

Dept. of Microelectronics and Information Technology

Course Notes for Logic Programming, 2G1121 (FoV 4057)

Latest update September 14, 2002

Thomas Sjöland, sjoland@it.kth.se

Laboratory of Electronic and Computer Systems

IMIT, Department of Microelectronics and Information Technology

IT-Universitetet

KTH, The Royal Institute of Technology

Introduction

These lecture notes are meant as a complement to the textbook and to the programming manual. We give here the structure of the course as it has been redesigned in 2002. It now contains six two hour lectures and four two hour tutorials where the course material is presented. Two two-hour lectures are scheduled to handle presentations of projects in class.

There is a project task of 2p which you can perform alone or in a group of at most four students. Apart from participating in lectures and tutorials you need to study a lot on your own (with e-mail feedback from the course leader).

An examination of 2p is given at the end of the course where you can get extra points if your project task has been approved before the examination date.

We use the book “The Art of Prolog: advanced programming techniques” by Leon Sterling and Ehud Shapiro. Second Edition. ISBN 0-262-19338-8. Programs and examples that can be useful when reading that book are available via anonymous ftp from `mitpress.mit.edu` in the directory `pub`. Reading recommendations are associated to the different chapters below.

An alternative book that we recommend is “Logic, Programming and Prolog” by Ulf Nilsson and Jan Małuszyński. Second Edition. ISBN 0-471-95996-0. It is available on the www as a pdf-file at <http://www.ida.liu.se/>. The chapters 13, 14 and 15 of that book are not required reading for the examination. You can also skip sections 3.5, 4.3, 4.4, 4.5, 4.6, 4.7, 6.3, 8.5 and 8.6.

For the project task, but also for the written examination you will find the SICStus Prolog Manual useful. The latest version of information on SICStus Prolog including an online manual in HTML is available from SICS at <http://www.sics.se/sicstus/>.

Contact the course leader to get access to a binary version of SICStus Prolog to use on your own computer during the course. D-students can also use the common facilities provided by NADA in the “datorlab” at campus. E-students and others should contact the course leader if they want to solve the need for computing resources in some other way.

It is also possible to use other prolog systems. There are links to other systems on the course web page.

Contents

1	Theory for Logic Programming	11
1.1	Logic as a Programming Language	12
1.2	History	13
1.3	Programs are Theories	14
1.3.1	Representing Data in a Logic Program	15
1.3.2	Representing a Program	16
1.4	Program Representing Family Relations	19
1.5	Logical Variables - Binding	20
1.5.1	Family Relations Continued: Rules	21
1.6	The Equality Theory - Substitutions	21
1.6.1	Testing Equality	22
1.6.2	Substitutions	22
1.6.3	Unifier	22
1.6.4	Most General Unifier	23

1.6.5	Unification	23
1.6.6	Quantification of Variables and Parameters	24
1.6.7	Declarative and Procedural Reading of Programs	25
1.6.8	Model Theory: Least Herbrand Model	26
1.6.9	Proof Theory	27
1.6.10	Conclusion	31
2	Good Programming Style	33
2.1	Specifying the Use of a Procedure	34
2.1.1	Mode Declarations	34
2.2	Arithmetic Expressions	35
2.2.1	Expression Evaluation and Unification	35
2.2.2	Different Equals	36
2.3	Recursive Programming Techniques	37
2.3.1	Composing Recursive Programs	37
2.3.2	Reasoning About Natural Numbers	37
2.3.3	Recursive Arithmetic	38
2.3.4	Programming with Recursive Structures	39
2.3.5	More on Syntax	40
2.3.6	Programming with Lists	43
2.3.7	Dictionaries	47

<i>CONTENTS</i>	7
2.3.8 Final Example	48
2.4 Efficient Representations	49
2.4.1 Example: Sets Represented as Lists	50
3 Advanced Recursive Techniques	51
3.1 Iterative and Recursive Definitions	51
3.1.1 Accumulating Parameters	52
3.1.2 Programming with Abstract Data Types	54
3.1.3 Difference Lists	56
4 Search Based Programming	61
4.1 Search	62
4.1.1 Controlling Search	62
4.1.2 Example: Execution of a Program with Cuts	63
4.1.3 Inserting Cuts in Programs	64
4.1.4 Green Cut: An Example	65
4.1.5 Green Cut: An Example (cont.)	65
4.1.6 Red Cut: An Example	65
4.1.7 Red Cut: Another Example	66
4.1.8 Implementation of if-then-else	67
4.2 Using and Proving Negative Information?	67
4.2.1 Negation: SLDNF-Resolution	71

4.3	Generate-and-Test Algorithms	72
4.3.1	An Example in Prolog	72
4.4	Searching in a State-Space	75
4.4.1	Puzzle: Missionaries and Cannibals	77
4.5	Different Search Methods	78
4.5.1	Game Playing	81
5	Logic and Grammars	83
5.1	Grammars	83
5.1.1	Context Free Grammars	84
5.1.2	Recognizers for Languages: Lexers and Parsers	88
5.1.3	A Grammar for Horn Clause Statements	91
5.2	DCGs, Lexers and Parsers	91
5.2.1	Definite Clause Grammars	91
5.2.2	Compilation of DCGs Into Prolog	93
5.3	Compiling	96
5.3.1	Phase 1: The Lexical Analyser	97
5.3.2	Phase 2: The Grammatical Analyser, Parsing	97
5.3.3	Phase 3: The Transformer	98
5.3.4	Phase 4: Code Generation	98
5.3.5	BASIC Programs- Concrete Syntax	98

<i>CONTENTS</i>	9
6 BASIC Interpreter	105
7 Program Transformation, Meta-programming	113
7.1 Transforming Programs	113
7.1.1 Program Transformation Rules	114
7.1.2 Equality Reordering	114
7.1.3 Equality Removal	114
7.1.4 Clause Level Transformation	114
7.1.5 Removal of Failing Clauses	115
7.1.6 Removal of Repeated Goals	115
7.1.7 Reordering of Goals	115
7.1.8 Partial Evaluation	116
7.2 Stepwise Enhancement	116
7.3 Functions are Deterministic Relations	117
7.3.1 Higher Order	117
7.3.2 Functional techniques, <code>apply</code> , <code>map</code>	117
7.3.3 All-Solutions Predicates	118
7.4 Meta-Programming	119
7.4.1 What is a Meta-Statement?	120
7.4.2 Meta-Logic	120
7.4.3 Ground Representation of Facts and Rules	121

7.4.4	Why Self-Interpreters?	122
7.4.5	Non-Ground Representation of Facts and Rules	123
7.4.6	Meta-Logical Predicates in SICStus	124
7.4.7	Support for Dynamically Changing Knowledge Bases	129
8	Expert Systems	131
8.1	Production Rules	132
8.1.1	Rule Base as Prolog Clauses	132
8.1.2	Uncertainty	133
8.1.3	User Interface - Dialogue	133
8.1.4	Explanation Facilities	133
8.1.5	Knowledge Acquisition	133
8.1.6	Expert System: Diagnosis	134
8.1.7	Expert Systems: An Example	135
9	Constraint Logic Programming (CLP)	139
9.1	Constraint Logic Programming	139
9.1.1	Constraint Programming - A Planning Perspective	142
9.1.2	OR-Techniques	146
9.1.3	OR-Techniques vs. Constraints	150

Chapter 1

Theory for Logic Programming

Outline

- Informal Introduction to Logic Programming Theory
- Data in Logic Programs: Constants, Structures, Logical Variables
- Equality Theory, Unification
- Logic Programs: Definite (Horn) Clauses
- Procedural - declarative
- Imperative - logical style
- Model Theory (least Herbrand model, term interpretation)
- Proof Theory and Operational Semantics (resolution, proof trees)
- Concurrent logic programming models
- Parallel execution on multiprocessors
- Elementary programs
- Simple databases

Sterling and Shapiro ch. 1, 2, 4, 5, 6 (not 2.5, 4.3, 5.2, 5.3, 5.5, 5.6, 6.3)
Nilsson and Małuszyński ch. 1, 2, 3, 6

1.1 Logic as a Programming Language

The idea of Logic Programming is to use a limited, but efficient, algorithm for proving theorems as the basis of a programming language. Logic Programming is a powerful paradigm with roots in research on Artificial Intelligence, which tries to bridge the gap between system specification and coding. The subset of first order logic used to express logic programs is often suitable as a language for system specification. A logic program has a *declarative* reading as a specification (a theory) expressed in a subset of first order predicate logic. Sometimes logical specifications are directly executable. The program is then said to have a *procedural* reading. When it is possible to use the same code for specification and execution the semantic gap between specification and implementation is eliminated. Even when this is not possible or desired, and the specified data and relations have to be *encoded* or *programmed* using more primitive structures, this often gives programs a natural appearance that can be a strong help in the creation of error-free code. A logical specification can also consider data objects to represent programs, proofs etc. These can be manipulated by a meta-interpreter allowing flexible programming techniques to be utilised.

Problem areas where Logic Programming is particularly useful are:

- Theorem proving
- Interpreters and compilers
- Graph-algorithms
- Protocol verification
- Knowledge representation
- Expert systems, decision support systems
- Planning and scheduling
- Processing of natural language
- Database modelling

In logic programming a program is a theory consisting of a set of formulae in *clausal form* (see 1.3.2 later). You can ask questions in the form of conjunctive propositions to the system. The construction of a proof is considered as equivalent to an execution. The possibility to search for a proof via alternative routes in a controlled fashion and the flexibility that you have in choosing different ways to construct a proof are also attractive both for the programmer and the programming system constructor. If a direct construction (without search) of a proof is made, the complexity of such an execution is of the same order as that of a traditional programming language. This gives a program that can be understood as an *executable specification*.

Since any programming language can be given a “logical” operational semantics in the form of proof rules (see course on semantics, 2G1117), a natural question that arises is what is specific to logic programming. A common answer is that logic programming is *declarative*, that is that a program can be read as axioms, where the meaning of the program is given without reference to the order of execution. In logic this is called model theory (see 1.6.8). Another answer, which does not involve model theory, is that in logic programming all steps of a computation can be considered as logically valid inference steps. This approach to the semantics is called proof theoretical using a term from logic. You can in principle write programs in a subset of any programming language that can be considered to be *logic programs*.

1.2 History

The idea of using logic as a programming language has been present in the computer science community for a long time. Programs to prove theorem were implemented early, but with only limited capacity regarding speed and storage and more seriously with inherent problems of search complexity. The use of the so called resolution principle to guide the construction of a proof in a theorem prover as suggested by Alan Robinson in the mid-sixties was an important contribution towards the goal of actually designing a programming language based on the construction of logical proofs. These ideas that were developed by Robert Kowalski and his colleagues at Imperial College in London from the mid-seventies coincided with work on executable grammars for parsing for instance of natural language performed by Alain Colmerauer and his colleagues in Marseille and giving the first implementation of Prolog in 1972. The first

efficient compiler for Prolog occurred in Dec-10 Prolog by David Warren et. al. in 1977 [Wa1] and the later so called WAM (Warren Abstract Machine) by the same author. Since a few years Prolog is standardised by ISO.

Logic Programming has attracted the interest of many researchers and it is still an active field of research, nowadays mostly in the form of various extensions and incorporations of several programming paradigms in the same framework. Although this is not the case for Prolog as you will see, logic programming taken more generally lends itself to a parallel execution with automatic distribution of small tasks over a network of processing elements. The possibility to automatically synthesise or transform programs using logical axioms was an idea which also occurred in the seventies that is still actively pursued by researchers. As in functional programming, which shares many ideas with logic programming, there is for instance research on partial evaluation systems to make programs more efficient and program analysis systems to find information for debugging or more efficient compilation methods.

Current state-of-the-art logic programming systems such as SICStus Prolog [SICStus] are among the most efficient problem oriented programming systems. They contain so called constraint programming (see chapter 9) as an integrated part and give support for interoperability with other programming systems.

A recent approach is to incorporate logic programming in so called *multi-paradigm-systems*, such as Mozart/Oz (see <http://www.mozart-oz.org>). Other than supporting many different programming paradigms in a coherent system, the focus in Mozart/Oz is on internet awareness and distributed and concurrent programs. This system is for instance used in the course 2G1126 on Distributed Computer Systems.

1.3 Programs are Theories

In logic programming programs are (first order) theories, sets of relations in some world of objects.

1.3.1 Representing Data in a Logic Program

The data items of a logic program, the logical *objects* (or *terms*) are represented as structures. A structure is a named frame containing components which are structures, constants or variables. Each structure has a fixed number of components (fixed arity). A special structure is the constant which could be considered to be a structure without components. To discriminate a structure from a variable the syntactic convention is that normally the name (the functor) of a structure begins with a small letter. Any string may, though, be considered to be a structure identifier if it is contained within single quotes (or using some other special syntax, see manual [SICStus]).

A World of Structures: (constants, structures)

Individual Constants (Atoms): a b foo 4711 -37 34.5 'X'

Compound structures (Trees): foo(a,b,-4711,bar(b,a))

Arity. A structure `foo` with 4 arguments is indicated thus: `foo/4`. A constant has no arguments, so we say for instance `a/0`.

There is a special syntax for dealing with structures that can be considered to be *lists*. We will return to this later.

Syntax of Variables

There are also variables. These range over structures and constants. Variables in logic programs can occur wherever constants or structures occur. They can be bound to structures, constants and to other variables.

The syntactic convention adopted in Prolog for representing variables is that variables begin with an upper case letter. The syntactic convention keeps the understanding of the program unambiguous. Here are some examples of variables:

X Y Z Foo Y123 Hello This_is_a_very_long_variablename_indeed

`_` is a void, that is uninteresting, (single-occurrence), variable which can be bound to any term. If you use several voids each is considered as a variable with a unique occurrence.

1.3.2 Representing a Program

A *program* is a set of relation definitions, named *predicates*, and a *query*, that is a logical formula to be proven assuming the defined relations as axioms. A logic program is sometimes named a *definite program*.

Query, Goal : Formula to be Verified (or Falsified)

Questions posed to the system are named *goals* or *queries* and are of the form $?- Q_1, \dots, Q_n$.

`,` is read as “and” and the Q_i are literals representing elementary relations to prove.

Definite Clauses: Relation (Predicate) Definitions

The classical form of a definition in logic programming uses a set of *definite clauses* written in a form where a positive literal P has exactly one occurrence in each clause:

$$P \text{ or not } Q_1 \text{ or } \dots \text{ or not } Q_n$$

To avoid the negation operator for literals this can be expressed as $P \text{ if } Q_1 \text{ and } \dots \text{ and } \dots Q_n$.

Definite clauses are also named *Horn Clauses* after the logician Alfred Horn.

Definitions in Prolog

In Prolog, a relation (predicate) is defined as a collection of definite clauses of the form $P :- B$, where P is named the *head* and B is named the *body*. The body of a clause is (a conjunction of) elementary literals. $:-$ is read as “if”, $,$ is read as “and”.

There are two forms of definitions:

- **Facts or Assertions:**

Atomic formulae of the form $P :- \text{true}$. also written P .

A fact represents a known truth about its parameters.

- **Rules or Conditional Facts:**

Conditional formulae of the form $P :- Q_1, \dots, Q_n$.

P is called the head and Q_1, \dots, Q_n the body of the clause and Q_1, \dots, Q_n are atomic formulas (literals, relation symbols), predefined or defined by another clause.

The symbol $;$, interpreted as, “or” is **syntactic sugar**, that is, it is not strictly necessary. It can be used to avoid defining auxiliary predicates or to reduce the number of clauses. A clause of the form

$$P :- Q_1, (Q_2 ; Q_3), Q_4.$$

is the same as

$$P :- Q_1, Q, Q_4.$$

$$Q :- Q_2.$$

$$Q :- Q_3.$$

We assume here that the variables of $(Q_2 ; Q_3)$ become the arguments of the predicate Q .

The definite clauses can be written as a relation definition where there is only one occurrence of the head of a clause for each defined predicate P :

P if $(Q_{11}, \dots, Q_{1n}) ; \dots ; (Q_{m1}, \dots, Q_{mn})$.

$1..m$ and $1..n$, are index sets large enough to cover all goal atoms, denoted by Q_{ij} .¹

If all goals Q_{ij} are true the clause is written P .

Elementary Literals (Atoms)

Some of the Q_{ij} may be predefined relation symbols ($=$, true , false , etc.) These literals cannot be redefined (only used in queries and definitions). Prolog contains many other built-in predicates, but they are not covered by the theory (see manual [SICStus]).

Clause Syntax

In Prolog relation symbols and predicate names use the same syntax as structures. The two sets are distinguished by their position in a program.

Example:

```
p(17).
p(X) :- X<8, q(X).
p(X) :- q(X), X=s(Y), p(Y).
```

In English the above example could be stated as follows:

- The property p holds for the constant 17.
- The property p holds for a term denoted by the variable X if $X<8$ and q holds for X .

¹We have simplified a bit by using the same n for all clauses, that is we assume the same number of goals in all clauses. This is not essential, and you should think about clauses with a different n for each clause.

- The property p holds for X if q holds for X , X equals a term $s(Y)$ and p holds for Y .

When several clauses exist with the same name, they represent alternative facts and rules about this relation. The conjunctive connective `,` connects the goals in the body together, all of which should be true for that alternative of the relation to hold.

1.4 Program Representing Family Relations

```
%male(Person)
male(stig).
male(erik).
male(jonas).

%female(Person)
female(eva).
female(lisa).
female(lena).

%parent(Parent, Child)
parent(erik, lisa).
parent(eva, lisa).
parent(erik, jonas).
parent(stig, erik).
parent(lena, jonas).
parent(ulf, erik).

?- parent(lena, jonas).
Yes.

?- parent(X, jonas).
X=erik;
X= lena.
```

```
?-parent(X,Y).
```

```
X=erik, Y=lisa;  
X=eva, Y=lisa;  
X= erik, Y=jonas;  
...
```

; is input by the user to indicate that another solution is wanted.

1.5 Logical Variables - Binding

- Variables range over structures
- Variables can be bound to structures and to other variables
- The semantics of variables is single-assignment

The variables of an ordinary imperative language (C, PASCAL, FORTRAN) are abstractions of cells in the computer's memory and the imperative program describes successive manipulations that are to be undertaken on these memory cells in order to transform the input data into the output. The order of these manipulations is therefore in many cases crucial in imperative programs.

In contrast, variables in logic programs are bound during the construction of a proof (an execution). A logic variable may be in either of two states during the execution: *bound* or *unbound*. It starts in the unbound state and during the computation it may become bound. Once a variable has been bound to some structure, or to another variable, it is not possible to release this binding in the rest of the proof.

After a proof has been found the bindings on the variables included in the query are presented to the user. The bindings of variables represent the minimum restrictions made on the values of the variables to make the proof valid. The order in which the bindings are made does not matter for the correctness of the proof. The order of bindings in a logic program might, however, affect the behaviour, e.g. termination, of a program depending on different allowed

orderings and in combination with some other operations the order in which the bindings are produced might change the result.

When a branch of a sub-proof fails and an alternative branch is tried (normally through the *backtracking* mechanism, see section 1.6.9) a bound variable can become restored to the unbound state.

1.5.1 Family Relations Continued: Rules

```
father(Dad, Child) :- parent(Dad, Child), male(Dad).
mother(Mum, Child) :- parent(Mum, Child), female(Mum).
sibling(X,Y) :- parent(Z,X), parent(Z,Y).
cousin(X,Y) :- parent(Z,X), parent(U,Y), sibling(Z,U).
```

```
?- father(X,Y).
X=erik, Y=jonas.
X=erik, Y=eva.
```

```
?- mother(erik, jonas).
No.
```

```
?- mother(Erik, jonas).
```

Succeeds – Why?

1.6 The Equality Theory - Substitutions

The traditional way to introduce the equality theory of logic programming involves substitutions.

First consider what it means for two terms to be equal. Here we use the name *Term* for the union of the set of variables and the set of objects.

1.6.1 Testing Equality

X equals Y iff

X is an unbound variable or Y is an unbound variable

or

X and Y are (bound to) the same constants

or

X and Y are structures with the same functor and arity

for example X is $\mathfrak{t}(Z_1, \dots, Z_n)$ and Y is $\mathfrak{t}(U_1, \dots, U_n)$

and for all arguments $1 \leq i \leq n$: Z_i equals U_i .

It is understood that the values of the variables are understood in the context of a set of variable bindings, an *environment*. An environment in logic programming is a substitution which we will define next.

1.6.2 Substitutions

A *substitution* is a function $Subst: Var \rightarrow Term$

We may represent a substitution as a set of simple bindings from a variable to some term $\{X/\mathfrak{t}1, Y/\mathfrak{t}2\}$. We may also write a substitution as a formula, a conjunction of simple equalities of the form $v=\mathfrak{t}$ where a variable v occurs on the left hand side in at most one of the simple equalities.

A substitution θ can be applied to terms (written $t\theta$) and also to formulae, or to another substitution σ (written $\sigma\theta$).

1.6.3 Unifier

A unifier is a substitution θ such that $s\theta \equiv t\theta$.

(applying the substitution θ to s and to t creates identical terms).

1.6.4 Most General Unifier

A unifier θ is more general than another unifier σ
iff there exists another unifier ω such that $\omega\theta = \sigma$.

A unifier θ is the most general unifier of two terms
iff θ is more general than any other unifier of the two terms.

Example: $t(X, Y, Z)$ and $t(U, V, W)$ are unified by $\{X/a, Y/b, Z/c, U/a, V/b, W/c\}$

but consider also for instance $\{X/U, Y/V, Z/W\}$ and $\{U/X, V/Y, W/Z\}$ which are mgus.

1.6.5 Unification

The operation of unifying two terms is called *unification*. It is common to talk about the result of the unification operation as a unification also, while it is perhaps more clear to talk about the substitution. An algorithm that constructs most general unifiers is a *unification procedure*.

Unification in a logic program is expressed with the predicate `=/2`. For instance the unification literal

```
foo(bar(X,W),Z,W)=foo(A,B,baz(Z,47))
```

makes the two terms equal by generating the mgu
 $A=bar(X,W), Z=B, W=baz(Z,47)$.

The terms are then both equal to `foo(bar(X,baz(B,47)),B,baz(B,47))`, the least limited term that unifies both of them. `Z` and `B` are considered to be equal, but as yet unknown. The variable `X` is left unbound.

Since the most general unifier is unique (up to renaming of variables), unification can be understood as a function *mgus*: $Subst \times Term \times Term \rightarrow Subst$

Compound Terms for the Description of Persons

```

male(person(erik,50,180cm)).
male(person(jonas,25,_)).

parent(erik,jonas).
parent(erik,eva).
parent(lena,jonas).

father(Dad,Child) :-
    Dad=person(DadName,-,-),
    Child=person(ChildName,-,-),
    parent(DadName,ChildName),
    male(Dad).

?- father(person(-,50,-), person(X,-,-)).
X=jonas;
X=eva.

```

How does the unification algorithm work here?

1.6.6 Quantification of Variables and Parameters

Variables occurring in the head part of clauses are called *parameters*. Other variables are sometimes called *local variables*. Parameters are implicitly for-all-quantified. Local variables are implicitly existentially quantified. For instance:

```
p(X,Y) :- q(X,Z), r(Z,Y).
```

is understood as

$$\forall X, Y : (p(X, Y) \leftarrow \exists Z : (q(X, Z), r(Z, Y)))$$

Parameters are often confusingly named *global variables* as opposed to *local variables*.

But if X is *global* and Y is *local*, what is Y , if $X=Y$ occurs in the program?

(The terminology often used is that unification of a global variable with a local variable *makes* the local variable global, but this notion relies on the order of the execution, while it is preferable to use a terminology that relates only to the clause read as a logical formula.)

Example Continued: Recursive Rules

```
%ancestor(Ancessor,Descendant):- ...
ancestor(X,Y) :- parent(X,Y).
ancestor(X,Y) :- parent(X,Z), ancestor(Z,Y).
```

```
?- ancestor(X, Y).
```

What is the result?

1.6.7 Declarative and Procedural Reading of Programs

A logic program can be understood in either of two ways. First it can be seen as a set of Horn clauses specifying facts about data. This is the *declarative* or *model-theoretical reading* of a logic program. Second, considering the particular execution mechanism used in the logic programming system at hand, it can be viewed as a program describing a particular execution. This is the *procedural reading* of a logic program.

In order to control the program flow, certain non-logical (or extra-logical) predicates are used which do not express any facts about the terms we are talking about. (see sections 1.6.9, 4.5 on search).

1.6.8 Model Theory: Least Herbrand Model

When reading a program as a specification we need to determine the meaning of the symbols. A term interpretation, or Herbrand interpretation is an association of a unique function to each functor occurring in the program and an association of sets of tuples of terms to relations. An interpretation is a model for a program if all statements in the interpretation are true.

The least Herbrand model is the least such model. For definite clauses such a model always exists. The model can be inductively built up from the relation symbols and the terms built from constants and terms in the program by constructing a fixpoint using the monotone T_P -operator. (see Nilsson and Małuszyński p. 29 ch 2.4), $ground(P)$ is the set of all ground instances of clauses in a program P (assume always at least one functor or constant and only finite structures).

$$T_P(I) := \{A_0 | A_0 : -A_1, \dots, A_m \in ground(P) \& \{A_1, \dots, A_m\} \subset I\}.$$

Start from the empty theory and determine the least fixpoint for $I = T_P(I) \cup I$.

The least Herbrand Model computed in this way contains only finite structures.

Note that the model does not contain variables.

Infinite Structures

Assuming that the least Herbrand Model defines the meaning of the program one must realise that unification must preserve the property that infinite (cyclic) terms are not allowed. This requires an occurs-check in the unification algorithm prohibiting for example $X=f(X)$ from generating

$$X=f(f(f(f(f(f(f(\dots\dots\dots))$$

This is very inefficient so for practical reasons occurs-check is the responsibility of the programmer. In critical cases a special test must be performed after the unification.

Note that the primitive `=/2` in Prolog allows infinite terms such as `X=f(X)` (rational trees). A theoretically sound unification algorithm is provided as `unify_with_occurs_check/2` but is not the one used for `=/2`. since it is less efficient. Normally this does not cause any problems in real programs, but one should be aware of it.

There are alternative semantic formulations, for instance for handling constraint logic programming [?], that model rational trees.

1.6.9 Proof Theory

Execution is a Search for a Proof or Failure

The execution (computation) of a logic program can be considered as a constructive proof procedure, where the system attempts at proving (or failing to prove) a given goal statement using the definitions as its environment of computation. If a proof succeeds under some restrictions on the variables in the given goal, these restrictions are shown as *bindings* of the variables.

Search Trees and Proof Trees

A logic program execution can be seen as a traversal of a tree, a *search*, which when successful generates a proof tree. A *proof* is one path from the root to a success leaf in a proof tree.

An example of a proof tree:

Consider the variable free program:

```
foo :- bar.
bar :- bar1, baz1.
bar :- bar2, baz2.
bar1. bar2. baz1. baz2.
```

We can draw the two possible proof trees like this (with the root at the bottom):

$$\frac{\frac{\text{bar1} \quad \text{baz1}}{\text{bar}}}{\text{foo}}$$

$$\frac{\frac{\text{bar2} \quad \text{baz2}}{\text{bar}}}{\text{foo}}$$

Often it is considered natural to draw a tree with the root at the top as indicated in figure 1.1. We also want to make a distinction between the search tree that is constructed by the execution of the program and the subset of the search tree that only contain successful branches, the proof tree. The proof tree gives the set of successful proofs, while the search tree contains also branches that are failed proof attempts, and (in principle) infinite branches that represent neither success nor failure.

Proving Goals in Prolog: Matching and Unification

When attempting to prove an atomic goal $p(\text{Args})$ a logic programming system finds the clauses in the database defining the relation $p/1$ and tries to prove one of them. This is accomplished through first matching the arguments, Args in the goal against the terms in the head of the clause. This matching operation tries to check whether the two matched terms can be made to represent the same values. The process is called *unification*. A unification may succeed or fail depending on the data involved. If it succeeds, some bindings of the variables might be produced, representing the minimal restrictions necessary for this branch of the proof to succeed. If the clause has a body, the goals of the body must also be proven. When an assertion, that is a clause for p without a body, is proved, the proof of the current goal is finished and the resulting bindings can be presented to the next higher level, which for the topmost goal is the user.

If a unification operation fails, the alternative clauses for the predicate in which the unification occurs are tried. If no alternatives remain, the proof of the predicate fails, possibly triggering alternatives to be tested to the clause in which the call was made. This mechanism is called *backtracking* and since the proof proceeds to the bottom goal in each branch before trying any alternative

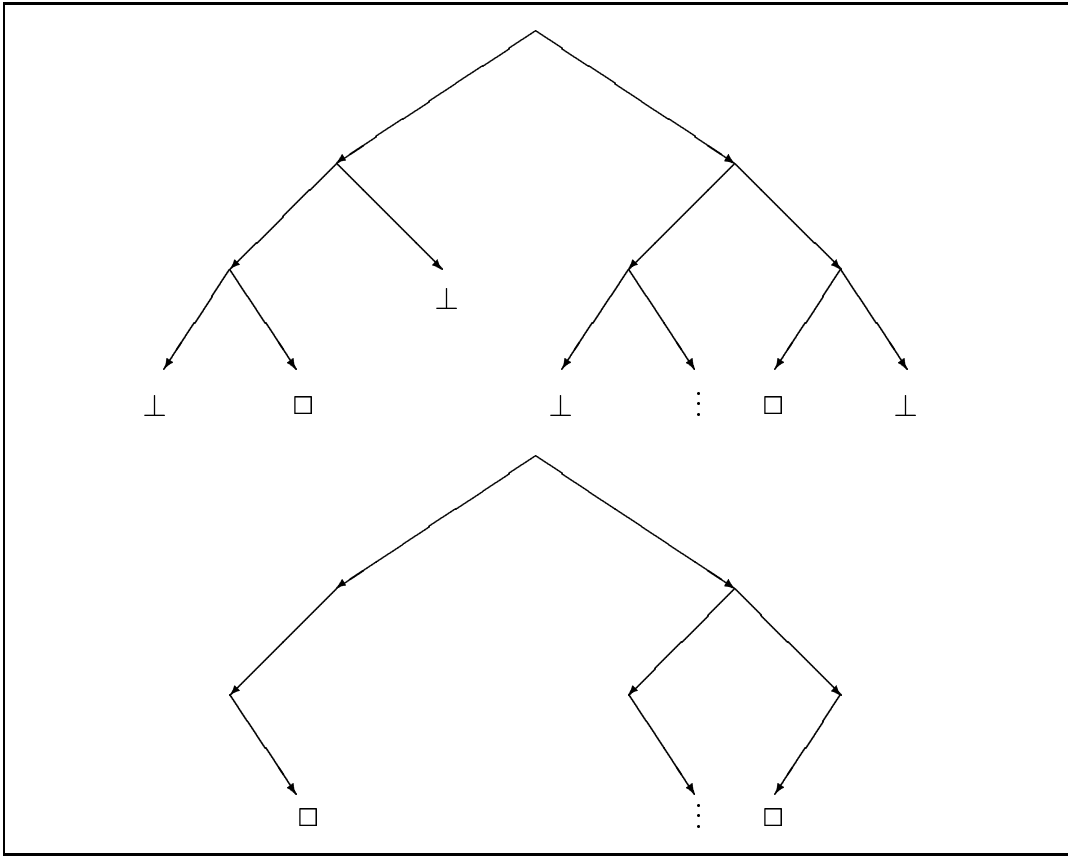


Figure 1.1: Search Tree (top) and Corresponding Proof Tree.

and since the goals of the body of a clause are tried in a left-to-right order, the execution mode of Prolog is called *depth-first, left-to-right with backtracking*. (We will discuss other strategies in section 4.5).

SLD-Resolution

Consider a program P containing clauses of the form

$$B_0 \leftarrow B_1, \dots, B_n.$$

The SLD-resolution rule (p. 43 Nilsson and Małuszyński):

$$\frac{\langle -A_1, \dots, A(i-1), A_i, A(i+1), \dots, A_m \quad B_0 \leftarrow -B_1, \dots, B_n}{\langle -(A_1, \dots, A(i-1), B_1, \dots, B_n, A(i+1), \dots, A_m)\theta} \text{ SLD-resolution}$$

where

1. A_1, \dots, A_m are atomic formulas (goals)
2. $B_0 \leftarrow B_1, \dots, B_n$ is a (renamed) definite clause in P
3. $mgu(A_i, B_0) = \theta$

The rule does not specify which A_i is selected; this is done with a *goal selection function* (in Prolog this is the leftmost one). Also there may be alternatives for the choice of clause $B_0 \leftarrow B_1, \dots, B_n$. The order is determined with a *clause selection rule* (in Prolog the clauses are tried in the order given in the program).

Soundness of SLD-resolution with Respect to the Least Herbrand Model:

Any query (goal) that is provable with SLD-resolution is a logical consequence of the program. (That is, it is included in the Herbrand Model).

Completeness of SLD-Resolution (with Respect to the Least Herbrand Model):

Any query (goal) that is (true) in the least Herbrand model is provable with SLD-resolution.

In the case of an infinite SLD-tree, the selection function has to be fair (as in breadth first search, see section 4.5). For finite SLD-trees left-first-with-backtracking as used in Prolog gives a complete method.

When you study this chapter again, consider especially these topics:

- Operational semantics of (constraint) logic programs
- Model theoretic semantics for (constraint) logic programs
- The concept of *intended model*
- *soundness* and *completeness* of the proof procedures
- how the two (operational and model theoretic) semantics are related

1.6.10 Conclusion

- LP can be used as a uniform language for representing databases, for example data structure and queries can be written in a single language
 - LP extends traditional databases by having recursive rules and structured data facilities.
-

Chapter 2

Good Programming Style

Outline

- Programming techniques
- Arithmetic in Prolog. `is/2`, `:=/2`
- Different primitives for equality, comparing `=/2`
- Recursive definitions.
- Programming with data structures: terms, trees and lists.
- `list/1`. `append/3`. `reverse/2`.
- Dictionaries. `lookup(Key,Value)` defined with lists.

Sterling and Shapiro ch. 2, 3, 6, 7, 8, 13 (not 2.4, 2.5, 3.6, 6.3, 7.6, 13.3, 13.4)

SICStus Prolog Manual

Nilsson and Matuszyński ch. 7 (except 7.3)

2.1 Specifying the Use of a Procedure

For serious projects it is good programming practice to specify the intended use of important procedures, such as predicates in a library.

A comment in prolog is written in one of two ways:

- Using the comment brackets: `/* this is a comment */`
- Inserting the comment character `%` making the rest of the line a comment

2.1.1 Mode Declarations

A common way of commenting code in logic programs is to indicate how the predicate is intended to be used. For instance for a definition:

```
%foo(+,?,?,+,-)
foo(A,B,C,D,E) :-
```

the *mode declaration* `foo(+,?,?,+,-)` indicates that the first and fourth arguments should always be instantiated (used as input), that the last argument should never be instantiated (always used as output) and that the second and third arguments can be either instantiated or uninstantiated. Some Prolog systems are able to utilize this information for generating more efficient code for instance, but in standard Prolog systems such as SICStus Prolog mode declarations are a support to the human eye only.

Sometimes a convention is used where the mode is written as a prefix to a descriptive name for the parameter, for instance;

```
%foo(+Age,?Name,?Address,+Income,-Tax)
```

You might want to have a somewhat more expressive specification of the interface to a predicate that can also describe the expected form of the arguments and give some information about how the predicate behaves. For instance this could be given as a comment of the following form:

```
% procedure foo(T1,T2,...Tn)
```

```

%
% Types:  T1:  type 1
% T2:  type 2
% T3:  type 3
% ...
% Tn:  type n
%Relation scheme:
%Modes:  (input arguments T1,T2) (output arguments T3,...,Tn)
%Multiplicities of solution:  deterministic (one solution only)

```

2.2 Arithmetic Expressions

2.2.1 Expression Evaluation and Unification

`is/2` a built-in predicate for evaluation of arithmetical expressions

`?- Value is Expression`

- `Expression` is evaluated and then unified with `Value`.

`is/2` evaluates expressions to numbers containing:

- `+` `-` `*` `/` `//` `mod` - plus, minus, multiplication, division, integer division, remainder
- `/\` `\/` `#` `\` `<<` `>>` - bitwise conjunction, disjunction, exclusive or, negation, shift
- `abs(X)`, `min(X)`, `max(X)`, `sin(X)`, `cos(X)`, `sqrt(X)`.

(for a complete list, see the SICStus Prolog manual)

For example

`?- X=1+2.` - Yes, since unification `X='+'(1,2)` succeeds

`?- 3 is 1+2.` - Yes, since unification `3=3` succeeds

?- 2 is 1+2. - No.

?- X=2, Y is 1+X*3. -Yes, Y=7

?- X=2, 4 is X*X. - Yes.

?- Z is 1+x. - instantiation error, x is a constant

?- Z is 1+X. - instantiation error, X is not instantiated

?- 2 is 1+X. - instantiation error, X is not instantiated

Automatic type conversion takes place in arithmetic

- Typical error, failing to unify floating point numbers.

?- 3 is (1/2)*6.

No, since unification $3=3.0$ fails!

- Dont use unification with floating point numbers!
- Beware of automatic conversion!

2.2.2 Different Equals

= - unification

:=, (= \=) - arithmetic, boolean equal (not equal)

?- X=2, X=Y. - yes. X=2, Y=2.

?- X:=2, X:=Y. - instantiation error

?- X=2, Y=2, X:=Y. -yes. X=2, Y=2.

?- X is 2+2, Y is X. - yes. X=2+2, Y=4.

2.3 Recursive Programming Techniques

- composing recursive programs
- natural numbers
- dictionaries

2.3.1 Composing Recursive Programs

1. Think about declarative meaning of recursive data type (a definition)
2. Write down recursive clause and base clause
3. Run simple examples - check different goals
4. Check what is happening (do you get a correct result?)

Typical Errors

- Missing (or erroneously failing) base case
- Error in data structure representation
- Wrong arity of structures
- Mixing an element and a list
- Permuted arguments

2.3.2 Reasoning About Natural Numbers

Consider Unary Syntax

0 - denotes zero

s(0) - denotes one

...

s(s(s(...s(0)...)))

Defining the Natural Numbers

```
natural_number(0).
natural_number(s(X)) :- natural_number(X).
```

Plus

```
plus(0, X, X) :- natural_number(X).
plus(s(X), Y, s(Z)) :- plus(X, Y, Z).
?- plus(s(0), 0, s(0)). - checks 1+0=1. Yes.
```

```
?- plus(X, s(0), s(s(0))). - checks X+1=2, (expresses minus) X=s(0).
?- plus(X, Y, s(s(0))). - checks X+Y=2, giving a pair of natural numbers, whose sum equals 2.
```

```
X=0, Y=s(s(0)); X=s(0), Y=s(0); X=s(s(0)), Y=0.
```

Less or Equal

```
le(0, X) :- natural_number(X).
le(s(X), s(Z)) :- le(X, Z).
```

Multiplication

```
times(0, X, 0) :- natural_number(X).
times(s(X), Y, Z) :- plus(Y, Z1, Z), times(X, Y, Z1).
```

2.3.3 Recursive Arithmetic

Consider the Programs:

```
sum([],0).
sum([H|T],S) :- sum(T,V), S is H+V.
```

```
sum0([],0).
sum0([H|T],S) :- sum0(T,V), S=H+V.
```

Elaborate the proof trees for `sum([1,2,3],S)` and `sum0([1,2,3],S)` and compare the values for `S` in the respective cases.

...

...

...

...

...

...

...

2.3.4 Programming with Recursive Structures

Trees

For example :

`void` - denotes empty tree (our choice)

`tree(Element, Left, Right)` - denotes a tree, where `Element` is a root and `Left, Right` are subtrees

```
tree(5, tree(8, void, void), tree(9, void, tree(10, void, void)))
```

Defining a Tree

```
binary_tree(void).
binary_tree(tree(Element, Left, Right)) :-
    binary_tree(Left), binary_tree(Right).
```

Membership

```
tree_member(X, tree(X, _, _)).
tree_member(X, tree(Y, Left, _)) :- tree_member(X, Left).
tree_member(X, tree(Y, _, Right)) :- tree_member(X, Right).
```

2.3.5 More on Syntax

The question of what is the right syntax for logic programming is not settled, and probably never will be. Prolog syntax has been standardised by ISO, but if you consider logic programming in other systems supporting other programming paradigms, such as Mozart/Oz [Mozartsite] or Prolog interpreters written in LISP [CaKa], SCHEME [SCHEME], C or Java other considerations might lead to a different syntax.

An *s-expression* is a general representation form for data used in LISP, the more modern variant SCHEME, and in some Prolog dialects, especially those embedded in a LISP/SCHEME environment. S-expressions are written using extremely simplistic syntactic rules. An s-expression is either a *syntactic atom*, for instance a number or the special atom nil, or a binary tree constructed using the *cons* operator ($A.B$) where A and B are also s-expressions. S-expressions are very convenient to represent data and programs. This is well exploited in the programming languages LISP/SCHEME and occurs in many functional and logical programming languages. A list is a particular form of s-expression, where the s-expression A is seen as an element of the list, and B is the rest of the list (another list) or the atom nil, which stands for the *empty list*. When s-expressions are used for logic programming it is convenient to represent logical variables by annotating the name of the variable, for instance by prefixing with the symbol ?.

All data and programs can be represented using this syntax. This is convenient since all tools are oriented towards representing programs and data in this form, but many find this kind of syntax hard to read for humans. Traditional syntaxes for first order logic that you find in the literature are normally considered easier to read than s-expressions. Special syntax make programs more readable. The lack of non-binary trees in s-expressions requires special care to achieve an efficient system.

Syntax of Lists in Prolog, LISP/SCHEME and Mozart

In Prolog lists are also used, although they are not strictly necessary. The empty list `NIL` of LISP/SCHEME is `[]` in Prolog. and `nil` in Mozart/Oz. The dotted pair (cons operator) of LISP/SCHEME, for example `(A . B)` is in Prolog written `'.'(a,b)` or using the more common syntactic convention `[a|b]`. In Mozart/Oz the syntax is `a|b` for a dotted pair.

To summarize the Prolog list syntax:

`[]` is the empty list

`'.'(H,T)` or `[H|T]` is a list where T is a list
`[|]` represents the constructor `'.'`

Simpler Syntax Through Syntactic Sugar:

Lists can often be written in a shorter form. In Prolog there is a syntactic convention (similar to that in LISP/SCHEME) that allows repression of a tail known to be `[]`.

For example we have these syntactic equivalences in Prolog:

`'.'(H, []) == [H|[]] == [H]`

`[a|[]] == [a]`

`[a|[b|[]]] == [a,b]`

```
'.'(1, '.'(2, '.'(3, []))) == [1|[2|[3|[]]]] ==
== [1,2|[3|[]]] == [1,2,3|[]] == [1,2,3]
```

The notation `(a b c)` in LISP/SCHEME for the list of three elements `a`, `b` and `c` is written `[a,b,c]` in Prolog.

The same list can of course be written as explicit cons cells:

In LISP/SCHEME: `(a . (b . (c . NIL)))`

In Prolog: `[a|[b|[c|[]]]]`

In Mozart/Oz as in many functional languages an infix cons operator is used instead: `a|b|c|nil`. In SICStus Prolog if `'.'` is defined as an infix operator (see syntactic support in 2.3.5 below or in the manual) you can consider `H.T` as equivalent to `'.'(H,T)`. Then you can write this list in Prolog as `a.b.c.[]`.

Here are some examples of lists:

- List with one element: `[A]`
- List with two elements: `[A,B]`
- List with head and tail: `[H|T]`
- List of lists: `[[A,B],[C,d],[e,F,g|T]]`
- `[erik]`
- `[person(erik,-,-),[jonas,lena],eva | Rest]`

Efficient Prolog-interpreters and compilers often make use of structures with higher arity than two to improve efficiency, rather than cons cells used in pure s-expressions.

Syntactic Support

Most current Prolog systems, such as SICStus Prolog, use a flexible method allowing the user to redefine the syntax “on-the-fly” by adding declarations of priority and pre/in/post-fixity of operators. Using `op/3` properties (priority, prefix, infix, postfix, associativity) of operators can be defined and then used (see manual [SICStus] for details). Using this mechanism requires a high level of sophistication on behalf of the user and even though the intention is the opposite, overuse of this feature might make the resulting programs *less* readable.

2.3.6 Programming with Lists

Defining a List:

```
list(L) :- L=[] . - defines the basis
list(L) :- L=[_|T], list(T) . - defines the recursion
```

Alternative Definition with the Parameter Implicit:

```
list([]).
list([_|T]) :- list(T).
```

Some Common Predicates and Their Different Uses

`member/2`, `append/3`, `reverse/2`

checking membership

```
member(X, [X|_]).
member(X, [_|T]) :- member(X,T).
```

This relation expresses facts about membership in a list. The meaning of the first clause is that an element, X , is the first member of a list.

The symbol $_$ stands for the *void* variable which can denote any term, in this case the rest of the list.

The second clause expresses the fact that the element X is a member of the list $[_|T]$ if X is a member of the rest of the list, here denoted by T .

A proof of the relation (predicate) `member/2` will succeed if the query is matched against a goal fulfilling the conditions stated in any of these two clauses. The second clause in the definition of `member` is recursively defined.

Discussion on different uses of `member/2`

?- `member(a, [b,f,a,c])` . - to test for membership

?- `member(X, [b,f,a,c])` . - to bind an element to unbound variable X

?- `member(a,L)` . - to generate a list where a is an element somewhere

Elaborate the search/proof tree with unification and backtracking in these cases

...

...

...

...

...

...

...

Concatenation of lists

`append([],L,L)` .

`append([H|T],L,[H|R]) :- append(T,L,R)` .

Note that the definition above is equivalent to the following where the parameters (the variables in the head, the arguments) and unification involving those are made explicit:

```
append(A,B,C) :- A=[], B=C.
append(A,B,C) :- A=[H|T], C=[H|R], append(T,B,R).
```

This definition states what it means for two lists to be appended. When executed the predicate can append two lists.

That is after attempting to prove the statement `append([a,b],[c,d,e],X)` the variable `X` will be bound to the list `[a,b,c,d,e]`.

Discussion on how `append/3` can be used differently

`append/3` can also be used for decomposing lists!

If one attempts to prove for instance: `append(X,[g,e,f],[a,b,c,g,e,f])` the resulting binding to the variable `X` will be the list `[a,b,c]`.

?- `append([a,b],[c],X)` . - addition of two lists

?- `append(Xs,[a,d],[b,c,a,d])` . - finds the initial differing part

?- `append(Xs,Ys,[a,b,c,d])` . - partitions a list into two lists

Backtracking gives several alternatives:

```
A=[], B=[a,b,c,d]
A=[a], B=[b,c,d]
A=[a,b], B=[c,d]
A=[a,b,c], B=[d]
A=[a,b,c,d], B=[]
```

Note that `append([1,2],foo,R)` succeeds! What is the result?

Why is this safer definition not used?

```
append([],L,L) :- list(L).
append([H|T],L,[H|R]) :- append(T,L,R).
```

Elaborate some more search/proof trees

...

...

...

...

...

...

...

...

...

Reverse

```
reverse([], []).  
reverse([H|T],R) :- reverse(T,W), append(W,[H],R).
```

The last definition is the so called naive reverse, so named since its execution is highly inefficient. It is suitable as a common benchmark program, though.

elaborate some search/proof trees:

...

...

...

...

...

...

...

Typical error: wrong assembly of a resulting list

2.3.7 Dictionaries

Finding and Adding a Value

A dictionary, that is a map of pairs $Key \rightarrow Value$ can be represented as an (ordered) binary tree with nodes such as `tree(Key, Value, Left, Right)`.

```
lookup(Key,tree(Key,Value, Left, Right), Value).
lookup(Key, tree(Key1,Value1, Left, Right), Value) :-
    Key < Key1,lookup(Key, Left, Value).
lookup(Key, tree(Key1, Value1, Left, Right), Value) :-
    Key > Key1, lookup(Key, Right, Value).
```

We can then define a dictionary and ask questions:

```
?- Dict=tree(1, fifi, _C, tree(2, mumu,_B, _A)),
    lookup(1, Dict, V),
    lookup(2, Dict, W).
```

`V=fifi, W=mumu.`

We can even use the definition of `lookup` to construct a dictionary and ask questions:

```
?- lookup(1, D, fifi),
    lookup(2, D, mumu),
    lookup(1,D, X).
```

```
D=tree(1, fifi, _C, tree(2, mumu, _B, _A)), X=fifi.
```

NB: for finding a key of a value, the traversal of the tree should be implemented.

```
?- Dict= tree(1, fifi, _C, tree(2, mumu, _B, _A)),
    lookup(Key1, Dict, fifi),
    lookup(Key2, Dict, mumu).
```

This does not work. Why?

2.3.8 Final Example

Define a predicate

`unsort_deg(Xs, D)` that, given a list of numbers, finds its unsort degree `D`.

The unsort degree of a list is the number of pairs of element positions in the list such that the first position precedes the second in the list, but the number occupying the first position is greater than the number occupying the second position.

Some examples:

the unsort degree of the list `[1,2,3,4]` is 0

the unsort degree of the list `[2,1,4,3]` is 2

the unsort degree of the list `[4,3,2,1]` is 6.

2.4 Efficient Representations

By choosing the appropriate representation we can achieve more efficient programs. Let us consider an example.

Given a representation for a game board as nested lists

```
LL= [[ - , - , - ],[- , - , - ],[- , - , - ]]
```

Let us define the predicate `ij_replace(X,Y,LL,KK,I,J)` for known `I, J, LL` and `Y`. `X` and `KK` are the returned arguments.

The predicate is rewritten to handle the alternative representation which is more efficient, at least for access

```
LL=1(1( - , - , - ),1( - , - , - ),1( - , - , - ))
```

```
ij_replace(X,Y,LL,KK,I,J) :-
    functor(LL,Name,Arity)           % determines Name and Arity
    functor(KK,Name,Arity)           % constructs a new data structure KK
    arg(I,LL,L),                     % finds the Ith row of the array
    arg(I,KK,K),                     % inserts row K with index I into KK
    i_replace(X,Y,L,K,J),            % replace X at column J in L with Y to K
    all_but_i(Arity,I,LL,KK).        % Copy all rows but row I from LL to KK
```

```
i_replace(X,Y,L,K,I) :-
    functor(L,Name,Arity),           % determines Name and Arity
    functor(K,Name,Arity),           % Constructs a new data structure K
    arg(I,L,X),                     % determines X as argument I in L
    arg(I,K,Y),                     % inserts Y as argument I in K
    all_but_i(Arity,I,L,K).          % copies all elems but I from L to K
```

```
all_but_i(0,I,L,K).
all_but_i(N,I,L,K) :-
    N>0, \+ (N=I),                 % finds the Nth elem X of the array L
    arg(N,K,X),                     % inserts elem X with index N into K
    N1 is N-1,
```

```

    all_but_i(N1,I,L,K).           % go ahead
all_but_i(N,I,L,K) :-
    N>0, N=I,                     % do nothing in this case
    N1 is N-1,
    all_but_i(N1,I,L,K).         % go ahead

```

Other data structures to represent arrays are trees of various kinds, such as binary trees, 2-3-trees and assoc structures (AVL-trees). The latter kind are available as a library, `library(assoc)`, in SICStus Prolog.

2.4.1 Example: Sets Represented as Lists

Even finite sets cannot be represented directly as objects in standard logic programming languages. Sets can instead be represented by the existing datatypes in a convenient way by enforcing an order on a structure used to store the set. For instance we can use an ordered list (or tree) where each element has a unique occurrence and where all operations are assumed to take ordered unique lists as input and produce ordered unique lists. If the sets are allowed to contain uninstantiated elements, however, we have problems with enforcing the requirement that the lists are ordered and unique, since the requirement may be violated in a later stage. Consider for instance `[X,Y,Z]` as a representation of a set with three uninstantiated elements. Of course if `X=Y` is executed, the list no longer contains unique elements. Perhaps even more obvious is that we cannot ensure that the order of the elements is the one intended until the elements are at least partially known.

Here we could add and discuss the definitions to achieve set operations. Try to formulate the predicates yourself. Ask if you get stuck!

When using a data structure to encode some richer mathematical structure the guarantee of invariants such as those discussed here is the responsibility of the programmer.

Chapter 3

Advanced Recursive Techniques

Outline

- iterative - recursive definitions
- last call optimizations
- Programming with accumulating parameters
- Programming with differential structures
- Abstract data types, separation of data definitions and the logic

Sterling and Shapiro ch. 7, 8, 13.3, (not 13.4), 15
Nilsson and Matuszyński ch. 7.3

3.1 Iterative and Recursive Definitions

Last-Call Optimization

Last-call optimization, a.k.a. tail recursion, is used by many modern compilers to drastically reduce the amount of needed stack space for the virtual

machine. Without going into technicalities we can just say that frames for variables and control information for the execution may be reused for the last call in a recursive definition. This gives an execution that makes many uses of recursion behave just as loop constructions in other languages. Memory is immediately reused and the system does not use stackspace unnecessarily. In logical languages such as Prolog the compiler can do a much better job than for instance a C-compiler since there are stronger invariants that can guide the compilation, so there is no need for an explicit loop construction. Some languages such as Mozart and some functional languages introduce loop constructions, but then often as syntactic sugar (translated to another internal form before compilation) for essentially recursive constructions.

Definitions that allow the last-call optimisation are called *iterative*.

3.1.1 Accumulating Parameters

Using *accumulators* is useful to achieve iterative rather than recursive definitions. An accumulating parameter is carried along the recursive steps and is transferred to the output variable in the base case.

For an example of this technique consider defining reverse with an accumulating parameter avoiding the call to `append` that occurs in the naive version.

reversing lists

a) naive reverse (*using append in each recursion step*)

```
reverse([], []).
reverse([X|Xs], Ys) :- reverse(Xs, Zs), append(Zs, [X], Ys).
```

b) *efficient version with accumulating parameter*

```
reverse(Xs, Ys) :- reverse(Xs, [], Ys).
reverse([], Acc, Acc).
reverse([X|Xs], Acc, Ys) :- reverse(Xs, [X|Acc], Ys).
```

advice: draw simple SLD-tree and check substitutions carefully!

Note that `append` is not used in the version with the accumulator.

Example t.2.1

Define the predicate for computing the factorial of a given integer.

a) Recursion:

```
factorial(0, 1).
factorial(N, F) :-
    N > 0,
    N1 is N - 1,
    factorial(N1, F1),
    F is N * F1.
```

b) Iteration with an Accumulator:

```
factorial(N, F) :- factorial(N, F, 1).
factorial(0, F, F).
factorial(N, F, F1) :-
    N > 0,
    N1 is N - 1,
    F2 is N * F1,
    factorial(N1, F, F2).
```

Example t.2.2

Define the predicate for computing the sum of members of integer-lists.

a) Recursion:

```
sumlist([], 0).
sumlist([I|Is], Sum) :-
    sumlist(Is, Sum1),
    Sum is Sum1 + I.
```

b) Iteration (with Accumulator):

```
sumlist(List, Sum) :- sumlist(List, Sum, 0).
sumlist([], Sum, Sum).
sumlist([I|Is], Sum, Sum1) :-
    Sum2 is Sum1 + I,
    sumlist(Is, Sum, Sum2).
```

3.1.2 Programming with Abstract Data Types

Programming with *abstract data types* usually means separation of data definitions and the operations over the data (the program's "logic"). This methodology allows changes in the data representation without change in the code using the operations.

1. Specify a set of objects
2. Specify a set of operations (relations, functions) on the objects
3. Allow access to objects only through *access predicates*

For instance you may define an abstract data type for graphs and define operations for insertion and deletion of nodes and links etc. The same idea can be used in logic programming. Here the operations are relations.

You can make a program more efficient by changing only the access predicates, that is the definitions of the encoding of the abstract data.

Assume the data type predicates specifying operations on a given representation of lists: (We use here the default syntax for lists and the built-in unification for defining the equality.)

```
cons(H,T,[H|T]).
```

```
nil([]).
```

```
equal(X,X).
```

Abstract Form of append/3

```
append(A,B,C) :- nil(A), equal(B,C).
```

```
append(A,B,C) :- cons(H,T,A), cons(H,R,C), append(T,B,R).
```

Note that the representation of lists can be changed without changing the algorithmic code defining `append` by replacing these datatype predicates:

```
cons(H,T,foo(T,H)).
```

```
nil(bar).
```

The two representations shown here are equally efficient, but if you for instance use a tree instead of cons cells to represent the (abstract) list, operations of finding and inserting the last element in a list is made substantially more efficient.

Using this method allows change in representation without change in the code of the algorithm.

Consider as an exercise how to define the operations for finding and insertion of elements in a list that is represented as a reasonably balanced tree.

- A good project task is to show how program transformation techniques (see later) can be used to make programs using an abstract data type more efficient.

3.1.3 Difference Lists

In real applications it is often essential to find a more efficient representation than for instance lists. In [Wa2] a compiler written in Prolog for a small PASCAL-like language is presented that shows the usefulness of the techniques with difference structures introduced here and in the course book.

A difference list is a structure $A-B$ such that A is a list and B is a list where B coincides with the ending of the list A . The structure is used to represent the beginning of A up to but not including the common part.

Syntax

The syntax of a difference list is $D1-D2$ where $D1$ is a list, which ends with $D2$

$D-D$ represents the empty list

$[a, b, c] - []$ represents the ordinary list $[a, b, c]$

$[a, b, c|T] - T$ and $[a, b, c|R] - R$ both represent the list $[a, b, c]$

For instance:

$[a, b, c] - [c]$ represents the list $[a, b]$

$[a, b, c, d, e] - [d, e]$ represents the list $[a, b, c]$

Programming with Differential Structures

Appending Two d-lists :

$[a, b, c]$ is represented by $[a, b, c|R1] - R1$

$[d]$ is represented by $[d|R2] - R2$

$[a,b,c,d,e,f]$ is represented by $[a,b,c|R1]-R2$, assuming $R1=[d,e,f|R2]$.

We can define the `append_dl` operation on difference lists in one clause:

```
append_dl(A-B,B-C,A-C).
```

Check how this works with an example :

```
?- append_dl([a,b,c|T]-T,[d,e,f|S]-S,U-V).
T=[d,e,f|S],
U=[a,b,c,d,e,f|S],
V=S
```

Note that the definition does not work backwards to divide lists into components in any other than the obvious way

```
?- append_dl(X,Y,[a,b,c,d,e,f|S]-S).
```

```
X=[a,b,c,d,e,f|S]-_A,
Y=_A-S ? ;
no
?-
```

But you can use it for testing

```
?- append_dl([a,b,c|A]-A,[d,e,f|B]-B,[a,b,c,d,e,f|S]-S).
A=[d,e,f|S] ? ;
no
?-
```

To be certain that the `append_dl` works it is also necessary that the lists appended are of the most general form before the call that is $[a|T]-T$ with T as an unbound variable.

Consider for example what happens if you do

```
?- L=[a|T]-T, dl_append(L,L,M).
```

Given a sound unification (where $X=f(X)$ fails) it should fail, but in Prolog you might get an infinite (cyclic) structure.

The advantage of difference lists can be used for reverse:

```
reverse(X,Y) :- dreverse(X,Y-[]).
dreverse([X|T],C-D) :- dreverse(T,C-[X|D]).
dreverse([],A-A).
```

In the above definition (from Sterling and Shapiro) the first argument is not a d-list, though. The reason for this omission is not discussed.

The next comment will perhaps be more understandable at a later stage, but it fits in naturally here, so bear with me at this point:

Consider an attempt to define `rev_dl/2` that takes d-lists as arguments and is completely isomorphic to the standard naive reverse:

```
rev_dl(A-A,A-A) :- var(A),!.
rev_dl([X|T]-W,C-D) :-
    rev_dl(T-W,U-E),
    append_dl(U-E,[X|Z]-Z,C-D).
```

The call to `var(A)` in the base case is necessary to avoid infinite lists (also called rational lists) in the solutions. We are only interested in finite terms. The reason infinite lists occur is that SICStus like most Prolog systems do not use the expensive occur check in unification, that is it is possible to get solutions to unifications like $X=f(X)$.

Using `var(A)` and `!` in the program will guarantee that it behaves as expected, (deterministic and terminating).

The call to `append_dl` is unnecessary and can be compiled away by a program transformation.

Note that `append_dl(A-B,B-C,A-C)` can be written

```
append_dl(A-X,B-Z,Y-C) :- X=B, Y=A, Z=C.
```

We can therefore replace the call with equalities:

```

rev_dl(A-A,A-A) :- var(A), !.
rev_dl([X|T]-W,C-D) :-
    rev_dl(T-W,U-E),!,
    E=[X|Z], C=U, Z=D.

```

Reduction of the number of equalities gives fewer variables

```

rev_dl(A-A,A-A) :- var(A),!.
rev_dl([X|T]-W,C-D) :-
    rev_dl(T-W,C-[X|D]).

```

Quicksort Defined with d-lists:

```

quicksort(Xs,Ys) :- quicksort_dl(Xs, Ys-[]).
quicksort_dl([X|Xs], Ys-Zs) :-
    partition(X, Xs, Littles, Bigs),
    quicksort_dl(Littles, Ys-[X|Z1]),
    quicksort_dl(Bigs, Z1-Zs).
quicksort_dl([], Xs-Xs).

```

```

partition(X,[Y|Ys],[Y|Ls],Bs) :- X>Y, partition(X,Ys,Ls,Bs).
partition(X,[Y|Ys],Ls,[Y|Bs]) :- X<Y, partition(X,Ys,Ls,Bs).
partition(_,[],[],[]).

```

Queues

A Queue May be Implemented as a Difference List:

Using difference lists to define queue operations:

```
append_dl(A-B,B-C,A-C).
```

```
nil_dl(A-A).
```

```
qempty(Q) :- nil_dl(Q).
```

```
%enqueue(Element, OldQueue, NewQueue)
```

```
enqueue(X,Qin,Qout) :- append_dl(Qin,[X|T]-T,Qout).
```

```
%dequeue(Element, OldQueue, NewQueue)
```

```
dequeue(X,Qin,Qout) :- append_dl([X|T]-T,Qout,Qin).
```

```
%an input list for queueing
```

```
S=[in(5), in(9), in(10), out(X1), out(X2), in(4)]
```

```
queue(S) :- queue(S, Q-Q).
```

```
queue([], Q).
```

```
queue([in(X)|Xs], Q) :- enqueue(X, Q, Q1), queue(Xs, Q1).
```

```
queue([out(X)|Xs],Q) :- dequeue(X, Q, Q1), queue(Xs, Q1).
```

Chapter 4

Search Based Programming

Outline

- Proof trees
- Cut
 - Execution of a program with cuts
 - Insertion of cuts in your own program, green and red cuts
 - Implementation of if-then-else
 - Avoiding useless backtrack points without using cut
- Negation, SLDNF
 - Basic concepts
 - Execution of programs with negation
 - Implementation of negation
- Generate-and-test
- State-space programming
- Puzzle-solving: Missionaries and cannibals
- Graph theoretical examples: Euler paths, Hamilton paths

- Alternative search methods
- Game-playing

Sterling and Shapiro ch. 5.5, 11, 14, 20, 21 (not 11.7, 14.4)
Nilsson and Matuszyński ch. 4, 5, 11, 12, A.3

4.1 Search

Execution with SLD-resolution is considered as a search of a proof in a depth-first left-to-right mode. Examples are elaborated to show in an abstract way how one could think about what a Prolog engine is doing while constructing the proof of a query (or a goal, as it is also called).

4.1.1 Controlling Search

The execution of a Horn Clause program can be viewed as a search tree traversal where the search strategy is depth-first left-to-right with backtracking. Sometimes when a solution to a sub-problem has been found, no other solutions to it or to earlier proved subgoals of the current goal need to be considered.

By using the non-logical primitive predicate `!`, named *cut*, you reach the effect of cutting away alternative branches to subgoals in the clause that is currently being proved. The alternatives on a higher level, that is to the clause which the current goal is a part of, are kept. This can decrease the amount of unnecessary computation.

`!` can be placed in the body of a clause or a goal as one of its atoms to cut branches of an SLD-tree

`p(X) :- q(X), !, r(X).`

Effects

! divides the body into two parts. When ! is reached, all backtracking of the left-side of the body is disallowed. The execution of the right-side of the clause body continues as usual.

New matches of the head of the clause are disallowed, for example backtracking is stopped one level up in the SLD-tree.

Consider:

```
P :- Q, !, R.
```

```
P :- ...
```

```
P :- ...
```

Cut Performs Two Operations:

1. Removes alternatives to Q that have not been tried when passing the cut
2. Removes alternatives to P that have not been tried when passing the cut

! must be used with care since correct solutions may be inhibited.

4.1.2 Example: Execution of a Program with Cuts

Consider the following program:

```
top(X,Y) :- p(X,Y).
```

```
top(X,X) :- s(X).
```

```
p(X,Y) :- true(1), q(X), true(2), r(Y).
```

```
p(X,Y) :- s(X), r(Y).
```

```
q(a).  q(b).  
r(c).  r(d).  s(e).
```

```
true(X).
```

```
?- top(X,Y).
```

- In the given program (seven answers)
- When `true(1)` is replaced by `cut` (five answers)
- When `true(2)` is replaced by `cut` (three answers)

4.1.3 Inserting Cuts in Programs

Green cuts only affect the efficiency of the computation, while *red cuts* remove solutions. This distinction most often assumes knowledge of the mode of the predicate so good programming practice says that you add mode declarations to any definition where `cut` plays a role (even in used definitions).

Green Cut

- Increases efficiency
- does not change the semantics of a program
(cuts away only failing branches in an SLD-tree)

Red Cut

- Changes the semantics of a program
(also cuts away success branches in an SLD-tree)
- in general, red cuts are considered harmful

4.1.4 Green Cut: An Example

The use of ! in `merge/3` to sort two sorted lists together to form a new sorted list. Given two sorted integer-lists `Xs` and `Ys`, construct a sorted integer-list `Zs`, containing elements from `Xs` and `Ys`.

```
%merge(+,+,?)
merge([], Ys, Ys).
merge(Xs, [], Xs).
merge([X|Xs], [Y|Ys], [X|Zs]) :- X < Y, merge(Xs, [Y|Ys], Zs).
merge([X|Xs], [Y|Ys], [X, Y|Zs]) :- X=Y, merge(Xs, Ys, Zs).
merge([X|Xs], [Y|Ys], [Y|Zs]) :- X > Y, merge([X|Xs], Ys, Zs).
```

4.1.5 Green Cut: An Example (cont.)

```
%merge(+,+,-)
merge([], Ys, Ys) :- !.
merge(Xs, [], Xs) :- !.
merge([X|Xs], [Y|Ys], [X|Zs]) :- X < Y, !, merge(Xs, [Y|Ys], Zs).
merge([X|Xs], [Y|Ys], [X, Y|Zs]) :- X=Y, !, merge(Xs, Ys, Zs).
merge([X|Xs], [Y|Ys], [Y|Zs]) :- X > Y, !, merge([X|Xs], Ys, Zs).
```

4.1.6 Red Cut: An Example

Find the minimum of two integers.

It might be tempting to define

```
%minimum(+,+,-)
minimum(X,Y,X) :- X<=Y, !.
minimum(X,Y,Y).
```

But consider the mode violating query `minimum(4,5,5)`, which gives `yes!`

```
?- minimum(4,5,Z) . - Yes, Z=4.
?- minimum(5,4,Z) . - Yes, Z=4.
?- minimum(4,5,5) . - Yes.
```

Correction

The correct form works also for `minimum(4,5,5)`, that is it fails:

```
%minimum(+,+,-)

minimum(X,Y,Z) :- X<Y, !, Z=X.
minimum(X,Y,Y) :- Y<X.
```

An optimized version that still works (note the more liberal mode):

```
%minimum(+,+,?)

minimum(X,Y,Z) :- X<Y, !, Z=X.
minimum(X,Y,Y) .
```

4.1.7 Red Cut: Another Example

Checking membership in a list.

```
% memberchk(+,+)
memberchk(X, [X|Xs]) :- !.
memberchk(X, [_|Ys]) :- memberchk(X, Ys).
```

```
?- memberchk(a, [b,c,a,d]) . - checks membership
```

```
?- memberchk(X, [b,c,a,d]) . - generates elements from a list
```

NB: It generates only the first element

```
?- memberchk(b, Z) . - generates lists
```

NB: It generates only one list (containing b)

Check the example lookup/3 from section 2.3.7!

4.1.8 Implementation of if-then-else

```
P :- (Condition -> TruePart; ElsePart).
```

is implemented as

```
P :- Condition, !, TruePart.
P :- ElsePart.
```

For example we can now define

```
%minimum(+,+,?)
minimum(X,Y,Z) :- (X<Y -> Z=X; Z=Y).
```

4.2 Using and Proving Negative Information?

First order logic allows a negation operation to precede any formula. In logic you may use rules such as this:

$$\frac{(\text{not } A) (A \text{ or } B)}{B}$$

Prolog does not handle negation in this general way. We should consider some (deliberate) limitations:

- The (Herbrand) interpretation of terms as syntactic functions is important for the understanding of negation. For instance the equality $f(\mathbf{X})=g(\mathbf{X})$ is always false in logic programs, while in first order logic it might be true for some models (those where f and g represent the same function).
- In a logic programming language there is no explicit representation of either disjunctive or negated axioms. In fact the only allowed definitions are of positive literals (*atoms*).

The handling of negation in Prolog is based on the *closed world assumption*.

This reads in ordinary English:

That which is not stated explicitly is false.

The clauses of a logic program are considered to be positive statements about the world. A negated query `not_p(X)` should succeed if the proof of the statement `p(X)` fails and it should fail if the proof of `p(X)` succeeds. The way this is implemented is through two clauses:

```
%not_p(++) (assuming that ++ stands for a ground term)
not_p(X) :- p(X), !, false.
not_p(_).
```

The predefined predicate `false` is used in order to prevent a branch from succeeding. Since it is preceded by a `!` the second clause is not applicable and the query fails. This works perfectly as long as the variable `X` is fully instantiated when the call to `not_p(X)` is being performed. But consider the following program:

```
animal(cow).

not_animal(X) :- animal(X), !, false.
not_animal(_).

?- not_animal(X), X=house.
```

The declarative understanding of this program is intuitively clear:

A house is not an animal so the query should succeed.

Unfortunately it fails because the call `not_animal(X)` fails since there is something which can be bound to the as yet unbound variable `X`, namely `cow`, which certainly is an animal. That the clause is really stating something about a `house` is not clear to the interpreter at this stage and the cut takes away the possibility for the program to succeed in this case! By reordering the goals in the query we reach a program that behaves correctly and succeeds:

```
?- X=house,not_animal(X).
```

A slight modification to the original example gives this program:

```

animal(cow).

not_animal(X) :- animal(X),!, false.
not_animal(_).

not_not_animal(X) :- not_animal(X),!, false.
not_not_animal(_).

?- not_not_animal(X), X=house.

```

The way this program is stated it will succeed in proving that a house is *not* a non-animal, therefore (given a classical interpretation of negation) it proves that a house is an animal!

Now this may indeed suggest that something is very wrong with the standard execution mechanism for Prolog with respect to negation. But a closer inspection shows that we were a little too quick in our statement about the declarative meaning of the program above. Declaratively speaking the program states that the property `not_animal` holds for any term and also that the property `not_not_animal` holds for any term. The `!` used in the first of the cases is used in order to rule out the alternatives that were not intended, but this scheme does not work when the terms are not instantiated upon calling the predicate. So this implementation of negation in Prolog does not correctly reflect the meaning of negation in logic unless the terms are fully instantiated (ground). Therefore the suggestive shorthand `not p(X)` is not used in Prolog but instead `\+` is used (for the operator *not provable*).

Unfortunately this type of behaviour of a program can be difficult to spot in a more realistic program and therefore one must be cautious when using negated clauses. Problems of this kind are overcome in Prolog systems where a dynamic execution mechanism waiting for variables to have values is used. For instance the Mozart/Oz system uses the more modern mechanism of *entailment of concurrent constraints* which allows data-dependent execution strategies to be implemented.

The Closed World Assumption and the Negation-as-failure Principle

The Closed World Assumption (cwa):

If P is not derivable (with SLD-resolution), the negation of P is considered true.

Implementation of Negation as Failure (naf):

The statement $\neg A$ is derivable if A is a formula which cannot be derived by SLD-resolution.

Consider the problem with infinite SLD-trees.

More precisely: The statement $\neg A$ is derivable if the goal A has a finitely failed SLD-tree.

We still have the problem, when A has variables:

```
unmarried_student(X) :- \+ married(X), student(X).
student(erik).
married(jonas).
```

Implementation of \neg :

\neg can be defined by using the built-in `call(P)` that tries to prove the relation with name and arity corresponding to the term P . (see section 7.4 on meta-programming).

```
\+ Goal :- call(Goal), !, fail.
\+ Goal.
```

Example, a Program with a Single Clause:

```
p(a).
?- p(X). Yes, X=a.
?- \+ p(b). Yes, (since p(b) fails)
```

?- \+ \+ p(a). Yes, (since \+ p(a) fails)

Handling of negation with the negation-as-failure principle works well for ground goals. Sometimes it works fine also for non-instantiated variables:

?- \+ \+ p(X). Yes, (since X is not instantiated X is not bound).

If you however try the query \+ \+ p(X), X=b. you will see that this also succeeds. This seems unnatural and is understood by considering how the proof for \+ \+ P works. Variable bindings made in the proof for P are not kept. This shows the typical case of a program that uses an unsound negation.

To summarize: *it is the responsibility of the programmer that uses of + can be understood as negation.*

4.2.1 Negation: SLDNF-Resolution

The combination of SLD-resolution to resolve positive literals and negation as failure (NF) to resolve negative literals is shown by this example:

```
foundation(X) :- on(Y, X), on_ground(X).
on_ground(X) :- \+ off_ground(X).
off_ground(X) :- on(X, Y).
above(X, Y) :- on(X, Y).
above(X, Y) :- on(X, Z), above(Z, Y).
on(c, b).
on(b, a).
```

Four Kinds of SLDNF-Derivation

- Refutations (that end with success branches)
- Infinite derivations
- (finitely) failed derivations
- Stuck derivations (if none of the previous apply)

NB: Check Examples in the Nilsson and Małuszyński, pp. 71-73

4.3 Generate-and-Test Algorithms

Generate-and-test, particularly suited to search problems, is a technique in algorithm design, which defines two parts of an algorithm:

- the first *generates* the set of candidate solutions one by one
- the second *tests* the candidates

4.3.1 An Example in Prolog

```
find(X) :- generate(X), test(X).
```

Important optimisation: to push the tester inside the generator as deep as possible.

```
find(X) :-  
    generate1(X), test1(X),  
    generate2(X), test2(X),  
    generate3(X), test3(X),  
    generate4(X), test4(X).
```

Example 1: Finding Parts of Speech in a Sentence

```
verb(Sentence,Word) :- member(Word,Sentence), verb(Word).
```

```
noun(Sentence,Word) :- member(Word,Sentence), noun(Word).
```

```

article(Sentence,Word) :- member(Word,Sentence), article(Word).

noun(man).

noun(woman).

article(a).

verb(likes).

?- noun([a, man, likes, a woman], N).

N=man;
N=woman

```

NB. member/2 should not contain cuts. Why?

Example 2

N-queens: Place N queens on a NxN chess-board in such a way that any two queens are not attacking each other.

The answer is a list of positions for queens, for example [3,1,4,2].

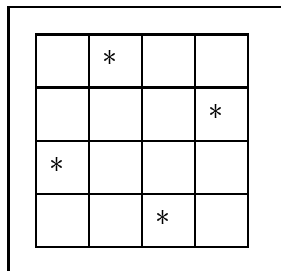


Figure 4.1: N-queens, a solution for N=4.

Optimizations are achieved by pushing the test for validity forward, for instance testing each element immediately rather than generating the whole solution and testing afterwards.

a) Naive Generate and Test:

```

queens(N,Qs) :-
    range(1,N,Ns),      % Ns is the list of integers [1..N]
    permutation(Ns,Qs), % Qs is a permutation of Ns
    safe(Qs).           % true, if the placement Qs is safe

range(M,N,[M|Ns]) :- M<N, M1 is M+1, range(M1,N,Ns).
range(N,N,[N]).

```

Example 2 (cont.)

```

permutation(Xs,[Z|Zs]) :-
    select(Z,Xs,Ys),
    permutation(Ys,Zs).
permutation([],[]).

select(X,[X|Xs],Xs).
select(X,[Y|Ys],[Y|Zs]) :- select(X,Ys,Zs).

safe([Q|Qs]) :- safe(Qs), \+ attack(Q,Qs).
safe([]).

attack(X,Xs) :- attack(X,1,Xs).
attack(X,N,[Y|Ys]) :- X is Y+N; X is Y-N.
attack(X,N,[Y|Ys]) :- N1 is N+1, attack(X,N1,Ys).

```

Example 2 (cont.)**b) When Generating a Position of a Queen - Test Whether it is Permitted:**

```

queens(N, Qs) :- range(1, N, Ns), queens(Ns, [], Qs).
queens(UnplacedQs, SafeQs, Qs) :-
    select(Q, UnplacedQs, UnplacedQs1),
    \+ attack(Q, SafeQs),
    queens(UnplacedQs1, [Q|SafeQs], Qs).
queens([], Qs, Qs).

```

select/3, attack/2 are the same as in a).

4.4 Searching in a State-Space

- Loop-avoidance in searching for a path
- Efficiency issues
- Different search strategies

Many problems are formulated as finding a path in a graph containing nodes (states) and edges (a transition relation):

given some start-state s_0 and a set of goal-states

determine whether there exists a sequence:

$$s_0 \longrightarrow s_1, s_1 \longrightarrow s_2, \dots, s_{(n-1)} \longrightarrow s_n,$$

such that s_n belongs to a set of goals.

States can be seen as nodes in a graph whose edges represent the pairs in the transition-relation, then the problem reduces to that of finding a path from the start-state to one of the goal-states.

Finding a Path

```

path(X,X).
path(X,Z) :- edge(X,Y), path(Y,Z).

edge(X,Y) :- ... % define construction/finding of the next node

```

For example,

```

edge(a,b). edge(b, c). edge(c,d).

?- path(a,d). - Yes.
?- path(a,X). - Yes. X=a; X=b; X=c; X=d.
?- path(X,d). - Yes. X=d; X=a; X=b; X=c.

```

Loop Detection

```

path(X,Y) :- path(X,Y,[X]).
path(X,X,Visited).
path(X,Z,Visited) :-
    edge(X,Y),
    \+ member(Y,Visited),
    path(Y,Z,[Y|Visited]).

member(X,[X|_]) :- !.
member(X,[_|_]) :- member(X,_).

```

Returning the Path as an Answer to the Goal

```

path(X,Y,Path) :- path(X,Y,[X],Path).
path(X,X,Visited,Visited).
path(X,Z,Visited,Path) :-
    edge(X,Y),
    \+ member(Y,Visited),

```

```

    path(Y,Z,[Y|Visited],Path).

member(X,[X|Y]) :- !.
member(X,[Y|Z]) :- member(X,Z).

```

4.4.1 Puzzle: Missionaries and Cannibals

- Three missionaries and three cannibals must cross a river.
- The only available boat will hold only two people at a time.
- There is no bridge, the river cannot be swum, and the boat cannot cross the river without someone in it.
- The cannibals will eat any missionaries they outnumber on either bank of the river.

The problem is to get everyone across the river with all the missionaries uneaten.

Example 3

```

puzzle(Moves) :- path(state(3,3,left),state(3,3,right),Moves).
path(InitNode,FinalNode,Path) :-
    path(InitNode,FinalNode,[InitNode],Path).
path(InitNode,FinalNode,-,[]) :- InitNode=FinalNode, !.
path(Node0,FinalNode,VisitedNodes,[Arc|Path]) :-
    edge(Node0,Arc,Node1),
    \+ member(Node1,VisitedNodes),
    path(Node1,FinalNode,[Node1|VisitedNodes],Path).

```

Example 3 (cont.)

```

edge(state(M0,C0,L0),move(M,C,D),state(M1,C1,L1)) :-
    member(M,[0,1,2]),
    member(C,[0,1,2]),
    M+C >=1,
    M+C <2,
    M0 >= M,
    C0 >= C,
    M1 is 3-(M0-M),
    C1 is 3-(C0-C),
    (M1:=0 ; M1:=3; M1:= C1),
    (L0=left ->
                                     (D=leftRight, L1=right)
                                     ;
                                     (D=rightLeft, L1=left)).

```

4.5 Different Search Methods

There are different search methods possible to find a proof.

- Depth-first - interprets current set of nodes as a stack
- Breadth-first - interprets current set of nodes as a queue
- Bounded-depth - controls the depth of the search
- Iterative-deepening - controls the depth of the search
- Heuristic search methods - use domain specific knowledge

The depth-first strategy described above (see section 1.6.9) is used in Prolog and it is normally the most efficient proof mechanism regarding space consumption since most of the memory allocated for the proof of a branch can be reclaimed after the proof attempt has finished. It is also simple to manage for the programmer.

Other techniques with nice theoretical properties, such as *breadth first* interpreters, that is proof procedures that try all alternatives concurrently (or variants thereof) are not so common since their memory requirements often develop exponentially during a proof. A breadth first interpreter has, though, the advantage of providing the proof theoretical property of *fairness*, that is all true statements which follow from the clauses of a program are provable (as long as the amount of available memory is sufficient). A looping clause which might prevent the standard interpreter from finding a solution simply by not terminating causes no problem (except from using resources) in a fairness-preserving interpreter. Since such a clause may inhibit the computation completely if depth-first semantics is used, it must be considered as a programming error in Prolog, even though the program may have a reasonable declarative meaning.

Iterative deepening is a technique that maximizes the allowed depth in a proof attempt that otherwise executes depth-first, as SLD-resolution. If a solution cannot be found with a given limit for the maximum depth in the search, the proof attempt is restarted with an increased maximum depth. This technique elegantly combines the efficiency of depth-first search with the fairness of breadth-first. It might seem silly at first glance to throw away all that has been done, since that search has to be reiterated in the next attempt, but it turns out that it is often more costly to try to keep intermediary information than to reiterate the search. Consider the number of nodes in a tree with branch factor m . At depth n the number of nodes is $m^n - 1$, while down to depth $n + 1$ the number of nodes is $m^{(n+1)} - 1$ which means that the number of new nodes to explore is as many as has been explored already, thus leading to $(m^n - 1)/(m^{(n+1)} - 1) < 50\%$ redundant computations when iterating to the next level at any level. The cost of keeping information about the nodes that have been passed makes a breadth-first search impractical.

Parallelism and Concurrency

Concurrent Logic Programming:

Flexible execution of a program using concurrency is described for instance in [Sha] or in Nilsson and Małuszyński.

essense

Goals in the body of a clause can execute as interleaving processes

$p(X) :- q(X), r(X).$

Synchronisation on variable unification

$X=Y$ means wait until X is bound before unification

effects

Divides the body into two parts and switches execution between the goals.

Deadlock is possible.

Parallel Logic Programming:

Computation can be distributed over several computers possibly sharing memory.

OR-Parallelism

Copying of backtrack stack (KABUWAKE, SICS: Ali / R. Karlsson)
 sharing of binding environment (Aurora Prolog, Gigalips: Carlsson, Warren, Overbeek)

AND-Parallelism

independent AND (Hermenegildo et.al., CIAO-Prolog)

stream-and (Montelius,Penny, parallel AKL)

To summarize:

- **Top-down** - recursively find supporting rules for query
- **Bottom-up** - inductively generate facts from known facts
- **Serial** - alternatives one at a time, backtrack for more

- **Or-parallel** - to prove “A or B” try proving either “A” or “B” in parallel, collect all solutions or choose one (first found, best etc.)
- **Concurrent** - to prove “A & B” try proving “A” and “B” concurrently sharing binding environments
- **AND-parallel** - to prove “A & B” where “A” and “B” have nothing in common, do “A” and “B” in parallel in separate binding environments, then combine results and go on
- **Iterative deepening** - search all solutions down to a maximum depth, then increase max
- **Tabled execution** - keep ongoing calls in a table to avoid redundant work

Heuristic Search

To solve larger problem, some domain-specific knowledge must be added to improve search efficiency. The term heuristic is used for any advice that is often effective, but is not guaranteed to work in every case.

4.5.1 Game Playing

Some words on game programming with the **minimax** method.

Game Trees

A game tree is an explicit representation of all possible plays of the game. The root node is the initial position of the game, its successors are the positions which the first layer can reach in one move, their successors are the positions resulting from the second players replies and so on.

The trees representing the games contain two types of node: *max* at even levels from the root, and *min* nodes at odd levels of the root.

A search procedure combines an evaluation function, a depth-first search and the minimax backing-up procedure.

Chapter 5

Logic and Grammars

Outline

- Grammars
 - Context free grammars
 - Context dependent grammars
 - DCG - translating grammars into logic programs
 - How to define a lexer and a parser for a language

Sterling and Shapiro ch. 19, 24

Nilsson and Matuszyński ch. 10

SICStus Prolog Manual

5.1 Grammars

A *grammar* describes the valid strings in a language.

5.1.1 Context Free Grammars

A context free grammar is a 4-tuple $\langle N, T, P, S \rangle$

where N and T are finite, disjoint sets of non-terminal and terminal symbols respectively.

$N \times (N \cup T)^*$ denotes the set of all strings (sequences) of terminals and non-terminals.

P is a finite subset of $N \times (N \cup T)^*$.

S is a non-terminal symbol called the start symbol.

The empty string is denoted by ϵ

The elements of P are usually written in the form of production rules:

$A ::= B_1, \dots, B_n$ where $(n > 0)$

$A ::= \epsilon$ ($n = 0$)

Example 1

<code><sentence></code>	<code>::=</code>	<code><noun-phrase><verb-phrase></code>
<code><noun-phrase></code>	<code>::=</code>	<code>the <noun></code>
<code><verb-phrase></code>	<code>::=</code>	<code>runs</code>
<code><noun></code>	<code>::=</code>	<code>engine</code>
<code><noun></code>	<code>::=</code>	<code>rabbit</code>

`<sentence>` derives the strings `the rabbit runs` and `the engine runs`.

How?

Example 2

```

prod_rule(sentence, [noun_phrase, verb_phrase]).
prod_rule(noun_phrase, [the, noun]).
prod_rule(verb_phrase, [runs]).
prod_rule(noun, [rabbit]).
prod_rule(noun, [engine]).

```

An Interpreter for Production Rules:

```

derives_directly(X, Y) :-
    append(Left, [Lhs|Right], X),
    prod_rule(Lhs, Rhs),
    append(Left, Rhs, Temp),
    append(Temp, Right, Y).

```

```

derives(X, X).
derives(X, Z) :-
    derives_directly(X,Y),
    derives(Y,Z).

```

```
?- derives([sentence], X).
```

Yes.

What is X unified to?

Example 3

```

sentence(Z) :- append(X, Y, Z), noun_phrase(X), verb_phrase(Y).
noun_phrase([the|X]) :- noun(X).

```

```

verb_phrase([runs]).

noun([rabbit]).
noun([engine]).

append([],X, X).
append([X|Xs], Y, [X|Zs]) :- append(Xs, Y, Zs).

?- sentence([the, rabbit, runs]).
    Yes.

?- sentence([the, X, runs]).
    X=rabbit;
    X=engine

?- sentence(X).

```

Example 4

Usage of Difference-Lists:

```

sentence(X0-X2) :-
    noun_phrase(X0-X1), verb_phrase(X1-X2).

noun_phrase([the|X]-X2) :- noun(X-X2).

verb_phrase([runs|X]-X).

noun([rabbit|X]-X).
noun([engine|X]-X).

?- sentence([the, rabbit|X] - X).
?- sentence([the, rabbit, runs] - []).
?- sentence([the, rabbit, runs, quickly] - [quickly]).
?- sentence(X).

```

Example 5**Collecting the Result:**

```

sentence(s(N, V), X0-X2) :-
    noun_phrase(N, X0-X1), verb_phrase(V, X1-X2).

noun_phrase(np(the, N), [the|X]-X2) :- noun(N, X-X2).
verb_phrase(verb(runs), [runs|X]-X).

noun(noun(rabbit), [rabbit|X]-X).
noun(noun(engine), [engine|X]-X).

?- sentence(S, [the, rabbit, runs] - []).
   Yes.
   S=s(np(the, noun(rabbit)), verb(runs)).

```

Context Dependent Grammars

```

<sentence>           ::= <noun-phrase>(X) <verb>(X)
<noun-phrase>(X)    ::= <pronoun>(X)
<noun-phrase>(X)    ::= the <noun>(X)
<verb>(singular)   ::= runs
<verb>(plural)     ::= run
<noun>(singular)    ::= rabbit
<noun>(plural)      ::= rabbits
<pronoun>(singular) ::= it
<pronoun>(plural)   ::= they

```

Example 6

```

sentence(X0-X2) :- noun_phrase(X, X0-X1), verb(X, X1-X2).

```

```
noun_phrase(X, X0 - X2) :- pronoun(X, X0-X2).
noun_phrase(X, [the|X0]-X2) :- noun(X, X0-X2).
```

```
verb(singular, [runs|X]-X).
verb(plural, [run|X]-X).
```

```
noun(singular, [rabbit|X]-X).
noun(plural, [rabbits|X]-X).
```

```
pronoun(singular, [it|X]-X).
pronoun(plural, [they|X]-X).
```

5.1.2 Recognizers for Languages: Lexers and Parsers

A program that recognizes a string in a formal language is often divided into two distinct parts:

Lexer: translation from lists of character codes to lists of tokens

Parser: the translation from lists of tokens to parse trees

A Recognizer for S-Expressions

Earlier we discussed the s-expression syntax. We will show how a program identifying s-expressions looks in Prolog.

A Backus-Naur Grammar Describing what a Legal S-Expression:

```
<s-expr>      ::= <s-atom> | ( <s-exprs> [ . <s-expr> ] )
<s-exprs>     ::= <s-expr> [<s-exprs>]
<s-atom>      ::= [<blanks>] <non_blanks> [<blanks>]
<non_blanks> ::= <non_blank> [<non_blanks>]
<blanks>      ::= <blank> [<blanks>]
```

A <non-blank> is a character with a character code greater than 32 (decimal).
 A <blank> is a character with a character code less than or equal to 32.

The input to the parser is represented as a list of characters (or character codes).

Coding a Recognizer for the S-Expression Grammar:

The predicate `sExpr(L1,L2,S)` succeeds if the beginning of the list `L1` is a list of characters representing an s-expression. As a result of the computation the structure `S` contains a structure (a list) representing the s-expression. The rest of the input text is stored in the list `L2`.

```
p :- sExpr((A.B),Out,S), write(Out), write(S).
```

```
sExpr(In,Out,S) :-
    (blanks(In,I0); In=I0),
    (sAtom(I0,Out,S)
     ;
    lpar(I0,I1),
    sExprs(I1,I2,S,Last),
    (dot(I2,I3),
     sExpr(I3,I4,Last)
     ;
    I4=I2,
    Last=[]),
    rpar(I4,Out)).
```

```
sExprs(In,Out,[H|T],Last) :-
    sExpr(In,I1,H),
    (sExprs(I1,Out,T,Last)
     ;
    Out=I1, T=Last).
```

```
sAtom(In,Out,A) :- nonBlanks(In,Out,A).
```

```

nonBlanks(In,Out,[H|T]) :-
    nonBlank(In,I1,H),
    (nonBlanks(I1,Out,T)
    ;
    I1=Out,
    T=[]).

blanks(In,Out) :- blank(In,I1), (blanks(I1,Out) ; I1=Out).

blank(In,Out) :- In=[C|Out], C=<32.

nonBlank(In,Out,C) :- In=[C|Out], C>=48.

lpar(In,Out) :- In=[#(|Out].

rpar(In,Out) :- (blanks(In,I1); In=I1), I1=[#|Out].

dot(In,Out) :- (blanks(In,I1); In=I1), I1=[#.|Out].

```

Some comments could be made about this program. The first thing one can notice is the close similarity of the grammar representation of the syntax to be parsed and the logic program that actually expresses these facts and which is able to perform the task. The usage of the logical connectives ‘,’ and ‘;’ for conjunction and disjunction respectively has not been explained but these connectives have their expected meaning. It should be noted that in the general case the connective ‘;’ represents a division of the proof into two independent parts whereas the connective ‘,’ connects two subproofs which should both hold for the statement to be true. The two connectives (‘,’ and ‘;’) are the only ones that can occur in the body of a clause.

5.1.3 A Grammar for Horn Clause Statements

```

<clause>      ::= <head> [ :- <body> ]
<head>        ::= <goal>
<body>        ::= <goal> [ , <body> ]
<goal>        ::= <name>( <termlist> ) | <cut> | <>false>
<termlist>    ::= <term> [ , <termlist> ]
<term>        ::= <variable>
               | <dataconstructor> [ ( termlist ) ]
               | <constant>
               | <list>
               | <integer>
               | <expression>
<list>        ::= [ <term> [ <moreterms> ] [ | <term> ] ]
<moreterms>   ::= , <term> [ <moreterms> ]
<expression> ::= <term> <operator> <term>
               | <operator> <term>
<operator>    ::= < | =< | >= | > | + | - | *
<cut>         ::= !
<>false>       ::= false

```

Try to write a lexer/parser for the language defined by this grammar! You may of course vary the grammar to define some other language of your choice.

5.2 DCGs, Lexers and Parsers

5.2.1 Definite Clause Grammars

When dealing with parsing problems many variables have to be used. In order to make such programs more readable a special syntax has been introduced in Prolog, *definite clause grammars*. When such a description is encountered, the system automatically compiles it into a Prolog program.

Definite Clause Grammars, DCGs, are a generalisation of Context Free Grammars, CFGs, allowing variables to express dependencies.

Formalism

$\langle N, T, P \rangle$

N - a possibly infinite set of atoms (non-terminals)

T - a possibly infinite set of terms (terminals)

P is in $N \times \langle N, U, T \rangle^*$ - a finite set of production rules

N and T are disjoint

Syntax

- Terminals are enclosed by list-brackets
- Non-terminals are written as ordinary compound terms or constants
- ', ' separates terminals and non-terminals in the right-hand side
- --> separates head and body
- Extra conditions, in the form of Prolog procedure calls, may be included in the right-hand side of a grammar rule. Such procedure calls are written enclosed in {} brackets
- the empty string is denoted by the empty list []

Example 1

```
sentence --> noun_phrase, verb_phrase.
noun_phrase --> [the], noun.
verb_phrase --> [runs].
noun --> [rabbit].
noun --> [engine].
```

```
?- sentence(X,_).
Yes.
X=[the, rabbit, runs|A]-A ;
X=[the, engine, runs|A]-A.
```

5.2.2 Compilation of DCGs Into Prolog

$p(t_1, \dots, t_n) \text{ --> } T_1, \dots, T_m$

is translated into the clause:

$p(t_1, \dots, t_n, X_0, X_m) \text{ :- } T_1', \dots, T_m'$

where X_1, X_m are distinct variables and each T_i' is of the form:

- $q(S_1, \dots, S_n, X_{i-1}, X_i)$ if T_i is of the form $q(S_1, \dots, S_n)$
- $C(X_{i-1}, X, X_i)$ if T_i is of the form $[X]$
- $T, X_{i-1}=X_i$ if T_i is of the form $\{T\}$
- $X_{i-1}=X_i$ if T_i is of the form $[\]$

The formalism allows Prolog code to occur inside $\{ \}$ brackets in the grammar rules.

$C(L_1, X, L_2)$ is a predefined predicate for parsing X in the list L_1 giving L_2 .

Example 2

`sentence --> noun, verb.`

`expr(X) --> term(Y), [+], expr(Z), {X is Y + Z}.`

translated to:

```
sentence(A, C) :- noun(A, B), verb(B, C).
```

```
expr(X, X0, X4) :-
    term(Y, X0, X1),
    C(X1, +, X2),
    expr(Z, X2, X3),
    X is Y + Z,
    X3=X4.
```

Example 3

```
<expr> ::= <term> + <expr>
<expr> ::= <term> - <expr>
<expr> ::= <term>
<term> ::= <factor> * <term>
<term> ::= <factor> / <term>
<term> ::= <factor>
<factor> ::= 0|1|2|...
```

Example 3 (cont.)

```
expr(X) --> term(Y), [+], expr(Z), {X is Y + Z}.
expr(X) --> term(Y), [-], expr(Z), {X is Y-Z}.
expr(X) --> term(X).
```

```
term(X) --> factor(Y), [*], term(Z), {X is Y*Z}.
term(X) --> factor(Y), [/], term(Z), {X is Y/Z}.
term(X) --> factor(X).
```

```
factor(X) --> [X], {integer(X)}.
```

```
?- expr(X, [2, *, 2, +, 4, *, 4], []).
```

Gives the answer $X=20$.

NB: avoid rules, which lead to left hand recursion, like:

```
expr --> expr, [+], expr.
```

Example 4

Write a DCG which accepts strings in the language $a^n b^m c^n$, ($n, m \geq 0$).

a) if n and m are fixed: $n=1, m=2$

```
abc --> a, b, c.
a --> [].
a --> [a].
b --> [].
b --> [b].
b --> [bb].
c --> [].
c --> [c].
test(String) :- abc(String, []).

?- test(X). Yes. 12 answers.
?- test([a, bb]). Yes.
```

Example 4 (cont.)

Write a DCG which accepts strings in the language $a^n b^m c^n d^m$, ($n, m \geq 0$).

b) general case

```
abcd(N,M) --> lit(a,N), lit(b,M), lit(c,N), lit(d,M).
lit(L, 0) --> [].
lit(L, I) --> [L], lit(L, I1), {I is I1+1}.
test(String) :- abcd(N, M, String, []).

?- test([a, b, b, c, d, d]). Yes.
```

5.3 Compiling

In logic programming *compiling* has the same purpose as in other programming systems:

To make programs run faster by transforming the program to an operationally equivalent representation using the static interdependencies known at compile time to produce a more or less specialized representation.

This representation is then executed by some *virtual machine*. The machine can have very high level instructions which are much more semantically powerful than the instructions of a traditional computer. Properly designed this can simplify the compiling possibly with the penalty of slower execution speed.

The classical phases of compiling are here recognized in the different transformations described below:

1. Lexical Analyser, *lexer*
2. Grammatical Analyser, *parser*
3. Optimizer
4. Code Generator

These phases take the form of functions that each one from a specified input produces one particular output. The structure of the data that may occur on the interfaces between these modules is specified in terms of *abstract syntax trees*. An abstract syntax is an unambiguous description of the structures used to represent a program. It can be viewed as a declaration of the type of the input and output of various optimizing transformations applied to the program to reach executable and/or more efficient forms with an equivalent meaning. Representing the set of legal programs as an abstract tree is handy in a logic programming environment.

The input form of the program is a list of characters. The code for reading a file into a list of character codes is the predicate `char_infile/2` found in section 9.1.3.

5.3.1 Phase 1: The Lexical Analyser

The lexical analyser transforms the program text which is represented as a list of characters to a list of tokens. This saves space and enables faster handling of the structure in the parsing phase. In particular the annoying effect of backtracking within syntactically atomic units, which has to be explicitly avoided if the representation in the form of a flat character list is kept does not occur if a lexical analysing phase is included. Redundant syntactic features are represented in a unique form in the output from this phase. To make the lexical analyser a deterministic predicate, that is to ensure that a failure in the later phase cannot be avoided by backtracking in the lexical analyser, (this is important for performance reasons, especially in the handling of syntactical errors), certain restrictions must be made on the allowed syntactic entities. The limitations are similar to those used in a recursive descent analyser. More specifically the decision about what is a lexically significant symbol must not be dependent on the meaning of the context in which it occurs.

The lexical analyser produces a simplified and more memory efficient representation of the program wherein reserved tokens have been identified. Integers, constant names and variable names are recognized and tagged by the lexical analyser so that no ambiguities are introduced.

The purpose of the lexical phase is to simplify the work in the actual parsing of the statements.

5.3.2 Phase 2: The Grammatical Analyser, Parsing

A grammatical analyser, a *parser*, accepts programs and transforms them into some suitable representation, a syntax-tree. This is also called *translation from concrete to abstract syntax*. The parser takes the list of tokens produced by the lexical analyser as input and produces an abstract tree, describing the program, if it conforms to the concrete syntax, otherwise it fails. The allowed statements are represented as parts of the token list according to the grammar stated below. Priorities and associativities of logic connectives and arithmetic operators are resolved. The arithmetic expressions occurring as terms in the source code are transformed to the same representation as that for nested function invocations and the comparison expressions are transformed into atomic operations.

5.3.3 Phase 3: The Transformer

Syntax trees are used for different *static checks and optimisations*. The transformation phase traverses the abstract tree generated by the parser and creates a different one. For example function calls can be unfolded and transformed, redundant equalities removed, unnecessary variables extracted and new necessary variables introduced. The transformation could also allow various transformations, such as *common subexpression elimination* and reordering of operations.

The abstract tree output from a transformer conforms to the abstract syntax. The use of the same abstract tree for input and output in the optimising predicates enables short-circuiting around any one of the optimising functions. This is an important characteristic of a transformer when it comes to questions of modularity and experimentation. Different optimising functions can be easily inserted without the need for a redesign of internal interfaces.

5.3.4 Phase 4: Code Generation

Code is finally produced from the syntax tree. The (virtual) machine instructions and the state-space of the machine can be described using abstract trees and semantic actions associated with them. This phase transforms the output from the third phase into a machine program in a fairly straightforward manner.

5.3.5 BASIC Programs- Concrete Syntax

```
DIM A(15)
FOR X=1 TO 15 STEP 1
A(X)=X-1
PRINT X,A(X)
GOTO 10
10 REM 10
NEXT X
```

Lexer for the BASIC Language

```

lex([]) --> [].
lex(T) --> space(sp), {!}, lex(T).
lex([H|T]) --> item(H), {!}, lex(T).

space(sp) --> [C], {spacechar(C)}, spaces.

item(nl) --> [10].
item(':') --> ":".
item('.') --> ".".
item(',') --> ",".
item('(') --> "(".
item(')') --> ")".
item(Op) --> oper(Op).
item(Aop) --> aop(Aop).
item(V) --> ident(Vi), {name(V,Vi)}.
item(int(I)) --> integer(I).
item(string(S)) --> [34], {!}, string(St), {!,name(S,St)}.
item(I) --> [C], {\+ spacechar(C)}, notspaces(R), {!,name(I,[C|R])}.

string([H|T]) --> [H], {\+ H=34, !}, string(T).
string([]) --> [34].

spaces --> [C], {spacechar(C),!}, spaces.
spaces --> [].

spacechar(32).
spacechar(9).

integer(I) --> [C], {C>=48, C<58, A is C-48}, integer0(A,I).

integer0(A,I) --> [C], {C>=48, C<58, A0 is C-48+A*10, !}, integer0(A0,I).
integer0(I,I) --> [].

ident([C|R]) --> [C], {C>=65, C<91}, ident0(R).

ident0([C|R]) --> [C], {(C>=65, C<91), !}, ident0(R).

```

```
ident0(I) --> ident1(I).
```

```
ident1([C|R]) --> [C], {(C>=48, C<58), !}, ident1(R).
```

```
ident1([36]) --> [36], {!}.
```

```
ident1([]) --> [].
```

```
notspaces([C|T]) --> [C], {\+ C<33, !}, notspaces(T).
```

```
notspaces([]) --> [].
```

```
oper(oper(<>)) --> <>.
```

```
oper(oper(=<)) --> =<.
```

```
oper(oper(>=)) --> >=.
```

```
oper(oper(<)) --> <.
```

```
oper(oper(>)) --> >.
```

```
oper(oper(=)) --> =.
```

```
aop(aop(+)) --> +.
```

```
aop(aop(-)) --> -.
```

```
aop(aop(*)) --> *.
```

```
aop(aop(/)) --> /.
```

```
readbasic(File,B) :-
    char_infile(File,L),
    !,
    lex(Tokens,L,[]),
    !,
    basic(B,Tokens,[]).
```

Parser for the BASIC Language

The lexer has produced a list of tokens.

The parser now has to identify the valid language constructs.

```
basic([H|T]) --> block(H), {!}, basic(T).
```

```
basic([]) --> [].
```

```

block(L) --> [nl], {!}, block(L).
block([label(Lb1)|S]) --> [int(Lb1)], stmts(S).

stmts(S) --> [nl], {!}, stmts(S).
stmts([H|T]) --> stmt0(H), {!}, [nl], stmts(T).
stmts([]) --> [].

stmt0([H|T]) --> stmt(H), {!}, reststmt(T).

reststmt(R) --> [:], {!}, stmt0(R).
reststmt([]) --> [].

items([H|T]) --> [H], {\+ H=nl,!}, items(T).
items([]) --> [].

stmt(nl) --> [NL].
stmt(rem(L)) --> [REM], items(L).
stmt(onerrorgoto(S)) --> [ONERRORGOTO], label(S).
stmt(goto(S)) --> [GOTO], label(S).
stmt(gosub(S)) --> [GOSUB], label(S).
stmt(on_goto(E,S)) --> [ON], expr(E), [GOTO], labels(S).
stmt(on_gosub(E,S)) --> [ON], expr(E), [GOSUB], labels(S).
stmt(return) --> [RETURN].
stmt(stop) --> [STOP].
stmt(dim(L)) --> [DIM], expr0s(L).
stmt(data(L)) --> [DATA], expr0s(L).
stmt(read(L)) --> [READ], exprs(L).
stmt(print(L)) --> [PRINT], exprs(L), {!}.
stmt(print([])) --> [PRINT].
stmt(for(I,From,To,Step)) --> [FOR,I,oper(=),From,TO,To,STEP,Step].
stmt(for(I,From,To,int(1))) --> [FOR,I,oper(=),From,TO,To].
stmt(next(I)) --> [NEXT,I], {!}.
stmt(if(Cond,S,Next)) --> [IF], condition(Cond),
    {!}, ifbody(S), elsebody(Next).
stmt(assign(Var,Expr)) --> expr(Var), [oper(=)], expr(Expr).
stmt(error([H|T])) --> [H], {\+ H=nl, \+ H=int(_)}, items(T).

ifbody(B) --> [THEN], ifbody0(B).
ifbody(B) --> ifbody0(B).

```

ifbody0([goto(S)]) --> label(S).
 ifbody0(S) --> stmt0(S).

elsebody(Next) --> [ELSE], {!}, ifbody(Next).
 elsebody([none]) --> [].

label(S) --> [int(S)].

labels([H|T]) --> label(H), {!}, labels(T).
 labels([]) --> [].

expr0s([H|T]) --> expr0(H), {!}, expr0s(T).
 expr0s([]) --> [].

exprs([H|T]) --> expr(H), restexprs(T).

restexprs([H|T]) --> [,], {!}, expr(H), restexprs(T).
 restexprs([]) --> [].

expr(E) --> expr0(E1), restexpr(E1,E).

restexpr(E1,expr(O,E1,E2)) --> [aop(O)], {!}, expr(E2).
 restexpr(E,E) --> [].

expr0(func(E,I)) --> [E,[], {\+ E=()}, exprs(I), []], {!}.
 expr0(E) --> [(), expr(E), []], {!}.
 expr0(real(R)) --> [int(E1),.,int(E2)],
 {!, name(E1,X1), name(E2,Y1), append(X1,[46|Y1],T), name(R,T)}.
 expr0(int(E)) --> [int(E)], {!}.
 expr0(string(E)) --> [string(E)], {!}.
 expr0(E) --> [E], {\+ E=n1, \+ E=(,\+ E=), !}.

condition(C) --> cond0(C1), restcond(C1,C).

restcond(C1,cond(O,C1,C2)) --> logop(O), {!}, condition(C2).
 restcond(C,C) --> [].

logop(AND) --> [AND].

logop(OR) --> [OR].

```

cond0(true) --> [TRUE].
cond0(false) --> [FALSE].
cond0(not(C)) --> [NOT], cond0(C).
cond0(cond(C,E1,E2)) --> expr(E1), [oper(C)], expr(E2).
cond0(C) --> [()], condition(C), [()].

pre([], []).
pre([for(I,From,To,Step)|T], [for(I,From,To,Step,Body)|S]) :-
    !, append(Body0, [next(I)|Rest], T),
    pre(Body0, Body),
    pre(Rest, S).
pre([H|T], [H|R]) :- pre(T, R).

expand_for([for(I,From,To,Step,Body)|S],
    [assgn(I,From), label(Forlabel),
    if(cond(>, I, To), [goto(Exitlabel)], [none]),
    Ebody,
    assgn(I, expr(+, I, Step)),
    goto(Forlabel),
    label(Exitlabel)|R]) :- !,
    newlabel(Forlabel),
    newlabel(Exitlabel),
    expand_for(Body, Ebody),
    expand_for(S, R).
expand_for([H|T], [H|R]) :- expand_for(T, R).
expand_for([], []).

:- dynamic mylabel/1.

mylabel(mlbl(0)).

newlabel(mlbl(L)) :-
    retract(mylabel(mlbl(L))),
    L0 is L+1,
    assert(mylabel(mlbl(L0))).

```

Chapter 6

BASIC Interpreter

Outline

- Example: an interpreter for a programming language
-

An Interpreter for an Imperative Language, (BASIC)

Consider the operational semantics of an imperative language. Given a program and a state, the memory if you will, containing a mapping from a set of variables to their current values, one of which being a current program counter and a stack of return addresses.

An interpreter can be formulated as a transition relation $p/2$ from states to new states

$$S_0 \text{ ---> } S_1 \text{ ---> } S_2 \text{ ---> } \dots \text{---> } S_n \text{ --->}$$
$$p(S_0, S_1), p(S_1, S_2), p(S_2, S_3), \dots, p(S_{n-1}, S_n)$$

For instance $x:=E$ for an expression E would be modeled as a predicate

```
assign(x,E,StateIn,StateOut) :- ...
```

```
prog(In,Out) :-
    assign(z,x,In,T1),
    assign(x,y,T1,T2),
    assign(y,z,T2,Out).
```

The state transformation could be expressed using the grammar notation

```
prog --> assign(z,x), assign(x,y), assign(y,z).
```

loosely corresponding to a procedure in an imperative language:

```
proc {Prog Z := X X := Y Y := Z }
```

- State transition predicates are expressed in a perfectly declarative way
- They still look very similar to the usage of assignment in an imperative language
- The predicates express the change of a thread of state objects logically

Compare this with how assignment is modeled in operational semantics of imperative languages.

This is elaborated below for some statements in the language BASIC using the accumulating parameter technique.

Try to implement such an interpreter yourself for some other language containing different language constructs (good project task if it is properly packaged and your code is documented).

Here is the Interpreter for the BASIC Language:

```

:- use_module(library(random)).
?- init_random(0).

basic(File) :-
    abolish(program/2),
    write(## compiling ##(File)), nl,
    readbasic(File,B), flatten(B,Bf),
    pre(Bf,P0), expand_for(P0,P1), flatten(P1,P),
    assert(program(File,P)),
    !,
    write(### run ###(File)), nl,
    basinterp(P,0).

:- dynamic program/2.
:- dynamic showlabel/0.
:- dynamic showrandom/0.

basinterp(Prog,Rseed) :-
    basinterp(Prog,Prog,[],DataUt,[stop]), nl,
    write(##### Dataout from basic program: ##), nl,
    write(DataUt),nl.

basinterp([label(Lbl)|Rest],Prog,DataIn,DataUt,Cont) :-
    !, (showlabel -> write(label(Lbl)), nl ; true),
    insert(var(pc,Lbl),DataIn,Data0),
    !, basinterp(Rest,Prog,Data0,DataUt,Cont).
basinterp([rem(_)|R],Prog,Di,Do,Cont) :-
    !, basinterp(R,Prog,Di,Do,Cont).
basinterp([onerrorgoto(_)|R],Prog,Di,Do,Cont) :-
    !, basinterp(R,Prog,Di,Do,Cont).
basinterp([goto(Lbl)|_],Prog,Di,Do,Cont) :-
    !, append(_, [label(Lbl)|Stmts],Prog),
    !, basinterp([label(Lbl)|Stmts],Prog,Di,Do,Cont).
basinterp([gosub(Lbl)|R],Prog,Di,Do,Cont) :-
    !, append(_, [label(Lbl)|Stmts],Prog),
    !, basinterp([label(Lbl)|Stmts],Prog,Di,Do,[R|Cont]).

```

```

basinterp([on_goto(E,S)|R],Prog,Di,Do,Cont) :-
    !, eval(E,V,Di), index(V,S,I),
    !, basinterp([goto(I)|R],Prog,Di,Do,Cont).
basinterp([on_gosub(E,S)|R],Prog,Di,Do,Cont) :-
    !, eval(E,V,Di), index(V,S,I),
    !, basinterp([gosub(I)|R],Prog,Di,Do,Cont).
basinterp([if(Cond,True,False)|R],Prog,Di,Do,Cont) :-
    !, eval(Cond,Cval,Di),
    (Cval=true, append(True,R,Body),
    !, basinterp(Body,Prog,Di,Do,Cont)
    ;
    Cval=false, append(False,R,Body),
    !, basinterp(Body,Prog,Di,Do,Cont)).
basinterp([for(I,From,To,Step,Body)|R],Prog,Di,Do,Cont) :-
    !, append(Body,[for(I,expr(+,I,Step),To,Step,Body)],Dobody),
    !, basinterp([assgn(I,From),
    if(cond(<=,I,To),Dobody,[none])|R],Prog,Di,Do,Cont).
basinterp([none|R],Prog,Di,Do,Cont) :-
    !, basinterp(R,Prog,Di,Do,Cont).
basinterp(none,Prog,Di,Do,[H|T]) :-
    !, basinterp(H,Prog,Di,Do,T).
basinterp([return|_],Prog,Di,Do,[H|R]) :-
    !, basinterp(H,Prog,Di,Do,R).
basinterp([],Prog,Di,Do,[H|R]) :-
    !, basinterp(H,Prog,Di,Do,R).
basinterp([stop|_],_,D,D,-) :- !.
basinterp([dim([])|R],Prog,Di,Do,Cont) :-
    !, basinterp(R,Prog,Di,Do,Cont).
basinterp([dim([func(Nm,[Len])|T])|R],Prog,Di,Do,Cont) :-
    !, ilength(Len,Array), init(nil,Array),
    !, basinterp([dim(T)|R],Prog,[array(Nm,Array)|Di],Do,Cont).
basinterp([data(L)|R],Prog,Di,Do,Cont) :-
    !, get_element(data(D),Di,Dt), append(D,L,DL),
    !, basinterp(R,Prog,[data(DL)|Dt],Do,Cont).
basinterp([read([])|R],Prog,Di,Do,Cont) :-
    !, basinterp(R,Prog,Di,Do,Cont).
basinterp([read([H|T])|R],Prog,Di,Do,Cont) :- atom(H),
    !, get_element(data([Element|Data]),Di,Dt),
    insert(var(H,Element),Dt,Ds),
    !, basinterp([read(T)|R],Prog,Ds,Do,Cont).

```

```

basinterp([read([func(X,[I])|T])|R],Prog,Di,Do,Cont) :-
    !, atom(X), eval(I,Index,Di),
    delete(data([Yval|Data]),Di,Ds),
    delete(array(X,Array),Ds,Dt),
    ainsert(Index,Array,Aout,Yval),
    !, basinterp([read(T)|R],Prog,[data(Data),array(X,Aout)|Dt],Do,Cont).
basinterp([assgn(X,Y)|R],Prog,Di,Do,Cont) :- atom(X),
    !, eval(Y,Yval,Di), insert(var(X,Yval),Di,Dt),
    !, basinterp(R,Prog,Dt,Do,Cont).
basinterp([assgn(func(X,[I]),Y)|R],Prog,Di,Do,Cont) :-
    !, atom(X), eval(Y,Yval,Di), eval(I,Index,Di),
    delete(array(X,Array),Di,Dt),
    ainsert(Index,Array,Aout,Yval),
    !, basinterp(R,Prog,[array(X,Aout)|Dt],Do,Cont).
basinterp([print([])|R],Prog,Di,Do,Cont) :-
    !, basinterp(R,Prog,Di,Do,Cont).
basinterp([print([H|T])|R],Prog,Di,Do,Cont) :-
    !, eval(H,V,Di), writeus(V),
    !, basinterp([print(T)|R],Prog,Di,Do,Cont).
basinterp(X,Prog,Di,Do,Cont) :-
    write(### Not implemented. ###(X)),nl,
    Di=Do.

```

/* Evaluate Arg1 to Arg2 in Environment Arg3 */

```

eval(cond(AND,C1,C2),C,D) :-
    !, eval(C1,V1,D), eval(C2,V2,D), and(V1,V2,C).
eval(cond(OR,C1,C2),C,D) :-
    !, eval(C1,V1,D), eval(C2,V2,D), or(V1,V2,C).
eval(cond(=,C1,C2),C,D) :-
    !, eval(C1,V1,D), eval(C2,V2,D), eq(V1,V2,C).
eval(cond('<',C1,C2),C,D) :-
    !, eval(C1,V1,D), eval(C2,V2,D), lt(V1,V2,C).
eval(cond(>,C1,C2),C,D) :-
    !, eval(C1,V1,D), eval(C2,V2,D), gt(V1,V2,C).
eval(cond('<=',C1,C2),C,D) :-
    !, eval(C1,V1,D), eval(C2,V2,D), le(V1,V2,C).

```

```

eval(cond(>=,C1,C2),C,D) :-
    !, eval(C1,V1,D), eval(C2,V2,D), ge(V1,V2,C).
eval(cond(<>,C1,C2),C,D) :-
    !, eval(C1,V1,D), eval(C2,V2,D), ne(V1,V2,C).
eval(true,V,-) :- !, V=true.
eval(false,V,-) :- !, V=false.
eval(int(I),V,-) :- !, V=int(I).
eval(real(I),V,-) :- !, V=real(I).
eval(func(RND,[int(From),int(To)]),V,-) :-
    !, random(From,To,R), V=int(R).
eval(func(LEN,[String]),V,D) :-
    !, eval(String,S,D), ilength(V,S).
eval(func(NUM$,[I]),V,D) :-
    !, eval(I,int(S),D), name(S,V).
eval(func(INT,[I]),V,D) :-
    !, eval(I,J,D), realorint(J,S),
    R is integer(S), V= int(R).
eval(func(N,[I]),V,D) :-
    !, atom(N), eval(I,J,D),
    member(array(N,A),D),
    !, index(J,A,V).
eval(string(S),V,-) :- !, name(S,V).
eval(expr(Op,E1,E2),V,D) :-
    !, eval(E1,V1,D), eval(E2,V2,D), operate(Op,V1,V2,V).
eval(X,V,D) :- atom(X), get_element(var(X,V),D,-).

and(true,true,V) :- !, V=true.
and(X,Y,false).

or(false,false,V) :- !, V=false.
or(_,-,true).

eq(X,X,V) :- !, V=true.
eq(X,Y,false).

ne(X,X,V) :- !, V=false.
ne(X,Y,true).

```

```
lt(int(I),int(J), V) :- I<J, !, V=true.
lt(int(I),int(J), V) :- V=false.
```

```
gt(int(I),int(J), V) :- J<I, !, V=true.
gt(int(I),int(J), V) :- V= false.
```

```
ge(int(I),int(J), V) :- I<J, !, V= false.
ge(int(I),int(J), V) :- V=true.
```

```
le(int(I),int(J), V) :- J<I, !, V= false.
le(int(I),int(J), V) :- V=true.
```

/* Evaluate Arg1 to Arg2 in Environment Arg3 */

```
realorint(N,I) :- (N=real(I) ; N=int(I)).
```

```
operate(+,int(I1),int(I2),R) :- !, E is I1 + I2, R=int(E).
operate(-,int(I1),int(I2),R) :- !, E is I1 - I2, R=int(E).
operate(*,int(I1),int(I2),R) :- !, E is I1 * I2, R=int(E).
operate(/,int(I1),int(I2),R) :- !, E is I1 / I2, R=real(E).
operate(+,N1, N2, R) :-
    realorint(N1,I1), realorint(N2,I2),
    !, E is I1 + I2, R=real(E).
operate(-,N1,N2,R) :- realorint(N1,I1), realorint(N2,I2),
    !, E is I1 - I2, R=real(E).
operate(*,N1,N2,R) :-
    realorint(N1,I1), realorint(N2,I2),
    !, E is I1 * I2, R=real(E).
operate(/,N1,N2,R) :-
    realorint(N1,I1), realorint(N2,I2),
    !, E is I1 / I2, R=real(E).
operate(+,string(S),X,R) :- name(S,L), operate(+,L,X,R).
operate(+,X,string(S),R) :- name(S,L), operate(+,X,L,R).
operate(+,L1,L2,L) :- append(L1,L2,L).
```

/* Change value of Arg1 in List Arg2 producing Arg3 with Arg4 */

```
ainsert(int(1),[_|T],[Val|T],Val).
ainsert(int(I),[X|T],[X|Rest],Val) :-
    I>1, I1 is I-1, ainsert(int(I1),T,Rest,Val).
ainsert(int(I),_,_,_) :- I<1, abort.
```

```
/* Change value of Arg1 in environment Arg2 giving env Arg3 */
```

```
insert(Var,EnvI,[Var|EnvO]) :-
    Var=var(Nm,_), delete(var(Nm,_),EnvI,EnvO), !.
insert(Var,EnvI,[Var|EnvO]) :-
    Var=array(Nm,_), delete(array(Nm,_),EnvI,EnvO), !.
insert(Var,EnvI,[Var|EnvI]).
```

Chapter 7

Program Transformation, Meta-programming

Outline

- Partial evaluation
- Higher order programming, defining and using `map`, `foldr` `foldl`

Sterling and Shapiro ch. 13, 16, 18

7.1 Transforming Programs

This section describes transformations of pure logic programs.

In *program transformation* both the input and the output is a program. The applied techniques range between a more or less direct mapping between program representations in different formalisms to advanced techniques like partial evaluation [Ka, Sah] based on knowledge of the expected use of the program stated in the form of annotations or otherwise deduced from the computational context. From the introduction of the idea of logic programming

there has been a realisation among researchers of the significance of program derivations where proof procedures from symbolic logic are used in order to deduce specialised instances of general programs [HanTa, Han]. One attractive aspect of such techniques is that proofs of correctness can be established in a fairly straight-forward manner.

7.1.1 Program Transformation Rules

Since logic programs are defined as axioms, it is quite natural to define logically based transformation rules, usable to improve programs. Here are some examples of such rules:

7.1.2 Equality Reordering

$$\frac{P :- e1, q, e2}{P :- e1, e2, q} \text{ Equality reordering}$$

Motivation: in a (pure) logic program the search will be made more efficient if information about equalities is known as soon as possible to the search procedure.

7.1.3 Equality Removal

$$\frac{P :- \dots e1 \dots e1 \dots}{P :- \dots e1 \dots} \text{ Equality removal}$$

Motivation: in a (pure) logic program statements about equalities need not be repeated. Unification is idempotent, that is the bindings of variables achieved by applying a unification $\text{Term1}=\text{Term2}$ will not be extended by a second application of the same unification.

7.1.4 Clause Level Transformation

A clause (j) is a specialised version of a clause (i)

if $(P_i :- Q_1, \dots, Q_m)\theta = P_j :- Q_1, \dots, Q_m$ in a program:

.
 .
 (i) $P_i :- Q_1, \dots, Q_m$
 .
 .
 (j) $P_j :- Q_1, \dots, Q_m$
 .
 .

When clause (j) is a specialised version of clause (i) it is redundant (in a pure logic program).

But: specialisation can increase efficiency and also make an otherwise looping program terminate. That is it might be advantageous to generate different versions for different forms of the arguments and then apply other transformations to the resulting clauses.

7.1.5 Removal of Failing Clauses

Suppose that $(P_i :- Q_1, \dots, Q_m)\theta = P_j :- \text{false}$

This specialised clause may be removed, since it can never contribute to a solution. Moreover, clauses $C_k :- \dots$ containing a goal that only matches the head P_j can be converted into $C_k :- \text{false}$, and the process can be repeated.

7.1.6 Removal of Repeated Goals

$$\frac{P :- \dots, Q_n, \dots, Q_n, \dots}{P :- \dots, Q_n, \dots} \text{ Removal of repeated goals}$$

The reasoning concerning equalities above holds for goals in general in a pure logic program.

7.1.7 Reordering of Goals

$$\frac{P :- \dots, Q_i, \dots, Q_k, \dots}{P :- \dots, Q_k, \dots, Q_i, \dots} \text{ Reordering of goals}$$

This rule can for instance be used to push recursive calls towards the end of a clause.

7.1.8 Partial Evaluation

$$\frac{P :- Q. \quad Q :- (B1 ; B2).}{P :- B1. \quad P :- B2.} \text{ fold/unfold}$$

Applying the rules above in combination with rules like those given earlier can create more efficient programs (note that variables must be renamed and the introduction of explicit equalities might be necessary to keep the semantics of P).

For pure logic programs techniques of this kind are named partial deduction. It is much harder to devise partial evaluation transformation rules which guarantee that the operational semantics of Prolog is kept even in the presence of non-logical goals such as cut (!), I/O etc. See for instance the thesis by Sahlin [Sah] or the corresponding web page <http://www.sics.se/isl/mixtus.html>

7.2 Stepwise Enhancement

Reuse of software components is desired in general. One way is to define well-documented, general and usable libraries. Another is to identify program *skeletons* that indicate the control flow. The skeleton is then enhanced by adding operations. This methodology is discussed in Sterling and Shapiro.

```
list([X|Xs]) :- list(Xs).
list([]).
```

```
sumlist([X|Xs],S) :- sumlist(Xs,S0), S is S0+X.
sumlist([],0).
```

```
length([_|Xs],L) :- length(Xs,L0), L is L0+1.
length([],0).
```

```
sum_length_list([X|Xs],S,L) :-
```

```
sum_length_list(Xs,S0,L0), S is S0+X, L is L0+1.  
sum_length_list([],0,0).
```

7.3 Functions are Deterministic Relations

A function can be defined as a relation where there is one unique value in the output domain for each input tuple. This means, that once a function has succeeded in supplying a value, no alternative branches need to be tried. A first order function $f : Term \rightarrow Term$ can for instance be encoded as a definition in a logic program as $f(X,Y)$ with a unique Y for each X .

The relation f/n corresponding to a function with $(n-1)$ arguments should be deterministic considering well defined uses, that is, when all arguments (except possibly the one corresponding to the output value of the function) are fully instantiated.

7.3.1 Higher Order

With *higher-order programming* is meant programs where the data items are programs. In functional programming this comes naturally. In logic programming that is defined for first-order theories, higher-order programming goes outside the theory, and needs to be encoded. Higher-order functions are not directly expressible since functions are not logical objects in the first-order logical model of a program.

7.3.2 Functional techniques, apply, map

Many techniques from the functional programming paradigm using higher order programming can be used.

A technique which is very powerful and popular in functional programming involves the use of functions that take functions as arguments. Consider for example `apply`, `map`, `foldr`, `foldl` etc. Such techniques can be adapted to logic programming.

For a Predicate `foo/n` We Can Use This General Form (Schema)

```
apply(foo,X1...Xn) :- foo(X1...Xn).
```

There is also a built-in predicate `call/1` that executes a query corresponding to the term `foo(X1...Xn)`

We can for instance write

```
?- call(foo(X,Y,Z)).
```

Which succeeds exactly when `foo/3` is defined and `?- foo(X,Y,Z).` succeeds.

Mapping a Predicate `Predname(In,Out)` to Each Element of a List

We define `map_list(L1,Predname,L2)` that applies a binary predicate to a list `L1` generating another list `L2`.

```
map_list([X|Xs],Predname,[Y|Ys]) :-
    apply(Predname,X,Y), map_list(Xs,Predname,Ys).
map_list([],_,[]).
```

7.3.3 All-Solutions Predicates

It might be useful to collect several solutions in a list. Prolog gives support for this through some higher-order predicates.

```
father(sven,olle).
father(sven,lisa).
father(bengt,lisa).
father(bengt,sven).
```

```
children(X,Kids) :- findall(Kid, father(X,Kid),Kids).
```

The query

```
?- children(bengt,Kids).
```

```
gives
Kids=[lisa,sven]
```

```
The query
?- findall(F, father(F,Kid),Fathers).
gives
Fathers=[sven,sven,bengt,bengt]
```

Instead of a single solution collecting all fathers to some child we might want a separate solution for each child. There is another set predicate for this.

```
?- bagof(F, father(F,Kid),Fathers).
Kid=lisa, Fathers=[sven,bengt]
Kid=sven, Fathers=[bengt]
Kid=olle, Fathers=[sven]
```

It is often sensible to present sorted lists of unique solutions. This is achieved by `setof`.

```
?- setof(F, father(F,Kid),Fathers).
Kid=lisa Fathers=[bengt,sven]
Kid=sven Fathers=[bengt]
Kid=olle Fathers=[sven]
```

7.4 Meta-Programming

- What is a metastatement?
- Meta-logic predicates
- `solve`, augmenting `solve`
- Mixing object and metalevel programming
- Support for dynamically changing knowledge bases
- Knowledge management using `assert/retract`

Sterling and Shapiro ch. 10, 17, 22

Nilsson and Matuszyński ch. 8, 9

7.4.1 What is a Meta-Statement?

A metastatement is a statement about statements.

- Stockholm is a nine-letter word.
- 'X+1-Y' has the size five.
- 'X+1-Y' contains two variables.
- This statement is true.
- This statement is false.
- 'P :- Q1...Qn.' is a clause.
- This is a metastatement.

7.4.2 Meta-Logic

Meta-logic refers to reasoning about a formalisation of some (other) logical system. If the meta-logic deals with itself it is called *circular* or *meta-circular*. In logic programming meta-logic has two meanings.

1. Using logical inference rules expressed as axioms with a meta-interpreter
2. Expressing properties of the proof procedure

For the latter case the term meta-logical predicate is used, for instance in the SICStus manual.

7.4.3 Ground Representation of Facts and Rules

The logical approach to metaprogramming requires a clear division between object level and meta-level.

- Formulas are represented as ground facts, where in particular variables on the object level are represented as constants on the meta-level.
- Each constant of the object language is represented by a unique constant of the meta-language
- Each variable of the object language is represented by a unique constant of the meta-language
- Each n-ary functor of the object language is represented by a unique n-ary functor of the meta-language
- Each n-ary predicate symbol of the object language is represented by a unique n-ary functor of the meta-language
- Each connective of the object language is represented by a unique functor of the meta-language (with corresponding arity)

For instance a clause

$$p(X) : \neg q(X, a), p(b).$$

could be represented by the ground clause.

$$\text{clause}(\text{if}(p(\text{var}(x)), \text{and}(q(\text{var}(x), a), p(b)))).$$

The SLD-resolution rule (p. 43 Nilsson and Małuszyński):

$$\frac{\langle -A_1, \dots, A(i-1), A_i, A(i+1), \dots, A_m \quad B_0 \langle -B_1, \dots, B_n \rangle}{\langle -(A_1, \dots, A(i-1), B_1, \dots, B_n, A(i+1), \dots, A_m)\theta}$$

The SLD-rule is encoded as a predicate:

```

step(Goal,NewGoal) :-
    select(Goal,Left,Selected,Right),
    clause(C),
    rename(C,Goal,Head,Body),
    unify(Head,Selected,Mgu),
    combine(Left,Body,Right,TmpGoal),
    apply(Mgu,TmpGoal,NewGoal).

```

- `select/4` describes the relation between a goal and the selected subgoals
- `clause/1` describes the property of being a clause in the object language
- `rename/4` describes the relation between four formulas such that two are uniquely renamed variants of the other two
- `unify/3` describes the relation between two atoms and their mgu
- `combine/4` describes the relation between a goal and three conjunctions
- `apply/3` describes the relation between a substitution and two goals

The generation of a sequence of goals using this rule is:

```

derivation(G,G).
derivation(G0,G2) :- step(G0,G1), derivation(G1,G2).

```

7.4.4 Why Self-Interpreters?

Flexibility

- Alternative search strategies
- Debugging
- Programs that change during their execution
- Collecting the actual proof of a satisfied goal (explanation)
- Non-standard logics: fuzzy logic, nonmonotonic logic, modal logic, cfg, DCG (see grammars)
- Program transformation, program verification, program synthesis

7.4.5 Non-Ground Representation of Facts and Rules

Efficiency of the previous ground representation is low. So you might want to use object language variables for the metalevel also. This seems straightforward, but mixing object level and metalevel has important semantic consequences.

Consider this representation (with `if` and `and` as infix operators):

```
for facts: ax(Fact if true).
```

```
for rules: ax(Head if Body).
```

A Meta-Interpreter for Pure Prolog Using the Above Representation

```
solve(true).
solve(P) :- ax(P if Q), solve(Q).
solve(P and Q) :- solve(P), solve(Q).
```

In order to extend the meta-interpreter to handle also non-logical features of Prolog a different interpreter must be written. For instance `!` (`cut`) is hard to handle.

An Augmented Meta-Interpreter Generating a Proof

This meta-interpreter generates a proof.

```
solve(true,true).
solve(P,if(P,ax(P if Q),Qt)) :- ax(P if Q), solve(Q,Qt).
solve(P and Q,and(Pt,Qt)) :- solve(P,Pt), solve(Q,Qt).
```

An Augmented Meta-Interpreter for Depth-Bounded Pure Prolog

```

solve(true,N) :- N>=0.
solve(P,N) :- N>0, ax(P if Q), N1 is N-1, solve(Q,N1).
solve(P and Q,N) :- solve(P,N), solve(Q,N).

```

This meta-interpreter finds a proof with a search tree depth of at most N levels.

```

depth(0).
depth(N) :- depth(N1), N is N1+1.

solve(G) :- depth(N), solve(G,N).

```

`solve/1` tries at gradually deeper levels of the tree, so called *iterative deepening* (see section 4.5).

7.4.6 Meta-Logical Predicates in SICStus

Meta-logical predicated are built-in predicates that perform operations that require reasoning about:

- the *current instantiation* of terms
- *decomposing terms* into their constituents

Instantiation Checking

`var(X)` - checks that X is uninstantiated variable (not a structure)

`nonvar(X)`- opposite to `var/1`

`ground(X)` - checks that X is completely instantiated

Example 9.1

```
?- var(X), X=1 -Yes, X=1.
?- X=1, var(X). -No.
?- nonvar(father(X,Y)). -Yes.
```

Define the predicate `plus/3`, which uses built-in arithmetics and performs plus and minus.

```
plus(X, Y, Z) :- nonvar(X), nonvar(Y), Z is X+Y.
plus(X, Y, Z) :- nonvar(X), nonvar(Z), Y is Z-X.
plus(X, Y, Z) :- nonvar(Y), nonvar(Z), X is Z-Y.
```

Type Checking

`integer(X)` - checks that `X` is instantiated to an integer

`float(X)` - checks that `X` is instantiated to a float

`number(X)` - checks that `X` is instantiated to a number

`atom(X)` - checks that `X` is instantiated to an atom (that is a non- variable term of arity 0, other than a number)

`atomic(X)` - checks that `X` is instantiated to an atom or number

`simple(X)` - checks that `X` is uninstantiated or instantiated to an atom or number

`compound(X)` - checks that `X` is instantiated to a term of arity higher than 0, that is a list or a structure

`==, (\==)` - syntactic equality (inequality) of terms

`?=` - terms identical or cannot unify

```
?- X=2, X==Y. - No.
```

```
?- ?=(X,Y). - No.
```

Decomposing Terms

```
functor(Term, FunctorName, Arity)
```

the functor of term `Term` has name `FunctorName` and arity `Arity`

For example

```
?- functor(father(erik,jonas),father,2). - Yes.
?- functor(father(erik,jonas),F,A). - Yes, F=father, A=2.
?- functor(Term,father,2). - Yes, Term=father(_A,_B).
?- functor(Term,father, N). - instantiation error
```

```
arg(N, Term, Argument)
```

the `N`th argument of a compound term `Term` is `Argument`

For example,

```
?- arg(1, father(erik, jonas), Arg). - Yes. Arg=jonas.
?- arg(2, father(erik, X), jonas). - Yes. X=jonas.
?- arg(N, father(erik, jonas), jonas).- instantiation error

?- arg(2, Y, jonas) - instantiation error
?- arg(1, father(X, Y), Z). - Yes. Z=X.
?- arg(3, father(X, Y), Z). - No.
Term=..List ( =.. is called univ)
```

`List` is a list whose head is the atom corresponding to the principal

functor of `Term`, and whose tail is a list of the arguments of `Term`.

For example,

```
?- father(person(erik,A,b), person(jonas,X,Y))=..List.

- Yes.
List=[father,person(erik,A,B),person(jonas,X,Y)].
```

```
?- Term=..[father, erik, X].
- Yes.
Term=father(erik,X).

?- father(erik, jonas)=..[father, erik, jonas].
- Yes.
```

Example 9.2

Define the predicate `subterm/2`, for checking if `Arg1` is a subterm of `Arg2`.

```
%subterm(Arg1,Arg2)
subterm(T, T).
subterm(S, T) :-
    compound(T),
    functor(T, F, N),
    subterm(N, S, T).
subterm(N, S, T) :-
    N > 1, N1 is N-1,
    subterm(N1, S, T).

subterm(N, S, T) :-
    N > 0, arg(N, T, Arg),
    subterm(S, Arg).
```

Example 9.3

Define the predicate `subterm/2`, for checking if `Arg1` is subterm of `Arg2`

(this time using other meta-logical predicates)

```
%subterm(Arg1,Arg2)
subterm(T, T).
```

```

subterm(S, T) :-
    compound(T),
    T=..[F|Args],
    subtermList(S, Args).
subtermList(S, [Arg|Args]) :-
    subterm(S, Arg).
subtermList(S, [Arg|Args]) :- subtermList(S, Args).

```

Define univ using functor/3 and arg/3

a) Term is given:

```

Term=..[F|Args] :-
    functor(Term, F, N),
    args(T, Args, 0, N).

args(_, [], N, N).
args(T, [Arg|Args], I, N) :-
    I < N,
    I1 is I+1,
    arg(I1, T, Arg),
    args(T, Args, I1, N).

```

Define univ/2 using functor/3 and arg/3

b) List is given:

```

T=..List :- List=[F|Args],
    length(Args, N),
    functor(T, F, N),
    args(Args, T, 1).

```

```

args([], _, _).
args([Arg|Args], T, N) :-
    arg(N, T, Arg),
    N1 is N+1,
    args(Args, T, N1).

```

7.4.7 Support for Dynamically Changing Knowledge Bases

Prolog contains some predicates used for manipulating the program seen as a database. The most common ones are `assert`, `retract`, `abolish`, etc. (see manual). These are support predicates used to modify of a program “on-the-fly”. The most natural use of these primitives is for small databases, but they can be used also for programs that modify rules at run-time. Since these modifications do not allow the code to be compiled, it is not recommended to use these techniques unless other methods are deemed unsuitable.

`assert(Clause)` - `Clause` is added to the program

`clause(Head, Body)` - finds a clause with the head `Head` and the body `Body`

`retract(Clause)` - `Clause` is erased from the program

For example:

```

member(X, [X|Ys]).
member(X, [Y|Ys]) :- member(X, Ys).

```

```
?- clause(member(H1, H2), B).
```

Yes.

```
H1=X, H2=[X|Ys], B=true;
```

```
H1=X, H2=[Y|Ys], B=member(X, Ys).
```

NB: Meta-level reasoning requires care to be taken concerning the range of the variables.

Chapter 8

Expert Systems

Outline

- Forward chaining - Backward chaining
- Meta-programming to form an expert system with a clearly separated inference engine, user interface and knowledge base.
- Expanding the meta-interpreter `solve` to handle a diagnosis application malfunction as discussed in Nilsson and Małuszyński.

In particular we discuss:

- Allowing user input for symptoms in a diagnosis application (abduction)
- Avoid asking the user the same question twice
- Utilise bounded depth search
- Utilise iterative deepening search
- Generate a proof object to facilitate handling of how/why questions
- A few brief comments on treating negative knowledge and prioritised rules.

Expert System =
 Knowledge Base (KB)
 +Inference Engine (IE)
 +User Interface (UI)

[Kowalski]

Expert System Shell = (IE) + (UI)

8.1 Production Rules

Often knowledge base consists of a set of *production rules*, having, for example, the form

IF A1 AND(OR) A2 THEN C1 AND C2

and *forward* or (and) *backward-chaining* is employed depending whether one starts from assumptions to conclusions or from conclusions (hypothesis) to assumptions.

We can encode such rules directly (on the object level) as clauses in Prolog. The knowledge base is composed from a set of clauses and SLD-resolution is employed as inference engine. SLD-resolution is a backward chaining proof procedure.

8.1.1 Rule Base as Prolog Clauses

IF A1 AND (A2 OR A3) THEN C1 AND C2

C1 :- A1, (A2 ; A3).

C2 :- A1, (A2 ; A3).

8.1.2 Uncertainty

Expert systems might contain rules and/or facts which hold with some degree of uncertainty.

```
IF it is summer AND there are no clouds
THEN the temperature is above 20 degrees C
CERTAINTY 80%
```

8.1.3 User Interface - Dialogue

An expert system is often not automatic. User interaction guides the search.

- Graphical user interface
- Dialogue

8.1.4 Explanation Facilities

Expert systems should be able to *explain* its conclusions to different people (experts, programmers, users).

- **How** did you come to your conclusion?
- **Why** does A follow from B ?

8.1.5 Knowledge Acquisition

An expert system should allow *incremental updates* of the knowledge base and rule base.

Knowledge can be acquired through **dialogue** with an expert, or through **analysis** of the system.

8.1.6 Expert System: Diagnosis

```

IF Y is a necessary component for X and Y is malfunctioning
    THEN X is also malfunctioning.
IF X exhibits a fault-symptom Z
    THEN either X is malfunctioning
        or there exists another malfunctioning component
            which is necessary for X.
IF there exists a component which is
    necessary for X and which malfunctions
    THEN X has an indirect fault.

```

```

malfunctions(X) :- needs(X,Y), malfunctions(Y).
malfunctions(X) :- symptom(Y,X), not indirect(X).

```

```

indirect(X) :- needs(X,Y), malfunctions(Y).

```

```

needs(car,fuel_system).
...
needs(electric_system,fuse).

```

Abduction

The knowledge base is usually *incomplete*.
symptoms are not known, but need to be established by asking questions in a *dialogue*.

KB + cause \mid - symptom
 Finding cause is named *abduction*.

Self-Interpreter Generating a Proof

```

solve(true,true).
solve(P,proof(P,Qt)) :- kb(P if Q), solve(Q,Qt).
solve(P and Q,and(Pt,Qt)) :- solve(P,Pt), solve(Q,Qt).

```

Query-the-User

```

solve(true).
solve(P and Q) :- solve(P), solve(Q).
solve(symptom(X,Y)) :- confirm(X,Y).
solve(P) :- kb(P if Q), solve(Q).

confirm(X,Y) :-
    write('Is the '),
    write(Y), tab(1),write(X), write('? '),
    read(yes).

```

Knowledge Base

```

ax(malfunctions(X) if possible_fault(Y,X) and symptom(Y,X)).
ax(possible_fault(flat,tyre) if true).

:- solve(malfunctions(X)).
Is the tyre flat?
yes (user input)
X=tyre

```

(anything but `yes` as answer makes the query fail)

8.1.7 Expert Systems: An Example

We develop an expert system, which finds a name of a fruit when some fruit characteristics such as shape, diameter, surface, colour and the number of seeds are given. The rules are taken from a book by Gonzalez: “The engineering of knowledge-based systems.” (p91). We interleave the rules with corresponding definitions as Prolog clauses. The parameters must be repeated in each clause possibly as anonymous void variables, `_`, since Prolog uses matching. When the number of parameters to consider in a rule becomes larger it might make sense to package the parameters in for instance a dictionary (see section 2.3.7).

```

% Rule 1:
% IF Shape=long and Colour=green or yellow
% THEN Fruit=banana
fruit(Name,Shape,Diameter,Surface,Colour,
      FruitClass,SeedCount,SeedClass) :-
      Shape==long, (Colour==green ; Colour==yellow),
Name=banana.
% Rule 2:
% IF Shape=round or oblong and Diameter > 4 inches
% THEN FruitClass=vine
fruit(Name,Shape,Diameter,Surface,Colour,
      FruitClass,SeedCount,SeedClass) :-
      var(FruitClass),
      (Shape==round ; Shape==oblong),
      integer(Diameter),
      Diameter>4,
      FruitClass=vine,
      fruit(Name,Shape,Diameter,Surface,Colour,
            FruitClass,SeedCount,SeedClass).

% Rule 3:
% IF Shape=round and Diameter < 4 inches
% THEN FruitClass=tree
fruit(Name,Shape,Diameter,Surface,Colour,
      FruitClass,SeedCount,SeedClass) :-
      var(FruitClass),
      Shape==round,
      integer(Diameter),
      Diameter<4,
      FruitClass= tree,
      fruit(Name,Shape,Diameter,Surface,Colour,
            FruitClass,SeedCount,SeedClass).

% Rule 4:
% IF SeedCount=1
% THEN SeedClass=stonefruit
fruit(Name,Shape,Diameter,Surface,Colour,
      FruitClass,SeedCount,SeedClass) :-
      var(SeedClass),

```

```
integer(SeedCount), SeedCount:=1,
SeedClass=stonefruit,
fruit(Name,Shape,Diameter,Surface,Colour,
FruitClass,SeedCount,SeedClass).

% Rule 5:
% IF Seedcount>1
% THEN SeedClass=multiple
fruit(Name,Shape,Diameter,Surface,Colour,
FruitClass,SeedCount,SeedClass) :-
var(SeedClass),
integer(SeedCount), SeedCount>1,
SeedClass=multiple,
fruit(Name,Shape,Diameter,Surface,Colour,
FruitClass,SeedCount,SeedClass).

% Rule 6:
% IF FruitClass=vine and Colour=green
% THEN Fruit=watermelon

% Rule 7:
% IF FruitClass=vine and Surface=smooth and Colour=yellow
% THEN Fruit=honeydew

% Rule 8:
% IF FruitClass=vine and Surface=rough and Colour=tan
% THEN Fruit=cantaloupe

% Rule 9:
% IF FruitClass=tree and Colour=orange and SeedClass=stonefruit
% THEN Fruit=apricot

% Rule 10:
% IF FruitClass=tree and Colour=orange and SeedClass=multiple
% THEN Fruit=orange

% Rule 11:
% IF FruitClass=tree and Colour=red and SeedClass=stonefruit
% THEN Fruit=cherry
```

```
% Rules 6-11 have this form:
fruit(Name,Shape,Diameter,Surface,Colour,
      FruitClass,SeedCount,SeedClass) :-
      FruitClass==tree,
      Colour==red,
      SeedClass==stonefruit,
      Name=cherry.

% Rule 12:
% IF FruitClass=tree and Colour=orange and SeedClass=stonefruit
% THEN Fruit= peach

% Rule 13:
% IF FruitClass=tree and Colour=red or yellow or green
      % and SeedClass=multiple
% THEN Fruit=apple

% Rule 14:
% IF FruitClass=tree and Colour=purple and SeedClass=stonefruit
% THEN Fruit=plum

% Example Query:
% ?- fruit(Name,round,3,Surface,red,FruitClass,1,SeedClass).
      Yes. Name=cherry.
```

Chapter 9

Constraint Logic Programming (CLP)

Outline

- Constraints (CLP) and practical applications

Nilsson and Matuszyński ch. 14
SICStus Prolog Manual

LP is naturally extended to *constraint logic programming*, CLP, which allows the use of relations over other domains than terms (finite sets (FD), reals, rationals). This leads to a change in the logical models considered for program. Unification is considered as constraint solving in a domain of terms, while relations over terms are treated by *constraint solving*.

CLP is useful in many practical application since it avoids search by using constraint propagation and more expressive domains.

9.1 Constraint Logic Programming

Collect not only equalities, but other conditions

Basic Concepts:

- Proof Trees
- Conditional Answers
- To increase efficiency
- to allow more flexible problem formulation

Solve equation system at the end with a special constraint solver for some domain.

$X < Y$ not an error even though X and Y are not known.

CLP(FD) : Finite Domains:

$X \text{ in } S$ means X is in the set S

$X > Y$ means finite domain element X denotes the elements greater than the elements denoted by Y .

Constraint variables take finite sets as values, not only individual values.

CLP(R): Real Numbers, CLP(Q): Rational Numbers:

The sets that can be expressed are limited to solutions to linear equations.

(non-linear are considered combinatorically unfeasible)

Global Constraints:

Specifying relation schema for several constraint variables using one global statement.

- Allows advanced low-level optimisations to be used by the system.
- Avoids repetition of similar constraints over perhaps thousands of variables.

Global constraints are an active research topic.

Application Areas of Constraints:**Planning and scheduling**

Planning problems occur in time-tabling in railways, schools, production planning, and in geometric planning, etc.

Some companies that have use logic programming techniques here (to my knowledge):

- Ovako Steel, SJ, French Railway company, Dassault Electronique

Configuration management

- Invited presentation about use of Prolog and Constraint Logic Programming in a company

A representative of a company [Tacton], talks about a significant problem in configuration and how their software solves the problem using SICStus Prolog with CLP(FD).

Some points of the presentation:

- Configuration is an NP-complete problem

compromise needed, some common approaches:

- reduce interactivity
- put responsibility on the end-user
- incomplete searches

- Features-and options configurations
- Compositional configuration
- Product structure handled with boolean conditions
- General constraint solver using CLP(FD)
- Constructive search

Constraint programming generalises logic programming to deal with semantically richer data domains. Novel systems for running parallel and concurrent programs which are distributed for instance on the internet make networked interactive applications everyday commodities and created new possibilities and challenges.

the following section is taken from a report co-authored with members of a team at SICS specializing in constraint based planning and scheduling lead by Per Kreuger.

9.1.1 Constraint Programming - A Planning Perspective

Constraint programming (CP) [Hen89, Tsa93] is based on the idea of an abstract space of statements (restrictions or conditions), a constraint space.

Some of these statements can be understood as fully determined. Take for instance the claim that a given train will depart from Avesta 15.05 Thursday

April 15th in the year 2001. Other statements are less exact for instance that a steel manufacturer needs to transport between 320 and 380 kilotons of steel from Hofors and Hellefors to Malmö next year. Both these statements can be represented as conditions in a constraint programming system. These are named *constraints* or restrictions on the value space for the variables contained by the statement.

It is certainly non-trivial to determine in a space of such statements how for instance these two statements are related given some mathematical model of a planning problem but under certain circumstances it is possible to make calculations with such abstract objects and to determine for instance logical consistency, that is to decide whether the statements are possibly both true or not.

It is also possible to compute one or several *witnesses*, that is assignments of values to all variables in a space. This is called to *enumerate the search space* for a given problem and in general it contains the utilisation of a search procedure. Such witnesses can in the production planning domain for instance be concrete production plans.

If many such plans are generated they can be compared with respect to different measures of *cost*. To find the best plan is modelled as an optimisation problem in the constraint system.

Much of the search when enumerating the search space for a given problem can often be eliminated with a technique named *constraint propagation*. This means briefly that each statement that is not completely determined is considered as a temporarily interrupted computation which can be made to interact with other similar concurrent computations. Computations are interrupted when the information that is needed to determine a value is missing but they continue once the information is later available.

A constraint programming system can be seen as a set of parallel processes which communicate, interact and are synchronised via shared variables in a dataflow graph. Whether it is possible to compute solutions for a given problem or not is to a large extent depending on the expressive power of the language being used to express statements about the problem. To formulate a mathematical model of some real process in such a language is in general a very hard problem. In spite of this there have been good results by using

constraint techniques to model and solve known hard planning problems for instance many classical scheduling and resource allocation problems.

Consider the example with a departure time. If it is known one might like to represent it as a number. Assume now that it is not known, but that there is nevertheless some information available about it, for instance that it must be between 11 and 12 some given day and that it must follow after the arrival time for some other trip.

This is a very strong restriction of the value for the departure time. Yet it contains an ambiguity which separates it in a fundamental way from the totally determined one.

A number of constraint statements can for instance express the relation between this departure time and other departure times (or arrivals) and can be said to represent a specification of a production plan. The more determined it is, the closer it is to a finished plan with totally determined times.

In principle it is possible to construct a production plan by successively adding more and more information to a space containing conditions (a constraint space).

Finite Domains

The constraint programming systems that have been most actively developed the last ten years are those that handle *finite domains*. In such a system each variable can take on values from a finite set of discrete values. This type of variable is natural to use to model discrete entities such as the number of engines or staff that have been allocated to a given task. They are, however, unnecessarily restrictive when the modelling concerns values that can be assumed to vary over continuous domains (with an infinite number of possible values), for instance time.

Global Constraints

A global constraint is a way to express properties of many variables in one statement.

The first type of constraint that was studied in constraint programming was constraints that relate two variables, for instance $<, \leq, \neq, =$ etc. In contrast to these simple binary constraints the focus has in recent years more and more been on complicated constraints between an unlimited number of variables. Examples of such constraints are such that relate variables with the value of a linear sum or such that maintain pairwise disequality of an arbitrarily large set of variables.

Such constraints can in principle often be encoded in terms of a set of simpler binary constraints which semantically have the same meaning. This is rarely practical, however, since an efficient solution is often too computationally complex to be realised by simply considering the variables in a pairwise fashion. The expression *global constraints* for this type of constraints was introduced in [BC94] and refers to arguments that can be made over a multitude of variables related with a non-binary condition.

Global constraints are abstractions of more complicated properties of problems and enables computations on a more detailed model. Many times methods from operations analysis or algorithm theory, which operate on graphs, can efficiently and naturally be integrated into a constraint programming system as global constraints. This is an active and very promising research area in constraint programming. For a systematic description of a large number of global constraints see [Bel00].

Constraint solving is often limited by a prohibitively large search space, for instance by the enumeration of many redundant variants. The use of meta-operations such as constraint relaxation, redundant modeling [CLW96] etc. could be utilised to improve the search behaviours of a solver. Another approach is to use abstraction to speed up search problems. This has been investigated in AI for general search problems [HMZM96, HPZM95, Yan97].

Prolog SICStus Prolog [SICStus], the dominating logic programming language, now standardised, is developed by adding new expressive possibilities, for instance constraints, object orientation, functional and meta-logical extensions and embeddings, parallel execution models and interoperability with other systems such as databases, GUI toolkits, Java etc.

Mozart-Oz [Mozart] is especially targeted at modern intra/inter-net applications. It is designed to handle concurrency and multi-paradigm programming. This language utilises logical variables as in logic programming, but bases its operational semantics not on SLD-resolution as Prolog does, but on a rewrite semantics driven by entailment as in concurrent constraint programming [SarThesis]. This approach, while more general and flexible than that of Prolog, naturally embeds logic programming as one of its programming paradigms [Mozartsite] together with higher-order functional and object-oriented programming. Constraint programming is also well supported [MuPoSchWu95, MuWu95]. If needed, the user can explicitly program the search.

9.1.2 OR-Techniques

Techniques from operations analysis, for instance linear programming (LP) and integer programming (IP) handles models efficiently where most of the variables are continuous and where a simple and well defined cost function well captures the “goodness” of different solutions to a given problem. OR-techniques are often based on massive computations where input data is given in the form of a complete problem description.

Linear Programming

Linear programming is used to model optimising problems where the cost function can be expressed as a linear sum over continuous variables and where the conditions are linear inequalities over these.

A linear program:

$$c_1x_1 + \cdots + c_nx_n \tag{9.1}$$

is maximised where

$$\begin{aligned} a_{11}x_1 + \cdots + a_{1n}x_n &\leq b_1 \\ &\vdots \\ a_{m1}x_1 + \cdots + a_{mn}x_n &\leq b_m \end{aligned} \tag{9.2}$$

and

$$x_1 \geq 0, \dots, x_n \geq 0 \tag{9.3}$$

Simplex is the oldest and most well known algorithm to solve this type of problem. Despite that Simplex in most applications has very good complexity properties (polynomial time complexity) there is for certain classes of problems specialised algorithms which are even more efficient. An example of such an algorithm is *network flow optimisation* (see section 9.1.2).

The main disadvantage with Simplex is that it works only if all the conditions are linear and the variables are continuous. The latter requirement excludes all disjunctive conditions (decision problems) where the value of a function depends on a boolean variable. The research field studies the problems that occur when you loosen the first demand. In OR it is called non-linear programming and will not be further covered here. The problems that can be formulated when variables are allowed to vary over discrete values are studied in the area of integer programming. Sometimes problems where both types of variables occur are called *mixed integer programming*.

Network Flows is a collecting concept for many types of linear programming algorithms which handle optimisation of flows in networks. The area is well researched and many problems are classified. Three important sub-problems in network flow optimisation are *shortest path*, *maximal flow* and *minimal cost*. When computing the shortest path the issue is to find the shortest path between some or all points in a network. Maximal flows encode problems where the flow between pairs of points in the network should be maximised. In minimal cost problems costs are assigned for the flow between different points and the cost of sending flows from one or more points in the network to one or more destinations is minimised.

Since this area is large, the algorithms are fairly dissimilar. For minimal cost problems specialised variants of *Simplex* are often used.

Integer Programming

Many times introducing integer variables can be avoided by using more or less advanced modelling tricks that encode a problem with integer demands as a linear problem. This often demands a comparatively specialised mathematical competence and is far from always possible.

	LP (lin progr.)	IP (integer progr.)	CP (constraint progr.)
domains	continuous	integ/mixed	integ/(fin. set), intervals
cost function	optimisation	optimisation, search	domain limitation
scalability	polyn. compl.	asome problems	decision problems
search	SIMPLEX	problem dependence	method integration

Table 9.1: Some Properties of a Selection of Techniques

Allowing integer conditions gives a significantly more free modelling, but search must instead be introduced in the algorithms. The search mechanisms that have been developed to solve this type of problem use the cost function in a direct way and solve *linear relaxations* (for parts) of the real problem in each step (iteration).

Many heuristic methods have been developed to make the search converge faster for certain classes of problems. Given a concrete problem, to determine whether it can be modelled as an instance of one of these well studied problem classes also requires specialised mathematical competence.

Lagrange-Relaxation is a technique which can be seen as systematically reformulating integer demands as parameters in the cost function in a corresponding linear program. These parameters are then adjusted by solving the relaxed problem with respect to the parameterised cost function. In each step the parameters are adjusted based on the results from the preceding iteration.

Lagrange-relaxation has successfully been used to solve for instance very large vehicle rotation problems [Löb98] and it is also one of the most important techniques to solve pairing problems (see below). For a general introduction to Lagrange-relaxation and also descriptions of a number of so called local search mechanisms which are not treated in this article see [Ree95a].

Pairing Algorithms

Most optimising systems for staff planning in the transport sector (that is for travelling staff) work in two steps:

1. Produce a number of possible circuits, that is jobs containing tasks (legs) that can be executed in order by for instance one person. Each circuit should also (sometimes by adding passive transports) describe a cycle in the track net graph, that is for instance starting and ending in the same location.
2. Solve an optimising problem that contains in choosing among the above generated circuits a subset such that all tasks are a part of at least one cycle and so that a global cost is minimised.

Step two is from a mathematical point of view simple to formulate, even if the size of the problems can many times make them hard to solve. In order to reach as good results as possible you must therefore in step one generate as many candidates as you can handle computationally in step two. It is however in practice often impossible to consider all possible circuits, so the selection that is being made is of utmost importance for the result.

Step two can simplified be described as constructing a matrix, with the tasks as rows and the circuits generated in step one as columns. In the meeting points between the tasks and circuits there is the value 1 if the task is a part of the circuit, and 0 otherwise. Furthermore it is required that each row shall sum up to at least the number of persons needed to perform the job. The task for the optimisation algorithm is now to assign boolean (0/1) values so that the cost is minimised.

There are special algorithms for 0/1-matrices that are very efficient and which can handle large data sets.

Step one is not as simple, and it is here that the commercial solvers differ. This is the step that is named *pairing*. It is important that a good selection of alternative circuits is generated in this step, since these are the only candidates considered in the search for solutions.

The circuits should also satisfy conditions that encode laws and union agreements, which are often hard to represent in a correct and efficient way. It is far from clear that agreements and legislation are mathematically consistent and there is always room for interpretations that often vary locally. Since agreements also change with time, it is important that they are represented in such a way that they are easy to maintain.

In addition a number of heuristically motivated generation conditions are represented and cost parameters, which limit the choice of circuits to those that are considered reasonable.

These two sets of conditions are then used in different methods to generate candidates for circuits for step two. Two main methods are used in this context: *Integer programming* with *Lagrange-relaxation* and *column generation* (see above and for instance [DSD84, RS94]). This is still an active area of research within OR.

9.1.3 OR-Techniques vs. Constraints

OR-methods are very efficient when the model of the real problem suits well into some well known class of problems and the problem is pure, that is independent of a context which is hard to describe or too complex. They are therefore often less suitable in an early stadium of the planning process when many parameters are yet unknown. On the other hand they have a given place once the search space of the problem has been shrunk with other methods and when you want to compare results of strategic choices in well delimited sub-problems. You can see many of these methods as planning primitives, methods that can be used to investigate properties of the problem.

In contrast, the techniques that have been developed in constraint programming, using finite domains, work well also when a majority of the variables model naturally discrete entities, when the cost function is hard to determine and when the model contains complicated (for instance non-linear) conditions. As mentioned, many of the best approaches to solving scheduling problems during the latest ten years have been introduced as global constraints [BLP95, BLPN95, CL96, CP89, CP94].

Constraint techniques offer the possibility to maintain a dynamically changing space of statements that represent all currently possible (sub)-plans given the constraints from customer demands, resource limitations and cost considerations. It is for instance possible to choose to optimise parts of a plan late and incrementally and leave the rest of the plan only partially determined. Another advantage is that it is possible to incorporate techniques from operations analysis into the constraint paradigm in the form of global constraints. This makes constraint programming an *integrative* project, where techniques from

different areas are collected and made available to modelling experts without requiring from them a detailed algorithmic competence. In this way these two classes of techniques can be said to complement each other. Much current research deals with finding suitable combinations of OR-techniques and constraints.

Appendix: Useful Prolog Predicates

```
a_list([]).
a_list([_|T]) :- a_list(T).

append([],L,L).
append([H|T],L,[H|R]) :- append(T,L,R).

flatten(L,R) :- flatten(L,R, []).
flatten([H|T],L,R) :-
    H=[_|_],
    !,
    flatten(H,L,R0),
    flatten(T,R0,R).
flatten([A|T],[A|R],L) :- flatten(T,R,L).
flatten([],L,L).

isinteger([H|T1],[H|T3],T2) :- isdigit(H), isnumber(T1,T3,T2).

isnumber([H|T1],[H|T1]) :- isnondigit(H).
isnumber([H|T1],[H|T3],T2) :- isdigit(H), isnumber(T1,T3,T2).

% All elements of a list are equal (or initialised to I)
member(X,[X|_]).
member(X,[_|T]) :- member(X,T).

% alternative formulation
member(X,L) :- delete(X,L,_).
```

```

delete(X, [X|T], T).
delete(X, [H|T], [H|R]) :- delete(X, T, R).

get_element(X, L, R) :- delete(X, L, R), !.
get_element(Name, L, L) :- write(undefined(Name)).

rev([], Y, Y).
rev([X1|X], Y, W) :- rev(X, [X1|Y], W).

reverse(X, Z) :- rev(X, [], Z).

lastelement([X], X) :- !.
lastelement([_|T], E) :- lastelement(T, E).

init(I, []).
init(I, [I|T]) :- init(I, T).

index(int(1), [X|_], X).
index(int(I), [_|T], X) :- I > 1, I1 is I-1, index(int(I1), T, X).
index(int(I), _, _) :- I < 1, abort.

ilength(int(0), []).
ilength(int(I), [_|R]) :- I > 0, I1 is I-1, ilength(int(I1), R).

% unique_list(L,U) returns a list where each element occurs only once.
unique_list([], []).
unique_list([H|T], U) :- member(H, T), !, unique_list(T, U).
unique_list([H|T], [H|U]) :- unique_list(T, U).

quicksort([X|Xs], Ys) :-
    partition(X, Xs, Littles, Bigs),
    quicksort(Littles, Ls),
    quicksort(Bigs, Bs),
    append(Ls, [X|Bs], Ys).
quicksort([], []).

partition(X, [Y|Ys], [Y|Ls], Bs) :- X > Y, partition(X, Ys, Ls, Bs).
partition(X, [Y|Ys], Ls, [Y|Bs]) :- X =< Y, partition(X, Ys, Ls, Bs).
partition(_, [], [], []).

```

Definitions Needed for Higher-Order Functions

```

relation(Rlist1(P),X) :- list1(P,X), !.

map_access([Hx|Tx],[Hy|Ty],E,V) :-
    ((E=Hx, V=Hy) ; map_access(Tx,Ty,E,V)).
map_update(L1,L2,L3,L4,E,V) :- L1=[], L2=[], L3=[E], L4=[V]
    ;
    L1=[H1|T1], L2=[H2|T2],
    (H1=E, L3=L1, L4=[V|T2]
    ;
    noteq(H1,E), L3=[H1|T3], L4=[H2|T4], update(T1,T2,T3,T4,E,V)).

```

Logical Input and Output

While input and output represents no particular theoretical problem to a (sequential) imperative language it certainly is one to a language in the Prolog-family. The normal implementation of I/O in Prolog uses traditional non-logical predicates like `get` and `put` to read and write a character on the output device respectively. This is natural if you regard the Horn clause program without respect to the declarative meaning of the clauses involved. But if you want to see the program as clauses stating truths about the world these predicates with side-effects on the outside world are dubious.

It is possible to overcome most of the difficulties with input by considering input as streams. This can be accomplished by defining a predicate that associates a list of characters with the characters in an input file. After this predicate has succeeded the list contains the characters of the file. This list can be unified with other lists in order to check the content. In this way any I/O-stream can be incorporated in a logic language in an acceptable way, logically speaking.

```

char_infile(FileName,List) :-
    open(FileName,read,S),
    readchars(S,List),

```

```
close(S),  
!.
```

```
readchars(S,L) :- get0(S,C), readchars0(S,L,C).
```

```
readchars0(_,L,-1) :- !, L=[].
```

```
readchars0(S,L,C) :- L=[C|R], readchars(S,R).
```

Output, however, has to be implemented as predicates with side-effect. A simple such predicate is `write(X)` which writes `X` on standard-output. Another is `put(C)` which writes the ascii-code denoted by `C` on standard-output. If the output predicates only occur last in the top goal it is guaranteed that no output is generated from any failing branch.

Bibliography

- [SCHEME] H. Abelson and G. Sussman, with J. Sussman. *The Structure and Interpretation of Computer Programs*, 2nd edition. MIT Press. ISBN 0-262-01153-0.
- [BLP95] P. Baptiste and C. L. Pape. “A Theoretical and Experimental Comparison of Constraint Propagation Techniques for Disjunctive Scheduling”. In the *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence* Montreal, Quebec, pp. 400-606, 1995.
- [BLPN95] P. Baptiste, C. Le Pape, and W. Nuijten. Incorporating efficient operations research algorithms in constraint based scheduling. In *Proceedings of the first international joint workshop on artificial intelligence and operations research*, Timberline Lodge, Oregon, 1995.
- [BC94] N. Beldiceanu and E. Contejean. “Introducing Global Constraints in CHIP”. In *Mathematical Computer Modelling* 20(12): 97–123, Pergamon Press Ltd. 1994.
- [Bel00] N. Beldiceanu. Global constraints as graph properties on structured networks of elementary constraints of the same type. Research R:2000-01, SICS, 2000.
- [CP89] J. Carlier and E. Pinson. “An Algorithm for Solving the Job-Shop Scheduling Problem”. In *Management Science* 35(2): 164-176, 1989.
- [CP94] J. Carlier and E. Pinson. “Adjustments of Heads and Tails for the Job-Shop Scheduling Problem”. In the *European Journal of Operational Research*, 78:146-161, 1994.

- [CaKa] M. Carlsson and K. M. Kahn. LM-PROLOG Users Manual. UP-MAIL, Uppsala University, Box 2059, S-750 02 Uppsala, Sweden.
- [SICStus] M. Carlsson et. al. *SICStus Prolog Programming Manual* at <http://www.sics.se/sicstus.html>
- [CL96] Y. Caseau and F. Laburthe. Improving branch and bound for jobshop scheduling with constraint propagation. Technical report, Laboratoire d'Informatique de l'Ecole Normale Supérieure LIENS, Département de Mathématiques et d'Informatique, 45 rue d'Ulm, 75232 Paris Cedex 05, France, 1996.
- [CLW96] A. B. M. W. Cheng, J. H. M. Lee and J. C. K. Wu. Speeding up constraint propagation by redundant modeling. In *Second International Conference on Principles and Practice of Constraint Programming CP'96*, volume 1118 of *LNCS*, pages 91–103, Cambridge, Massachusetts, USA, Aug 1996. Springer-Verlag. Available in <http://www.cse.cuhk.edu.hk/is1/dPub2.html#Li>.
- [DSD84] J. Desrosiers, F. Soumis, and M. Desrochers. Routing with time windows by column generation. *Networks*, 14:545–565, 1984.
- [Fromherz91] M. P. J. Fromherz. *Explore/L An Object-Oriented Logic Language*. Institut für Informatik der Universität Zürich, Nr 91.06, June 1991
- [Han] Å. Hansson. A formal development of programs. (Ph.D. Thesis, University of Stockholm 1980-01-25)
- [HanTa] Å. Hansson and S.-Å. Tärnlund. A natural programming calculus. (6th Int. Joint Conference on Artificial Intelligence, Tokyo, 20-24 August 1979)
- [Mozart] S. Haridi, P. Van Roy, P. Brand and C. Schulte. “Programming Languages for Distributed Applications”. Invited paper in *New Generation Computing*, Vol. 16, No.3, pp 223-261, 1998. Ohmsa Ltd. and Springer Verlag., Tokyo.
- [Mozartsite] S. Haridi, P. Van Roy, P. Brand, C. Schulte et. al. <http://www.mozart-oz.org/>.
- [Hen89] P. Van Hentenryck. *Constraint Satisfaction in Logic Programming. Programming Logic Series*. The MIT Press, Cambridge, MA, 1989.

- [Ka] K. M. Kahn. Partial evaluation techniques for Prolog. (AI Magazine spring 1984)
- [HMZM96] R. C. Holte, T. Mkadmi, R. M. Zimmer and A. J. MacDonald. Speeding up problem solving by abstraction: a graph oriented approach. *Artificial Intelligence*, 85:321–361, 1996.
- [HPZM95] R. C. Holte, M. B. Perez, A. J. Zimmer and R. M. MacDonald. Hierarchical a*: Searching abstraction hierarchies efficiently. 1995.
- [Vermeir et.al.91b] E. Laenens, D. Vermeir, Univ. of Antwerp, Belgium, N. Leone, P. Rullo. *Efficient query evaluation in a language combining object-oriented and logic programming*. report from ESPRIT project 2424, 1991, CRAI Italy.
- [Vermeir et.al.89] E. Laenens, D. Vermeir and B. Verdonk. *LOCO, a Logic-based Language for Complex Objects*. In *Proc. of ESPRIT89 Technical Conf.*, Project 2424, pp.604-616 , 1989 Philips Applications and Software Services, Eindhoven, The Netherlands and Dept. of Computer Science, Univ. of Antwerp, Belgium.
- [Löb98] A. Löbel. *Optimal Vehicle Scheduling in Public Transit*. Ph. D. thesis, TU Berlin, 1998. Shaker-Verlag, Aachen.
- [McCabe89] F. G. McCabe. *Logic and Objects* Ph.D. thesis, Dept. of Computing, Imperial College of Science and Technology, 1989
- [Moss90] C. Moss. *An Introduction to Prolog++*, Res. rep. DOC 90/10 Imperial College of Science, Technology and Medicine, London, 1990
- [MuPoSchWu95] M. Müller, K. Popov, C. Schulte and J. Würtz. *Constraint Programming in Oz*. DFKI, Saarbrücken, Tyskland, 1995
- [MuWu95] M. Müller and J. Würtz. *Finite Domain Programming in Oz*. DFKI, Saarbrücken, Tyskland, 1995.
- [Sah] D. Sahlin. *Partial Evaluation of Full Prolog*. Ph.D. Thesis. KTH, SICS 1991.
- [Sha] E. Shapiro. A subset of concurrent Prolog and its interpreter. ICOT Technical Report TR-003, February 1983 and Weizmann Institute Technical Report CS83-12, Aug.-83.

- [Sjöland92] T. Sjöland. *Objektorientering i SICStus Prolog*. SICS technical report T92:01, January 1992 (in Swedish).
- [Sjöland92b] T. Sjöland. *Using SICStus Objects for Graphical User Interfaces*. In the 5th Multi-G conference, Stockholm, Sweden. Also research report R92:13, October 1992.
- [Vasey, Spencer et.al.90] P. Vasey, C. Spencer, D. Westwood and A. Westwood. *Prolog++, version 1.0 Programming Reference Manual* Logic Programming Associates Ltd., London, England, 1990
- [Vermeir et.al.91a] D. Vermeir. *A simple interface from LOCO to X-Windows* report from ESPRIT project 2424, 1991, Univ. of Antwerp., Dept. of Mathematics and Computer Science.
- [Wa1] D. Warren. Implementing Prolog, parts 1 and 2. (Dept. of AI, Edinburgh University. 1977)
- [Wa2] D. Warren. Logic Programming and Compiler Writing. in *Software Practice and Experience*, Vol 10, 97-125 1980.
- [Ree95a] C. Reeves, editor. *Modern Heuristic Techniques for Combinatorial Optimization*. McGraw-Hill International (UK) Ltd., 1995.
- [RS94] C. C. Ribeiro and F. Soumis. A columns generation approach to the multiple-depot vehicle scheduling problem. *Operations Research*, 42(1):41–52, 1994.
- [SarThesis] V. A. Saraswat. *Concurrent Constraint Programming Languages*. Ph.D. Thesis 1989, published by MIT Press 1993.
- [Tacton] K. Orsvärn et. al. *Tacton Systems*.<http://www.tacton.com/>
- [Tsa93] E. Tsang. *Foundations of Constraint Satisfaction*. Academic Press, 1993.
- [Yan97] Q. Yang. *Intelligent planning A decomposition and abstraction base approach*. ISBN 3-540-61901-1. Springer-Verlag, Berlin, 1997.