1. Definition ... a finite set of unambiguous, executable instructions that will ultimately terminate if followed.
   - Imperative languages
2. Algorithmic Discovery
   - Top Down, Stepwise refinement ...
     - do the “clear beepers out of corners of the room” problem
   - Bottom Up ... same problem ...
   - Modular Design ...
     - DEFINE-NEW-INSTRUCTION turnright AS ITERATE 3 TIMES turnleft;
   - Pre and Post Conditions
     - At beginning of cycle, robot is along wall facing next wall
     - At the end of cycle the beeper is cleared, robot has turned the corner to face the next wall

3. Algorithmic Representation
   - Pseudo-Code (go-to-the-wall)
   - Flow Chart
   - Mathematical Statement \[ n_0 = 1; n_i = i \cdot n_{i-1} \quad i = 1..50 \]
   - Syntactically correct instructions (WHILE front-is-clear DO move;)
   - Depends on elements of the language

4. Elements of the Language
   - Primitives (move, turnleft, pickbeeper, putbeeper, turnoff)
   - Flow Control
     - Sequence
     - Branching (IF-THEN-ELSE)
       \[ \text{Testing} ... \text{front-is-clear, facing-north, any-beepers-in-beeper-bag, etc} \]
     - Loops (ITERATE n TIMES, WHILE-DO)
   - Block Structuring BEGIN .. END
   - Modularization … functions, procedures (DEFINE-NEW-INSTRUCTION)
5. Karel is an “interpreted” language
   - compilation
   - interpretation
   - advantages and disadvantages of each

6. Sample translation into DRC assembly
   While front-is-clear Do
   Begin
     move;
   End;

   Partial solution ... *very inefficient* ... for concept demo purposes only
Data structure for wall

- keep n-s walls and e-w walls separate
- n-s walls ... memory location = Street*10+avenue
- 1 if present, 0 if absent

Wall structure in memory

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>20</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>30</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>40</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>50</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
Note ... the program will take up a lot of space, more than 10 memory locations ... therefore we’d have to locate the program in “high memory” for example in location 1000 and put a JPM 1000 at location 0. Obviously, the current memory of DRC is not sufficient to do this problem except in concept.
In the program:

Registers 0, 1 and 2 hold karel’s position and direction
  • R0 holds the street,
  • R1 holds the avenue,
  • R2 holds the direction (n=0, e=1, s=2, w=3)

To calculate the memory addresses we need to multiply street values by 10 … therefore we need a multiply instruction
  • this can be done by extending the language or by creating a special routine that we do each time it is needed … put values in the registers, jump to the routine, return to the original location
  • we’ll opt for creating an instruction
    o mul Ra, Rb, Rc Ra <- Rb * Rc 17RRR
Translation into DRC of ... WHILE front-is-clear DO move;  
Note: efficiency can be improved for 
east/west movements by incrementing/decrementing R3 (the memory address) and jumping to the ldr instruction.

beq R2, north
ldi R9, 1
sub R9, R2, R9
beq R9, east
ldi R9, 2
sub R9, R2, R9
beq R9, south
ldi R9, 3
sub R9, R2, R9
beq R9, R2, west

north: ... # lines of code if karel is facing north
...

east:  ldi R4, 10  #code if karel is facing east

next:  mul R3, R0, R4
       add R3, R1, R3
       ldr R4, R3
       bne R4, stop
       inc R1
       jmp next

stop:  stp

south:...  # lines of code if karel is facing south
...

west: ...  # lines of code if karel is facing west
...

// Check the solution using as a starting point ... R0 = 2, R1 = 1, R2 = 1
Representation of a language ... using Karel as an example

- Two common forms
  - Bubble Diagrams
  - BNF (Backus-Naur Form)
  - advantages and disadvantages
BNF for Karel

BNF is the Backus-Naur Form of describing a programming language

::= this symbol is called a production. Each definition in BNF is called a production.

< > words within the angle braces are names of objects

| a symbol which represents alternative choices

[ ] anything within square braces is repeated 0 or more times

program ::= BEGINNING-OF-PROGRAM
    [<new-instruction>]
    BEGINNING-OF-EXECUTION
    <instruction> [<instruction>]
    END-OF-EXECUTION
    END-OF-PROGRAM

<new-instruction> ::= DEFINE-NEW-INSTRUCTION <name> AS
    BEGIN <instruction> END ;

(instruction) ::= <primitive> | <conditional> | <repeat> | <new-instruction> *

<primitive> ::= move; | turnleft; | pickbeeper; | putbeeper; | turnoff;

<conditional> ::= IF <test> THEN BEGIN <instruction> END
    ELSE BEGIN <instruction> END;
    IF <test> THEN BEGIN <instruction> END;
<repeat> ::= WHILE <test> DO BEGIN <instruction> END;
          | ITERATE <integer> TIMES BEGIN <instruction> END;

<test> ::= <side>-is-clear | <side>-is-blocked | next-to-a-beeper |
         not-next to a beeper | facing-<direction> | not-facing-<direction> |
         any-beepers-in-beeper-bag | no-beepers-in-beeper-bag

<side> ::= front | left | right

<direction> ::= north | south | east | west

<name> ::= <letter> [<character>]

<letter> ::= a..z | A..Z

<character> ::= <letter> | <digit> | -

<digit> ::= 0..9

<integer> ::= <digit> [<digit>]    // practical limit of 5 digits

* a new instruction must be previously defined to be considered an instruction in this production.