Constructing Programs that we Understand

Most programs are poorly understood.

There are unexpected "bugs".

We have to "try" them to find out what they will do.

They are always being "fixed" or "improved".

The only solution to this problem requires

- precise description of component programs, and
- discipline in the way we construct new programs from old ones.

Today's Questions:

(1) If we have a set of primitive programs, how can we combine them to construct bigger, more useful, programs?

(2) If we have a set of previously constructed programs, how can we combine them to construct bigger, more useful, programs?

(3) What does a program that "does nothing" really do?

Two Programs that "Do Nothing"

Definition 1: A program that quits. We call this program **abort**. It does nothing and nothing can ever happen after it runs.

Definition 2: A program that changes nothing. We call this program **skip**.

- The first does nothing because it never even allows the program after it to run.
- The second does nothing because it terminates without making any changes to the state.

This example illustrates the perils of using English (or any other natural language) to describe programs. English blurs the difference.

Our set of building blocks includes both of these programs.

Programming: Constructing Programs

Intuitively, programming is "telling the computer what to do".

More professionally, programming is the construction of bigger programs from smaller ones.

We almost never "instruct" the machine in detail. Our job is to use previously written programs in new ways. We "instruct" the computer to use those programs. They are our building-blocks.

We must understand, the building-blocks, and the constructors, in detail.

If we are given the mathematical description of our building-blocks, we must be able to produce the mathematical description of the programs that we produce.

Our products will be someone else's building-blocks.

Program Constructors and their Description

A *constructor* is a way of combining two more programs to get a constructed or larger program.

We are going to learn 4 forms of program construction:

- sequential composition
- conditional execution
- union (of conditional programs)
- iteration.

Any program constructed with only these tools is a well-structured program. Well structured programs are more easily understood.

We must assume that we have precise descriptions of the component programs.

We must know how to find a precise description of the constructed programs from the component programs.

Only then will the programs that you construct be well understood programs.

What Should We know about how building blocks and constructed programs?

We should know the answers to the following questions:

- Which states can we start the program in if we want termination to be possible?
- Which states can we start the program in if we want termination to be certain? (*safe states*)

If we start a program in a state where termination is possible, what final states are possible?

The Constructor ";"

If A and B are programs, A;B is a program.

Intuitively, A;B means do A, then do B.

A leaves the machine in some state; B starts in that state. The constructed program starts in the state that a started in and ends in the state that b stops in.

A safe state for A;B must be a safe state for A and, must always lead to safe states for B. Otherwise it isn't safe.

Note that this constructor works for any programs A and B, not just primitive programs. A and B could be thousands of lines long and very complex. This simple definition remains valid.

This constructor is found in every programming language, usually denoted by ";" or a new line character.

Using ";": Some Simple Examples

In the following assume we have only two (finite) integer variables, x and y, available. x' denotes the value of the variable x after the program executes; 'y denotes the value of y before any execution.

(1) Problem: (x'= 'x+1) ∧ (y' = 'y+1)
Solution: x ⇐ x+1; y ⇐ y+1
Solution: y ⇐ y+1; x ⇐ x+1

(2) Problem: (x'= 'x+'y) ∧ (y' = 'y+1)
Solution: x ⇐ x+y; y ⇐ y+1
Solution: y ⇐ y+1; x ⇐ x+y-1

(3) Problem: (x'= 'x+y') ∧ (y' = 'y+1)
Solution: x ⇐ x+y+1; y ⇐ y+1
Solution: y ⇐ y+1; x ⇐ x+y

(4) Problem: (x'= 'x+'y) ∧ (y' = 'x+1)
Solution: y ⇐ x +1; ???
Solution: x ⇐ x+y; y ⇐ x - y +1

Even in these simple examples, programming can be tricky, mistakes can be made, and one must work with discipline and care. Some simple problems cannot be solved without imposing limits on the values of x and y or by using an extra variable.

Swapping Two Values

Consider the following problem:

$$(x' = 'y) \wedge (y' = 'x)$$

Remember that we have only 2 variables!

Can we do it? - Under certain conditions.

Solution: x ⇐ x + y; y ⇐ x - y; x ⇐ x - y

The condition: x must have enough states to store x+y. If the initial values are too large, it fails.

What if we have a third variable, t?

Solution: t ⇐ x; x ⇐ y; y ⇐ t

Question: Does the specification allow t to change?

Answer: Yes, the specification places no restrictions on what you do to t.

Note: Space-Generality trade-off. Without the extra variable we cannot swap all possible values.

Using ";" With Powerful Programs

Suppose that, instead of our normal arithmetic operations we had operations on arrays.

For example, "A >20" is an expression whose

value is an array, same shape as A, with a ***true***

where A had an element with value greater than 20 and a ***false*** in all other positions.

B⇐ A>20 assigns that value to B.

A/B, where A and B have the same shape and A is an array whose elements consist of ***true*** and ***false*** is an expression that "filters" B, its value is an array with the corresponding elements of B in positions where there is a ***true*** in A and 0 elsewhere.

ΣA is an expression whose value is the sum of the values of the elements in A.

Problem: Consider the effect of:

B⇐A>20; C⇐B/A; D ⇐ ΣC

What does this program do?

A general notation for constructing programs

The important principles of program construction apply to all practical languages.

In this class we will use a simple notation to describe how our programs are constructed.

These program plans can be translated into any other "imperative" programming language.

Partial Syntax for Programs

<program> ::= <simple program>
|<composed program>

<simple program> ::=

<primitive program>
| (<program>)
| *more to come*

<composed program>::=
<simple program> ; <simple program>
|<composed program> ; <simple program>

<primitive program> will not be fully defined but will include <expression>, <assignment> **skip**, **abort,** and *more to come*

The above is the first step towards defining a complete notation for designing programs.

I call the notation (language) DAD. It features simplicity, ease of analysis, and generality. We use it as a tool for program planning.

Do we Need Other Program Constructors?

";" is surprisingly powerful.

With a "rich" set of primitive programs, we can do a great deal just by sequencing the invocations of those programs.

The programming language APL is famous for it's "one-liners". All they use is ";" to achieve composition of the powerful built-in functions.

APL's primitive libraries were *not* built using ";" alone. We do need more.

- We need to limit the conditions under which a program will be executed (conditionals).
- We need to provide for alternatives (branches).
- We need to provide for iteration (loops).

We will have to define these program constructors precisely, telling how to find the function/relation of the constructed program.

Guarding Programs

We need to tell the computer under what conditions a program may be executed.

We will do this by providing programs that say "yay" or "nay", i.e. ***true*** or ***false***, to the execution of a program. The information will be left in the unnamed variable #, the place where expression values are deposited.

These programs are normal boolean expressions. They evaluate to ***true*** or ***false***. We use them as *guards*. Guards should not change the state of any other variables.

Definition: If "g" is a guard, and "P" is a program then, "g $\rightarrow$P" is a *guarded program.*

Note**:** a guarded program is not a program.

Meaning: The program, P, should be executed only if the guard, g, evaluates to true.

Describing Guarded Programs

For any guarded program we want to know:

- What will the guarded program do if executed?
- When is termination of the guarded program guaranteed?

In what states would the guard say ***true***, but termination is not guaranteed?

What does a guarded program mean?

- The guarded program terminates only if the guard is ***true*.**
- In those cases it behaves exactly as g;P would.
- There may be states where the guard itself might not terminate, or states where the guard yields ***true*** but terminates in a state where P is not guaranteed to terminate.
- These states are "traps"; the computer can get trapped by using g to see if P can be executed and believing what g reports.

A guarded program should not be used in states that are traps for it.

Combining Guarded Programs using "|"

It is useful to combine guarded programs than to combine unguarded ones. The guards can be used to tell the computer when to consider each of the guarded programs.

If A, B, ... are guarded programs then,

(A| B| ...) is a program.

Intuitively:

- One of the programs whose guard is true will be selected and executed.
- If no guard is true, the program will abort.
- If two or more guards are true, we are introducing nondeterministic behaviour.
- The guards should be such that there are no "trap" states.

Syntax for Programs

<program> ::= <simple program>
| <composed program>

<simple program> ::=

<primitive program>

| (<program>)

| **(<guarded program list>)**

| *more to come*

<guard> ::= <boolean expression>

<guarded program> ::=

<guard> → <simple program>

<guarded program list> ::=

<guarded program> | <guarded program list> '|' <guarded program>

<composed program>::=

<simple program> ; <simple program>

|<composed program> ; <simple program>

Examples using guarded programs.

Problem:
Y'=MAXIMUM('X1,'X2)∧NC(X1,X2)

Solution: (X1 ≤ X2 → Y⇐ X2
| X2 ≤ X1→ Y⇐X1
)

Problem: y' = SQRT(|'x|)

Solution: (x<0 → y⇐SQRT(-x)
|x>0 → y⇐ SQRT(x)
| x=0 → y⇐0
)

What does the following program do?

(X > 7 → X ⇐ X+1
| X <7 → X ⇐ X-1
| X=7 → X ⇐ 100
)

Vector Function Table

	‘X > 7	‘X= 7	‘X< 7
X’ =	‘X + 1	100	‘X- 1

H_1 H_2 G

A Non-deterministic Program

Problem:

Vector Relation Table

	‘X > 7	‘X = 7	‘X < 7
X’ |	X’ = ‘X + 1	(X’ = 8) ∨ (X’ = 6) ∨ (X’ = 100)	X’ = ‘X-1

H_1 H_2 G

Solution:

(X > 6 → X ⇐ X+1
| X < 8 → X ⇐ X-1
| X = 7 → X ⇐ 100
)

"Divide and Conquer" Programming

The guarded program is there to make sure that a program will only be executed when the guard says ***true***. Every programming language has a similar feature, called conditional statement.

Using the guarded program list we can provide a list of alternatives. The constructed program will do what would have been done by one of its programs with a guard that says true.

To check a guarded program list:

- Make sure that all the states will have at least one guard true.
- Check the guarded programs one at a time to see if they will do the right thing when the guard is true. You never need to look at two guarded programs at once.

This is the part of the "*divide and conquer*" approach to program construction/analysis. "Divide and Conquer" is the only way to master complexity in programs. Never try to understand (or write) a whole program at once.

Why do we Need Iteration?

Without iteration the number of state changes that can happen is limited by the length of a program.

- Everything is done only once.
- To do a lot, we must write a lot.

"Iteration" is a fancy word for repetition.

Each time that we execute a program that is being iterated, part of that program must determine whether the iterated program should be executed again or not.

In some languages, the decision about repetition seems to be made outside the iterated program, but this is misleading. The check for continuing is made every time the program is executed and so is part of what is being repeated.

Syntax of Iteration in the Planning Language

If P is a program,

it P ***ti***

is a program.

Execution of P will either be followed by another execution of P or stop in accordance with decisions made within P.

To determine whether to continue or stop we introduce two new primitive programs. These are used to indicate whether or not the iteration should continue.

☛, pronounced “go”

●, pronounced “stop”

If ☛ is executed, during P, P will be repeated.

If ● is executed, iteration will stop.

If both are executed, the latest execution determines the effect.

Examples:

● ; ● ; ☛ is equivalent to ☛

☛ ; ☛ ; ● is equivalent to ●

The body, P, of *it* P *ti*

P can be any program provided that it executes

either ● or ☛ at least once in its first execution.

If P never executes ●, iteration never terminates.

Non-terminating programs are useful in real-time systems but those are beyond the scope of this course.

In this course we are primarily interested in terminating programs.

One more useful Primitive program.

init, which is “$ = ***start***”, a boolean expression sets # to ***true*** if $ = ***start***

init allows a program to do something special on the first execution of its body.

init is only useful in the body of a loop.

The Meaning of *it* P *ti*

Wrapping a program P in iteration brackets means that

- P will be executed at least once.
- During each execution of P either ☛ or ● must be executed.
- When the execution of P is complete, the iteration will either continue or stop depending on whether ☛ or ● was the last of those programs to be executed.

Other Iteration Constructs

In the following, B represents a boolean expression, P an arbitrary program, A, S, and C arithmetic expressions, and I is an integer variable.

- **while** B **do** P

  it (B → (P;☛) | ¬ B → ●) ***ti***

- **repeat** P **until** B

  it P; (B → ● | ¬ B → ☛) ***ti***

- **repeat** P **while** B

  it P; (B → ☛ | ¬ B → ●) ***ti***

- **for** I ⇐ A **step** S **until** C **do** P

  I ⇐ A;

  it (I ≤ C→ (P; I⇐I+S;☛) | I > C → ●) ***ti***

Using itti we can accomplish the iteration statements of other languages. First we design the loop, then we pick the best construct.

Examples

Problem: x' = min('x,20)

Solution:

it (x>20 → (x⇐ x-1; ☛) | ¬ (x>20) → ●) ***ti***

Solution:

(x>20 → x⇐ 20 | ¬ (x>20) → **skip**)

Problem:

('y≥0) ∧ ('x='x') ∧ (y'=0) ∧ (z'= 'x × 'y)

Solution: z ⇐ 0;

it (¬ (y=0) → (z ⇐ z+x; y ⇐ y-1; ☛)
| (y=0) → ●)
ti

This program doesn't stop for negative 'y!

Problem:

(('y>0) ∧ ('x = x') ∧ (y' = 0) ∧ (z' = 'x × 'y)) ∨ ((¬ ('y>0)) ∧ ('x = x') ∧ ('y= y')∧ (z' =0))

Solution: z ⇐0;

it ((y>0) → (z⇐ z+x;y⇐ y-1;☛)
| ¬(y>0) → ●)

- ***ti***

Final Syntax for Program Planning

<program>::=<simple program> |
<composed program>
| <guarded program list>

<simple program> ::=
<primitive program> | (<program>)
| ***it*** **<program>** ***ti***

<guard> ::= <boolean expression>

<guarded program> ::=
<guard> → <simple program>

<guarded program list> ::=
<guarded program> |
<guarded program list> '|'
<guarded program>

<composed program>::=
<simple program> ; <simple program>
|<composed program> ; <simple program>

<primitive program> will include <expression>, <assignment>, ●, ☛, **skip**, **abort**, and **init**.

This is the whole syntax,

With it you can plan any program.

You can only produce well-structured programs.

It is easy to analyse.

It is easy to translate into other languages.

It makes excellent documentation.

The important properties of DAD

(1) Allows any known "fixed" algorithms.
(2) Symmetry supported in programs.
(3) Full Nesting of programs. (Box structure)
(4) Any program can be a statement in a bigger program.
(5) Complete semantics.
(6) Rules fit on one page.

Naming Programs

To ease readability we often give programs names.

Placing the name of a program, in a larger program has the same semantics as copying the text of the named program into that program.

Checking Termination

Problem: x' = min('x,20)

Solution:

it (x>20 → (x⇐ x-1; ☛) | ¬ (x>20) → ●) ***ti***

Solution:

it x>20;(#→ (x⇐ x-1; ☛) | ¬ (#) → ●) ***ti***

How can we be sure that a program will always terminate?

(1) Find a quantity that always decreases whenever the body executes ☛.

(2) Show that whenever that quantity is not positive, the body will execute ●.

How can we be sure that the programs above will always terminate?

(1) The value of x - 20 decreases whenever ☛ is executed

(2) Whenever x - 20 is not positive, ● will be executed.

Note that many useful programs do not always terminate. We always need to know whether or not a program will terminate.

Euclid's algorithm for Greatest Common Divisor (GCD)

Problem:
('x > 0) ∧ ('y > 0) ∧ (x' = y' = GCD('x,'y))

Some mathematical facts:

- GCD is only defined for positive arguments.
- If x> y and x and y are not negative, GCD(x-y,y) = GCD(x,y).
- for positive x, GCD(x,x) = x

Euclid's algorithm for Greatest Common Divisor (GCD)

Solution:

((x > 0) ∧ (y > 0) →
it
(x>y→ (x⇐x-y; ☛)
| y>x → (y ⇐ y-x; ☛)
| x=y → ●)
ti)

Note that the outer guard prevents termination of the program when ‘x = ‘y = 0.

The iteration will terminate if ((‘x>0)∧(‘y>0))∨(‘x=‘y).

(1) The value of **max(x,y) - gcd(x,y)** decreases, and x and y remain positive, whenever ☛ is executed with both x and y positive. When x=y this quantity is zero.

(2) The program will execute ● if and only if x=y.

- If both x and y are initially positive, or both are initially 0, the iteration will stop.

The Advantages of Thinking in Terms of Monotonically Decreasing Quantities.

The conventional way of thinking about termination is to try to think about all the possible things that might happen.

There are usually many possible execution sequences.

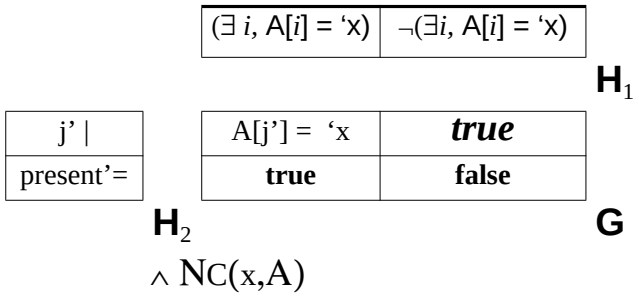
Often we overlook some of them.

Looking at this decreasing quantity allows us to avoid trying to find all the sequences.

It is simple and certain.

If the loop terminates, there must always be a quantity that decreases with each execution of the body.

PROBLEM: Searching array A with indices 1 ... n



PROBLEM: Searching array A with indices 1 ... n

SOLUTION:

j ⇐ 1; present⇐ **false**;
it (A[j]=x→ (present⇐**true**;●)
| ¬(A[j]=x)→ (j<n→(j⇐ j+1;☛) | j ≥ n→ ●))
ti

SOLUTION:

j⇐ n; present⇐ **false**;
it (
(A[j] = x → (present ⇐ **true**; ●) |
¬(A[j] = x)→(j >1;
(#→ (j⇐ j -1; ☛) | ¬(#)→ ●))))
ti

Exercises

(3) Explain the differences between these programs.
(4) Do they always get the same answer?
(5) What is the monotonically decreasing quantity for each?

Finding the maximum in part of an array, A, of N elements

Problem: partmax

$(\forall\ i, j \le i \le N \Rightarrow A[i] \le max') \wedge$

$(\exists_k\ j \le k' \le N \wedge A[k'] = max' \wedge NC(A,j))$

Solution:

$max \Leftarrow A[j]; i \Leftarrow j; k \Leftarrow j;$

it

$(i < N \rightarrow ($☛$; i \Leftarrow i + 1;$

$(max < A[i] \rightarrow (max \Leftarrow A[i]; k \Leftarrow i)$

$\mid max \ge A[i] \rightarrow$ **skip**$)$

$)$

$\mid i \ge N \rightarrow$ ●$)$

ti

Silly Sort

Problem: permutation('A, A') $\wedge$
$(\forall\ i, (1 \le i < N) \Rightarrow (A[i+1]' \le A[i]'))$

Solution:

$j \Leftarrow 1;$

it

$(j < N \rightarrow ($partmax; $A[k] \Leftarrow A[j]; A[j] \Leftarrow max;$

$j \Leftarrow j + 1;$☛$)$

$\mid j \ge N \rightarrow$ ●$)$

ti

Reminder: partmax

$(\forall\ i, j \le i \le N \Rightarrow 'A[i] \le max') \wedge$
$j \le k' \le N \wedge A[k'] = max' \wedge NC(A,j)$

Note how this program was constructed using a previously built program.

Note that we did not have to look at partmax, but we must look at it's description.

There are much better ways to sort!

Summary

Programs should be constructed using previously defined programs, not written from start to end.

One can avoid big programs by building programs from previously available, precisely described, parts.

It is essential to have a precise description of those parts.

These programs can be systematically checked.

- Are all cases covered by a guarded program list?
- Is each part correct for the cases that it covers?
- Under what initial conditions will the iterations terminate?

It is hard enough to get these little programs correct. Tiny changes have a big effect. We must stress writing well-structured small programs.

These principles apply, and can be applied, no matter what programming language is used.

How can *anyone* understand a large program?

The most complex program that we have seen up to now has included only one loop and a few guarded program lists.

These "simple" programs are not simple. A great many details must be correct for them to work.

Small lapses of attention are "fatal". Small changes can have a very big effect.

There is no such thing as "almost right".

Textbook programs can be much larger. Some may be about 100 or more lines.

They are still small programs.

Industrial programs of 10,000 lines are small.

Typical programs are hundreds of thousands.

Many systems involve millions of lines.

How can such programs be right?

How dare we rely on them?

"Loop Invariants"

A *loop invariant* is a predicate that:

- will be ***true*** whenever execution of the loop begins,
- will be maintained (kept ***true***) by each execution of the body.

If you learn to think in terms of loop invariants you won't have to think in terms of lengthy sequences of cases.

There is a loop invariant for every loop and in every practical programming language. In the next pages you will see many examples.

Loop invariants are another tool to help us avoid having to try to find all possible sequences.

Instead of asking what changes during the loop, we will ask what stays the same.

It makes the thinking easier.

Finding maximum and minimum elements in an array.

PROBLEM: maxmin (Array X with indices 1 ... n**)**

Assume functions MAXIMUM, MINIMUM defined on arrays and subarrays.

	$(1 \le l \le u \le n)$	$\neg(1 \le l \le u \le n)$

H_1

i' |	X[i']=MINIMUM(X[*l*:u])	***true***
j' |	X[j']=MAXIMUM(X[*l*:u])	***true***

H_2 **G**

$\wedge$ NC (X, n, *l*, u)

Finding maximum and minimum elements in an array.

maxmin ≡
i⇐*l*; j⇐i;
(**integer** w; w⇐*l*;
it
(w<u → (☛;w⇐w+1;
(X[w] < X[i] → i⇐w
| X[w] > X[j] → j⇐w
| X[i] ≤X[w] ∧ X[w] ≤X[j] → **skip**))
| w≥u → ●)
ti)

invariant:

X[i]=MINIMUM(X[*l*:w]) ∧
X[j]=MAXIMUM(X[*l*:w])

w is a local variable.

monotonically decreasing quantity: u-w

Sorting an Array X [1 ... N], N> 1

Problem: permutation('X,X') ∧
(∀ i, 0 < i < N⇒ X'[i] ≤X'[i+1])
Solution:
i⇐ 1;
it
(i < N →(X[i] > X[i+1] →
(swap(X[i],X[i+1]);☛;i⇐ 1)
|X[i] ≤ X[i+1] → (i⇐ i+1; ☛))
| i ≥ N → ●)
ti

invariant: (∀ j, (0 < j < i) ⇒ X[j] ≤X[j+1])

monotonically decreasing quantity:

(number of out of order pairs, N-i).

Notes:
((a,b)<(c,d)) ≡ ((a < c) ∨ ((a=c) ∧ (b<d)))
Specification of swap(a,b) ≡
((a' ='b) ∧ (b' = 'a)) ∧ NC(all other variables)
Note that aliasing will limit our ability to implement this.

Summary

Building well-structured programs is the only way to reduce the number of bugs. Reducing the number of bugs is essential for reliabilty, safety, and sales.

Thinking in terms of invariants is the best way to design loops.

Finding the MDQ is the best way to be sure of termination.

We have to avoid trying to trace all the possible sequences.

We have to avoid long programs by using previously constructed programs.

Three principles:

- Divide and Conquer
- Monotonically decreasing quantities
- Loop invariants that "grow" to program specifications.

Be a program designer, not a language lawyer.

Design rules last. Programming languages change or become out-of-date.

Generalised Pattern Matching

Given two strings, one called **pat**, one called **dat**, find the occurrences of pat in dat.

This problem arises in many applications.

The programs run on long data files and should be as fast as possible.

Such programs should also be correct; some commercial offerings are unreliable.

Early solutions were *ad hoc* and complex.

Thorough research has led to far better algorithms.

This program is not easy; it is a famous algorithm. Many find it quite complex, but we can see view it as a set of simple programs.

Any well-structured program can be viewed as a set of simple programs.

That is the *only* way we can understand complex programs.

Looking for a match at position k?

integer array pat[0:p];
integer array dat[0:D]; **boolean** m;

Problem:

(m' = (∀ i, (0≤i≤p) ⇒ pat[i]=dat[k+i])) ∧ NC(pat,dat, p, D, k)

Solution:

(k≤ D-p;
(# → (integer i; i ⇐ 0;
it (pat[i]=dat[k+i];
(#→ (i<p;
(#→(i⇐ i+1;☛)|¬#→(m⇐ **true**;●)))
|¬# → (m ⇐ **false**; ●)))
ti)
| ¬# → m ⇐ **false**))

loop invariant:

(∀j, (0 ≤ j < i) ⇒ pat[j]=dat[k+j]) ∧
((i=p) ⇒ pat[p]=dat[k+p])

Note: To make the program a little more efficient, we had to make the invariant more complex.

decreasing quantity**:** p - i

Note that iteration may stop before (p-i) =0.

Is there a match at some position k, k ≥ 0?

Let's not throw away the failure information. We make i global so that we can use its value.

Problem**:** k ≥ 0 ⇒
((m' = (∀ i, 0 ≤ i ≤ p ⇒ pat[i]=dat[k+i])) ∧
(k≤ D-p ⇒
i'=minel[1]({i|(0≤i≤p)∧pat[i]≠dat[k+i]}∪{p+1}))
∧ NC(pat,dat, p, D, k))

Solution**:**

matchk ≡(k ≤ D-p;
(# → (i ⇐ 0;
it (pat[i]=dat[k+i];
(#→(i⇐ i+1; i ≤ p;
(#→ ☛ |¬# →(m ⇐ **true**;●)))
|¬# → (m ⇐ **false**; ●)))
ti)
|¬# → m ⇐ **false**))

if k≤ D-p, i' will indicate the first position in the pattern where the match failed. If i' >p, the match has succeeded. This will be useful later.

---

[1] Note: minel (x), where x is a non-empty set of integers, is the smallest value in x.

Is there a match at position k, k ≥ 0 ?

Solution:

matchk ≡ (k≤ D-p;
(# → (i ⇐ 0;
  it (pat[i]=dat[k+i];
  (#→(i⇐ i+1; i ≤ p;
    (#→ ☛ |¬# →(m ⇐ **true**;●)))
  |¬# → (m ⇐ **false**; ●)))
  ti)
|¬# → m ⇐ **false**))

loop invariant:

(∀j, (0 ≤ j < i) ⇒ pat[j]=dat[k+j])

The invariant has been simplified because i is allowed to increment after a match has been found.

decreasing quantity: p+1- i

Note that iteration may stop before (p+1-i) =0, but it will stop when the quantity gets to 0.

Finding the location of the next match

Problem: NC(pat, dat, p, D) ∧

	(∃ *l*, (*l* > ‘k) ∧ (∀ i, 0 ≤ i ≤ p ⇒ pat[i]=dat[*l*+i]))	¬ (∃ *l*, (*l* > ‘k) ∧ (∀ i, 0 ≤ i ≤ p ⇒ pat[i]=dat[*l*+i]))
		H1
k’ |	k’ =minel({*l* | (∀ i, 0 ≤ i ≤ p ⇒ pat[i]=dat[*l*+i]) ∧ (*l* >‘k)})	***true***
m’=	**true**	**false**
i’ |	i’ = p+1	***true***
H2		G

Solution:
nextk ≡
(***it***
(k<D-p → (k⇐ k + 1; matchk;
(m → ● | ¬ m → ☛))
| k ≥ D-p → (m ⇐ false; ●))
ti)
This program will be discussed further on the following slides.

Finding the location of the next match
Is this the best we can do?

If there is no match at position k, should we simply increment k by 1 or can we go further?

- Consider the pattern "dave". If the match failed on the "e", we could increment k by 3 without missing a possible match.
- Consider the pattern "ddde". If the match failed on the "e", we must increment by 1.

This pattern match is the inner loop of many long programs. Both pattern and data can be long. We should try to make this program as efficient as possible.

In the examples above, the amount of the increment depends on the pattern alone, not the data. It is a constant for most searches.

Time invested in computing properties of the pattern will be repaid later if the program is used to search large arrays.

Computing d(f)

d(f) ≡ the increment to the first possible match.

f is number of matches before failure.

f = 0 means the first character didn't match.

f = p+1 means a successful match.

If the pattern doesn't repeat, d(f) ≥ f.

If the pattern does repeat, we must consider a possible match where the repetition begins.

For example, consider the pattern "**abcabd**".

This pattern begins to repeat at position 3.

For f = 0, 1, or 2, the repetition is irrelevant. d(0) = d(1) = 1. d(2) = 2.

For f = 3 (no second a), we should begin to compare where the match failed, (increment by 3) but, because there is no "a", we can skip one more. d(3) = 4

If we found the second a, but not b, move 4.

If we fail at the end, we start at the repetition.

If we succeed, we move 6 over.

d(4) = 4, d(5) = 3, d(6) = 6

Computing d(f)

For a pattern **abcabc** we have:

d(0:6) = 1 1 2 4 4 5 3

d(5) is different from the previous example because the pattern is a complete repetition. If we failed to find the final c, we won't find it when we shift over by 3 positions either.

The success value (d(6)) is different because the pattern is a complete repetition and might be found again beginning where we found the second a.

Repetitions "don't count" if the value we were looking for when the match failed is the same as the value we would be looking for after advancing to the next possible match.

If the match succeeded, the repetition must always count.

Computing d[f]: finding repetitions

We do not want to recompute d(f) so we will compute it and store it in an array d[0:p+1]. First, we need to find relevant repetitions.

Problem: $0 < f \leq p+1 \Rightarrow$
$r' = (\forall j, 0 \leq j < f-m \Rightarrow pat[j]=pat[j+m]) \wedge NC(f,m)$

Solution: repatm ≡
(**integer** j; j ⇐ 0; r ⇐ **true**;
it
(j < f-m → (pat[j]=pat[j+m]***;***
(# → ***(***j ⇐ j+1;☛)
|¬#→ (r ⇐ **false**; ●)***))***
| j ≥ f-m → ●)
ti)

invariant:

r=(∀i, (0 ≤ i < j) ⇒pat[i]=pat[i+m])

decreasing quantity: f - m - j

Note that this will terminate immediately if f=m. Effectively, there is a zero-length repetition at f.

Computing d(f), for a value of f greater than 0

Problem: f >0 ⇒ (
NC(f) ∧ d'[f]=
minel({m|(m>0)∧(∀j,0 ≤ j < f-m ⇒
pat[j]=pat[j+m]) ∧ (¬(pat[f-m]=pat[f]))}))

Solution: setdf≡
(**integer** m; m ⇐ 1;
it
(m ≤ f → (repatm;
(r →
(f≤p→(pat[f-m]=pat[f]→(m⇐m+1;☛)
|pat[f-m]≠pat[f]→ ●)
|f>p→ ●)
|¬r→ (m ⇐ m +1;☛)))
| m > f → ●)
ti ;
d[f] ⇐ m)

This program is discussed on the next slide.

Computing d(f), for f other than 0

Solution: setdf≡
(**integer** m; m ⇐ 1;
it
(m ≤ f → (repatm;
(r →
(f≤p→(pat[f-m]=pat[f]→(m⇐m+1;☛)
|pat[f-m]≠pat[f]→ ●)
|f>p→ ●)
|¬r→ (m ⇐ m +1;☛)))
| m > f → ●)
ti;
d[f] ⇐ m)

invariant: m ≤ minel(
{q | (q>0) ∧ (∀j,0 ≤ j < f-q ⇒
pat[j]=pat[j+q]) ∧ (¬(pat[f-q]= pat[f]))})

decreasing quantity: (f-m+1)

repatm satisfies:
0 < f≤ p+1⇒
r'=(∀j,0≤j<f-m⇒pat[j]=pat[j+m])})∧NC(f,m)

Computing d[f], for 0 < f ≤ p+1

Problem: (∀ f, 0≤f ≤ p+1 ⇒
d'[f]= minel(
{m|(m>0)∧(∀j,0≤j<f-m ⇒
pat[j]=pat[j+m]) ∧ (¬(pat[f-m]=pat[f]))}))

Solution: (d[0] ⇐ 1;
f ⇐ 1;
it
(f ≤ p+1 → (setdf; f ⇐ f +1; ☛)
| f > p+1 → ●)
ti)

loop invariant:(∀ q, 0 ≤q < f ⇒
d[q]= minel({m|(m>0)∧(∀j,0≤j<q-m ⇒
pat[j]=pat[j+m]) ∧(¬(pat[q-m]=pat[q]))}))

decreasing quantity: (p+2 -f)

Back to Pattern Matching, using d[f]

Problem: NC(pat,dat,p,d)∧(((∀ i, 0 ≤ i ≤ p ⇒ pat[i]=dat[‘k+i])∧‘i=p+1∧(∀f,0≤f≤p+1⇒d[f]=d(f)))⇒

	(∃ *l*, (*l* > ‘k) ∧ (∀ i, 0 ≤ i ≤ p ⇒ pat[i]=dat[*l*+i]))	¬ (∃ *l*, (*l* > ‘k) ∧ (∀ i, 0 ≤ i ≤ p ⇒ pat[i]=dat[*l*+i]))

H1

k’ |	k’ =minel({*l* | (∀ i, 0 ≤ i ≤ p ⇒ pat[i]=dat[*l*+i]) ∧ (*l* >‘k)})	***true***
m’=	**true**	**false**
i’ |	i’ = p+1	***true***

H2 G

Solution: nextk ≡

(***it***

(k<D-p ⟶ (k⇐ k+d[i]; matchk;

(m⟶ ● | ¬ m ⟶ ☛))

| k ≥ D-p⟶ (m ⇐ false; ●))

ti)

matchk satisfies: k ≥ 0 ⇒

((m’ = (∀ i, 0 ≤ i ≤ p ⇒ pat[i]=dat[k+i])) ∧ (k≤D-p ⇒

i’=minel({i|(0≤i≤p)∧pat[i]≠dat[k+i]}∪{p+1}) ∧ NC(pat,dat, p, D, k))

Can matchk be improved?

Problem:

‘i=minel({j |(0≤j≤p+1) ∧
¬(pat[j]=dat[k-d[‘i]+j])}) ⇒
(m’ = (∀ j, 0 ≤ j ≤ p ⇒ pat[j]=dat[k+j])) ∧
(∀ f, 0≤f≤p+1⇒d[f]=d(f))∧ (k≤ D-p ⇒

i’=minel({j |(0 ≤ j ≤ p+1)
∧¬(pat[j]=dat[k+j])}) ∧ NC(pat,dat, p, D, k)

Solution:

matchk ≡(k ≤ D-p;
(# → (i ⇐ max(0,i-d[i]); m ⇐ **true**;
it (pat[i]=dat[k+i];
(#→(i⇐ i+1; i ≤ p; (#→ ☛|¬# →●))
|¬# → (m ⇐ **false**; ●)))
ti)
|¬# → m ⇐ **false**))

loop invariant:
(m = (∀ j, 0 ≤ j < i ⇒ pat[j]=dat[k+j])) ∧
(¬m ⇒ pat[i] ≠ dat[k+i])

decreasing quantity: p+1- i

Note that iteration may stop before (p+1-i)=0.

Building Blocks for Pattern Matching

Rather than try to write the algorithm in a single step, we built these components:

- A simple program to see if a match was present at a previously specified position.
- matchk, an improved version that reported where a match first failed.
- nextk, a program that, if run after one match has been found in the data, finds the next place where there is a match and reports where it begins.
- repatm, a program that looks for repetitions in the pattern, not in the data. The repetitions must begin at m and end where the match as failed.
- setdf, a program that computes d(f) the maximum safe displacement if f marks the place where a match failed (f > 0).
- A program that computed the displacement, d(f) for all possible f and stored their values in an array d[f].
- An improved version nextk, that uses d[f].
- An improved version of matchk that does not look at places where we already know there was a match.

The final versions of nextk and matchk can be the building blocks for programs to search long files.

Lessons to be Learned

- These are useful algorithms for many applications, the algorithms are useful in themselves.
- This is not an algorithm that the average engineer or programmer would think of.
- Engineers that make frequent use of programs, or who write frequently used programs should know the literature on algorithms.
- The conditions under which an algorithm will work must be carefully specified.
- This algorithm, written out in full, would be incomprehensible for most of us.
- We have presented (developed) it in small steps and there is no real need to look at it all together.
- The reason we don't have to look at it all at once is because we have precise descriptions of the parts.
- Even with this method, programming is, and always will be a very error prone process.
- Testing is essential.
- When efficiency is important, computations should be moved out of inner loops wherever possible.
- Small improvements in an algorithm often make an invariant more complex.

Structure can always be maintained!