1 Loop Invariants

26 August 1993

1. Control Structures
   (a) Straight line
   (b) Branch: `case`, `if then else`.
   (c) Loop: `for`, `repeat until`, `while do`.
2. Loops
   (a) `( r S u C ) <==> ( S ; w ~C d S )`
   (b) `( w C d S ) <==> ( if C { r S u ~C })`
   (c) `( for i=I to F do S ) <==> ( i=I ; w i<=F do {S;i++})`
   (d) So `while` can simulate anything, a `repeat` + `if` = `while`, and `for` is incable of inifite regress.
   (e) Proof: transformation of flow charts:
   (f) break statement, using flags, forcing the termination condition, goto's and returns. *Example:* For each thing in a list, do something; how to break out on error doing something?
3. Subroutines and Goto's: How to they fit in?
   (a) As straight-line code:

   > *The Golden Rule of Subroutines:* Call a subroutine when passing from one level of detail to another.

   (b) As loops: Recursion.
   (c) Not at all, goto's and spaghetti code, they just don't fit in, and so are abandoned.
4. Loop Invariants
   (a) Format: `{assert I} ; while C do { S; assert I };`
   (b) Loop Invariant $\mathcal{I}$, true at assertion.
   (c) Termination: advance towards $\mathcal{C}$ false.
   (d) Goal: invariant + termination = goal.
   (e) Initialization.

5. Invariant Example, max integer in array `A[1..n]`.
   (a) Invariant: $M = \max\{\, A[j] \mid 1 \le j \le i \,\}$.
   (b) Discussion: choice of invariant driven by the fact:
   There is no such thing as the maximum of a empty set.
   (c) Termination: $i = n$, advance i by one each time through the loop.
   (d) Goal, M is maximum in array.
6. Comment on Zeno's Paradox: Let $\mathcal{I}$ be Achilles' arrow is at distance d, then consider: `d=0; while d<1 do d=d+(1-d)/2;`

2 Loop Invariants continued

31 August 1993

1. Loop Invariants, another example: Find first negative number in an array.
   (a) Invariant: All numbers in $\{\, A[j] \mid 1 \le j < i \,\}$ are positive.
   (b) Termination: $i > n$ or $A[i] < 0$, increase i by one each time through the loop.
   (c) Goal = invariant + termination.
   (d) Initialization: $i = 1$.
   (e) Notes:
      i. Choice of invariant driven by the fact that the answer could be "none."
      ii. Lack of short-circuited AND in Pascal.
      iii. Sentinel bound: Let $A[n+1] = -1$.
2. Extended Example: greatest common divisor of two integers.
   (a) Divisibility: $\forall\, a, b \in \mathbf{Z},\ (a \mid b \iff \exists k \in \mathbf{Z} : ak = b)$.
   (b) Common Divisor: $\forall\, a, b \in \mathbf{Z},\ \mathrm{cd}(a, b) = \{\, c \in \mathbf{Z} \mid c \mid a, b \,\}$.
   (c) Greatest Common Divisor: $\forall\, a, b \in \mathbf{Z},\ \gcd(a, b) = \max \mathrm{cd}(a, b)$.
      i. $\gcd(2, 4) = 2,\ \gcd(a, ka) = a$.
      ii. $\gcd(5, 7) = 1,\ \gcd(p_1, p_2) = 1$, for two distinct primes. In general, an a, b such that $\gcd(a, b) = 1$ are called *relatively prime.*
      iii. $\gcd(1547, 560) = 7$.
   (d) Notes:
      i. $\forall\, a \in \mathbf{Z},\ 1 \mid a$, hence $\mathrm{cd}(a, b)$ is never empty.
      ii. $\forall\, a \in \mathbf{Z},\ a \mid 0$, hence $\mathrm{cd}(0, 0) = \mathbf{Z}$. We shall *define* $\gcd(0, 0) = 0$.

iii. $\forall\, c \in \mathrm{cd}(a,b),\ c \mid \gcd(a,b)$.

(e) Euclidean Algorithm:

Keep replacing the larger of a, b by their difference, until the answer becomes obvious.

(f) Key lemma: $\forall a, b \in \mathbf{Z},\ \gcd(a-b,b) = \gcd(a,b)$. *Proof:* Show $\mathrm{cd}(a,b) \subseteq \mathrm{cd}(a-b,b)$ and then that $cd(a-b,b) \subseteq \mathrm{cd}(a,b)$.

(g) Invariant: Let C be the gcd we seek. We keep integers a, b according to the invariant: $C = \gcd(a,b)$ and $a \geq b \geq 0$.

(h) Termination: $b = 0$, we advance by decreasing the sum $a + b$ each time through the loop.

(i) Goal = invariant + termination: $C = (a, 0) = a$.

(j) Initialization: Make invariant true by setting a and b to the input values, perhaps sign corrected and swaped.

3 Text files and strings

2 September 1993

1. Files, general files, streams, text files.
2. Assignment compatibility of files.
3. Text Files: *reference page 233–235, Oh! Pascal!*
   (a) A text file is a sequence of characters including special end-of-line and end-of-file characters.
   (b) File always ends with eoln eof sequence.
   (c) Each open file has a cursor over the next character to read.
   (d) Boolean valued function eoln is true if cursor over an eoln; eof is ture if cursor over an eof.
   (e) What character is at the cursor position of an eoln or eof is implementation dependent.
      i. In Standard Pascal, eoln is a blank and it is illegal to read and eof.
      ii. In Turbo Pascal, eoln is the two character sequence crlf (chr(13) chr(10)) and eof is a control-Z (chr(26)).
   (f) Read will not read past an eoln, readln must be called; write will not write an eoln, writeln must be called.
   (g) Example:

```
while not eof do begin
   while not eoln do begin
      read(Ch) ; write(ch) ; end ;
   readln ; writeln ; end ;
```

4. Strings. `var s : string [MAXLENGTH];` : *reference page 106, 301–303, Oh! Pascal!*
   (a) A string is "almost" an array of characters.
   (b) Strings have maximum lengths and current lengths.
   (c) Maximum lenghts do not affect type.
   (d) `length(s) : integer` is the string's current length.
   (e) `s[i] : char` is the current actual i-th character of the string, for $1 \leq i \leq$ `length(s)`
   (f) Other SYSTEM functions on strings are:
      i. `Concat(S1,...)->string`.
      ii. `Copy(S,Indx,Cnt)->string`.
      iii. `Insert, Delete, Pos`.
5. Reading words: Case study of the program on pages 241–242 of *Oh! Pascal!*.
   (a) Invariant: Exist `count` completed words to the left of the cursor; and `InAWord` is true iff cursor was just over a currently incomplete word.
   (b) Termination: advance one character towards the end of file.
   (c) Goal: assuming eof is after a eoln, count words because we completed the last word.
   (d) Initialization: no words and we were not over a completed word.
   (e) Each time through the loop:
      i. Case non-blank: We check that InAWord is true.
      ii. Case blank: We check that InAWord false, and if word just ended that count is incremented. By the invariant, word just ended if InAWord was true last time. That is, whenever InAWord goes from True to false.
   (f) Note filter style of the cascade of the two conditionals.
6. Notes:
   (a) Students didn't believe all Pascal's had strings.
   (b) Some students didn't remember arrays.

(c) Needed to introduce ASCII better.

(d) Some students didn't believe that a text file in Pascal needed an eoln before an eof. If not, readln is useless.

I should have done it this way:

(a) Characters and Ascii.

(b) Arrays and strings.

(c) Text files w/ specific advice for Ultrix, VMS and Turbo.

7. Streams are not supported in PASCAL!

4 Pointers

7 September 1993

1. Memory model. address → cell.
2. Cell contains an integer of fixed maximum size, or one symbol from a fixed finite set.
3. Address identifies in some way the cell, and are sequenced. Can think of the addresses as being integers.
4. Byte: often a cell is a byte. Everything is built up from bytes, and certain composites of consecutive bytes can be used just as easily as bytes.

```
8 oz   = 1 cup      2 cups  = 1 pint
8 bits = 1 byte     2 bytes = 1 word
```

   Also, quart is a longword; a half-gallon is a quad-word.

5. What is a byte? byte = char = ASCII.
6. What is a word? Integers, positive and negative, however:

```
MAXINT + 1 -> (-MAXINT)-1
2 * MAXINT -> -2
```

7. What is a quad word? Floating point computations.
8. What is an address? A variety of approaches which Pascal trys to make uniform.

   (a) Linear memory: an address is a big integer. *Ex.* MC68000, it is a long-word.

(b) Segmented memory: an address is a pair of integers which are not independent. *Ex.* Intel 8086, a pair of words which are added according to $a * 16 + b$.

(c) Trade-offs: speed and size in favor of Intel approach, ease of programming in favor of 68000 approach.

(d) Pages and Virtual: no fixed correlation between address and cells. Hardware negotiates invisibly to programmer. *Advantages:* Large, sparse memory space, use of fast expensive memory for some of the memory space.

9. How to compilers use addresses?

10. Address of variables: *Address as a static entity.*

    (a) Maintains a memory map for "dataland".

    (b) Assigns variables to a fixed address and reserves enough consecutive bytes for the variable.

    (c) This is why *define before use* and *type* are required.

    (d) Name of variable is turned into a use of its address at compile time.

11. Arrays and records, *Address as objects to calculate with.*

    (a) Arrays: uses the fact that memory addresses can be treated as integers. Assigns a *base address* and enough bytes for the entire array. Inserts code to calculate address of array element from base, index, and element size.

    (b) Non-zero based and multidimensional arrays.

    (c) Records: offset needed because it could be an array of records.

12. VAR parameters and *Indirect Addressing. Addresses as things you can store.*

(*) Recursion and stack based local var.

13. The Heap: a memory space available to the user to dynamically create variables while the program is running.

    (a) Getting memory, memory management. (Assume infinite, no reuse.)

    (b) Returning to the user a *pointer* to the memory.

    (c) A Pointer: A variable which stores an address.

    (d) The type of a pointer includes the type of what it points to. This is to make it harder to make mistakes.

    (e) There is the pointer, and what the pointer points to. There is also the address of where the pointer is stored. This last is hidden, the first is simply the pointer's name. The second is indirect addressing and uses the dereferencing arrow `^`.

5 Pointers

9 September 1993

1. New, Dispose and Nil.
2. Syntax: `var a : record a: integer ; end ;`
3. Syntax: `var p : ^ integer ;`
4. Syntax: `p^ := 1 ;`
5. Assignment compatibility of pointers. Based on the strict compatibility of what they point to.
6. Note: if "pointer to" introduced a new type, then it would be impossible to have a record include a pointer to itself.

6 Linked Lists

14 September 1993

1. A linked lists, the data structure.
2. Adding to ehad, traversing and printing.
3. General insertion and dummy headers.
4. Deletion.
5. Variants: last pointers, circular lists, doubly linked lists.
6. DOS file layout and the File Allocation Table.

7 Boolean Logic

16 September 1993

1. Boolean Algebra
   (a) Boolean variables receiving boolean valued expressions.
   (b) Operations AND, OR and NOT.
   (c) Absorptive Laws:
      i. $A \wedge (A \vee B) = A$.
      ii. $A \vee (A \wedge B) = B$.
      iii. *Proof:* Truth table method of proof.

(d) DeMorgan's Theorem:

i. $\neg(A \vee B) = \neg A \wedge \neg B$.

ii. $\neg(A \wedge B) = \neg A \vee \neg B$.

iii. *Proof:* Venn Diagram: a short version of Truth Table. Four points and circles enclosing points true for sentence.

(e) Distributive Laws:

i. $A \vee (B \wedge C) = (A \vee B) \wedge (A \vee C)$.

ii. $A \wedge (B \vee C) = (A \wedge B) \vee (A \wedge C)$.

iii. *Proof:* Venn Diagram: three circles enclosing eight points.

2. Examples:

   Numbers which are odd and positive are exactly those which are neither even nor negative.

   (a) $O(n) \wedge n > 0 = \neg(E(n) \vee n \leq 0) = \neg((E(n) \vee n = 0) \vee n < 0) = \neg(E(n) \vee n < 0)$.

   (b) The complement of the set $1, 3, 5, \ldots$ is the union of evens, $0, 2, 4, \ldots$ and negatives, $-1, -2, -3, \ldots$.

   However, the even and positive numbers do not include all which are neither odd nor negative, because zero should not be not included.

3. Stacks, Sedgewick's example ... didn't work.

8 Personal Summary

Pointers: best perhaps to keep to a cell and pointer model, where the cell is an array and can have certain addressing calculations. That is, it is important to see the cell as cut out of a linear space, but it is easier to draw as a cell.

Begin with a variable i. It has a name, a place in memory, and this place has an address. The name and the address are one. They are written beside a cell representing the memory location. Using a variable depends on whether it appears at the left or right of an assignment. Its use could be a retrieve at the location or a store to the location. The value of a variable is implicitely the retrieve from the location.

An array is a bunch of places starting at a location. The name could be synonymous with the address of the first cell. The data is placed into the cell in order so that arithmetic gives you direct access. Records and arrays of records are similar.

Indirect addressing is when you store the location in a variable. The variable is called a pointer.