT ( y == power (x, n) )
```

*写循环的 proof tableau 时一定要注意：

```
ASSERT (true)
assignment - C ...
ASSERT (I)      → 这个 assertion 是在 assignment 操作之后的
while (B)
    ASSERT (I && B)
    C
    ASSERT (I)
ASSERT (I && !B)    → 如果 I && !B 就是 post condition 的话，就可以结束了，但如果
ASSERT (Q)            有个另外的 post condition Q ，在这里要表示 I&&!B implies Q
```

Example 2:

```
I: i+j == 100 && 0 <= i <= 101
i = 0; j = 100;
while (i <= 100) {
    i = i+1; j = j-1; }
```

=>

```
     ASSERT(true)
     ASSERT(100==100 && 0<= 0 <= 101)
HA   i=0 ;
     ASSERT(i+100 ==100 && 0<= i <= 101 )
H.A  j = 100;
     ASSERT ( i+j == 100 && 0<= i<= 101 )
     while (i<= 100 ) {
        ASSERT ( i+j == 100 && 0<= i<= 101 && i<=100)
        // i+j == 100 implies i+j == 100 , 0<=i <= 101 && i<=100 implies 0<=i+1<=101
        ASSERT ( i+j == 100 && 0 <= i+1 <= 101 )
        ASSERT ( i+1 +j == 101 && 0<= i+1 <= 101 )
H.A     i=i+1 ;
        ASSERT ( i+j == 101 && 0<= i<= 101 )
        ASSERT ( i+j-1 ==100 && 0<= i<= 101 )
H.A     j = j-1
        ASSERT ( i+j == 100 && 0<= i<= 101 )
     }
     ASSERT ( i+j == 100 && 0<= i<= 101 && !(i<=100) )
     // i+j == 100 && !(i<=100) implies i==101 && j==-1
     ASSERT (i==101 && j == -1)
```

* Note that inference rule for while loops verifies only partial correctness，即这个方法检测不到 while loop停了没

⇓

```
i=0;  y=1;
while (i != n)      → 如果n<0, 这永不停
INVAR (i>=0 && y==power(x,i))
{   y = y*x ;
    i++;
}
```

i<=101 && i>100 implies i==101 ←

i==101 && i+j==100 implies j==-1

Example 3:

```
INVAR I: y==power(x,n-k) && 0<=k<=n
ASSERT(n>=0)
k=n;
y=1;
while (k>0) {
  INVAR(I)
  y=y*x;
  k=k-1;
}
ASSERT(y==power(x,n))
```

=>

```
ASSERT(n>=0)   // n>=0 implies n>=0
ASSERT( 1==power(x,0) && 0<=n<=n)
k=n;                                                        H.A
ASSERT( 1==power(x,n-k) && 0<=k<=n )
y=1;                                                        HA
ASSERT( y==power(x,n-k) && 0<=k<=n)
while (k>0) {
   ASSERT(y==power(x,n-k) && 0<=k<=n && k>0)
   // y==power(x,n-k) implies y*x == power(x,n-k+1)
   // k>0 && k<=n implies 0<=k-1<=n
   ASSERT( y*x == power(x,n-k+1) && 0<=k-1<=n)
   y=y*x;                                                   H.A
   ASSERT( y==power(x,n-k+1) && 0<=k-1<=n )
   ASSERT( y==power(x,n-(k-1)) && 0<=k-1<=n)
   k=k-1;                                                   HA
   ASSERT( y==power(x,n-k) && 0<=k<=n)
}
ASSERT(y==power(x,n-k) && 0<=k<=n && !(k>0))
// k>=0 and k<=0 implies k==0
// k==0 implies y==power(x,n)
ASSERT( y==power(x,n))
```

Find a proper Invariant I

* Often, the invariant may be obtained by generalizing the post-condition of the specification.
  - A simple technique that frequently works is to replace a "size" constant by a variable that is used as a counter.
    - For example, 讲的就是第一个 example, the assertion y == power(x,i) used in the invariant for the preceding loop is obtained from the post condition y== power(x,n) by replacing n by i
    - 在此基础上, it will usually be necessary to add range conditions on such a counter, 继续刚刚的 example: we added the assertion i>=0 to the invariant to ensure the power(x,i) was meaningful

所找的 invariant 必须要符合以下的条件:

1. It's preserved by loop body
2. It may be established initially by suitable assignment, taking into consideration the assumed pre condition
3. Together with the negation of the loop condition, it implies the desired post condition, perhaps after some finalizing code is executed

2nd extra page.

```
ASSERT( 1 <= n < max )
{ int i; i = 1;
  A[0] = 1;
  while( i < n ) { A[i] = A[i-1] + 3*i + 2;
                   i = i+1;
                 } //end while
}
ASSERT(ForAll(k = 0; k < n) A[k] == (3*k*k + 7*k + 2)/2 )
```

找 invariant

ASSERT (K=n <max)
{ int i;
i=1;
A[0]=1;
ASSERT (I)
while (i<n) { - - - -
i = i+1 }

① I 经过两次替换后 必定形成一个没有 for All 的式子，即 For All (K=0; K<n) 这个范围中只能有一个元素，且这个元素必定是 K=0，因为他 A[0] =1
利用 i=1，得知，此时 K已经等于0，要缩小范围，从 n下手，i = 1，
∴ For All (K=0; K<i) - - - - 必在 inavriant 中

② 已知 For All (K=0; K<i) - - - - 在 invariant 中
根据这个来判断 赋值 1和 i的关系
i+=1 ∴ 赋值 1 比 i 小
∴ 1 <=i ，又 ∵ !(i<n) → i>=n 要和 I 结合形成 post condition，∴ 1 <=i<=n 在 I 中

ASSERT (I && !(i<n))
ASSERT (for All (K=0; K<n) A[K] ==(3*K*K +7*K +2)/2

Invariant : 1 <=i <=n && For All (K=0; K<i) - - - -

Array Search

这是一个关于如何从一个嵌套代码中写proof tableau的例子，看清楚ASSERT是怎样继承的

```
ASSERT ( 0<=n<=max)
{ int i;
  present = false;
  i=0;
  while (i!=n) {
     if (A[i] == x) present = true;
     i++;
  } // end while
  ASSERT (present iff x in A[0:n-1])
Invariant: present iff x in A[0:i-1]
           && 0<=i<=n
```

proof tableau:

补上最上面这行以及最下面那行的原因是 i 等均属于 local variable, the pre- and post-conditions do not mention i

↳ Inference Rule for local variable:

$$\frac{P\ \{C\}\ Q}{P\ \{T\ I;\ C\}\ Q}$$

T: I的type

```
ASSERT( 0<=n<=max)
{
  int i;
  ASSERT( 0<=n< max)
  ASSERT( false iff x in A[0:-1] && 0<=0<=n)
  present = false;
  ASSERT( present iff x in A[0:-1] && 0<=0<=n)
  i=0
  ASSERT( present iff x in A[0:i-1] && 0<=i<=n)
  while (i!=n) {
      ASSERT (present iff x in A[0:i-1] && 0<=i<=n && i!=n)
      // i>=0 implies i>=0 , i!=n && i<n implies i<n
      ASSERT(present iff x in A[0:i] && 0<=i<n)
      if (A[i] == x) {
         ASSERT (present iff x in A[0:i-1] && 0<=i<=n && i!=n && A[i]==x)
         ASSERT( true iff x in A[0: i-1] && 0<=i<=n)
H.A      present = true
         ASSERT (present iff x in A[0:i-1] && 0<=i<=n) } // end if
      else {
         ASSERT (present iff x in A[0:i-1] && 0<=i<=n && i!=n && A[i]!=x)
         // present iff x in A[0: i-1] implies present iff x in A[0: i-1]
         ASSERT (present iff x in A[0:i-1] && 0<=i<=n) } end else
      ASSERT (present iff x in A[0:i] && 0<=i<n)
      ASSERT( present iff x in A[0: i] && 0<=i+1<=n)
H.A   i++
      ASSERT (present iff x in A[0:i-1] && 0<=i<=n)
  } end while
  ASSERT (present iff x in A[0:i-1] && 0<=i<=n && !(i!=n))
  // !(i!=n) && i<=n implies i==n
  // i==n implies present iff x in A[0:n-1]
  ASSERT (present iff x in A[0 : n-1])
}
ASSERT ( present iff x in A[0: n-1])
```

因为if/else {}中最后一条assertion，应该是紧随if/eles的，所以

应该移进去，因为从if出来之后才进行的i++，才得到这个

还有许多 Example:

① f=1; g=0; k=1;

```
while (k!=n)
INVAR (k>0 && f==Fib(k) && g==Fib(k-1))
  { t= f+g;   g =f;   f= t;   k++;
  }
```

Proof tableau:

```
ASSERT(true)
ASSERT(1 >0 && 1 == Fib(1) && g== fib(0))
f=1;
ASSERT(1 >0 && f == Fib(1) && 0== fib(0))
g=0;
ASSERT(1 >0 && f == Fib(1) && g== fib(0))
k=1;                                                          H.A
ASSERT(k>0 && f == Fib(k) && g== fib(k-1))
while (k!=n)
    ASSERT(k>0 && f == Fib(k) && g== fib(k-1) && k!=n)
    // k>0 implies k+1 >0, f== Fib(k) && g== fib(k-1) implies f+g
    // == Fib(k+1)
    ASSERT(k+1 >0 && f+g == Fib(k+1) && f == fib(k))
    t= f+g;                                                   H.A
    ASSERT(k+1 >0 && t== Fib(k+1) && f == fib(k))
    g=f                                                       H.A
    ASSERT(k+1 >0 && t== Fib(k+1) && g== fib(k))
    f=t;                                                      H.A
    ASSERT(k+1 >0 && f== Fib(k+1) && g== fib(k))
    k++;                                                      H.A
    ASSERT(k>0 && f == Fib(k) && g== fib(k-1))
}// end while
ASSERT(k>0 && f == Fib(k) && g== fib(k-1) && !(k!=n))
```

Fib(n) = Fib(n-1) + Fib(n-2)

For Statements :

for (A0; B; A1) C   Example ⇒   For (i=0 ; i<n ; i++) C

A0 and A1 are assignments

B is a boolean condition

C is a statement

等同于以下这个 while statement

A0; while (B) {C A1}

Corresponding proof tableau

```
ASSERT (P)
A0;          → while 外面执行
ASSERT (I)
while (B) {
    ASSERT (I && B)
    C
    A1       → while 里面执行
    ASSERT (I)
} // end while
ASSERT (I && !B)
ASSERT (Q)
```

Example:

```
ASSERT ( 0<=n<= max)
{ int i;
  present = false;
  for ( i=0 ; i<n ; i++)
     if (A[i] == x) present = true
  }
ASSERT ( present iff x in A[0:n-1])
```

换 while ⇒

```
ASSERT (0<=n <= max)
{ int i;
  present = false;
  i = 0;
  ASSERT (I)
  while (i <n)   {
      ASSERT (I && i<n)
      if (A[i] == x) {
          ASSERT(I && i<n&& A[i]==x)
          present = true
          ASSERT( I)
      else
          ASSERT (I && i<n && !A[i]==x)
          ASSERT (I)
      i++
      ASSERT (I) } // end while
ASSERT (I && !(i<n))
```

Array Component assignment → Formalize reasoning about array-component assignments

Code to interchange two array elements — assume all subscripts are within range of subscripts for A.

Not good

```
ASSERT(A[i] == x0 && A[j] == y0)
    A[i] = A[i] - A[j];
    A[j] = A[i] + A[j];
    A[i] = A[j] - A[i];
ASSERT(A[i] == y0 && A[j] == x0)
```

what can go wrong?

↳ if i == j, code does not work

Insted of modifying individual array elements, we have to modify the entire Array

Notation: $(A \mid I \mapsto E)[I'] = \begin{cases} E & \text{if } I == I' \\ A[I'] & \text{if } I' \,!{=}\, I \end{cases}$ 就是在A中把A[I]这个位置的值覆盖成E，然后取A[I']的值

Example (notation use):

A

0	1	2	3	4	5
8	7	9	3	2	4

$(A \mid 2 \mapsto 6)$

0	1	2	3	4	5
8	7	6	3	2	4

$((A \mid 2 \mapsto 6) \mid 4 \mapsto 9)$

0	1	2	3	4	5
8	7	6	3	9	4

$((A \mid 2 \mapsto 6) \mid 2 \mapsto 11)$

0	1	2	3	4	5
8	7	11	3	2	4

$$((A \mid I \mapsto E) \mid I \mapsto E') = (A \mid I \mapsto E')$$

Modified Hoare's Axiom Scheme

$[Q](A \mapsto A')$ { A[I] = E; } Q

where A' is $(A \mid I \mapsto E)$

⇒

```
ASSERT(Q)
    A[I] = E
ASSERT(Q)
```

→ Q中若提及Array A，则把上面的这个Q中关于A的部分换成(A | I ↦ E)

Example:

H.A

```
ASSERT((A|i↦3)[j] >= (A|i↦3)[i]))
    A[i] = 3;
ASSERT(A[j] >= A[i])
```

Rewrite pre condition inform that does not have array component substitution

i == j: 3 >= 3 true

i != j: A[j] >= 3

Logical formula precondition becomes

ASSERT(A[j] >= 3 || i == j)

Example 2

H.A

```
ASSERT(((A|i↦x) | k↦5) [j] == 0)
    A[i] = x
```

H.A

```
ASSERT((A|k↦5)[j] == 0)
    A[k] = 5
ASSERT(A[j] == 0)
```

Example:

```
ASSERT (0 <= n <= max) // Invariant I = 0<=i<=n && x != A[0: i-1] && A[n]==x
{ int i;
  ASSERT ( 0 <= n <= max)
  ASSERT (0 <= 0 <= n && x != (A | n ↦ x ) [0: -1] && (A | n ↦ x ) [n] == x)
  A[n] = x ;
  ASSERT ( 0 <= 0 <= n && x != A [0: -1] && A [n] == x)
H.A
  i = 0;
  ASSERT (0<=i<=n && x != A[0: i-1] && A[n]==x )
  while (A[i] != x) {
    ASSERT (0<=i<=n && x != A[0: i-1] && A[n]==x && A[i] != x)
```

// A[n] == x and A[i] != x implies i != n , 0<= i <= n and i != n implies 0 <= i < n

// 0 <= i < n implies 0 <= i+1 <= n , A[i] != x implies x != A[0 : i]

```
    ASSERT (0<=i+1<=n && x != A[0: i ] && A[n]==x )
H.A i ++
    ASSERT (0<=i<=n && x != A[0: i-1] && A[n]==x )
  }//end - while
  ASSERT (0<=i<=n && x != A[0: i-1] && A[n]==x && !(A[i] != x ))
  // A[i] == x && A[n] == x implies i == n, which implies following
  ASSERT ( (i<n) iff x in A [0:n-1])
H.A
  present = (i<n)
  ASSERT ( present iff x in A [0:n-1] )
```

Example:

```
ASSERT (A[j] == y && A[i] == x )
  ASSERT ( A[j] == y && A[i] == x)
  { int z ;
HA  ASSERT ( ( (A | i ↦ A[j] ) | j ↦ A[i] ) [i] == y && A[i] == x)
      z = A[i];
  ASSERT ( ( (A | i ↦ A[j] ) | j ↦ z ) [i] == y && z == x )
HA  A[i] = A[j]
    ASSERT ( (A | j ↦ z ) [i] == y && (z == x) )
    ASSERT ( (A | j ↦ z ) [i] == y && (A | j ↦ z) [j] == x)
H.A A[j] = z ;
  }
  ASSERT (A[i] == y && A[j] == x)
```

最外面 把 A[i] == A[i] , A[i]==y

Example 2

h.A

ASSERT$(((A \mid i \mapsto x) \mid k \mapsto y)[j] == 0)$

$A[i] = x$

h.A

ASSERT$((A \mid k \mapsto y)[j] == 0)$

$A[k] = y$

ASSERT$(A[j] == 0)$

Rewrite logical formula: 有的时候不一定都 hold，要讨论

$j == k$: $y == 0$ false

$j != k$: $j == i$: $x == 0$

$j != i$: $A[j] == 0$

$\Rightarrow (j != k \ \&\& \ j == i \ \&\& \ x == 0) \ || \ (j != k \ \&\& \ j != i \ \&\& \ A[j] == 0)$

Example: Shifting array elements from even numbered positions to a "contiguous chunk" in the beginning

interface const int n;

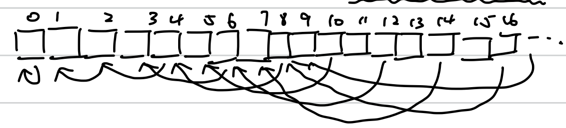
Entry A[2n]; // entries numbered 0, ..., 2n-1

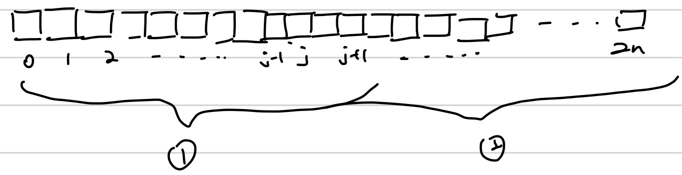
Precondition

$n >= 0$ && $A == A_0$

Post condition

For All $(i = 0; i < n)$ $A[i] == A_0[2i]$

∴ 

∴ 

(1) ForAll $(i = 0; i < j)$ $A[i] == A_0[2i]$:

Says what are values in modified part of array.

At end of loop implies post condition

(2) For All $(i = j; i < 2n)$ $A[i] = A_0[i]$:

Says what are values in unmodified part of array

This is needed to verify that Assignment $A[j] = A[2j]$ modifies invariant correctly

→ 是前面的 while loop 的一个概念：True before entering the loop; preserve by loop body

* Second part is needed because the code "moves around" element of the array (from unmodified → modified)

三

```
ASSERT(n >= 1 && A == A0)

j = 1;

while( j != n ) {
            A[j] = A[2*j];
            j++;
}
ASSERT(ForAll(i=0; i<n) A[i] == A0[2i]

invariant I: 1 <= j <= n &&
        ForAll(i=0, i<j) A[i] == A0[2i] &&
        ForAll(i=j, i<2n) A[i] == A0[i]
```

刚刚讲的極所以是拿来作 invariant 了

16

```
                                        ①                                ②
invariant I:
1<=j<=n && ForAll(i=0; i<j) A[i]==A0[2*i] && ForAll(i=j; i<2n) A[i]==A0[i]

ASSERT(n >= 1 && A == A0) /*implies below assertion*/
ASSERT(1<=1<=n && ForAll(i=0; i<1) A[i]==A0[2*i] &&
                  ForAll(i=1; i<2n) A[i]==A0[i])
j = 1;
ASSERT(I)
while( j != n ) {
ASSERT( I && j != n ) /* implies below assertion:*/
                      /* explain in class */
ASSERT(1<=j+1<=n && ForAll(i=0; i<j+1) (A | j +-> A[2*j])[i]==A0[2*i] &&
                    ForAll(i=j+1; i<2n) (A | j +-> A[2*j])[i]==A0[i])
      A[j] = A[2*j];
ASSERT(1<=j+1<=n && ForAll(i=0; i<j+1) A[i]==A0[2*i] &&
                    ForAll(i=j+1; i<2n) A[i]==A0[i])
      j++;
ASSERT(I)
 } //end while
ASSERT(I && j == n) /*implies below assertion*/
ASSERT(ForAll(i=0; i<n) A[i] == A0[2i])
```

this ~~next~~ page

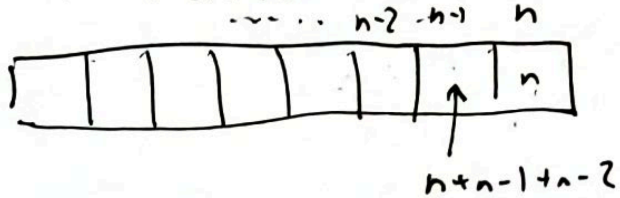
F_1

F_2

$+\!\!\rightarrow$ $\mapsto$

- j <= n && j != n implies j+1 <= n
- F_1 with j == i gives A[2*i] == A0[2*i] true by 2nd ForAll in invariant
  j != i (i = 0,...,j-1) is the first Forall statement in I
- F_2 is the same as 2nd ForAll of I except range of values is smaller

8) Example. All entries in segment A[0: max] are defined.



invariant says what are values in modified part of array

```
ASSERT(1 <= n < max)
{ int j; j = n-1;
  A[n] = n;        <- ASSERT(I)   { j = n-1
                                  { A[n] == n
while (j > 0) { A[j] = A[j+1] + j;
                j = j-1; } //end-while
}
          j=0
ASSERT( ForAll(k = 1; k < n+1) A[k] == (n-k+1)*(n+k)/2)
```

invariant I : $0 <= j <= n-1$ &&

ForAll (k=j+1; k< n+1) A[k] == (n-k+1)*(n+k)/2)

k<=n

20

2. As the invariant I we choose:

```
0 <= j <= n-1 && ForAll(k = j+1; k<n+1) A[k] == (n-k+1)*(n+k)/2
```

The proof tableau is as follows:

```
ASSERT(1 <= n < max) // n>=1 implies n-1 >= 0
ASSERT(0<=n-1 && true ) // variable declaration does not affect reasoning
{ int j;
ASSERT(0<=n-1 && n == (n-n+1)*(n+n)/2 ) //range of ForAll consists of n
ASSERT(0<=n-1<=n-1 && ForAll(k = n; k<n+1) (A|n+->n)[k] == (n-k+1)*(n+k)/2 )
(1)         j = n-1;
ASSERT(0 <= j <= n-1 && ForAll(k = j+1; k<n+1) (A|n+->n)[k] == (n-k+1)*(n+k)/2 )
  A[n] = n;
ASSERT(I)
while (j > 0) {
ASSERT( I && j > 0)
// j>0 implies j-1 >= 0 when j int
// I implies below ForAll() statement,
// and I implies A[j+1] + j == (n-j-1+1)*(n+j+1)/2  + j
// == (n*n + n - j*j + j)/2 == (n-j+1)(n+j)/2
(3) ASSERT(0<=j-1<=n-1 && A[j+1]+j == (n-j+1)*(n+j)/2 &&
        ForAll(k=j+1; k<n+1) (A|j+->A[j+1]+j)[k] == (n-k+1)*(n+k)/2)
//in below assertion write separately the equation when k==j
//and note (A|j+->A[j+1]+j)[j] == A[j+1] + j
ASSERT(0<=j-1<=n-1 && ForAll(k=j; k<n+1) (A|j+->A[j+1]+j)[k]==(n-k+1)*(n+k)/2)
                A[j] = A[j+1] + j;
ASSERT( 0 <= j-1 <= n-1 && ForAll(k = j-1+1; k<n+1) A[k] == (n-k+1)*(n+k)/2)
                j = j-1;
ASSERT(I)
} //end-while
}
ASSERT(I && j <= 0) //j<=0 && 0<=j implies j == 0
(2) //j==0 && I implies postcondition
ASSERT( ForAll(k = 1; k < n+1) A[k] == (n-k+1)*(n+k)/2 )
```

(4) The loop terminates because, by the invariant, j is non-negative and each iteration of the loop decrements j by one.

Example $\text{ASSERT}(((A \mid j \mapsto 3) \mid k \mapsto x+2)[m] > x+2)$

$$\text{H.A.} \uparrow \quad A[j] = 3$$

$$\text{ASSERT}((A \mid k \mapsto x+2)[m] > \underbrace{(A \mid k \mapsto x+2)[k]}_{x+2})$$

$$\text{H.A.} \uparrow \quad A[k] = x+2$$

$$\text{ASSERT}(\underline{A}[m] > \underline{A}[k])$$

Rewrite pre-condition as logical formula w/o array component assignment notation

- $m == k$: $x+2 > x+2$ false
- $m \,!= k$: $j == m$: $3 > x+2$
  - $j \,!= m$: $A[m] > x+2$

Formula:

$$(k \,!= m \;\&\&\; j == m \;\&\&\; 3 > x+2) \;||$$
$$(k \,!= m \;\&\&\; j \,!= m \;\&\&\; A[m] > x+2)$$

在做此類型的有关Array的“modified hoare axiom”时，推到上面pre condition之后，需要推出类似这样的邏輯关系，再根据这一层逻辑关系，确定这个

Non-interference Principle

ASSERT(P)
C
ASSERT(P)

is valid if

i) C has no assignment to variables used in P and
ii) C has no side-effects to variables in P

∴ 如果这个 hold → C 不影响 P

E.g. ASSERT(A == A0) ⇒ implies

ASSERT(ForAll (i=0; i<n) A[i] == A0[i])

Non Interference Principle

```
int k;
A[n] = target;
k=0
while (A[k] != target) k++;
present = (k < n);
```

no assignment / side effect to A[0 : n-1]

ASSERT(ForAll (i=0, i<n) A[i] == A0[i])

We have verified two correctness statement for the array search code C

Can combine w/ bounds as

ASSERT (0 <= n < max
&& A == A0)
C
ASSERT (present iff target in A[0: n-1]
&& ForAll (i=0; i<n) A[i] == A0[i])

↳ No assignment / side effect

inference rule

$$\frac{P_1 \{c\} Q_1 \qquad P_2 \{c\} Q_2}{P_1 \&\& P_2 \ \{c\}\ Q_1 \&\& Q_2}$$

Justification Using precondition strengthening

$P_1 \&\& P_2 \{c\} Q_i$, $i=1,2$

New inference rules Post-condition conjunction

$$\frac{P \{c\} Q_1 \qquad P \{c\} Q_2}{P \{c\} Q_1 \&\& Q_2}$$

$$\frac{P_1 \{c\} Q_1 \qquad P_2 \{c\} Q_2}{P_1 \| P_2 \ \{c\}\ Q_1 \| Q_2}$$

Intro to Computability (ch 12)

Algorithmic problem:

- potentially infinite set of inputs
  - a function that associate an output to each input

Solution for algorithmic problems:

- Algorithm that for each input computes the correct output

Unimplementable / Unsolvable

- many well defined and natural algorithmic problem can't be solved (i.e an algorithm does not exist)

Halting problem : Specification #判断一个函数会否在接收到一个 arg 后停下

Implement a function HALTS in C that receive two file. parameters func and arg

i) Must return true if the file func contains a C function definition with one file parameter, and the function ~~terminate~~ if applied to arg.

file.c

```
bool test (FILE f)
{ ... }
```

(ii) Returns false otherwise

Diagonalization

Technique for solving unsolvability

we assume HALTS can be implemented and derive a contradiction

```
bool halts (FILE func, FILE* arg) { ... }
```

assume this returns properly

test.c

```
int test (FILE* f ) {
    FILE* a = tempfile();
    copy (f, a) ;  /* copies file f to a */
    if (halts (f, a)) while (true){ }
    else return 0;
}
```

用 test 跑 halts (test, a)

停了的话就 ifinite loop

不停的话就 return false

Run test on test

* if halts return true → program will infinite loops
  should be false
  ⇒ ⇐
* if halts return false → program termines
  ⇒ ⇐ /* should be true */

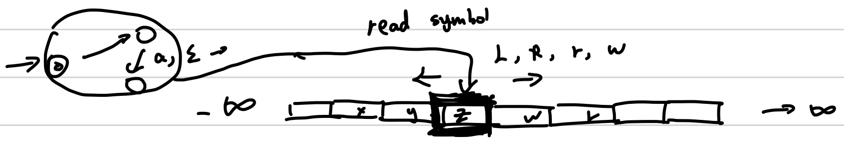
Assumption HALTS returns properly is false

⇒ HALTS is not solvable

可以理解第 2 part 的 contradiction：如果 halts 是 false，它停不下去，~~⇔~~ return 了

From a computability point of view, all high level programming languages (C, Java, Python) are equivalent: exact same function can be implemented in each language

* There exist various simple theorectical models that are equivalent to general purpose programming language
  - most well known is <u>Turing Machine</u>
  - Turing machine is a pushdown automata where the reading head can go into the stack and rewrite symbol inside of it.



<u>Church-Turing thesis</u>

Any function that can be implemented by an informal algorithm can be implemeted on a Turing Machine

- C.T is believe to be true
- impossible to prove it, because "informal algorithm" doesn't have a great definition

Reductions (和 365的一样)

↗ 解决B的方法可以拿来解决A

↓ A reduces to B

Algorithmic problem A reduces to problem B if a (supposed) algorithm for B can be used to construct algo for A

input → Algo for A → output

↕ use as a subroutine

Algo for B

Note the definition does not imply B has an algorithm

Example Know HALTS ( ) is unsolvable

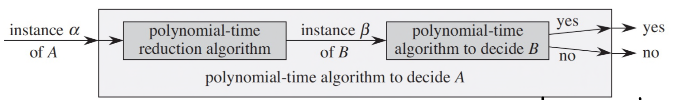
bool HaltsOnEmpty (FILE func)

* Returns true if func is a definition of an int function with one file parameter and that function halts when applied to the empty file
* Returns false otherwise

i) Halts on Empty reduces to general halts (is a special case of halts)

ii) In order to show Halts on Empty is unsolvable, need to reduce halts to halts on Empty.

↳ Reduction! "在解决一个问题之后，这个问题的答案可以解决另一个问题，那么另一个问题必定比直接解决的问题要简单"



Any instance of A can be transformed in polynomial time into an instance of B in an answer perserving way

P_1 : Given a set S of n integers, does S contain the value 4?

P_2 : Given a set S of n itegers, does S contain the target integer k?

↳ P_1 比 P_2 简单 ⟹ P_1 reduce to P_2 ✓ ⟹ P_1 比 P_2 简单

P_2 比 P_1 简单 ⟹ P_2 reduce to P_1 ⟹ ✓

S' 变成 $S-k+4$ → 存在4即存在 k

223 Notation : halts $\leq_p$ halts on Empty

reduces to Polynomial time

halts $\le_?$ Halts On Empty reduction

halts (Func , arg)
returns true if
func applied to arg
func (arg)

目标：用 haltsonEmpty 解决 halts

func, arg → Func, Arg

Implement a file transformation
merge(func, arg, FuncArg)

if func contains
```
int f (FILE* a) { c }
```
then merge writes FuncArg :
```
int f(FILE* b) {
    FILE* a = tmpfile();
    fprintf (a, "%s", "arg")

    c
}
```
new description, function ignores this

contents of file arg

把 func, arg 的行为复制到了 funcarg 中，这样 funcarg with empty file 的行为 = func on arg

Note : The code c uses file *a
Function written to FuncArg on any input behaves like func on input arg
所以 Func arg 只是把本来的 Func on arg 给 copy 下来

```
bool halts (FILE* func, FILE* arg) {
    FILE* FuncArg = tmpfile();
    merge (Func, arg, FuncArg);
    return HaltsOnEmpty (FuncArg)
}
```
Function written to FuncArg on the empty file behaves as func on input arg

Assuming HaltsOnEmpty() satisfies its specification, then implementation of halts is correct.

Halts reduce to HaltsonEmpty → HaltsOnEmpty is unimplementable / unsolvable

上面的只能在解决了 HaltsOnEmpty 后，才能解决 Halts()

Rice's Theorem $\leftarrow$ formalized as

Using similar "programming tricks" we can show that "practically all" algorithmic problem related to program correctness or termination are unsolvable

Decision Problem

Output is yes or no

Semantic Property of program:

property that relates to behaviour of program:

- on input x, on output y
- terminates on input x

Syntactic Property property depending on the code

- program has correct C syntax
- program has 1000 lines

Non-trivial decision property:

some inputs have the property and some do not

Rice's Theorem

All non-trivial semantic decision properties of programs are unsolvable

Semantic property: relates to input / output behaviour of the program

Unsolvable problems for strings

Post correspondance Problem (PCP)

Input: two sequences of strings

$(u_1, \dots u_k)$ $(V_1, \dots, V_k)$

Question: Does their exist a sequence of indices $i_1, \dots, i_m$

such that $u_{i_1}, u_{i_2}, \dots u_{i_m} = V_{i_1}, \dots V_{i_m}$

Note

the $1, \dots k$ may repeat in the sequence $i_1, \dots, i_m$

Example

(a^2, b^2, ab^2) (a^2b, ba, b)

Solution: $\begin{array}{c|c|c|c|} aa & bb & aa & abb \\ aab & ba & aab & b \end{array}$

Turing Machine halting problem reduces to PCP

$\hookrightarrow$ PCP is unsolvable

Using a reduction from PCP we can show various problems for CFG (context free grammar) are unsolvable.

- CFG equivalence
- does a CFG generate all terminal strings
- deciding ambiguity of CFG

Note CFG are useful because parsing can be done efficiently

Regular languages

- all "natural" decision problems are solvable for regular language
  - ↳ solvable in principle: we do not care about resources used by the algorithm

- There are known examples of unsolvable problems but have m "unnatural" / "Artificial" definition

- Computational complexity
  - * Existance of an algorithm does not imply it's solvable in practice
  - * Integer factorization
    - Trivially solvable in the computability sense
    - No efficient algorithm known
      also non hardness
  - * Many problems for regular language that can't be solved efficiently:
    - NFA minimization, NFA equivalence
    - regex minimization, regex equivalence
      → problems are PSPACE - complete