Figure 9.10: Swanston & Faraday Sts., frame 242. (a) Cars in intersection, original image. (b) Cars found by system, with orientations given by lines through cars, and their labels. Intersection boundary input manually, north is up. The interpretation follows:
*Give Way to oncoming \( \text{id(car,c2\_242,bel(1,1))} \) turning right from west should have (but hasn't) given way to \( \text{id(car,c1\_242,bel(1,1))} \) from east: \( \text{bel(0.40,0.97)} \)
Figure 9.11: Swanston & Faraday Sts., frame 316. (a) Cars in intersection, original image.
(b) Cars found by system, with orientations given by lines through cars, and their labels.
Intersection boundary input manually, north is up. The interpretation follows:

*Illegal: car id(car,e5216,bel(1,1)) from east is doing a right turn on the wrong side
of id(tInXn,1,bel) : bel(0.72,0.78)
Chapter 10

Conclusion

The philosopher Dan Dennett has described in [25] his concept of how consciousness is a product of a network of interacting agents, none of which is the "center" or "seat" of consciousness. In this book we have demonstrated a system for interpreting images which, likewise, has no central controller. Rather it is a network of interacting (peer) agents, which, nevertheless, produces stories about images. This is a far cry, of course, from consciousness, but it does demonstrate how interesting high-level concepts can be generated from low-level data through networks of agents. In this book, the agents (concept-frames, or in Hewitt’s language, actors) behave as Hewitt prescribed [38]. The relationship with Minsky’s frames [47], is also close, but his slots (or terminals) are not explicitly implemented in SOO-PIN, their role is taken by property lists attached to each concept-instance. Together with the high-level logic programming language in which SOO-PIN is implemented, property lists provide a flexibility well suited for building a wide variety of concept-frames necessary in a rich domain. At an engineering level, the SOO-PIN system has demonstrated that an Object-Oriented Concurrent Logic Programming approach is a viable means of producing useful image interpretation systems. Another capability shown to be feasible by SOO-PIN is the use of a form of Dempster-Shafer uncertainty calculus, which was shown to convey useful uncertainty values through the network and produce intuitively reasonable uncertainty values at the final interpretation stage.
At the level of the traffic scenario, the system generated reasonable high level interpretations which would be suitable for input into a number of systems, for instance, quite sophisticated traffic intersection statistics could be accumulated, which would be of use to highway engineers. It would not take too much extension to connect such a traffic interpretation system to actual traffic light control and adjust the lights as a function of the types of driving behaviors detected. In fact, in conjunction with a street-level camera, the system could detect and report more complex illegal movements of cars as part of a traffic law infringement system.

Clearly, the present traffic implementation has many limitations. For instance, the background subtraction technique for low-level processing is crude, and results in problems with shadows and obscuration by overhanging structures, and limits the obliqueness of the view to the near vertical (see Figures 9.5 9.6 9.11). The system would be more robust with an intelligent object recognition system, such as one based on evidence-based techniques [13][8]. Output from such low-level processors should include uncertainty values which could be incorporated with the uncertainty measures currently propagated through SOO-PIN.

Improvements to the traffic implementation could also be made in the range of concept-frames. For instance, the system was not developed to detect whether a car was intending to turn left or right while still in the road next to an intersection. Also, the whyStopped concept-frame could be extended to reason that cars stop because the car in front is blocking the way. The system could also be extended to deal with other legally significant entities like pedestrians, trams and emergency vehicles.

The uncertainty measure propagated through the system could be improved by making the system more reactive to it, for instance, eliminating hypotheses that have a belief below a certain threshold. The be1Upd message is a useful means of dealing with negative information (ie, the deletion of concept-instances) and of updating belief values around the network. This could have been implemented for the traffic scenario, except that, in some ways, the traffic scenario was quite simple, and there was no situations where
hypotheses need to be deleted. Another improvement would be to map the uncertainty values onto English phrases (hedges) for output, and it would be an interesting problem to determine the most reasonable English phrases for the various combinations of support and plausibility output by the system.

Future work for the SOO-PIN system also includes extending the network of frames into low-level processing, creating concept-frames concerned with segments, image attributes, labeled regions and objects. With such a system, it would be possible to re-segment portions of the image where high-level concept-frames expect to find objects (i.e., cars) while varying the segmentation parameters. This would be similar to the techniques used in Schema [27], Sigma [45] and that of Bell and Pau [7].

In the traffic scenario, SOO-PIN was working on sparse images derived from video image sequences. An obvious extension would be to deal with the sequence in its entirety. This would require a fundamental rethink of the architecture of the system, involving concepts of short-term and long-term memory, time-varying versus static objects, and a different form of output (whereas currently the system outputs interpretations when it has finished running, there would need to some way of determining when was the right time to generate output if it was running over a long sequence).

Finally, for the programmer, a useful improvement would be a graphical user interface (GUI) for building the system, similar to the system employed by Garvey [31] for his belief network. The current system was built using the high-level logic programming language Parlog++, which greatly facilitated the process, but it would be good to avoid the need for future programmers to learn this language. If there was a suitable GUI in which the logical nature and relationships of each concept-frame could be defined, then the Parlog++ code could be compiled from it. More speculatively, and in keeping with Cognitive Science, it would be a significant development if Machine-Learning could be involved in the construction (or evolution) of the network of concept-frames.
Appendix A

Parlog++ Procedures

In this appendix we list selected code in the Parlog++ language used in SOOPIN.

A.1 Switchboard Source Code

The first source code example is the switchboard which controls message flow around the network, and spawns new concept-frames as needed.

/* PARLOG++ 'switchboard' that accepts messages on the input stream
 and routes them to the appropriate concept-frame. If not found,
 it spawns the addressed concept-frame. */

switch.
    InStr istream, WriStr ostream

invisible ProcList state <= [], ObjType state <= switch,
    MsgCount state <= 0

clauses

    InStr::msg(NameTo,Msg) =>
        find_name(ProcList,NameTo,ProcInput,PLshort):
        %debug, inq next line
        ProcInput = [Msg|ProcInput]&
        WriStr::msg(NameTo,Msg)&        /*debug*/
        %writeMy(OutFile,msg(NameTo,Msg))&
        Count is MsgCount + 1,
writeCount(Count) &
MsgCount becomes Count &
ProcList becomes [proc(NameTo,ProcInput1)|PLshort] ;

InStr::msg(NameTo,Msg) =>
checkProc(NameTo):
Proc = .. [NameTo,[Msg|NewProcInput],NewProcOutput],
call(Proc),     /*parallel, spin off new process*/
WriStr::msg(NameTo,Msg),  /*debug*/
%writeMy(OutFile,msg(NameTo,Msg)) &

merge(NewProcOutput,InStr,NewInStr),
ProcList becomes [proc(NameTo,NewProcInput)|ProcList],
Count is MsgCount + 1,
writeCount(Count) &
MsgCount becomes Count &
InStr becomes NewInStr ;

InStr::msg(NameTo,Msg) =>
Count is MsgCount + 1,
writeCount(Count) &
MsgCount becomes Count &
WriStr::[unknown_proc,NameTo,Msg] & & ;

InStr::err(ErrMsg) =>
Count is MsgCount + 1,
writeCount(Count) &
MsgCount becomes Count &
WriStr::[error_rcvd,ErrMsg] & & ;

InStr::info(InfoMsg) =>
Count is MsgCount + 1,
writeCount(Count) &
MsgCount becomes Count &
WriStr::[info_rcvd,InfoMsg] & & ;

InStr::draw =>
drawNet(ProcList,NewProcList) &
WriStr::draw &
Count is MsgCount + 1,
writeCount(Count) &
MsgCount becomes Count &
ProcList becomes NewProcList & & ;

InStr::quit =>
WriStr::stopping_procs &
WriStr::msgCount(MsgCount) &
```prolog
stop_proc(ProcList) &

Instr::last =>
    WriStr::last &

Instr::BadMsg =>
    WriStr::[bad_message, BadMsg].

code

mode find_name(?,-^,^).

/* find process named, outputting its input stream and a proc list without that process. If not found then fail*/
find_name([proc(Name, ProcInput)|PLshort], Name, ProcInput, PLshort) <-
true;
find_name([Proc|PLRest], Name, ProcInput, PLshort) <-
    PLshort = [Proc|PLshorter] &
    find_name(PLRest, Name, ProcInput, PLshorter).

mode drawNet(?,-^).

drawNet(ProcList, NewProcList) <-
    open(network, write, NetFile) &
    drawProc(ProcList, NetFile, NewProcList) &
    close(NetFile) &
    display_network.

mode drawProc(?,-^).

drawProc([], []).

drawProc([proc(Name, ProcInput)|Rest], NetFile, [proc(Name, ProcInput1)|Rest1]) <-
    ProcInput = [dump(InList, ObjLevel)|ProcInput1] &
    writeNet(NetFile, InList, ObjLevel) &
    drawProc(Rest, NetFile, Rest1).

mode stop_proc(?).

stop_proc([]).

stop_proc([proc(Name, ProcInput)|Rest]) <-
    ProcInput = [] &
    stop_proc(Rest).

mode writeMsg(?,-^).
%write destination and type of message
```
writeMsg(OutFile, msg(NameTo, Msg)) <-
  functor(Msg, MsgType, Arity) &
  writeMy(OutFile, [NameTo, MsgType]).

mode writeCount(?).
%write count if its a multiple of 5
writeCount(Count) <-
  Rem is Count mod 5 &
  Rem =:= 0;
  write('***') &
  flush_output(user_output);
writeCount(Count).

end.
A.2 Give-Way Source Code

In the next example, a *giveWay* concept-frame is shown, that which detects
the existence of a give-way situation between a U-turning car and an oncoming
car.

giveWayUt. %give way to oncoming traffic when U-turning on road
   Out ostream
   invisible InstList state <= [], ObjType state <= giveWayUt,
   ObjLevel state <= 3, OutFile state
initial open(ObjType,write,OutFile) &
   writeMy(OutFile,............................................................................)
   xterm(ObjType) & & .
clauses
   last => writeMy(OutFile, [stopping, InstList]) &
      close(OutFile) & Out::last. /*dump insts & stop*/
   dump(OutInstList, OutObjLevel) =>
      OutInstList = InstList &
      OutObjLevel = ObjLevel & & ;
create(Id, Relns) =>
   getInst(Id, InstList, inst(TargId, Props, Justn), ExcInstList);
   writeList(OutFile, [create, unnecessary, as, Id, found]) &
   union(Relns, Props, NewProps) &
   InstList becomes [inst(TargId, NewProps, Justn) | ExcInstList]
else /*inst not found*/
   writeList(OutFile, [created, Id]) &
   checkAssns(Id) & /*send 'check' to likely
      associates with this inst Id*/
   InstList becomes
      [inst(Id, Relns, []) | InstList] & & ;
   /*dont process any more msgs until InstList is updated!*/
check(reln(composedOf, SendId, Bel), Done) =>
   %Note Done flag to delay sender until this object updated
   Done = yes &
   writeList(OutFile, [check, relation, composedOf, from, SendId]) &
   getType(SendId, SendType) &
   %get the road the 'right' is in
   Out::msg(SendType, getVal(SendId,
      reln(composedOf, id(road, any, bel), bel), RoadProps)) &
   Out::msg(SendType, getVal(SendId,
A.2. Give-Way Source Code

desc(from(any),bel,RtFrom))&
sendRoad(OutFile,InstList,RoadProps,RtFrom,SendId,
NewInstList)&

/**<found if FoundIds not empty*/
InstList becomes NewInstList&& ;

negCheck(Id) =>
  %Id is removed from the Property lists of all Insts
  writeList(OutFile,[Id,removed,from,all,relations])&
  delRef(InstList.Id,NewInstList)&
  InstList becomes NewInstList&& ;

anyInst(FoundList,Prop) =>
  writeList(OutFile,[enquiry,regarding,Prop])&
  /**<Note: this only checks properties in the PropList*/
  seek(InstList,Prop,FoundList)&& ;

getVal(WantId,Prop,FoundPropList) =>
  %propagate to other traffic obj
  getInst(WantId,InstList,inst(InstId,Props,Justn),ExcInstList):
  /**<fail if not found*/
  writeList(OutFile,[enquiry,to,WantId,re,Prop])&
  searchProps(Props,Prop,FoundPropList)&& ;

getVal(WantId,Prop,FoundPropList) =>
  writeList(OutFile,[enquiry,to,WantId,re,Prop,failed])&
  FoundPropList = []&& ; /**<If Id wrong, return []*/

updVal(Id,ReIns) =>
  getInst(Id,InstList,inst(TargId,Props,Justn),ExcInstList):
  writeList(OutFile,[update,Id,with,ReIns])&
  union(ReIns,Props,NewProps)&
  InstList becomes [inst(TargId,NewProps,Justn)|ExcInstList]

  else /*inst not found, so create it but dont checkAssms*/
  writeList(OutFile,[update,of,Id,with,ReIns,failed,not,found])&
  InstList becomes
  [inst(Id,ReIns,[])|InstList] && ;

inq => writeMy(OutFile,[inq,InstList])&& ;

WrongMsg => writeMy(OutFile,[bad_msg,WrongMsg]).

code
mode checkAssms(?,
^,?,?).
/** check normal associations of this object type to see if they exist in the
expected relationship. Note the result of this checking is returned in
msg 'updVal' */
checkAssms(InstList,InstList,InstId).

/*******************************************************************************/
mode sendProps(?,
,?).
/** send list of properties to another object instance */
sendProps(._[[]];
  %if no props, dont send anything
sendProps(id(ToType,ToNum,Bel),Props) <-
  Out::msg(ToType,updVal(id(ToType,ToNum,Bel),Props)).

/*******************************************************************************/
mode sendRoad(?,
,?,?,?,?`). %OutFile,InstList,RoadProps,
%RtFrom,SendId,NewIL
%sent to straight to get cars from other dirn
sendRoad(OutFile,InstList,[reln(composedOf,id(road,RoadNo,Bel1),Bel1)|Rest],
  [desc(from(Dir),BelId)|Rest2],SendId,NewIL) <-
  length(Rest,Len)&
  writeListCond(user_output,Len,[SendId,composed,of,
    more,than,one,inXn,Rest])&
  length(Rest2,Len2)&
  writeListCond(user_output,Len2,[SendId,from,more,
    than,one,dirn,Rest2])&
  Out::msg(straight,anyInst(FoundStr,reln(composedOf,
    id(road,RoadNo,Bel1),Bel1)))&
  getIdLFromInstL(FoundStr,FoundIds) & %\%ds
  checkTarget(OutFile,InstList,targetOf,FoundIds,
    SendId,NewIL);

sendRoad(OutFile,InstList,[],_,_,InstList); %empty compose
sendRoad(OutFile,InstList,[],_,_,InstList); %empty from list
%error
sendRoad(OutFile,InstList,Wrong1,Wrong2,Wrong3,InstList) <-
  writeList(OutFile,[bad,call,to,sendRoad,in,giveWayUt,ie,Wrong1,
    Wrong2,Wrong3]).

/*******************************************************************************/
mode checkTarget(?,
,?,?,?,?`). %OutFile,InstList,TargReln,CommStr,SendId,NewIL
%before creating the inst, check car is pointing toward subject
checkTarget(OutFile,InstList,TargReln,[Id|Rest],SendId,NewIL) <-
  Out::msg(uTurn,getValue(SendId,reln(composedOf,
    id(car,any,bel),bel),SendCarPropList))&
  getType(Id,IdType)&
A.2. Give-Way Source Code

```c
Out::msg(IdType, getValue(Id, reln(composedOf, id(car, any, bel), bel),
    CarPropList)) &
checkTarget2(OutFile, InstList, TargReln, Id, CarPropList,
    SendId, SendCarPropList, NewIL1) &
checkTarget(OutFile, NewIl1, TargReln, Rest, SendId, NewIL);

checkTarget(,, InstList, , [], , InstList).

/*********************
mode checkTarget2(?=?,?=?,?=?,?=?,?)
%get the single car from the lists CarPropList & RightCarPropList, send
%to Osr to find if car coming targets left. if so call create.
%first check case where sender is target of car to give way to
checkTarget2(OutFile, InstList, targetOf, Id, [reln(composedOf, CarId, Bel)]|Rest],
    SendId, [reln(composedOf, CarSendId, Bel2)|Rest1], NewIL) <-
Out::msg(car, getValue(CarId, desc(reversed, bel), Return)) &
reverseReln(targetOf, Return, RReln) &
checkOsr(RReln, CarSendId, CarId, FoundProps) &
condCreateInst(OutFile, InstList, Id, CarId, SendId, CarSendId,
    FoundProps, NewIL);
%next check case where sender is targeting car to give way
checkTarget2(OutFile, InstList, targeting, Id, [reln(composedOf, CarId, Bel)]|Rest],
    SendId, [reln(composedOf, CarSendId, Bel2)|Rest1], NewIL) <-
Out::msg(car, getValue(CarSendId, desc(reversed, bel), Return)) &
reverseReln(targeting, Return, RReln) &
checkOsr(RReln, CarSendId, CarId, FoundProps) &
condCreateInst(OutFile, InstList, SendId, CarSendId, 
    Id, CarId, FoundProps, NewIL);

checkTarget2(OutFile, InstList, , , [], SendId, , InstList) <-
    writeList(OutFile, ['Error in checkTarget2, got no cars composing',
        Id]);
checkTarget2(OutFile, InstList, , , , , [], InstList) <-
    writeList(OutFile, ['Error in checkTarget2, got no cars composing',
        SendId]).

/*******************
mode reverseReln(?=?,?=?)
%if Return from car contains desc(reversed, bel), ie is non-empty,
%then put 'B' on the end of input reln and pass back
reverseReln(Reln, [desc(reversed, BelID)], NewReln) <-
    concatStr([Reln, '''B'''], NewReln);
reverseReln(Reln, , Reln).

/*********************
mode condCreateInst(?=?,?=?,?=?,?=?,?)
%OutFile, InstList, SignDesc,
%StrId, StrCarId, StDir, SendId, CarSendId, SeDir, FoundProps, NewIL
```
% dont create if FoundProps is []
condCreateInst(_,InstList,_,_,_,_,[],InstList);
% test if new components are in existing Inst, create if not
condCreateInst(OutFile,InstList,Id,CardId,SendId,CardSendId,FoundProps,NewIL) <-
    duplist(InstList,[relin(composedOf,Id,be1),
    relin(composedOf,SendId,be1)],DupIds)&
%DupIds is list of Ids of Insts with components
DupIds =@ []:
createInst(OutFile,InstList,Id,CardId,SendId,
    CardSendId,[relin2(dum,Id,SendId,be1)],NewIL); %Ids
% or do nothing
condCreateInst(_,InstList,_,_,_,_,_,InstList).

/**********************
mode createInst(?,?,?,?,?,?,?),
% make new insts for members of CommStr
createInst (OutFile,InstList,Id,CardId,SendId,
    CardSendId,Justn,NewIL) <-
    getNewOrAddr(NewNo)&
    belTwoDfN(Justn,NewBel)&
    union(InstId,giveWayUt,NewNo,NewBel),[relin(composedOf,Id,be1),
    relin(composedOf,CardId,be1),relin(composedOf,SendId,be1),
    relin(composedOf,CardSendId,be1),
    desc(subject(CardId),bel)],Justn],InstList,NewIL)&
    getVel(CardSendId,vel)&
    velPhrase(vel,Phrase)&
    name(NL,[10,13])&
    Out::msg(result,result([''''Give Way to Oncoming: ''',CardSendId,NL,
    '''' U-turning''',Phrase,to,CardId,going,
    straight,'''': ''',NL,NewBel])&
    writeList(OutFile,[adding,inst,NewNo,composed,of,Id,SendId,CardId,
    CardSendId])&
    getNewOrAddr(NewNo1)&
    Out::msg(giveWay,create(id(giveWay,NewNo1,NewBel),
    [relin(givesWay,CardId,be1),relin(rightOfWay,CardSendId,be1)]))&
    sendProps(Id,[relin(partOf,Id,giveWayUt,NewNo,NewBel),be1])&
    sendProps(SendId,[relin(partOf,Id,giveWayUt,NewNo,NewBel),be1])].

/**********************
mode velPhrase(?,?), %OffendId,velocity phrase
% send msg to result conditional upon speed of OffendId
velPhrase([],''' gives way '''); %velocity unknown
velPhrase([Vel],'''has given way''' <-
    stopped(Vel,stopped):
    true;
velPhrase([Vel],'should have (but hasn't) given way');
velPhrase(Vel,_) <-
    writeMy([error,in,velPhrase,parameter,is,Vel])&
    fail.

//===--------------------------------------------------------------------------==/
mode getVel(?). %id(Car,CarNo,Bel),[desc(vel(X,Y),BelV)]
%get the velocity desc from car, if not there return [] %vel
getVel(id(Car,CarNo,Bel3),Vellist) <-
    Out: :msg(Car, getVal(id(Car,CarNo,Bel3), desc(vel(any,any),bel),
                        Vellist)).

end.
References


REFERENCES


REFERENCES


REFERENCES


REFERENCES


Index

A* search, 6
acceleration, 97
accuracy, 106
actors, 17, 24
acyclic networks, 66
affine transformation, 51
Allen, 27
Andre et al, 8
assignment of weights, 68
associative network, 6
assumption-based truth maintenance system, 7, 84
ATMS, 7, 84
ATN, 5, 6
augmented transition network, 5
background subtraction, 60
Bajcsy et al, 5, 26
Baldwin, 2, 3, 68, 71, 77, 80, 81
BaRT, 67
Bayes, 66, 72
Bayesian belief network, 10
belief, 3, 66
belief and vision, 81
belief interval, 69
belief pair, 69, 78, 81, 82, 84, 106
belief updating, 83
Bell and Pau, 10, 18, 121
belTwoOfN, 77, 79, 83
belUpd, 84
blackboard systems, 18
Bobick and Bolles, 10
Boddington et al, 7, 106
Bogler, 67
bottom-up, 4, 17, 27, 42, 47
Bunke, 28
burglars, 68, 70
car velocities, 52
CARRS, 7
causal support, 66
centroid, 48, 90
Cipolla and Yamamoto, 89
CITYTOUR, 8
Cohen et al, 25
collisions, 96, 100
combining belief, 77
combining evidence, 70, 82
combining independent propositions, 71
communication channel, 22
comparing match lists, 92
compatibility mappings, 67
compound objects, 30, 37, 46, 77
INDEX

concept-frame, 3, 24, 31, 83
concept-instance, 24, 82
conceptual dependency, 17
concurrent execution, 2, 42, 47
concurrent prolog, 20
conditional probabilities, 66
conflicting evidence, 70
conjunction, 72, 76, 79
connected labeled regions, 60
constraint propagation networks, 6
correspondence problem, 3, 89
curvature, 99
cutlery scenario, 3, 35
data channel, 31
data structure, 31, 82
deadlock concept-frame, 53
deletion of instances, 84
Dellepiane et al, 6
Dempster, 70
Dempster-Shafer, 66, 68, 77, 80
Dennett, 119
dependency, 72, 82, 84
disjunction, 74, 76, 79
domain knowledge, 1, 30, 106
Draper et al, 17
Dubois and Prade, 81
evidential reasoning, 67
evidential support, 66
existence criteria, 30, 83
Feller, 77
Feri et al, 9

first order predicate calculus, 20
frame of discernment, 66–68, 71,
78, 81
frames, 3, 16
FRIL, 68
fuzzy confidence measures, 86
Gabor filter, 89
Garvey, 68, 121
geometric scene description, 8
give-way concept-frame, 53
Govindaraju et al, 6
GRANT, 25
graph parsing, 4
Green, 24
Hayes, 30
heading angle, 52
Herskovits, 2, 15, 29
Hewitt, 2, 17, 21, 24, 119
high-level vision, 1, 2
highway traffic scenes, 67
Horn clause, 72, 81
Huang et al, 10, 67
HUGIN, 10, 67
human categorization, 14
hypothesize and test, 2, 4, 6, 10
idealized cognitive models, 14
identity, 82
identity problem, 30
illegality, 94, 100, 106
image sequences, 88
independence, 77, 80
intersection, 51
IPRS library, 60
justification list, 83, 84
Keller et al, 86
Lakoff, 2, 14, 19
linguistic hedges, 86
logic programming, 2, 10, 20, 81
low-level processing, 3, 48, 60
Lowrance et al, 67, 69, 70, 81
Mackworth, 7
major axis, 48, 52, 90
MAPSEE, 7
Markov trees, 66
mass distribution, 67, 69
Matsuyama and Hwang, 9
median filtering, 60
message passing, 31, 84
Minsky, 2, 16, 21, 119
Mohnhaupt and Neumann, 51, 89
Mulder et al, 6
multi-threaded systems, 5
multiple frames, 104
nearness predicate, 29, 85
negative information, 30, 42
negCheck, 84
network-of-frames, 28, 47
Neumann, 8, 18
Nieman et al, 6
non-monotonic reasoning, 30
object orientation, 2, 9, 21
optical flow, 88
orientation, 90, 91
pairing, 90, 91
parlog, 20
parlog++, 3, 23, 28, 31, 78, 81, 93
parsing, 5
Pearl, 66, 80, 81
pedestrian, 97
phase space, 51
PLANNER, 20
plausibility, 69, 76, 82
predicate calculus, 6
probability, 66, 77
procedural subroutines, 84
prolog, 10, 20
Provan, 7
proximity to boundary, 85
real images, 48
red light, 103
Reiter and Mackworth, 7
results, 103
Ringwood, 27
road, 51
Robinson, 20
rotation rates, 92
runtime experiments, 85
scene analysis, 2
scene model, 26
Schank, 2, 17
SCHEMA, 17
Schema, 121
Schirra et al, 8, 86
scripts, 17
segmentation problem, 105
semantic network, 6
SGI Personal Iris, 106
Shapiro and Takeuchi, 2, 22
SIGMA, 9, 18
Sigma, 121
single-threaded systems, 2, 5
situatedness, 29
Slezak, 2, 30
smalltalk, 21
SOCCER, 8
SOO-PIN, 1, 24, 81
sparse image sequences, 89
spatial database, 47, 48
spatial predicates, 29, 66, 84, 93
spatial prepositions, 15
Sun SparcStation II, 106
support, 69, 75, 82
switchboard, 28, 44

T-intersection, 51, 103, 105
targeting, 104
tense, 88, 97
token matching, 2, 88
top-down, 4, 17, 27, 42, 47
traffic, 2, 3, 48, 88, 103
traffic jam, 53
traj concept-frame, 93
trajectory, 90, 92
transition network, 2, 5

Tropf and Walters, 5
Tsotsos, 27
turn activity, 94
turn concept-frames, 51
Tweety, 80

U-turn, 106
uncertainty, 3, 66, 81, 88
velocity, 2, 3, 88, 89, 91, 104, 105
Venn diagram, 77
Verri and Poggio, 88
video images, 3, 88
visual tracking, 89

Wesley, 68
wheels scenario, 3, 42
whyStopped concept-frame, 97, 104
world model, 26
wrong interpretations, 106
wrong side of road, 94

XFIG, 53, 86, 103