Imperative Programming – Statements

Steven Zeil

Sep. 15, 2003

Contents

Imperative Languages

Imperative programming languages are characterized by

- assignment
- control flow

1. Statements
2. Data Structures
3. Procedure Activation

---

1 Statements

1. Structured Programming
2. Syntax
3. Invariants

1.1 Structured Programming

1. Static versus Dynamic Properties
2. Flat Control Structure
3. Structured Programming

1.1.1 Static versus Dynamic Properties

Consider a simple sequence of statements:

```
a := 0;
b := a + 1;
print (2*b);
```

In this case, the dynamic structure of the computation mirrors the static structure of the code.

Static properties of code are any properties that can be determined by a "single pass" examination of the code, without actually executing or simulating the execution of the code.

Dynamic properties of code are properties that can only be determined by executing (or simulating the execution of) the code.

What are some examples of static code properties?

What are some examples of dynamic code properties?

1.1.2 Flat Control Structure

A **basic block** of code is a sequence of consecutive statements such that all of the statements in the sequence are executed once if and only if the first is executed.

- "straight-line" code
- no jumps into the middle
- no jumps out of the middle

- In basic blocks, the static and dynamic structure are essentially identical.
- Clearly, as we introduce conditional and looping statements, the static and dynamic structure must diverge.
  - but how much?

  Consider the following FORTRAN code:

```
      SUBROUTINE BINSEARCH (A, N, X, I)
      DIMENSION A(1)
      LOW = 1
      HIGH = N
 1    IF (LOW > HIGH) GOTO 100
      MID = (LOW + HIGH) / 2
      IF (A(MID) - X)  2, 3, 4
 2    LOW = MID + 1
      GO TO 1
 3    I = MID
      GOTO 200
 4    HIGH = MID - 1
      GO TO 1
100   I = -1
200   CONTINUE
      END
```

Early FORTRAN control flow

GO TO unconditional branch to a statement identified by a numeric label

arithmetic IF `IF (`⟨*exp*⟩`) s,t,u`

If ⟨*exp*⟩ is negative, go to `s`. If it is zero, go to `t`. If positive, go to `u`.

FORTRAN control flow (cont.)

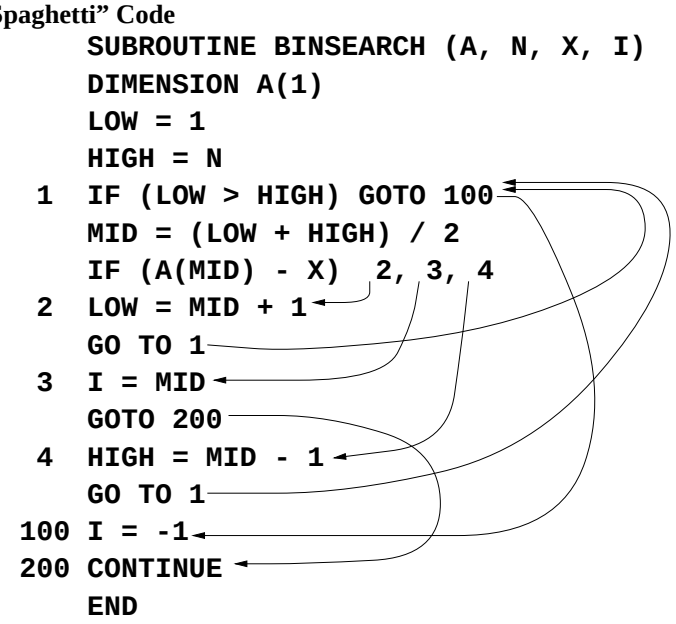
logical IF `IF (`⟨*exp*⟩`)` ⟨*stmt*⟩

If ⟨*exp*⟩ is true, then do the ⟨*stmt*⟩.

DO loop `DO s I = j, k, m`

Define a loop from here to statement `s`, with `I` initialized to `j`, incremented by `m` at the end of each loop, and loop execution ending after $\frac{k-j}{m}$ iterations.

"Spaghetti" Code



1.1.3 Structured Programming

Text: A program is **structured** if the flow of control through the program is evident from the syntactic structure of the program text.

More common: A program is **structured** if it is formed from a restricted set of single-entry, single-exit control flow constructs:

- sequencing
- selection
- iteration

Sequencing

Arranging statements into basic blocks:

⟨*stmt*⟩ ::= `begin` ⟨*sl*⟩ `end`

⟨*sl*⟩ ::=

| ⟨*stmt*⟩ `;` ⟨*sl*⟩

```
a := 1;
b := a - 1;
print 2*b;
```

Selection

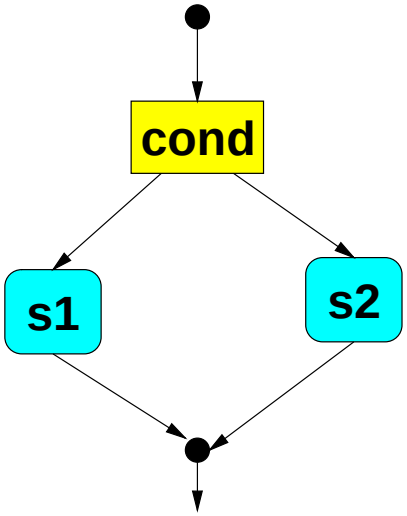
Selection refers to all control-flow statements used to choose one of several alternatives: `if`, `switch`/`case`, etc.

- Structured languages replaced FORTRAN's flat syntax with the idea of *nesting* statements.

Some alternate forms of if-then-else (Pascal, C, Ada):

⟨*stmt*⟩ ::= `if` ⟨*exp*⟩ `then` ⟨*stmt*⟩ `else` ⟨*stmt*⟩
⟨*stmt*⟩ ::= `if(` ⟨*exp*⟩ `)` ⟨*stmt*⟩ `else` ⟨*stmt*⟩
⟨*stmt*⟩ ::= `if` ⟨*exp*⟩ `then` ⟨*stmt-list*⟩
{`elsif` ⟨*stmt-list*⟩}
`else` ⟨*stmt-list*⟩ `endif;`

All are single-entry, single-exit:



Implementation of If

`if` ⟨*exp*⟩ `then` ⟨*stmt*$_1$⟩ `else` ⟨*stmt*$_2$⟩

	translation of ⟨*exp*⟩;
	if result is false goto L1;
	translation of ⟨*stmt*$_1$⟩;
	goto L2;
L1:	translation of ⟨*stmt*$_2$⟩;
L2:	...

Of course, if there is no `else` part, we just get

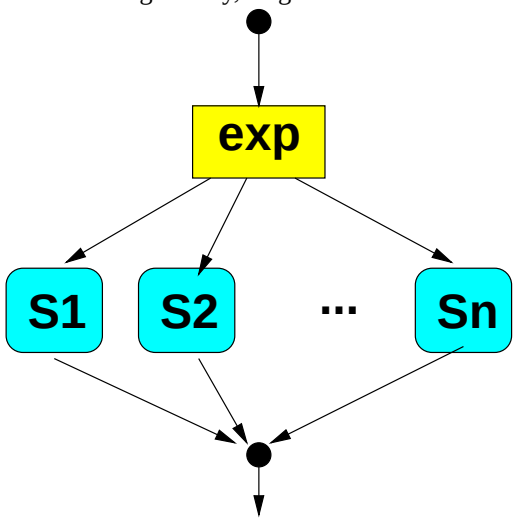
---

	translation of ⟨*exp*⟩;
	if result is false goto L1;
	translation of ⟨*stmt*$_1$⟩;
L1:	...

Many languages also feature a multi-way selection:
Pascal:

case ⟨*exp*⟩ **of**
⟨*const*$_1$⟩ : ⟨*stmt*$_1$⟩ ;
⟨*const*$_2$⟩ : ⟨*stmt*$_2$⟩ ;
⋮
⟨*const*$_n$⟩ : ⟨*stmt*$_n$⟩
end

All are single-entry, single-exit:



C's version is slightly different:

switch ⟨*exp*⟩ {
case ⟨*const*$_1$⟩ : ⟨*stmt*$_1$⟩ ; **break** ;
case ⟨*const*$_2$⟩ : ⟨*stmt*$_2$⟩ ; **break** ;
⋮
case ⟨*const*$_n$⟩ : ⟨*stmt*$_n$⟩ ;
end

Translation of these can be interesting.

- Treat as nested if-then-elses
- Jump table
- Hash table

---

case ⟨*exp*⟩ **of**
  ⟨$const_1$⟩ : ⟨$stmt_1$⟩;
  ⟨$const_2$⟩ : ⟨$stmt_2$⟩;
  ⋮
  ⟨$const_n$⟩ : ⟨$stmt_n$⟩
end

can be treated as

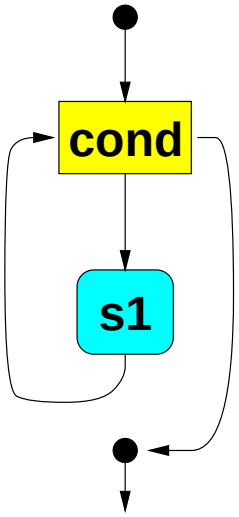
tmp := ⟨*exp*⟩;
if tmp = ⟨$const_1$⟩ **then** ⟨$stmt_1$⟩
else if tmp = ⟨$const_2$⟩ **then** ⟨$stmt_2$⟩
else
  ⋮
else if tmp = ⟨$const_n$⟩ **then** ⟨$stmt_n$⟩

---

while ⟨*exp*⟩ **do** ⟨*stmt*⟩
while (⟨*exp*⟩) ⟨*stmt*⟩
while ⟨*exp*⟩ **loop** ⟨*stmt-list*⟩ **end loop**;

(Pascal, C, Ada)

All are single-entry, single-exit:



Additional Loop Forms

repeat ⟨*stmt-list*⟩ **until** ⟨*exp*⟩
do ⟨*stmt*⟩ **while** (⟨*exp*⟩);

(Pascal, C)

loop ... **exit when** ⟨*exp*⟩; ... **end loop**;

(Ada)

Definite Loops

FORTRAN had a true definite loop:

DO ⟨*label*⟩ I = ⟨*start*⟩ , ⟨*last*⟩ , ⟨*increment*⟩

Number of iterations could be computed at loop entry, because

1. all 3 expressions were evaluated at entry
2. programs were forbidden to change the loop variable within a loop

---

Later languages adapted the DO loop, but relaxed the rules, making it indefinite.

FOR I := ⟨*start*⟩ **TO** ⟨*last*⟩ BY ⟨*increment*⟩

for I **in** ⟨*start*⟩ . . ⟨*last*⟩

(Pascal, Ada)

C's is particularly flexible:

for (⟨$stmt_1$⟩ ; ⟨*exp*⟩ ; ⟨$stmt_2$⟩) ⟨$stmt_3$⟩

translates as

⟨$stmt_1$⟩
while (!⟨*exp*⟩)
  {
    ⟨$stmt_3$⟩
    ⟨$stmt_2$⟩
  }

```
procedure BinSearch (
    var A: array[1..1000] of real;
   N: integer; X: real; var I: integer);
var  {Pascal version}
  Mid, Low, High: integer;
begin
  Low := 1; High := N;
  Mid := (Low + High) / 2;
  while (Low <= High)
        and (A[Mid] <> X) do begin
    if A[Mid] < X then
       Low := Mid + 1
    else
       High := Mid – 1;
    Mid := (Low + High) / 2
  end;
  if (Low <= High)
    I := Mid
  else
    I := –1;
end;
```

```
void binSearch (double *A, int N,
                double X, int* I)
{   /* C version */
  int mid, low, high;
  low = 0; high = N-1;
  mid = (low + high) / 2;
  while ((low <= high)
         && (a[mid] <> X)) {
    if (a[mid] < X)
       low = mid + 1;
    else
       high = mid - 1;
    mid = (low + high) / 2;
  }
  I =  (low <= high)? mid : -1;
}
```

```
procedure BinSearch (
   A: in array[<>] of Real;
   N: in Integer; X: in Real;
   I: out Integer) is
begin  // Ada version
  Mid, Low, High: integer;

  Low := 1; High := N;
  while (Low <= High) loop
    Mid := (Low + High) / 2;
  exit when A[Mid] = X;
    if A[Mid] < X then
       Low := Mid + 1
    else
       High := Mid - 1;
    end if;
  end loop;
  if (Low <= High)
    I := Mid
  else
    I := -1;
  end if;
end BinSearch;
```

To see the significance of the structured programming philosophy, compare the flows of control between these structured versions and the older FORTRAN version.

"Spaghetti" Code

```
      SUBROUTINE BINSEARCH (A, N, X, I)
      DIMENSION A(1)
      LOW = 1
      HIGH = N
    1 IF (LOW > HIGH) GOTO 100
      MID = (LOW + HIGH) / 2
      IF (A(MID) - X)  2, 3, 4
    2 LOW = MID + 1
      GO TO 1
    3 I = MID
      GOTO 200
    4 HIGH = MID - 1
      GO TO 1
  100 I = -1
  200 CONTINUE
      END
```

"Structured" Code

```
begin
  Low := 1; High := N;
  Mid := (Low + High) / 2;
  while (Low <= High) and (A[Mid] <> X) do begin
    if A[Mid] < X then
       Low := Mid + 1
    else
       High := Mid - 1;
    Mid := (Low + High) / 2
  end;
  if (Low <= High)
    I := Mid
  else
    I := -1;
end;
```

1.2 Syntax

Special problems arise determining where some statements end.

- Statement Lists
- If-then-else syntax
- Bracketing

1.2.1 Statement Lists

Pascal:

$$\begin{array}{rcl} \langle stmt \rangle & ::= & \texttt{BEGIN}\ \langle sl \rangle\ \texttt{END} \\ & | & \langle \textit{if-stmt} \rangle \mid \ldots \\ \langle \textit{if-stmt} \rangle & ::= & \texttt{IF}\ \langle exp \rangle\ \texttt{THEN}\ \langle stmt \rangle \\ & & [\texttt{ELSE}\ \langle stmt \rangle] \\ \langle sl \rangle & ::= & \langle stmt \rangle \\ & | & \langle stmt \rangle\ \texttt{;}\ \langle sl \rangle \end{array}$$

This allows

```
procedure p;
begin
  if a > b then
    a := b;
  writeln (a)
end;
```

But a common programming error would be

```
procedure p;
begin
  if a > b then
    a := b;
  writeln (a);
end;
```

Arguably, it even looks better this way!

To avoid this problem, Pascal later added

$$\langle stmt \rangle ::=$$

The empty string is an acceptable statement!

So now this is legal:

```
procedure p;
begin
  if a > b then
    a := b;
  writeln (a);
end;
```

How many statements in the outer statement list?

This is also legal Pascal:

```
procedure p;
begin
  if a > b then
    a := b;;;;
  writeln (a);;;;
end;
```

Adding the empty statement encourages another error, however.

```
begin
  if x >= 0 then
    write ('A');
  else
    write ('B');
end;
```

This leads to a syntax error, because the ';' puts two statements between the `then` and the `else`.

Pascal used ';' as a *separator*.

Most languages avoid Pascal's error-proneness by using ';' as a *terminator*, e.g., C:

$$\begin{array}{rcl} \langle stmt \rangle & ::= & \{\ \langle sl \rangle\ \} \\ & | & \langle \textit{if-stmt} \rangle \\ & | & \langle exp \rangle\ \texttt{;} \\ \langle \textit{if-stmt} \rangle & ::= & \texttt{if(}\ \langle exp \rangle\ \texttt{)}\ \langle stmt \rangle \\ & & [\texttt{else}\ \langle stmt \rangle] \\ \langle sl \rangle & ::= & \langle stmt \rangle \\ & | & \langle stmt \rangle\ \langle sl \rangle \end{array}$$

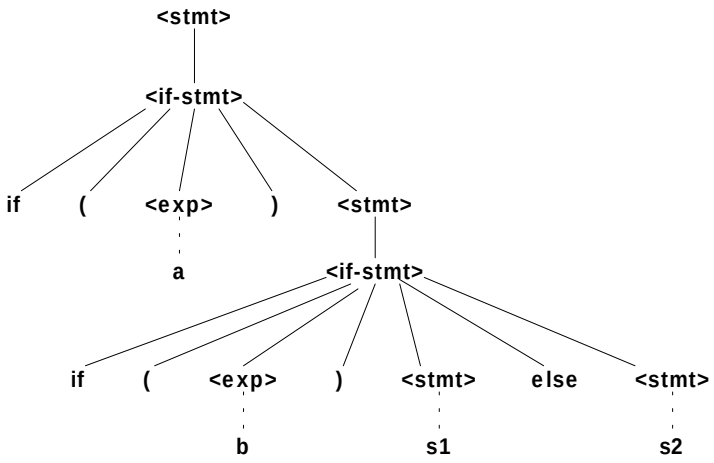
1.2.2 If-then-else syntax

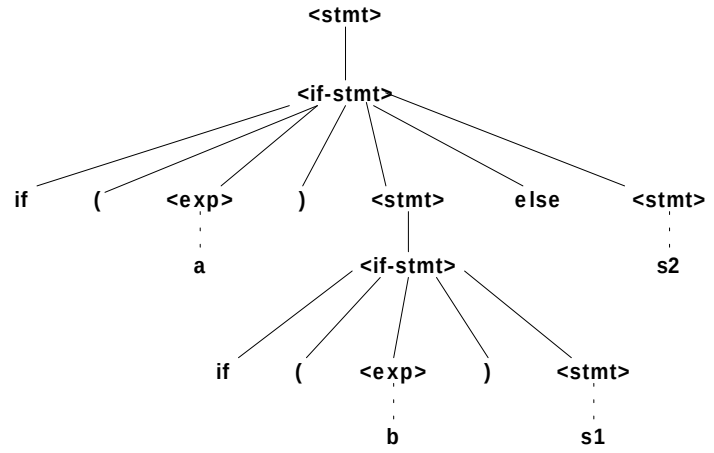
The simplest way to capture the structure of C (and Pascal) `if` statements is:

$$\begin{array}{rcl} \langle stmt \rangle & ::= & \langle \textit{if-stmt} \rangle \mid \ldots \\ \langle \textit{if-stmt} \rangle & ::= & \texttt{if(}\ \langle exp \rangle\ \texttt{)}\ \langle stmt \rangle \\ & | & \texttt{if(}\ \langle exp \rangle\ \texttt{)}\ \langle stmt \rangle \\ & & \texttt{else}\ \langle stmt \rangle \end{array}$$

But how can we parse

```
if (a) if (b) s1 else s2
```





This simple grammar:

$$\begin{array}{rcl} \langle \textit{stmt} \rangle & ::= & \langle \textit{if-stmt} \rangle \mid \ldots \\ \langle \textit{if-stmt} \rangle & ::= & \texttt{if(}\ \langle \textit{exp} \rangle\ \texttt{)stmt} \\ & \mid & \texttt{if(}\ \langle \textit{exp} \rangle\ \texttt{)else}\ \langle \textit{stmt} \rangle \end{array}$$

turns out to be ambiguous.

Although the grammar can be rewritten to remove the ambiguity

- it's not easy
- the resulting productions are unintuitive
- the ambiguity of the grammar may well reflect a weakness in the language design

If terminators

More modern languages resolve this problem by terminating the `if` with a reserved word.

$$\begin{array}{rcl} \langle \textit{stmt} \rangle & ::= & \langle \textit{if-stmt} \rangle \mid \ldots \\ \langle \textit{if-stmt} \rangle & ::= & \texttt{if}\ \langle \textit{exp} \rangle\ \texttt{then sl end if;} \\ & \mid & \texttt{if}\ \langle \textit{exp} \rangle\ \texttt{then sl else}\ \langle \textit{sl} \rangle \\ & & \texttt{end if;} \end{array}$$

1.2.3 Bracketing

The `if...end if` is an example of bracketing.

Bracketing helps define the nested structure of languages.

Pascal and C use a single bracketing construct (`BEGIN...END` and {...}).

Mismatched Brackets

What happens if a programmer leaves out a closing bracket?

```
{   /* C version */
  int mid, low, high;
  low = 0; high = N-1;
  mid = (low + high) / 2;
  while ((low <= high)
         && (a[mid] <> X) {
    if (a[mid] < X)
      low = mid + 1;
    else
      high = mid - 1;
    mid = (low + high) / 2;
  /* missing } */
  I =  (low <= high)? mid : -1;
}

void nextFunction()
{
```

Ada adopted distinct bracketing schemes for each construct.

```
procedure BinSearch (\ldots) is
begin  // Ada version
  Mid, Low, High: integer;

  Low := 1; High := N;
  while (Low <= High) loop
    Mid := (Low + High) / 2;
```

```
    exit when A[Mid] = X;
      if A[Mid] < X then
         Low := Mid + 1
      else
         High := Mid - 1;
      end if;
    end loop;
    if (Low <= High)
      I := Mid
    else
      I := -1;
    end if;
end BinSearch;
```

This allows most bracketing errors to be caught at the end of the next bracket.

1.3 Invariants

An **invariant** is a property that holds every time execution reaches a designated location in the code.

Invariants are useful in

- proving algorithms correct
- designing them correctly in the first place
- documenting the details of their operation

1.3.1 Who cares?

How do you know when your code is ready to release?

- when you're done testing it

Who cares? (cont.)

How do you know when you're done testing it?

- when you're bored with testing
- when you can't think of any more good tests
- when you've exhausted your budget for testing
- when it's the day before the Marketing dept. announced the product would be released
- when you've satisfied a scientifically chosen testing criterion
- when the code is bug-free

Testing is a sampling process, and no matter how well done, will not guarantee the code to be bug-free.

So what do you do?

- Use good testing practices, and hope for the best
- Employ statistical models of software reliability to predict when the code is "good enough"
- Prove the code correct
- Try to avoid bugs in the first place.

Editorial opinion:

Proving programs correct is most useful only where other techniques can't be applied (e.g., security)

But understanding how a program *could* be proven correct is an aid to designing it correctly.

1.3.2 Pre- and Postconditions

A **precondition** is an invariant attached just before the start of a construct. It describes the conditions under which that construct is supposed to work correctly.

A **postcondition** is an invariant attached just after a construct. It describes what that construct was supposed to accomplish.

You may have seen these ideas applied to documenting subroutines:

```
void binSearch (double *A, int N,
                double X, int* I)
/*
 * pre:  A contains N elements,
 *         sorted into ascending order
 * post: if X is in A[0..N-1],
 *         then A[I]==X
 *       if X is not in A[0..N-1],
 *         then I<0
*/
{...
```

These illustrate some of the cardinal rules of invariants:

- every invariant is a boolean condition
- every invariant should be checkable

I personally would recommend

- Try to express every invariant in a form from which it would be easy to write code to check it.
- Invariants should describe dynamic behavior, not repeat obvious static information.

```
void binSearch (double *A, int N,
                double X, int* I)
/*
 * pre:  for all 0<j<N,  A[j]>=A[j-1]
 * post: if there exists j, 0<=j<N,
 *         such that A[j]==X,
 *       then A[I] == X
 *       else I<0
 */
{...
```

Statement-Level Invariants

If we write invariants between statements, then one invariant may serve as

- the pre-condition for the following statement and
- the postcondition for the following one

```
mid = (low + high) / 2;
while ((low <= high) && (a[mid] <> X)) {
  /* {(low <= high) && (a[mid] <> X)
       && (mid == (low + high) / 2} */
  if (a[mid] < X)
```

Proof Rules for Partial Correctness

A program is **(totally) correct** if, given any input that satisfies the program's precondition, the program eventually halts with output that satisfies the program's postcondition.

A program is **partially correct** if, given any input that satisfies the program's precondition and for which the program eventually halts, the output satisfies the program's postcondition.

Notation

We will write general patterns of rules for statements as

$$\{\langle precond \rangle\}\ \langle stmt \rangle\ \{\langle postcond \rangle\}$$

We often use P, Q, and R to denote invariants, S for statements. We use a vertical bar to "givens" from "conclusions".

For example, we might write

$$\frac{x > 0}{\{y > 0\}\ y = x + y;\ \{y > 0\}}$$

Composition (Stmt Lists)

$$\frac{\{P\}\ S_1\ \{Q\}, \{Q\}\ S_2\ \{R\}}{\{P\}\ S_1; S_2\ \{R\}}$$

In other words,

- if
  - when P is a precondition for statement S_1, Q is a postcondition, and
  - when Q is a precondition for statement S_2, R is a postcondition,
- then when P is a precondition for the statement sequence $S_1; S_2$, R is a postcondition.

Selection (Conditionals)

$$\frac{\{P \wedge E\}\ S_1\ \{Q\}, \{P \wedge \neg E\}\ S_2\ \{Q\}}{\{P\}\ \text{if } E \text{ then } S_1 \text{ else } S_2\ \{Q\}}$$

While Loops

$$\frac{\{P \wedge E\}\ S\ \{P\}}{\{P\}\ \text{while } E \text{ do } S\ \{P \wedge \neg E\}}$$

P is a **loop invariant**, an invariant condition that is true upon entry to the loop, at the start of each iteration, and upon exit.

Finding a loop invariant is considered the key to understanding what a loop actually does.

Designing With Invariants

So what does all this buy us? let's look at the binary search code:

```
procedure BinSearch (
    var A: array[1..1000] of real;
    N: integer; X: real; var I: integer);
var  {Pascal version}
  Mid, Low, High: integer;
begin
  Low := 1; High := N;
```

```
  Mid := (Low + High) / 2;
  while (Low <= High)
        and (A[Mid] <> X) do begin
    if A[Mid] < X then
      Low := Mid + 1
    else
      High := Mid - 1;
    Mid := (Low + High) / 2
  end;
  if (Low <= High)
    I := Mid
  else
    I := -1;
end;
```

Are you absolutely sure that `Low` and `High` will never get "stuck" at the same values on two successive iterations? Not even when

- `Low = High`
- `Low + 1 = High`
- `Low + 2 = High`
- each of the above, and `Low` is even? odd?
- each of the above, and `High` is even? odd?

```
procedure BinSearch (
   var A: array[1..1000] of real;
   N: integer; X: real; var I: integer);
var  {Pascal version}
  Mid, Low, High: integer;
begin
  Low := 1; High := N;
  /* {X is not in A, or
   (A[j] == X for some Low<=j<=High)} */
  Mid := (Low + High) / 2;
  while (Low <= High)
        and (A[Mid] <> X) do begin
    if A[Mid] < X then
      Low := Mid + 1
    else
      High := Mid - 1;
    Mid := (Low + High) / 2
  end;
  if (Low <= High)
    I := Mid
  else
    I := -1;
end;
```

```
procedure BinSearch (
   var A: array[1..1000] of real;
   N: integer; X: real; var I: integer);
var  {Pascal version}
  Mid, Low, High: integer;
begin
  Low := 1; High := N;
  /* {X is not in A, or
   (A[j] == X for some Low<=j<=High)} */
  Mid := (Low + High) / 2;
  /* {(X is not in A, or
     (A[j] == X for some Low<=j<=High))
     and (Mid = (Low + High / 2)} */
  while (Low <= High)
        and (A[Mid] <> X) do begin
    if A[Mid] < X then
      Low := Mid + 1
    else
      High := Mid - 1;
    Mid := (Low + High) / 2
  end;
  if (Low <= High)
    I := Mid
  else
    I := -1;
end;
```

The condition

```
/* {(X is not in A, or
   (A[j] == X for some Low<=j<=High))
   and (Mid = (Low + High / 2)} */
```

is our loop invariant.

For shorthand, call it `Inv`.

```
procedure BinSearch (
   var A: array[1..1000] of real;
   N: integer; X: real; var I: integer);
var  {Pascal version}
  Mid, Low, High: integer;
begin
  Low := 1; High := N;
  /* {X is not in A, or
   (A[j] == X for some Low<=j<=High)} */
  Mid := (Low + High) / 2;
  /* {Inv} */
  while (Low <= High)
        and (A[Mid] <> X) do begin
    /* {Inv and (Low <= High)
        and A[Mid] <> X} */
    if A[Mid] < X then
```

```
      Low := Mid + 1
    else
      High := Mid - 1;
    Mid := (Low + High) / 2
  end;
  /* {Inv and (Low > High
        or A[Mid] = X)} */
  if (Low <= High)
    I := Mid
  else
    I := -1;
end;
```

The post-loop invariant makes it clear that, if we exit the loop, the program will return the correct value in `I`:

```
/* {(X is not in A, or
   (A[j] == X for some Low<=j<=High))
   and (Mid = (Low + High / 2)
   and (Low > High
        or A[Mid] = X)} */
```

simplifies to

```
/* {(X is not in A and Low > High)
  or
   (Low <= High and A[Mid] = X)} */
```

Looking at the middle of the loop, we have

```
while (Low <= High)
      and (A[Mid] <> X) do begin
  /* {Inv and (Low <= High)
       and A[Mid] <> X} */
  if A[Mid] < X then
    Low := Mid + 1
  else
    High := Mid - 1;
  Mid := (Low + High) / 2
end;
```

Since `Low <= High`, we know that `Low+High >= 2*Low`, so `Mid >= Low`. Similarly, we can argue that `Mid <= High`.

Let d denote the distance between `High` and `Low` at the beginning of the loop.

```
while (Low <= High)
      and (A[Mid] <> X) do begin
  /* {Inv and (Low <= High)
       and A[Mid] <> X
       and d = High-Low} */
  if A[Mid] < X then
    Low := Mid + 1
  else
    High := Mid - 1;
  Mid := (Low + High) / 2
end;
```

Now because `Low <= Mid`, we have `Low < Mid + 1`. So we can argue that

```
while (Low <= High)
      and (A[Mid] <> X) do begin
  /* {Inv and (Low <= High)
       and A[Mid] <> X
       and d = High-Low} */
  if A[Mid] < X then
    Low := Mid + 1
    /* {d > High-Low} */
  else
    High := Mid - 1;
  Mid := (Low + High) / 2
end;
```

because we know that `Low` just increased.

We can make a similar argument for `High`:

```
while (Low <= High)
      and (A[Mid] <> X) do begin
  /* {Inv and (Low <= High)
       and A[Mid] <> X
       and d = High-Low} */
  if A[Mid] < X then
    Low := Mid + 1
    /* {d > High-Low} */
  else
    High := Mid - 1;
    /* {d > High-Low} */
  Mid := (Low + High) / 2
end;
```

---

and then the rule for conditionals lets us say:

```
while (Low <= High)
      and (A[Mid] <> X) do begin
  /* {Inv and (Low <= High)
       and A[Mid] <> X
       and d = High-Low} */
  if A[Mid] < X then
    Low := Mid + 1
    /* {d > High-Low} */
  else
    High := Mid - 1;
    /* {d > High-Low} */
  /* {d > High-Low} */
```

```
    Mid := (Low + High) / 2
  end;
```

So at the end of each loop iteration, we are guaranteed that the distance between `Low` and `High` has actually been reduced.

This means that our loop will exit eventually.

OK, who are you trying to kid?

No, I don't do that in detail when I design algorithms.

But I do think in those terms:

- What is the loop invariant?
- What critical quantities do I need to track to tell if things are working properly?
  - Should those quantities be increasing, decreasing, or staying the same?

Invariants and Debugging

Programmers use invariants during testing and debugging by turning them into **assertions**, programming language code that checks an invariant and issues an error message if the check fails.

The `assert(e)` macro in C and C++ converts to

```
if (!e)
  error("assertion␣violated␣in␣line␣...
```

unless the compiler flag `NDEBUG` is set.

If that flag is set, the assertion is converted into a comment.

```
void binSearch (double *A, int N,
                double X, int* I)
{   /* C version */
  int mid, low, high, d;
  low = 0; high = N-1;
  mid = (low + high) / 2;
  while ((low <= high) && (a[mid] <> X)) {
    assert((d = high-low) >= 0);
    if (a[mid] < X)
       low = mid + 1;
    else
       high = mid - 1;
    assert (d > high-low);
    mid = (low + high) / 2;
  }
  I =  (low <= high)? mid : -1;
}
```

In other languages, assertions are formed using conventional if's.

It's not as easy to make them go away, however, when compiling the "production" version.