An introduction to CONSTRAINT HANDLING RULES

Jon Sneyers

August 2010

PART ONE

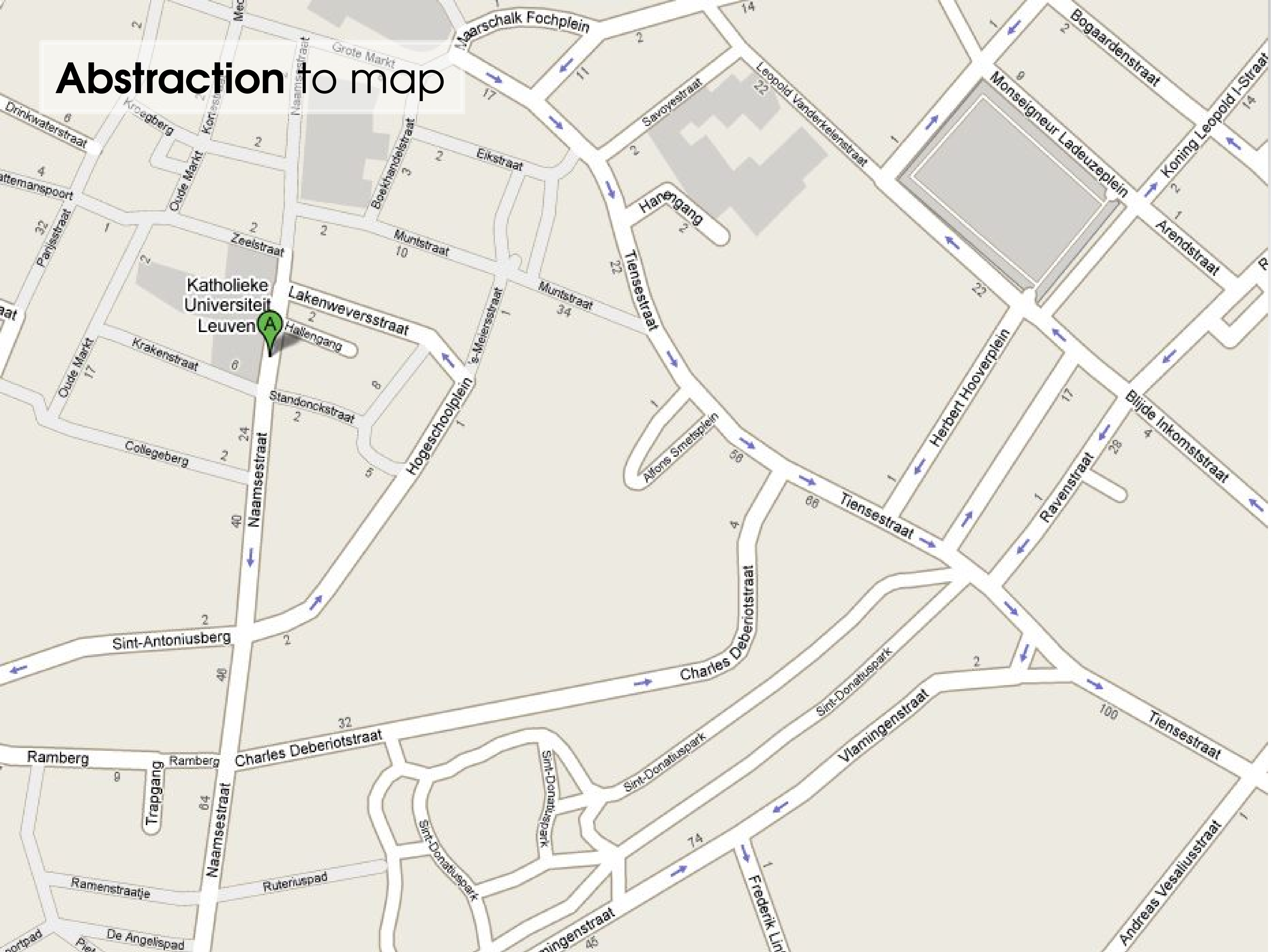
Introduction

How to get from A to B ?

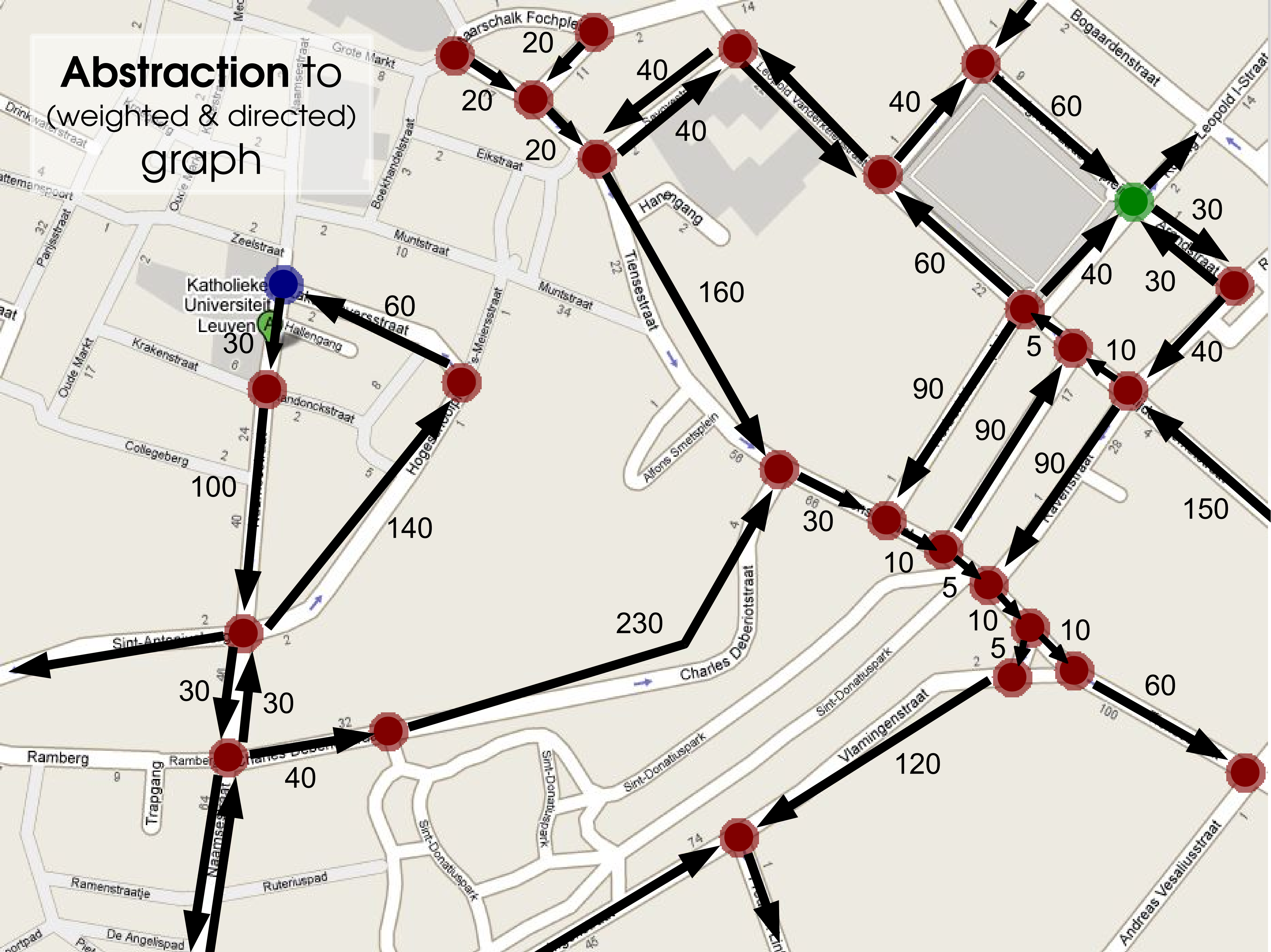
point A : Universiteitshallen
Naamsestraat 22, Leuven, Belgium


A
B
shortest path
(by car)
according to
Google
Maps

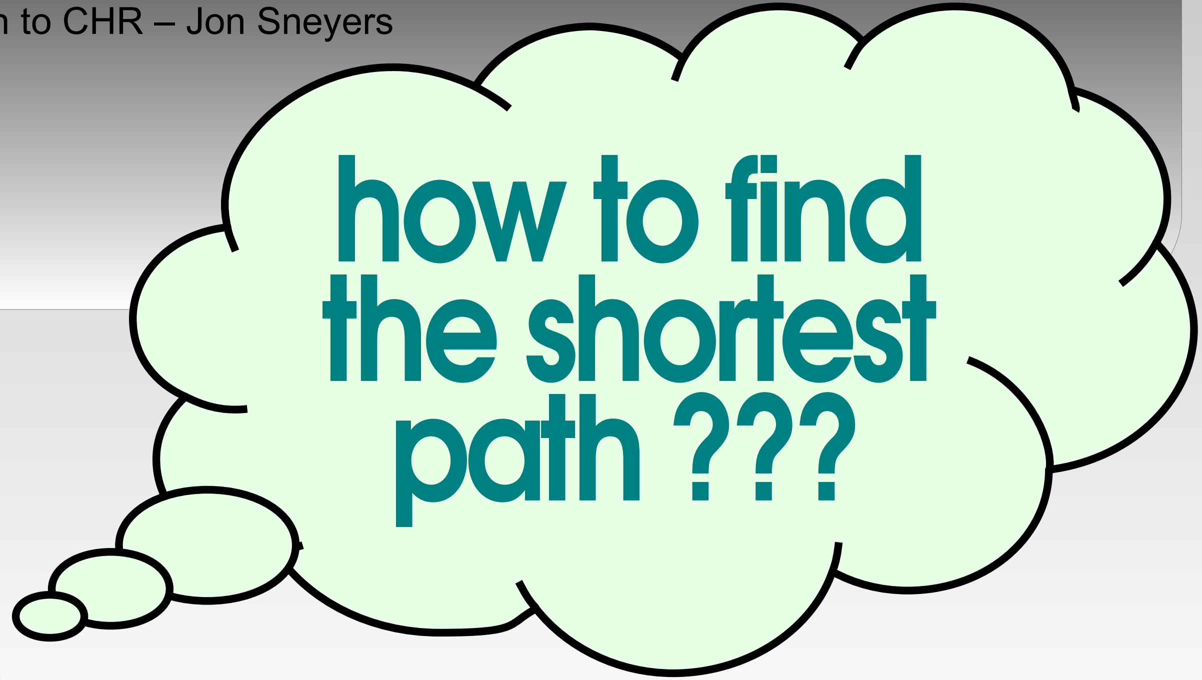
Abstraction to map



Abstraction to (weighted & directed) graph




how to find the shortest path ???
20
20
20
40
60
30
100
140
230
30
30
40
30
10
5
10
10
5
90
90
40
150
60
120
Katholieke Universiteit Leuven
Grote Markt
Naamsestraat
Drinkwaterstraat
Kroegberg
Oude Markt
Zeelstraat
Muntstraat
Eikstraat
Boekhandelstraat
Krakenstraat
Collegeberg
Hallengang
Charles Deberiotstraat
Alfons Smetsplein
Sint-Donatiuspark
Vlamingenstraat
Andreas Vesaliusstraat
Ramberg
Trapgang
Ramenstraatje
Ruteriuspad
De Angelispad
Parijsstraat



Edsger Dijkstra (1930-2002)
Dutch computer scientist

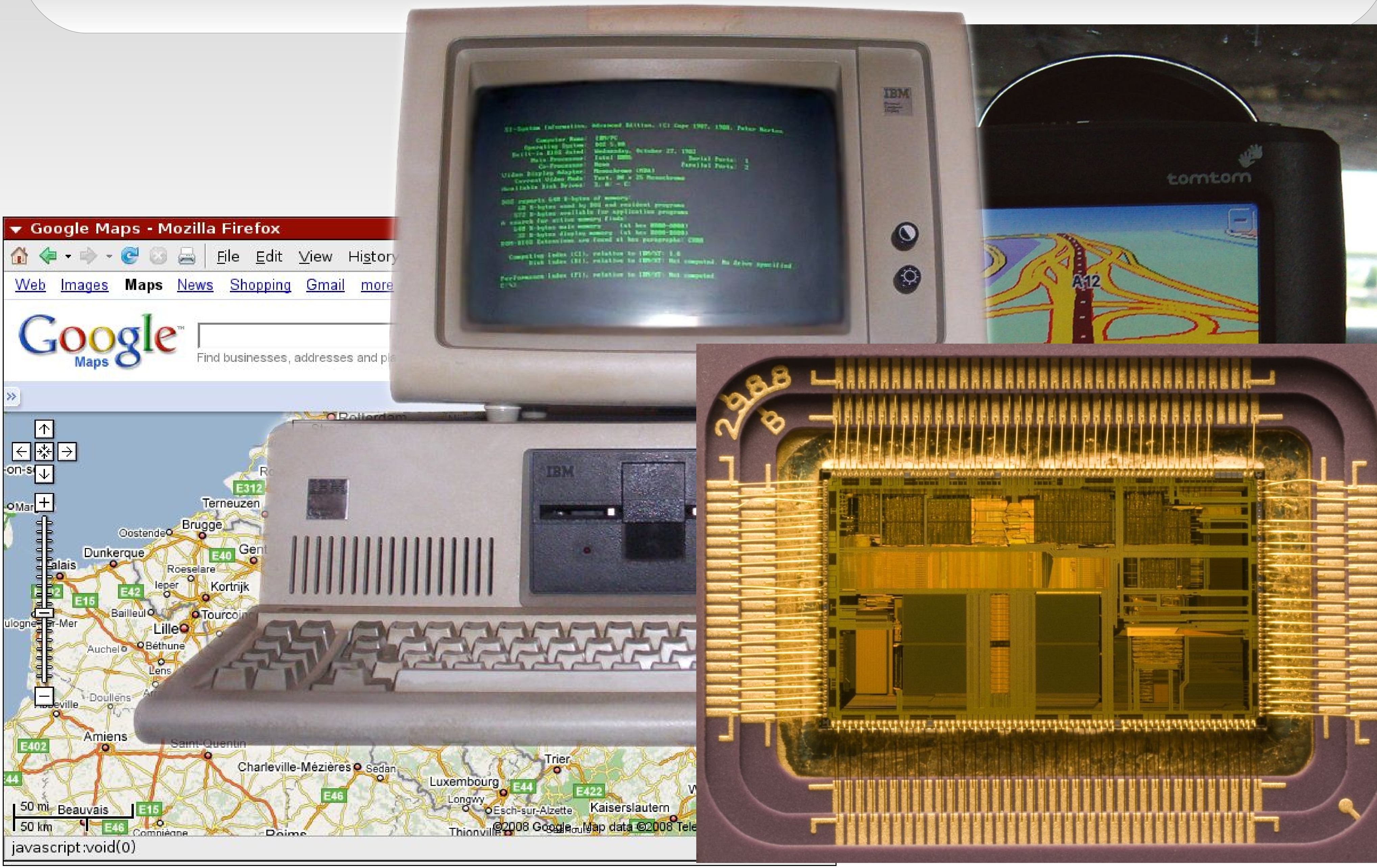
Dijkstra's algorithm:

1. distance(*start-point*) = *0*
2. pick a (not-yet-considered) point x with smallest distance, **LABEL(x)**
3. if ***end-point*** is considered, stop; otherwise go to step **2**

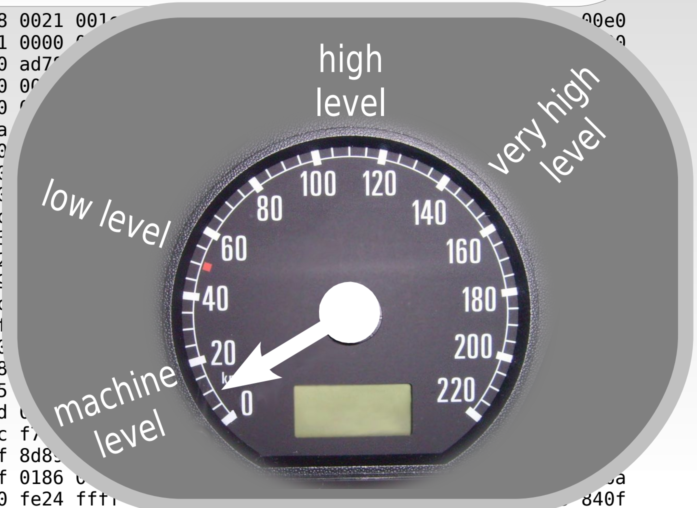
LABEL(x): for all arrows $x \xrightarrow{a} y$:
set distance(y) = distance(x) + a
(if the new distance *is shorter)*

Edsger Dijkstra (1930-2002)
Dutch computer scientist

How to do this **automatically** ?



Implementing Dijkstra's algorithm



...in machine code

and so on...

Implementing Dijkstra's algorithm

```
.L3:
   addl  $1, -148(%ebp)
   movzbl        -116(%ebp),
%eax
   movsbl        %al,%eax
   movl  %eax, -268(%ebp)
   cmpl  $100, -268(%ebp)
   je    .L7
   cmpl  $100, -268(%ebp)
   jg    .L11
   cmpl  $97, -268(%ebp)
   je    .L6
   cmpl  $97, -268(%ebp)
   jg    .L12
   cmpl  $0, -268(%ebp)
   je    .L2
   cmpl  $10, -268(%ebp)
   je    .L2
   jmp   .L4
.L12:
   cmpl  $99, -268(%ebp)
   je    .L2
   jmp   .L4
.L11:
   cmpl  $112, -268(%ebp)
   je    .L9
   cmpl  $116, -268(%ebp)
   je    .L10
   cmpl  $110, -268(%ebp)
   je    .L8
   jmp   .L4
.L9:
   cmpl  $0, -144(%ebp)
   jle   .L13
   movl  $0, -124(%ebp)
   jmp   .L15
.L13:
   movl  $1, -144(%ebp)
   leal  -220(%ebp), %eax
   movl  %eax, 16(%esp)
   leal  -204(%ebp), %eax
   movl  %eax, 12(%esp)
   leal  -119(%ebp), %eax
   movl  %eax, 8(%esp)
   movl  $.LC21, 4(%esp)
   leal  -116(%ebp), %eax
   movl  %eax, (%esp)
   call  sscanf
   cmpl  $3, %eax
   je    .L16
   movl  $1, -124(%ebp)
   jmp   .L15
.L16:
   leal  -119(%ebp), %eax
```

```
   movl  %eax, -276(%ebp)
   movl  $.LC22, -280(%ebp)
   movl  $3, -284(%ebp)
   cld
   movl  -276(%ebp), %esi
   movl  -280(%ebp), %edi
   movl  -284(%ebp), %ecx
   repz
   cmpsb
   seta  %dl
   setb  %al
   movl  %edx, %ecx
   subb  %al, %cl
   movl  %ecx, %eax
   movsbl        %al,%eax
   testl %eax, %eax
   je    .L18
   movl  $2, -124(%ebp)
   jmp   .L15
.L18:
   movl  -204(%ebp), %eax
   testl %eax, %eax
   jle   .L20
   movl  -220(%ebp), %eax
   testl %eax, %eax
   jg    .L22
.L20:
   movl  $3, -124(%ebp)
   jmp   .L15
.L22:
   movl  -204(%ebp), %eax
   addl  $2, %ea
   movl  $40, 4(
   movl  %eax, (
   call  calloc
   movl  %eax, -
   movl  -220(%e
   addl  $1, %ea
   call  calloc
   movl  %eax, -160(%ebp)
   movl  -220(%ebp), %eax
   movl  $4, 4(%esp)
   movl  %eax, (%esp)
   call  calloc
   movl  %eax, -188(%ebp)
   movl  -204(%ebp), %eax
   addl  $2, %eax
   movl  $4, 4(%esp)
   movl  %eax, (%esp)
   call  calloc
   movl  %eax, -192(%ebp)
   movl  -220(%ebp), %edx
   movl  -204(%ebp), %eax
```

```
   leal  (%edx,%eax), %eax
   leal  2(%eax), %edx
   movl  -220(%ebp), %eax
   leal  1(%eax), %ecx
   movl  -204(%ebp), %eax
   addl  $2, %eax
   movl  $4, 24(%esp)
   movl  %edx, 20(%esp)
   movl  $8, 16(%esp)
   movl  %ecx, 12(%esp)
   movl  $40, 8(%esp)
   movl  %eax, 4(%esp)
   movl  $.LC23, (%esp)
   call  printf
   cmpl  $0, -168(%ebp)
   je    .L23
   cmpl  $0, -160(%ebp)
   je    .L23
   cmpl  $0, -192(%ebp)
   je    .L23
   cmpl  $0, -188(%ebp)
   jne   .L27
.L23:
   movl  $4, -124(%ebp)
   jmp   .L15
.L27:
   movl  -160(%ebp), %eax
   movl  %eax, -156(%ebp)
   jmp   .L2
.L10:
   cmpl  $0, -140(%ebp)
   je    .L28
   movl  $5, -124(%ebp)
   jmp
.L28:
   movl
   movl
   movl
   movl
   leal
   movl  %eax, (%esp)
   call  sscanf
   cmpl  $1, %eax
   je    .L2
   movl  $6, -124(%ebp)
   jmp   .L15
.L8:
   cmpl  $0, -144(%ebp)
   jne   .L32
   movl  $7, -124(%ebp)
   jmp   .L15
.L32:
   cmpl  $0, -136(%ebp)
   je    .L34
   movl  $8, -124(%ebp)
```

```
   jmp   .L15
.L34:
   movl  $1, -136(%ebp)
   leal  -208(%ebp), %eax
   movl  %eax, 8(%esp)
   movl  $.LC25, 4(%esp)
   leal  -116(%ebp), %eax
   movl  %eax, (%esp)
   call  sscanf
   movl  %eax, -128(%ebp)
   cmpl  $0, -128(%ebp)
   jg    .L36
   movl  $9, -124(%ebp)
   jmp   .L15
.L36:
   movl  -208(%ebp), %eax
   testl %eax, %eax
   js    .L38
   movl  -208(%ebp), %edx
   movl  -204(%ebp), %eax
   cmpl  %eax, %edx
   jle   .L40
.L38:
   movl  $10, -124(%ebp)
   jmp   .L15
.L40:
   movl  $0, -196(%ebp)
   movl  -204(%ebp), %eax
   movl  %eax, -200(%ebp)
   jmp   .L2
.L7:
   movl  $11, -124(%ebp)
   jmp   .L15
   cmpl  %eax, -132(%ebp)
   jl    .L43
   movl  $13, -124(%ebp)
   jmp   .L15
.L43:
   leal  -224(%ebp), %eax
   movl  %eax, 16(%esp)
   leal  -212(%ebp), %eax
   movl  %eax, 12(%esp)
   leal  -216(%ebp), %eax
   movl  %eax, 8(%esp)
   movl  $.LC26, 4(%esp)
   leal  -116(%ebp), %eax
   movl  %eax, (%esp)
   call  sscanf
```

```
   cmpl  $3, %
   je
   movl
   jmp
.L45:
.L47:
   movl
   jmp
.L51:
   movl  -192(%ebp), %edx
   addl  $4, %edx
   movl  -216(%ebp), %eax
   sall  $2, %eax
   addl  %eax, %edx
   movl  (%edx), %eax
   addl  $1, %eax
   movl  %eax, (%edx)
   sall  $2, %eax
   addl  %edx, %eax
   sall  $3, %eax
   movl  %eax, %edx
   addl  -168(%ebp), %edx
   movl  -156(%ebp), %eax
   movl  %edx, 4(%eax)
   movl  -224(%ebp), %edx
   movl  -156(%ebp), %eax
   movl  %edx, (%eax)
   movl  -212(%ebp), %eax
   cmpl  -200(%ebp), %eax
   jge   .L52
   movl  -212(%ebp), %eax
   movl  %eax, -200(%ebp)
```

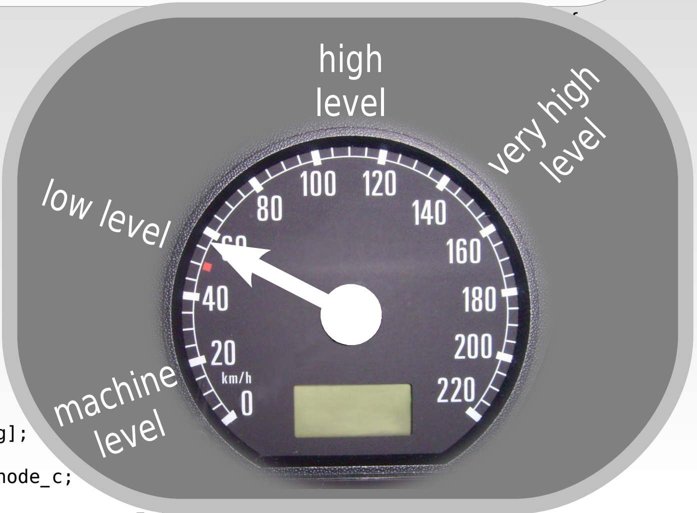
```
   movl  $16, -124(%ebp)
   jmp   .L15
.L2:
.L88:
   leal  -116(%ebp), %eax
   movl  %eax, (%esp)
   call  gets
   testl %eax, %eax
   jne   .L3
   movl  stdin, %eax
         %eax, (%esp)
         feof
         %eax, %eax
         .L61
         $17, -124(%ebp)
         .L15
   cmpl  $0, -148(%ebp)
   jne   .L63
   movl  $20, -124(%ebp)
   jmp   .L15
.L63:
   movl  -220(%ebp), %eax
   cmpl  %eax, -132(%ebp)
   jge   .L65
   movl  $18, -124(%ebp)
   jmp   .L15
```

and so on...

...in assembly

Implementing Dijkstra's algorithm

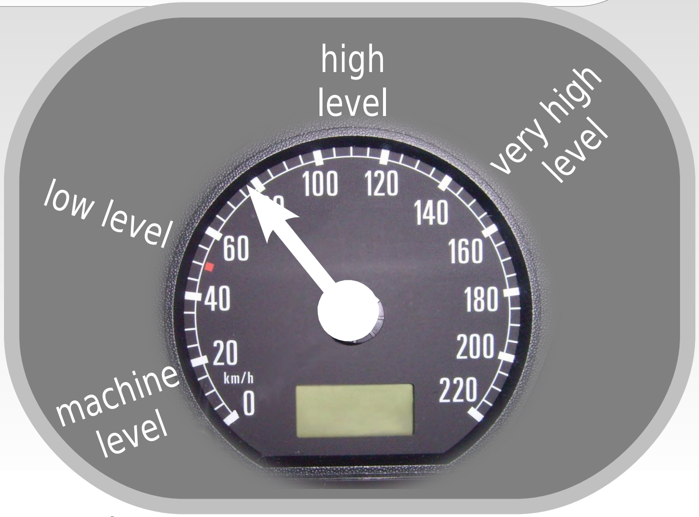
```
#define nod(node) (long)(node-nodes+1)
#define VERY_FAR  1073741823
#define NNULL     (node*)NULL
typedef struct fheap_st {
  node *min;
  long dist;
  long n;
  node **deg_pointer;
  long deg_max;
} f_heap;
f_heap fh;
node *after, *before, *father, *child,
*first, *last,
  *node_c, *node_s, *node_r, *node_n,
*node_l;
long dg;
#define BASE          1.61803
#define NOT_ENOUGH_MEM 2
#define OUT_OF_HEAP    0
#define IN_HEAP        1
#define MARKED         2
#define NODE_IN_FHEAP( node ) ( node ->
status > OUT_OF_HEAP )
void Init_fheap (n) long n; {
  fh.deg_max = (long) (log ((double) n) /
log (BASE) + 1);
  if ((fh.deg_pointer = (node **) calloc
(fh.deg_max, sizeof (node *)))
      == (node **) NULL)
    exit (NOT_ENOUGH_MEM);
  for (dg = 0; dg < fh.deg_max; dg++)
    fh.deg_pointer[[illegible]] = NNULL;
  fh.n = 0;
  fh.min = NNULL;
}
void Check_min (nd) [illegible]d;
  if (nd->dist < fh[illegible]
     fh.dist = nd-[illegible]
     [illegible]
  }
}
void Insert_after_min (nd) node *nd; {
  after = fh.min->next;
  nd->next = after;
  after->prev = nd;
  fh.min->next = nd;
  nd->prev = fh.min;
  Check_min (nd);
}
void Insert_to_root (nd) node *nd; {
  nd->heap_parent = NNULL;
  nd->status = IN_HEAP;
  Insert_after_min (nd);
}
void Cut_node (nd, father) node *nd,
*father; {
  after = nd->next;
  if (after != nd) {
      before = nd->prev;
      before->next = after;
      after->prev = before;
    }
  if (father->son == nd) father->son =
after;
  (father->deg)--;
  if (father->deg == 0) father->son =
NNULL;
}
void Insert_to_fheap (nd) node *nd; {
  nd->heap_parent = NNULL;
  nd->son = NNULL;
  nd->status = IN_HEAP;
  nd->deg = 0;
  if (fh.min == NNULL) {
      nd->prev = nd->next = nd;
      fh.min = nd;
      fh.dist = nd->dist;
    } else Insert_after_min (nd);
  fh.n++;
}
void Fheap_decrease_key (nd) node *nd; {
  if ((father = nd->heap_parent) == NNULL)
    Check_min (nd);
  else {
      if (nd->dist < father->dist) {
     node_c = nd;
     [illegible]le (father != NNULL) {
       [illegible]ut_node (node_c, father);
       [illegible]sert_to_root (node_c);
         if (father->status == IN_HEAP) {
        father->status = MARKED;
        break;
       }
       [illegible]de_c = father;
        father = father[illegible]heap_parent;
      }
    }
  }
}
node * Extract_min () {
  node *nd;
  nd = fh.min;
  if (fh.n > 0) {
      fh.n--;
      fh.min->status = OUT_OF_HEAP;
      first = fh.min->prev;
      child = fh.min->son;
      if (first == fh.min) first = child;
      else {
    after = fh.min->next;
      if (child == NNULL) {
        first->next = after;
        after->prev = first;
      } else {
        before = child->prev;
        first->next = child;
        child->prev = first;
        before->next = after;
        after->prev = before;
      }
    }
     if (first != NNULL) {
  node_c = first;
  last = first->prev;
  while (1) {
      node_l = node_c;
      node_n = node_c->next;
      while (1) {
    dg = node_c->deg;
    node_r = fh.deg_pointer[dg];
    if (node_r == NNULL) {
        fh.deg_pointer[dg] = node_c;
        break;
      } else {
        if (node_c->dist < node_r-
>dist) {
       node_s = node_r;
       node_r = node_c;
     } else node_s = node_c;
        after = node_s->next;
        before = node_s->prev;
        after->prev = before;
        before->next = after;
        node_r->deg++;
        node_s->heap_parent = node_r;
        node_s->status = IN_HEAP;
        child = node_r->son;
        [illegible]
     [illegible]de_r->son[illegible]
  node_s->prev = node_s;
        else {
       after = child->next;
       child->next = node_s;
       node_s->prev = child;
       node_s->next = after;
       after->prev = node_s;
     }
       }
    node_c = node_r;
    fh.deg_pointer[dg] = NNULL;
  }
      if (node_l == last) break;
      node_c = node_n;
    }
  fh.dist = VERY_FAR;
[illegible]
  Init_fheap (n);
  node_last = nodes + n;
  for (i = nodes; i != node_last; i++) {
      i->parent = NNULL;
      i->dist = VERY_FAR;
    }
  source->parent = source;
  source->dist = 0;
  Insert_to_fheap (source);
  while (1) {
      node_from = Extract_min ();
      if (node_from == NNULL) break;
      num_scans++;
      arc_last = (node_from + 1)->first;
      [illegible]
      fo[illegible] arc[illegible]node_from[illegible]
  arc_ij != arc_last; arc_ij++) {
        node_to = arc_ij->head;
        dist_new = dist_from + (arc_ij->len);
        if (dist_new < node_to->dist) {
            node_to->dist = dist_new;
            node_to->parent = node_from;
            if (NODE_IN_FHEAP (node_to))
Fheap_decrease_key (node_to);
            else Insert_to_fheap (node_to);
          }
      }
    }
  n_scans = num_scans;
  return (0);
}
```



...in C (programming language)

Implementing Dijkstra's algorithm

```
public final class DijkstraShortestPath<V, E> {
  private List<E> edgeList;
  private double pathLength;
  public DijkstraShortestPath(Graph<V, E> graph,
      V startVertex, V endVertex) {
         this(graph, startVertex, endVertex,
          Double.POSITIVE_INFINITY);
    }
  public DijkstraShortestPath(Graph<V, E> graph,
    V startVertex, V endVertex, double radius) {
     CFIterator<V, E> iter = new
     CFIterator<V,E>(graph, startVertex, radius);
    while (iter.hasNext()) {
       V vertex = iter.next();
       if (vertex.equals(endVertex)) {
        createEdgeList(graph, iter, endVertex);
        pathLength =
          iter.getShortestPathLength(endVertex);
        return;
       }
    }
    edgeList = null;
    pathLength = Double.POSITIVE_INFINITY;
 }
 public List<E> getPathEdgeList() {
   return edgeList;
 }
}
public class FHNode<T> {
  T data;
  FHNode<T> child;
  FHNode<T> left;
  FHNode<T> parent;
  FHNode<T> right;
  boolean mark;
  double key;
  int degree;
  public FHNode(T data, double key) {
      right = this;
      left = this;
      this.data = data;
      this.key = key;
  }
  public final double getKey() {
      return key;
  }
  public final T getData() {
      return data;
  }
}
public class FH<T> {
  private static final double oneOverLogPhi
   = 1.0 / Math.log((1.0 + Math.sqrt(5.0)) / 2.0);
  private FHNode<T> minNode;
  private int nNodes;
  public FH() {}
  public boolean isEmpty() {
      return minNode == null;
  }
  public void clear() {
      minNode = null;
      nNodes = 0;
  }
  public void decreaseKey(FHNode<T> x, double k) {
      x.key = k;
      FHNode<T> y = x.parent;
      if ((y != null) && (x.key < y.key)) {
          cut(x, y);
          cascadingCut(y);
      }
      if (x.key < minNode.key) {
          minNode = x;
      }
  }
  public void delete(  ode<T> x) {
      decreaseKey(x,     ble NEGATIVE INFINITY);
          eMin();
  }
  p   ic    d insert(   o  <T> n   e,      e  y) {
      no   key = key
      if  inNode    ll,
         node.left   inNode
         node.right = minNode.right;
         minNode.right = node;
         node.right.left = node;
         if (key < minNode.key) minNode = node;
      } else minNode = node;
      nNodes++;
  }
  public FHNode<T> removeMin() {
      FHNode<T> z = minNode;
      if (z != null) {
          int numKids = z.degree;
          FHNode<T> x = z.child;
```




```
              }
              nNodes--;
          }
          return z;
      }
      protected void cascadingCut(FHNode<T> y) {
          FHNode<T> z = y.parent;
          if (z != null) {
              if (!y.mark) {
                  y.mark = true;
              } else {
                  cut(y, z);
                  cascadingCut(z);
              }
          }
      }
```

and so on...

...in Java

Implementing Dijkstra's algorithm

```
dijkstra(Vertex, Ss):-
  create(Vertex, [Vertex], Ds),
  dijkstra_1(Ds, [s(Vertex,0,[])], Ss).
dijkstra_1([], Ss, Ss).
dijkstra_1([D|Ds], Ss0, Ss):-
  best(Ds, D, S),
  delete([D|Ds], [S], Ds1),
  S=s(Vertex,Distance,Path),
  reverse([Vertex|Path], Path1),
  merge(Ss0, [s(Vertex,Distance,Path1)], Ss1),
  create(Vertex, [Vertex|Path], Ds2),
  delete(Ds2, Ss1, Ds3),
  incr(Ds3, Distance, Ds4),
  merge(Ds1, Ds4, Ds5),
  dijkstra_1(Ds5, Ss1, Ss).

path(Vertex0, Vertex, Path, Dist):-
  dijkstra(Vertex0, Ss),
  member(s(Vertex,Dist,Path), Ss), !.

create(Start, Path, Edges):-
  setof(s(Vertex,Edge,Path),
  e(Start,Vertex,Edge), Edges), !.
create(_, _, []).

best([], Best, Best).
best([Edge|Edges], Best0, Be
  shorter(Edge, Best0), !,
  best(Edges, Edge, Best).
best([_|Edges], Best0, Best):-
  best(Edges, Best0, Best).

shorter(s(_,X,_), s(_,Y,_)):-X < Y.

delete([], _, []).
delete([X|Xs], [], [X|Xs]):-!.
delete([X|Xs], [Y|Ys], Ds):-
  eq(X, Y), !,

delete([X|Xs], [Y|Ys], [X|Ds]):-
  lt(X, Y), !, delete(Xs, [Y|
delete([X|Xs], [_|Ys], Ds):-
  delete([X|Xs], Ys, Ds).

merge([], Ys, Ys).
merge([X|Xs], [], [X|Xs]):
merge([X|Xs], [Y|Ys], [X|Z
  eq(X, Y), shorter(X, Y),
  merge(Xs, Ys, Zs).
merge([X|Xs], [Y|Ys], [Y|Z
  eq(X, Y), !,
  merge(Xs, Ys, Zs).
merge([X|Xs], [Y|Ys], [X|Zs]
  lt(X, Y), !,
  merge(Xs, [Y|Ys], Zs).
merge([X|Xs], [Y|Ys], [Y|Zs]):-
  merge([X|Xs], Ys, Zs).

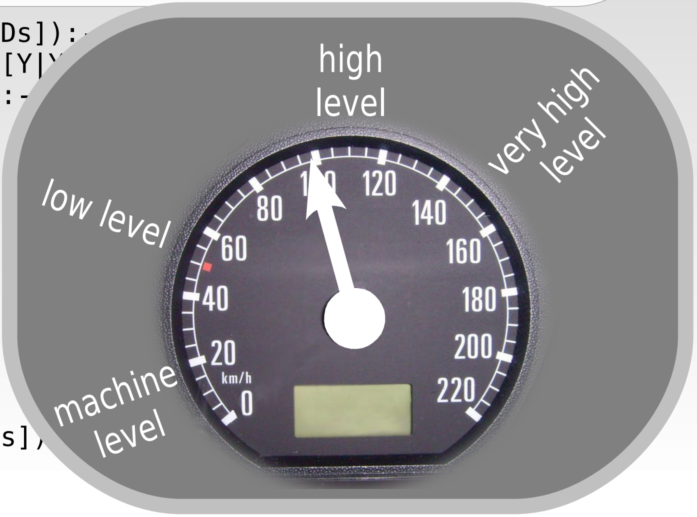
eq(s(X,_,_), s(X,_,_)).
lt(s(X,_,_), s(Y,_,_)):-X @< Y.

  ([], _, []).
in         )|X   ,     [     P)|Ys]):-
          +
  Inc  X   Inc

member(X, [
member(X, [_|Ys]):-member(X,

reverse(Xs, Ys):-reverse_1(Xs, [], Ys).
reverse_1([], As, As).
reverse_1([X|Xs], As, Ys):-reverse_1(Xs, [X|As], Ys).

e(X, Y, Z):-dist(X, Y, Z).
e(X, Y, Z):-dist(Y, X, Z).
```



...in Prolog

Implementing Dijkstra's algorithm

```
:- chr_constraint edge(+node,+node,+length), dijkstra(+node),
        distance(+node,+length), scan(+node,+length),
        relabel(+node,+length).
:- chr_type node == int.
:- chr_type length == number.

dijkstra(A) <=> scan(A,0).
scan(N,L), edge(N,N2,W) ==> L2 is L+W,relabel(N2,L2).
scan(N,L) <=> distance(N,L),
        (extract_min(N2,L2) -> scan(N2,L2) ; true).
distance(N,_) \ relabel(N,_) <=> true.
relabel(N,L) <=> decr_or_ins(N,L).


:- chr_constraint insert(+item,+key), extract_min(?item,?key),
              decr_or_ins(+item,+key), decr(+item,+key),
   mark(+item), ch2rt(+item), decr(+item,+key,+item,+item,+mark),
   findmin, min(+item,+key), item(+item,+key,+item,+item,+mark).

:- chr_type item == int.
:- chr_type key == number.
:- chr_type mark ---> m ; u.

insert(I,K) <=> item(I,K,0,0,u), min(I,K).

min(_,A) \ min(_,B) <=> A =< B | true.

extract_min(X,Y), min(I,K), item(I,_,_,_,_)
            <=> ch2rt(I), findmin, X=I, Y=K.
extract_min(_,_) <=> fail.

ch2rt(I) \ item(C,K,R,I,_)#passive
            <=> item(C,K,R,0,u).
ch2rt(I) <=> true.
```

```
findmin, item(I,K,_,
findmin <=> true.

item(I1,K1,R,0,_), i
         <=> K1 < K

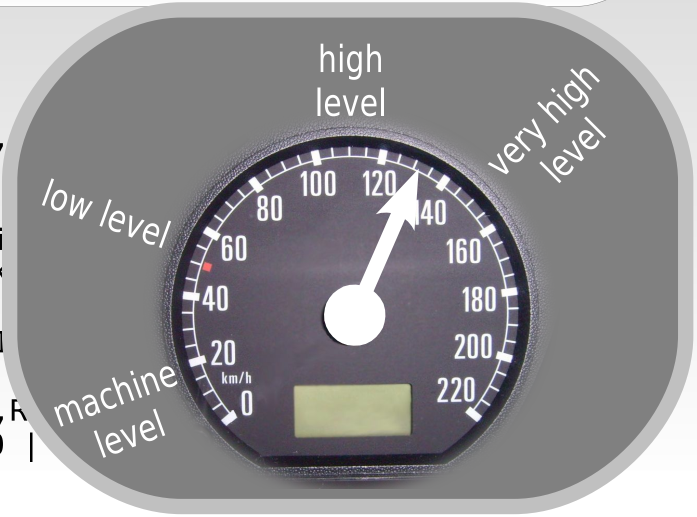
          ; item(I1

decr(I,K), item(I,0,R
          <=> K < 0 |
decr(I,K) <=> fail.

item(I,0,R,P,M), decr_or_ins(I,K)
          <=> K < 0 | decr(I,K,R,P,M).
item(I,0,_,_,_) \ decr_or_ins(I,K) <=> K >= 0 | true.
decr_or_ins(I,K) <=> insert(I,K).

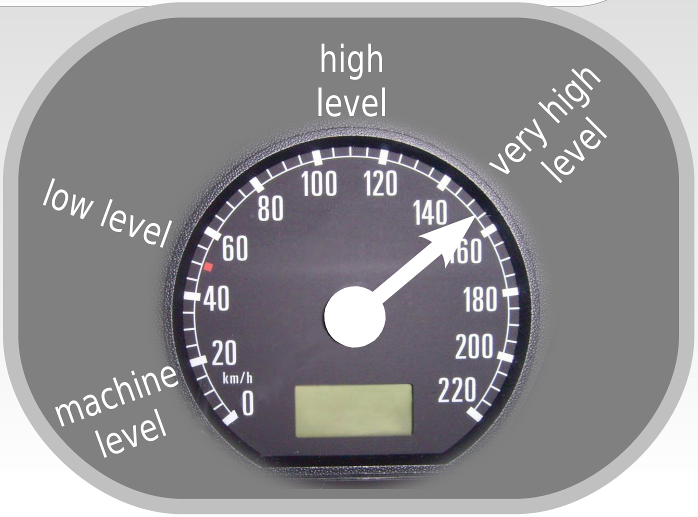
    r(I         min(I,K).
    I  R,       item(I,K,R,0,u).
item(P          cr(I,K,R,P,M)
                PK | item(I,K,R,P,M).
dec  I   R,  M    item(I,K,R,0,u), mark(P).

mark(I), item(I,K,R,0,_) <=> item(I,K,R-1,0,u).
mark(I), item(I,K,R,P,m)
          <=> item(I,K,R-1,0,u), mark(P).
mark(I), item(I,K,R,P,u) <=> item(I,K,R-1,P,m).
mark(I) <=> writeln(error_mark), fail.
```



...in CHR

Implementing Dijkstra's algorithm




```
:- chr_constraint   edge(+node,+node,+length),
                    source(+node),
                    distance(+node,+length).
:- chr_type node == int.
:- chr_type length == number.

1 :: source(V) ==> distance(V,0).
1 :: distance(V,D1) \ distance(V,D2) <=> D1 =< D2 | true.
D+2 :: distance(V,D), edge(V,C,W) ==> distance(W,D+C).
```

...in CHR^{rp}

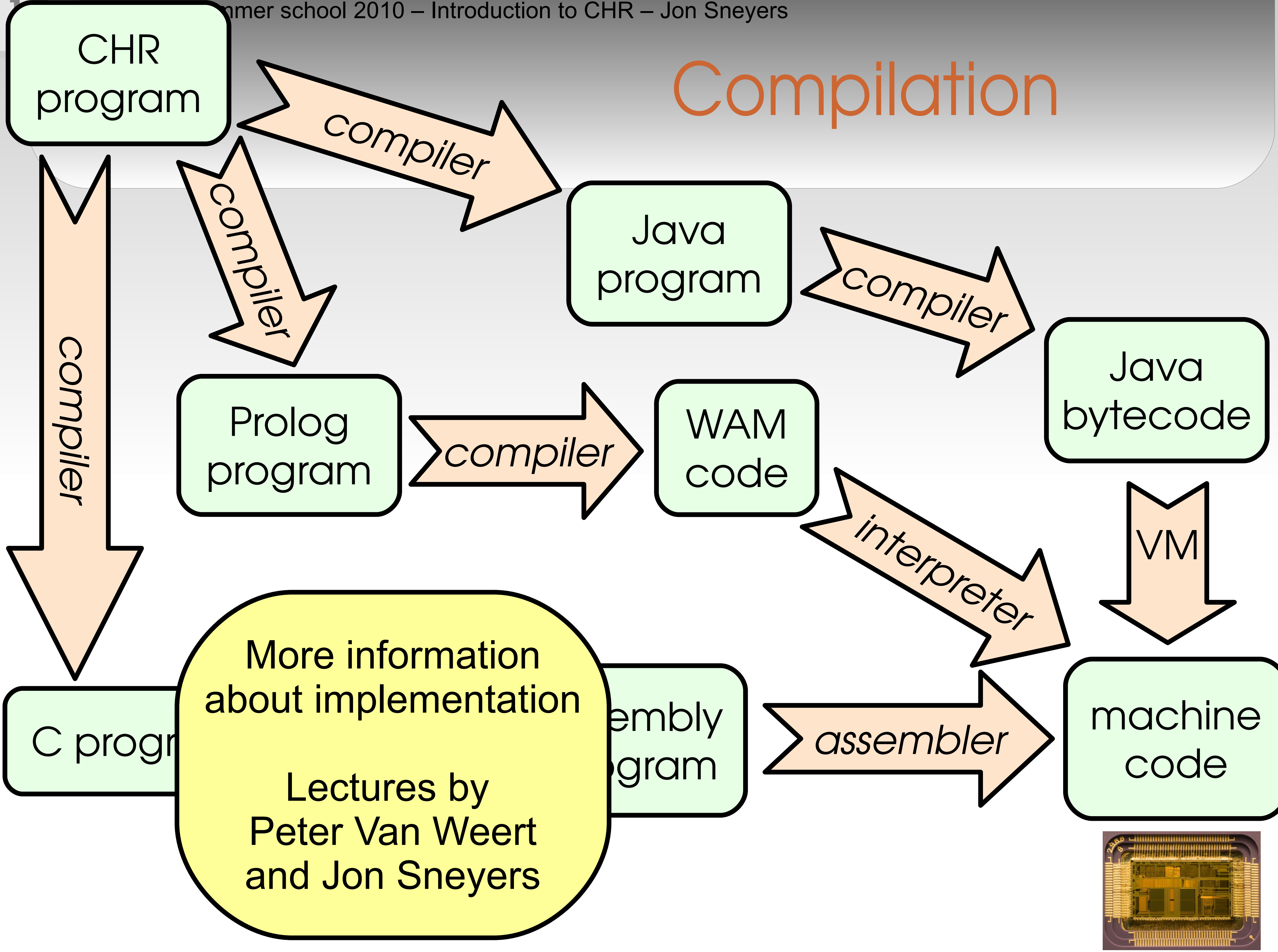
CHR = Constraint Handling Rules

- CHR is a very **high level** programming language
- based on **rules**
  - propagation rules:
    - clouds ⇒ forecast(rainy).
    - forecast(rainy) ⇒ bring(coat).
    - forecast(sunny) ⇒ bring(sunscreen).
  - simplification rules:
    - bring(coat), bring(sunscreen) ⇔ bring(umbrella).

驰

- stand-alone (CHR-only) or extending a **host language**

Compilation



Syntax of CHR

head:	CHR constraints
guard:	host language (built-in)
body:	CHR constraints + host language

- Propagation rule:

  head ⇒ guard | body.

  example: dist(A,D), road(A,B,L) ⇒ dist(B,D+L).

- Simplification rule:

  head ⇔ guard | body.

  example: dist(A,X), dist(A,Y) ⇔ $X \leq Y$ | dist(A,X).

Operational semantics of CHR

IF head **IN STORE (AND** guard **HOLDS), THEN...**

- Propagation rule: **... ADD** body **TO STORE**

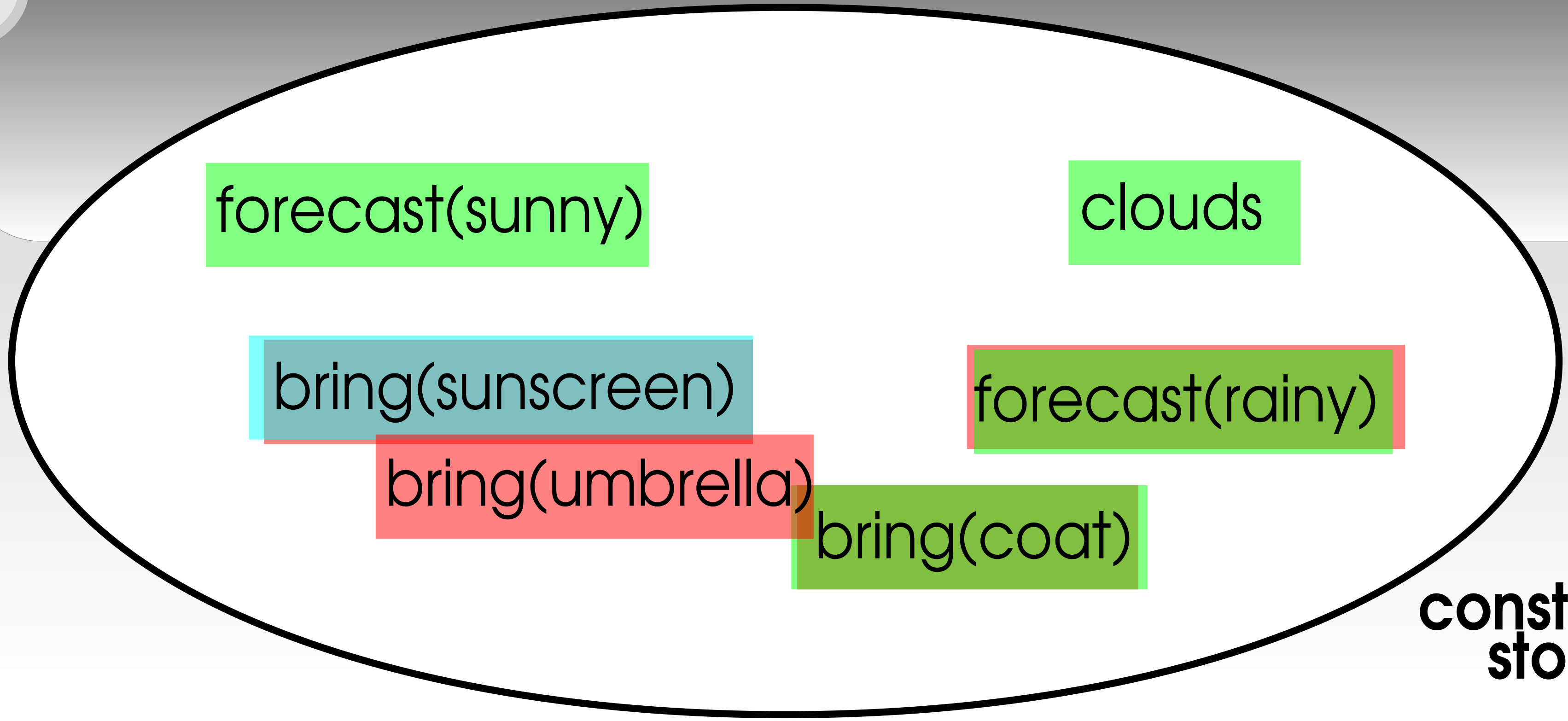
  head ⇒ guard | body.

  example: dist(A,D), road(A,B,L) ⇒ dist(B,D+L).

- Simplification rule: **... REPLACE** head **BY** body

  head ⇔ guard | body.

  example: dist(A,X), dist(A,Y) ⇔ X≤Y | dist(A,X).



constraint store

CHR program

propagation rules

```
clouds ⇒ forecast(rainy).
forecast(rainy) ⇒ bring(coat).
forecast(sunny) ⇒ bring(sunscreen).
```

simplification rule →

```
bring(coat), bring(sunscreen) ⇔ bring(umbrella).
```

Features of CHR

Embedded in a host language

CHR *extends* an existing programming language, e.g.
CHR(Prolog)
CHR(Haskell)
CHR(Java)
CHR(C)

⇒ dist(B,D+L).

- Simplification rule:

head ⇔ guard | body.

example: dist(A,X), dist(A,Y) ⇔ X≤Y | dist(A,X).

Features of CHR

Multiple heads

The head of a rule consists of an arbitrary number of CHR constraints (1 or more)

cf. Prolog: single-headed

- Propagation rule:

  head ⇒ guard | body.

  example: dist(A,D), road(A,B,L) ⇒ dist(B,D+L).

- Simplification rule:

  head ⇔ guard | body.

  example: dist(A,X), dist(A,Y) ⇔ X≤Y | dist(A,X).

Features of CHR

- Propagation rule:

  head ⇒ guard |

  example: dist(A,D), ro

- Simplification rule:

  head ⇔ guard | body.

  example: dist(A,X), dist(A,Y) ⇔ X≤Y | dist(A,X).

Multi-set semantics

The constraint store may contain the same constraint multiple times
{c} is not the same as {c,c}

cf. classical logic: $p \leftrightarrow p \wedge p$

Features of CHR

- Propagation rule:

Important remark:

in CHR(Prolog), we can still use Prolog disjunction or nondeterministic predicates in the body of rules!

CHR with disjunction/search is called **CHR$^{\vee}$**

ary.

d(A,B,L) ⇒ dist(B,D+L).

Committed-choice

Once a rule has been applied, it remains applied – no backtracking to try different derivation paths

cf. Prolog: choice-points and backtracking

head ⇔ guard | body.

example: dist(A,X), dist(A,Y) ⇔ X≤Y | dist(A,X).

Features of CHR

Logical semantics

CHR has a declarative semantics!

propagation = implication

- Propagation rule:

  head ⇒ guard | body.

  example: dist(A,D), road(A,B,L) ⇒ dist(B,D+L).

simplification = equivalence

- Simplification rule:

  head ⇔ guard | body.

  example: dist(A,X), dist(A,Y) ⇔

More information about logical semantics:

Lecture by Hariolf Betz

PART TWO

Writing CHR programs

CHR(Prolog) by example

- Simple example: color mixing in CHR
- We first declare CHR constraints as follows:

```
:- chr_constraint red, blue, yellow, purple, …
```

- Then we write the rules:

```
red, blue <=> purple.
blue, yellow <=> green.
yellow, red <=> orange.
```

CHR(Prolog) by example

- Simple example: color mixing in CHR

```
red, blue <=> purple.
blue, yellow <=> green.
yellow, red <=> orange.
```

- CHR program execution:
    - user gives a **goal**
    - rules are applied exhaustively
    - the remaining constraints are the **result**

CHR(Prolog) by exa[mple]

- Simple example: color mi[xing]

```
red, blue <=> purple.
blue, yellow <=> green.
yellow, red <=> orange.
```

- Example interaction:

```
?- blue, red.
purple
?- yellow, blue, red.
green
red
```

Why this answer?

(and not, say, "yellow, purple")

Refined semantics

Execution from left to right and from top to bottom (cf. Prolog)

Confluence

- Simp… n CHR

```
r1 @
r2 @
r3 @ yellow, red <=> oran
```

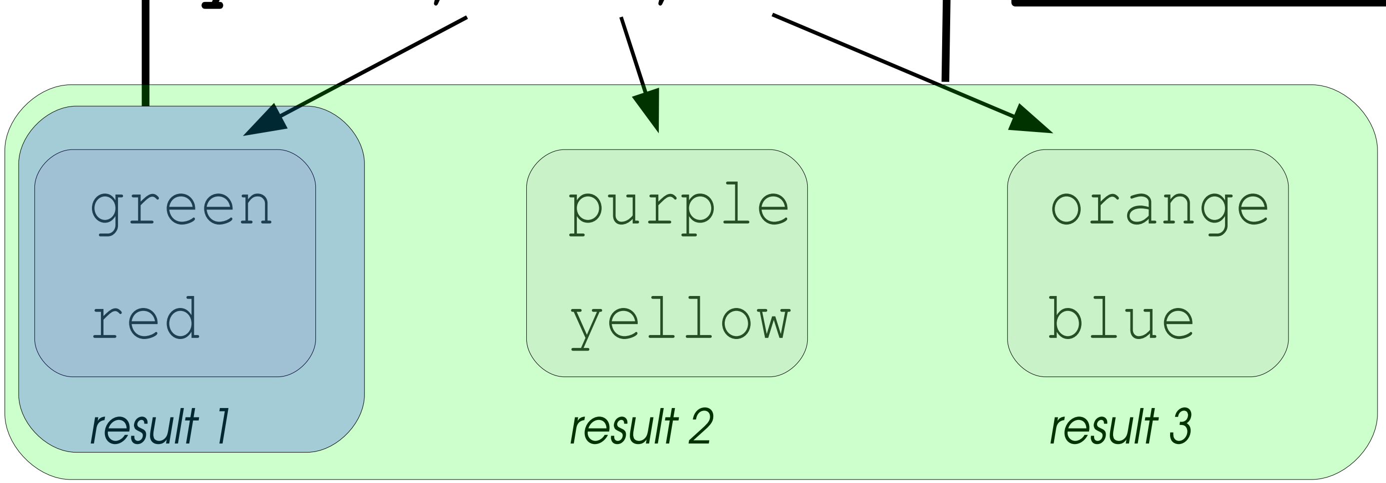
Refined semantics

Execution from left to right and from top to bottom (cf. Prolog)

Abstract semantics

allows rule application in any order

- `?- yellow, blue, red.`



Confluence

A CHR program is called **confluent** if
for any given goal, there is only one result,
regardless of the order in which rules are applied.
(so the color mixing program is not confluent)

Abstract semantics

allows rule appl ... er

- `?- yellow, blue, red.`

result 1	result 2	result 3
green red	purple yellow	orange blue

More information about confluence:

Lectures by Slim Abdennadher

Constraints with arguments

- Add anything to brown and it remains brown:

```
red, blue <=> purple.
blue, yellow <=> green.
yellow, red <=> orange.

brown, red <=> brown.
brown, blue <=> brown.
brown, yellow <=> brown.
brown, purple <=> brown.
...
```

Constraints with arguments

- From many 0-ary constraints to one unary constraint:

```
:- chr_constraint red, blue, yellow, purple, ...
red, blue <=> purple.
blue, yellow <=> green.
yellow, red <=> orange.


:- chr_constraint color/1.
color(red), color(blue) <=> color(purple).
...
```

Constraints with arguments

- Now we can write more general rules:

```
:- chr_constraint color/1.
color(X), color(Y) <=> mix(X,Y,Z) | color(Z).
color(brown), color(_) <=> color(brown).


% host language
mix(red,blue,purple).
mix(blue,yellow,green).
mix(yellow,red,orange).
```

Type and mode declarations

- Optionally, we can specify types and modes:

```
% no type/mode declaration:
:- chr_constraint color/1.

% only mode declaration:
:- chr_constraint color(+).      % ground argument

% type and mode declaration:
:- chr_constraint color(+colorname).
:- chr_type colorname ---> red ; blue ; yellow ;...
```

More information about type/mode declarations:

Lectures by Peter Van Weert

Simpagation rules

- So far we have only used simplification rules.
- Simpagation rules can be more concise/efficient:

```
% simplification rule:
color(brown), color(_) <=> color(brown).
```

> **"true" ?**
>
> In Prolog, "`true`" is a built-in that does not do anything. We use it to indicate an empty body.

```
% simpagation rule:
color(brown) \ color(_) <=> true.
```

Typical pattern #1: flattening lists

- We want to convert "`colors([red,green,blue])`" to "`color(red), color(green), color(blue)`"

```
:- chr_constraint color(+colorname).

:- chr_type colorname ---> red ; blue ; yellow ;...

:- chr_constraint colors(+list(colorname)).

:- chr_type list(T) ---> [] ; [T|list(T)].
```

```
colors([]) <=> true.

colors([C|Rest]) <=> color(C), colors(Rest).
```

(just like how you would do this in Prolog)

Typical pattern #2: "default constructor"

- Now we have not a fixed quantity of paint, but we specify the amount
- For backwards compatibility, we still have `color/1`

```
:- chr_constraint color(+colorname).
:- chr_constraint color(+colorname,+amount).
:- chr_type colorname ---> red ; blue ; yellow ;...
:- chr_type amount == float.


% we assume 1 liter of paint:
color(C) <=> color(C,1).
```

Typical pattern #3: maintaining a sum

```
:- chr_constraint color(+colorname,+amount).
```

```
color(C,A1), color(C,A2)
        <=> TA is A1+A2, color(C,TA).

color(C,0) <=> true.
color(X,A1), color(Y,A2)
        <=> mix(X,Y,Z) | TA is A1+A2, color(Z,TA).
```

Typical pattern #4: maximum

- Which color do we have the most of?

```
:- chr_constraint color(+colorname,+amount).
:- chr_constraint most(+colorname).
```

```
color(C,A) ==> most(C,A).
most(_,A1) \ most(_,A2) <=> A1 >= A2 | true.
```

CHR(Prolog) one-liners (1)

- Finding the minimum:

```
min(A) \ min(B) <=> A =< B | true.
```

```
?-  min(8), min(3), min(6), min(7).
min(3)
```

- Computing the sum:

```
sum(A), sum(B) <=> C is A+B, sum(C).
```

```
?-  sum(3), sum(5), sum(6).
sum(14)
```

CHR(Prolog

- Finding the min

```
min(A) \ min(B)
```

```
?- min(8), min(3), mi
min(3)
```

- Computing the

```
sum(A), sum(B)
```

```
?- sum(3), sum(5), su
sum(14)
```

Online algorithm

An online algorithm processes its inputs while they arrive (it does not need to see the full input to get started)

Using CHR often results in online algorithms

Anytime algorithm

An anytime algorithm can be interrupted during the computation to give a partial (approximate) result, from which it can then resume the computation

Using CHR often results in anytime algorithms

Concurrent algorithm

A concurrent algorithm can be executed in parallel (while sequential algorithms are hard to parallelize)

Using CHR often results in concurrent algorithms

CHR(Prolog) one-liners (2)

- Transitive closure

```
:- op(700,xfx,before).
:- chr_constraint before(+any,+any).
```

```
A before B, B before C ==> A before C.
```

```
?- a before b, b before c, c before d.
a before b
b before c
c before d
a before c
a before d
b before d
```

CHR(Prolog) one-liners (3)

- Naive merge-sort in $O(n^2)$ time

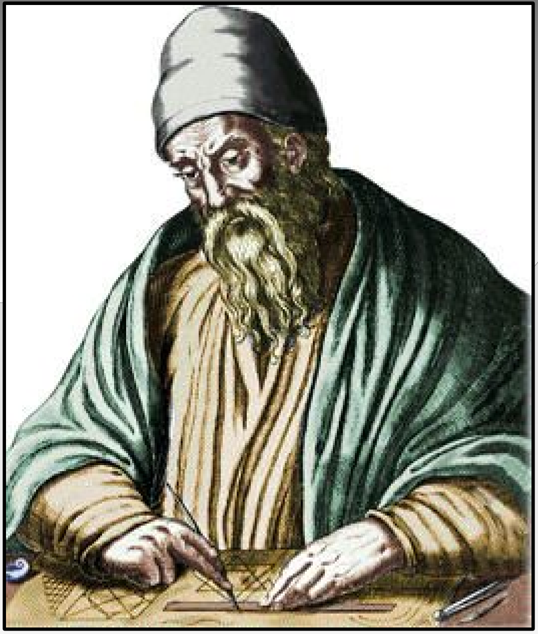
```
:- op(700,xfx,before).
:- chr_constraint before(+any,+any).

A before B \ A before C <=> B @< C | B before C.



?- 0 before foo, 0 before bar, 0 before baz, 0 before quux.
0 before bar
bar before baz
baz before foo
foo before quux
```

CHR(Prolog) two-liners (1)



- Greatest common divisor

  (Euclid's algorithm)

```
:- chr_constraint gcd(+int).

gcd(0) <=> true.

gcd(N) \ gcd(M) <=> N =< M | L is M mod N, gcd(L).
```

```
?- gcd(94017), gcd(1155), gcd(2035).
gcd(11)
```

CHR(Prolog) two-liners (2)

- Prime number generator
  (sieve of Eratosthenes)

```
:- chr_constraint prime(+int).

prime(N) ==> N>2 | M is N-1, prime(M).

prime(A) \ prime(B) <=> 0 =:= B mod A | true.
```

```
?- prime(10).
prime(2)
prime(3)
prime(5)
prime(7)
```

CHR(Prolog) two-liners (3)

- Fibonacci numbers

```
:- chr_constraint fib(+int,+int), upto(+int).

upto(_) ==> fib(0,1), fib(1,1).

upto(Max), fib(N1,M1), fib(N2,M2)
  ==> Max>N2, N2 is N1+1 |
    N is N2+1, M is M1+M2, fib(N,M).
```

```
?- upto(10).
fib(10,89)
fib(9,55)
fib(8,34)
fib(7,21)
fib(6,13)
...
```

CHR(Prolog) two-liners (4)

- Optimal merge-sort

```
:- op(700,xfx,before).
:- chr_constraint before(+any,+any), sort(+int,+any).

X before A  \  X before B
      <=> A @< B | A before B.


sort(N,A), sort(N,B)
      <=> A @< B | M is N+1, sort(M,A), A before B.


?- sort(0,foo), sort(0,bar), sort(0,baz), sort(0,quux).
bar before baz
baz before foo
foo before quux
sort(2,bar)
```

CHR(Prolog) two-liners (5)

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

- Soduko puzzle solver in CHR

```
:- chr_constraint given(+pos,+val), maybe(+pos,+list(val)).

given(P1,V) \ maybe(P2,L)
   <=> sees(P1,P2), select(V,L,L2) | maybe(P2,L2).

maybe(P,L) <=> member(V,L), given(P,V).

sees(X-_, X-_).
sees(_-X, _-X).
sees(X-Y, A-B) :- X//3 =:= A//3,  Y//3 =:= B//3.
```

```
?- given(a-1,5),given(f-4,3), ..., maybe(a-2,[1,2,3,...,9]), ...
given(a-1,5)
given(a-2,2)
given(a-3,7)
...
```

One last example...

- Simple less-than-or-equal constraint solver

```
:- op(700,xfx,leq).
:- chr_constraint leq/2.
reflexivity @ X leq X <=> true.
idempotence @ X leq Y \ X leq Y <=> true.
antisymmetry @ X leq Y, Y leq X <=> X=Y.
transitivity @ X leq Y, Y leq Z ==> X leq Z.
```

```
?- A leq B, B leq C, C leq A.
A = B
B = C
```

Differences between CHR and Prolog

	Prolog	**CHR**
basic elements	predicates	constraints
elements are defined by	clauses	rules
syntax	`head :- body.`	`head <=> guard | body.`
#heads	1	1, 2, 3, ...
definition selection condition	unification	matching + guard
different applicable definitions	try alternatives (backtracking)	committed-choice
no applicable definition	failure	suspension (delay) ↳ partial result

Committed-choice – different from Prolog!

- In Prolog, **backtracking** (proof search) is used to find a non-failing derivation
- In CHR there is no backtracking
- ```
  :- chr_constraint chr/0, output/1.
  chr <=> output(foo).
  chr <=> output(bar).
  prolog :- output(foo).
  prolog :- output(bar).
  ```

```
?- prolog.
output(foo) ;

output(bar)
```

```
?- chr.
output(foo)
```
```

Head matching – different from Prolog!

- In Prolog, **unification** is used to match clause heads
- In CHR, **matching** (one-way unification) is used
-
```
:- chr_constraint chr/1, output/1.
chr(foo) <=> output(bar).
prolog(foo) :- output(bar).
```

```
?- prolog(foo).
output(bar)

?- prolog(Variable).
output(bar)
Variable = foo

?- prolog(quux).
No
```

```
?- chr(foo).
output(bar)

?- chr(Variable).
chr(Variable)

?- chr(quux).
chr(quux)
```

PART THREE

Theory & Applications

History of CHR: some milestones

(not including applications)

1991 CHR is born, Thom Frühwirth

1993 First CHR compiler by Pascal Brisset

1995 Christian Holzbaur implements CHR(SICStus)

1998 confluence, program analysis (PhD Slim Abdennadher)

2002 Tom Schrijvers implements Leuven CHR system

2002-2005 optimized compilation (PhDs Gregory Duck, Tom Schrijvers)

2003 First CHR book [Frühwirth&Abdennadher, Essentials of Constraint Programming]

2004 refined semantics, Gregory Duck et al.

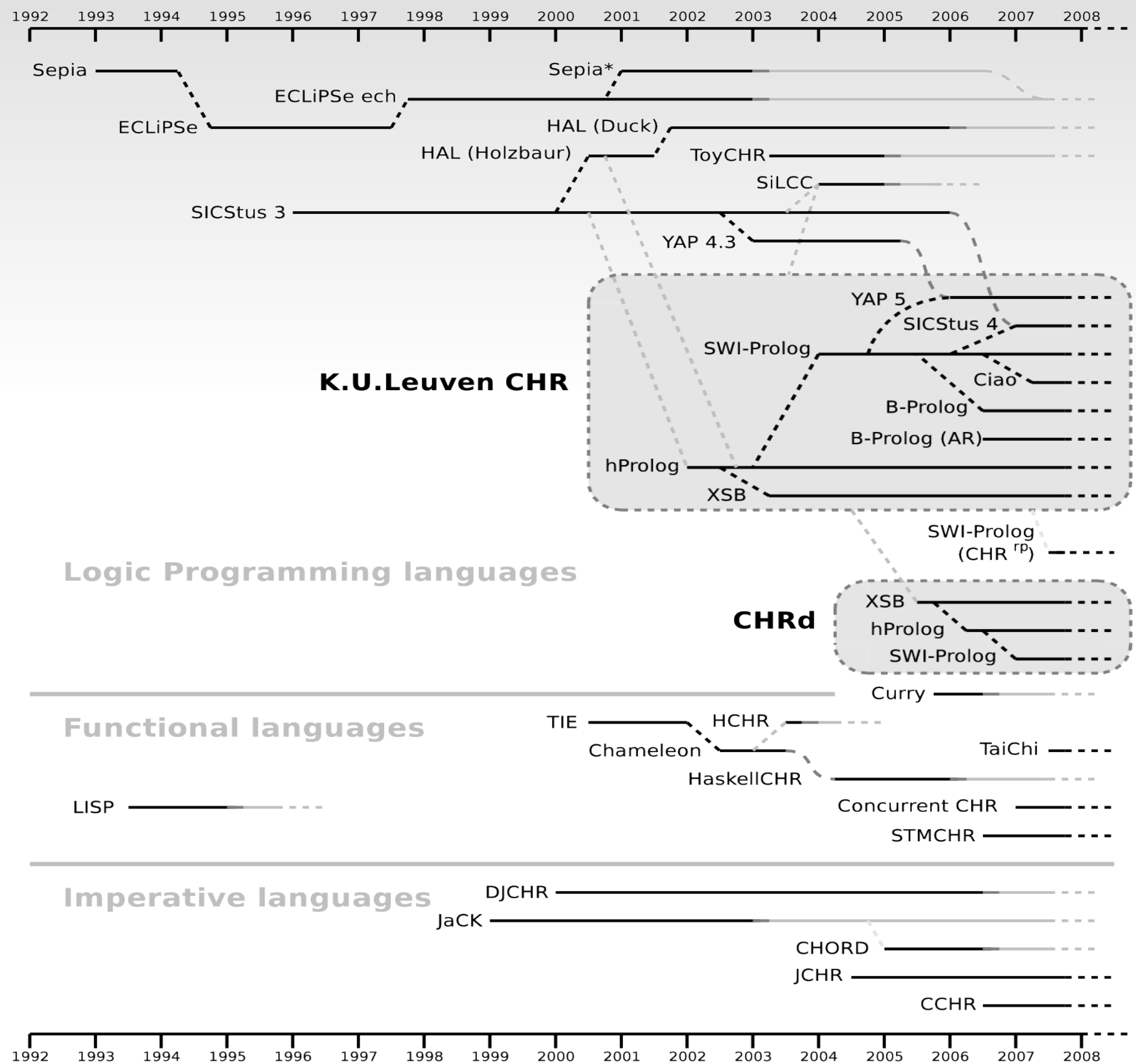
2004 First CHR workshop

2005- Peter Van Weert implements Leuven JCHR (Java)

2007 Sulzmann & Lam implement first concurrent system

2009 Second CHR book, sixth CHR workshop

CHR systems



More information about CHR systems:

Lectures by Peter Van Weert

Theory topics (1)

- Semantics
  - Declarative (logical) semantics
    - Classical logic (Frühwirth)
    - Linear logic (Hariolf Betz)
    - Transaction logic, …
  - Operational semantics
    - Abstract semantics
    - Theoretical semantics
    - Refined semantics (Duck et al)
    - Priority semantics (Leslie De Koninck)

Theory topics (2)

- Relationship to other formalisms
  - Term rewriting (ACD term rewriting, Duck, Stuckey et al)
  - Production rules / business rules (Van Weert)
  - Join-Calculus (Sulzmann and Lam)
  - Logical Algorithms (De Koninck)
  - Graph Transformation Systems (Raiser)
  - Petri nets (Betz)
  - ...

More information about related formalisms:

Lectures by Thom Frühwirth

Theory topics (3)

- Program analysis
  - Confluence (Abdennadher, Duck et al, Raiser&Tacchella, Haemmerlé&Fages, …)
  - Operational equivalence (Abdennadher&Frühwirth)
  - Termination (Frühwirth, Paolo Pilozzi, Dean Voets)
  - Complexity (Frühwirth&Schrijvers, Sneyers, De Koninck)
  - Abstract interpretation (Schrijvers, Stuckey, Duck)
  - …

More information about analysis:

Lectures by Slim Abdennadher,
Jon Sneyers (complexity),
Frank Raiser (state equivalence)

Application domains

- Constraint solvers
  - CHR was specifically designed for this
  - Some domains where CHR has been used:
    - Scheduling
    - Soft constraints
    - Spatio-temporal reasoning
    - Multi-agent systems
    - Semantic web
- General-purpose programming language
  - Many classical algorithms have been implemented in CHR in a very elegant and natural way - often more concise than pseudocode!

Application domains

- Programming language development
  - Type systems (e.g. Haskell type classes)
  - Abductive reasoning
  - Computational linguistics (NLP)
    - CHR Grammars (Dahl&Christi
  - Meta-programming
  - Testing & verification
- CHR can be used as a high-performance business rule engine (integrated in your favorite host language!)

More information about abduction and linguistics:

Lectures by Henning Christiansen

Good starting points

- **Book:** Thom Frühwirth, *Constraint Handling Rules*, Cambridge University Press, 2009. http://www.constraint-handling-rules.org
- **Introductory survey:** Thom Frühwirth, *Theory and Practice of Constraint Handling Rules*, Special Issue on Constraint Logic Programming (P. Stuckey and K. Marriott, Eds.), Journal of Logic Programming, Vol 37(1-3), 1998.
- **Advanced survey:** Jon Sneyers, Peter Van Weert, Tom Schrijvers and Leslie De Koninck, *As Time Goes By: Constraint Handling Rules — A Survey of CHR Research from 1998 to 2007*, Theory and Practice of Logic Programming, 2010.
- **CHR website** with bibliography: http://dtai.cs.kuleuven.be/CHR/