```
dict(S) := leaf(S).

leaf([]) :-
    → true.
leaf([insert(K,V) | S]) :-
    → node(S, K, V, L, R),
      leaf(L),
      leaf(R).
leaf([lookup(K,V) | S]) :-
    → V = not_found,
      leaf(S).

node([], _, _, L, R) :-
    → L = [],
      R = [].
node([insert(K1, V1) | S], K, V, L, R) :-
    → (       K1 = K
          →   node(S, K, V1, L, R)
      ;       K1 < K
          →   L = [insert(K1, V1) | L1],
              node(S, K, V, L1, R)
      ;       K1 > K
          →   R = [insert(K1, V1) | R1],
              node(S, K, V, L, R1) ).
node([lookup(K1, V1) | S], K, V, L, R) :-
    → (       K1 = K
          →   V1 = found(V),
              node(S, K, V, L, R)
      ;       K1 < K
          →   L = [lookup(K1, V1) | L1],
              node(S, K, V, L1, R)
      ;       K1 > K
          →   R = [lookup(K1, V1) | R1],
              node(S, K, V, L, R1) ).
```

In the following section on object-oriented programming, we will relate this programming technique to conventional object-oriented programming and its standard terminology.

## 3.3 OBJECT-ORIENTED PROGRAMMING

In this section, the basic techniques that allow us to do object-oriented programming in AKL are reviewed. Like the programming techniques in the previous section, they are not a contribution of this dissertation, but belong by now to logic programming folklore.

There is more than one way to map the abstract concept of an object onto corresponding concepts in a concurrent constraint language. The first and most widespread of these will be described here in detail [Shapiro and Takeuchi

1983]. It is based on the process reading of logic programs. Several embedded languages have been proposed that support this style of programming (e.g., [Kahn et al. 1987; Yoshida and Chikayama 1988; Davison 1989]). They are typically much less verbose, and they also provide more explicit support for object-oriented concepts. More modern treatments of objects in concurrent logic languages exist (see, e.g., [Smolka, Henz, and Würtz 1993]). This is discussed in Chapter 7.

As will be seen, in this framework there is no real need for an *implementation* of objects, unlike the case when one is adding object-oriented support to a language such as C. Following an object-oriented style of programming is a very natural thing.

As a point of reference, we will adhere to the object-oriented terminology of [Snyder, Hill, and Olthoss 1989]. The meaning of this terminology will be summarised as it is introduced.

### *3.3.1 Objects*

An *object* is an abstract entity that provides *services* to its clients. Clients explicitly *request* services from objects. The request *identifies* the requested service, as well as the object that is to perform the service.

Objects are realised as processes that take as input a stream (a list) of requests. The stream identifies the object. The data associated with the objects are held in the arguments of the process. An object definition typically has one clause per type of request, which performs the corresponding service, and one clause for terminating (or deallocating) the object. Thus, clauses correspond to *methods*. The requests are typically expressions of the form name(A, B, C), where the constructor “name” identifies the request, and A, B, and C are the arguments of the request.

The process description, the agent definition, is the *class*, the implementation of the object. The individual calls to this agent are the *instances*.

A standard example of an object is the bank account, providing withdrawal, deposits, etc.

```
make_bank_account(S) :=
        bank_account(S, 0).

bank_account([], _) :-
    → true.
bank_account([withdraw(A) | R], N) :-
    → bank_account(R, N - A).
bank_account([deposit(A) | R], N) :-
    → bank_account(R, N + A).
bank_account([balance(M) | R], N) :-
    → M = N,
        bank_account(R, N).
```

A computation starting with

make_bank_account(S),
S = [balance($B_1$), deposit(7), withdraw(3), balance($B_2$)]

yields

$B_1 = 0$, $B_2 = 4$

A bank account object is created by starting a process bank_account(S, 0) which is given as initial input an unspecified stream S (a variable) and a zero balance. The stream S is used to identify the object. A service deposit(5) is requested by binding S to [deposit(5) | $S_1$]. The next request is added to $S_1$, and so on.

In the above example, only one clause will match any given request. When it is applied, some computation is performed in its body and a new bank_account process replaces the original one. The requests in the above example are processed as follows. Let us start in the middle.

bank_account(S, 0), S = [deposit(7), withdraw(3), balance($B_2$)].

The bank account process is reduced by the clause matching the first deposit-request, leaving some computation to be performed.

N = 0+7, bank_account($S_1$, N), $S_1$ = [withdraw(3), balance($B_2$)].

This leaves us with.

bank_account(S, 7), S1 = [withdraw(3), balance($B_2$)].

The rest of the requests are processed similarly.

Finally, there are a few things to note about these objects. First, they are automatically *encapsulated*. Clients are prevented from directly accessing the data associated with an object. In imperative languages, this is not as self-evident, as the object is often confused with the storage used to store its internal data, and the object identifier is a pointer to this storage, which may often be used for any purpose.

Second, requests are entirely *generic*. The expression that identifies a request may be interpreted differently, and may therefore involve the execution of different code, depending on the object. This does not involve mandatory declarations in some shared (abstract or virtual) ancestor class, as in many other languages.

Third, *becoming* another type of object is extremely simple. Instead of replacing itself with an object of the same type, an object may pass its stream, and appropriate parameters, on to a new object. An example of this was given in the section on process structures, where a leaf process became a node process when a message was inserted into a binary tree.

### *3.3.2 Inheritance*

In the object-oriented paradigm, objects can be classified in terms of the services they provide. One object may provide a subset of the services of another object. This way an *interface hierarchy* is formed.

It is of course important, from a software engineering point of view, that the descriptions of objects higher up in the hierarchy can be reused as parts of the descendant objects. This is called *implementation inheritance* or *delegation*.

Delegation is easily achieved in the framework we describe. However, since requests are completely generic, it is also possible to design an interface hierarchy without it, if so desired.

Delegation is achieved by creating instances of the ancestor objects. The object identifier of (the stream to) this ancestor object is held as an argument of the derived object. The object corresponding to the ancestor could appropriately be called a *subobject* of the derived object. The derived object filters incoming requests and delegates unknown requests to its subobject.

Delegation is not restricted to unknown requests. We may also define what is elsewhere known as *after-* and *before-methods* by filtering as well. The derived object may perform any action before passing a request on to a subobject.

Let us derive from the bank account class a kind of account that does some form of logging of incoming requests. Let us say that it also adds a get_log service that returns the log. This is easy.

```
make_logging_account(S) :=
        make_bank_account(O),
        make_empty_log(Log),
        logging_account(S, O, Log).

logging_account([get_log(L) | R], O, Log) :-
    → L = Log,
        logging_account(R, O, Log).
logging_account([Req | R], O, Log) :-
    → O = [Req | O1],
        add_to_log(Req, Log, Log1),
        logging_account(R, O1, Log1).
logging_account([], O, _) :-
    → O = [].
```

With delegation, it is cumbersome to handle the notion of *self* correctly. Modern forms of *multiple inheritance*, based on the principle of specialisation, are also difficult to achieve.

Instead, it is quite possible to view inheritance as providing the ability to share common portions of object definitions by placing them in superclasses, which are then implicitly copied into subclass definitions. To exploit this view, syntactic support has to be added to the language, e.g., along the lines of Goldberg,

Silverman, and Shapiro [1992]. This view corresponds closely to that of conventional object-oriented languages.

### *3.3.3 Ports for Objects*

Ports are a special form of constraints, which, when added to AKL, or to any concurrent logic programming language, will solve a number of problems with the approach to object-oriented programming presented above. This section provides a preliminary introduction to ports. They, the problems they solve, and numerous examples of their use, are the topic of Chapter 7.

A *port* is a binary constraint on a bag (a multiset) of messages and a corresponding stream of these messages. It simply states that they contain the same messages, in any order. A bag connected to a stream by a port is usually identified with the port, and is referred to as a port. The open_port(*P*, *S*) operation relates a bag *P* to a stream *S*, and connects them through a port.

The stream *S* will usually be connected to an object. Instead of using the stream to access the object, we will send messages by adding them to the port. The send(*M*, *P*) operation sends a message *M* to a port *P*. To satisfy the port constraint, a message sent to a port will immediately be added to its associated stream, first come first served.

When a port is no longer referenced from other parts of the computation state, when it becomes garbage, it is assumed that it contains no more messages, and its associated stream is automatically closed. When the stream is closed, any object consuming it is thereby notified that there are no more clients requesting its services.

Thus, to summarise: A port is created with an associated stream (to an object). Messages are sent to the port, and appear on the stream in any order. When the port is no longer in use, the stream is closed, and the object may choose to terminate.

A simple example follows.

open_port(P, S), send(a, P), send(b, P)

yields

P = ⟨a port⟩, S = [a,b]

Here we create a port and a related stream, and send two messages. The messages appear in S in the order of the send operations in the composition, but it could just as well have been reversed. The stream is closed when the messages have been sent, since there are no more references to the port.

Ports solve a number of problems that are implicit in the use of streams. The following are the most obvious.

- If several clients are to access the same object, their streams of messages have to be merged into a single input stream. With ports, no merger has to be created. Any client can send a message on the same port.

- If objects are to be embedded in other data structures, creating e.g. an array of objects, streams have to be put in these structures. Such structures cannot be shared, since several messages cannot be sent on the same stream by different clients. However, several messages can be sent on the same port, which means that ports can be embedded.
- With naive binary merging of streams, message sending delay is variable. With ports, message sending delay is constant.
- Objects based on streams require that the streams are closed when the clients stop using them. This is similar to decrementing a reference counter, and has similar problems, besides being unnecessarily explicit and low-level. A port is automatically closed when there are no more potential senders, thus notifying the object consuming messages.

These and other problems and solutions are discussed in Chapter 7.

## 3.4 FUNCTIONS AND RELATIONS

Functions and relations are simple but powerful mathematical concepts. Many programming languages have been designed so that one of the available interpretations of a procedure definition should be a function or a relation. AKL has well-defined subsets that enjoy such interpretations, and provide the corresponding programming paradigms.

### *3.4.1 Functions*

The functional style of programming is characterised by the *determinate* flow of control and by the *non-cyclic* flow of data. There is no don't care or don't know nondeterminism—a single result is computed. Agents do not communicate bidirectionally—an agent takes input from one agent and produces output to another agent. The latter point is weakened somewhat if the language has a non-strict semantics, in which case “tail-biting” techniques are possible.

Many of the AKL definitions are indeed written in the functional style. For example, the “append”, “squares” and “fruits” definitions in the preceding sections are essentially functional, although the latter two were introduced as components in a process-oriented setting.

The basic relation between functional programs and AKL definitions may be illustrated by an example, written in the non-strict, purely functional language Haskell [Hudak and Wadler 1991]. (The appropriate type declarations are supplied with the functional program for clarity.)

```
data (BinTree a) => (Leaf a) | (Node (BinTree a) (BinTree a))

flatten :: (BinTree a) -> [a] -> [a]

flatten (Leaf x) l = x:l
flatten (Node x y) l = flatten x (flatten y l)
```

In AKL, a corresponding program may be phrased as follows.

```
flatten(leaf(X), L, R) :-
    → R = [X | L].
flatten(node(X, Y), L, R) :-
    → flatten(Y, L, L1),
      flatten(X, L1, R).
```

The main differences are that an explicit argument has to be supplied for the output of the "function", and that nested function applications are unnested, making the output of one call the input of another.

AKL is not a higher-order language, but can provide similar functionality in a simple manner. The technique has been known in logic programming for a long time [Warren 1982; Cheng, van Emden, and Richards 1990]. A term representation is chosen for each definition in a program, and an agent apply is defined, which given such a term applies it to arguments and executes the corresponding definition.

Let a term p($n$, $t_1$, …, $t_m$) represent a definition p / ($n$-$m$), which when applied to $n$-$m$ arguments $t_{m+1}$, …, $t_n$ calls p / $n$ with p($t_1$, …, $t_n$).

To give an example relating to the above programs, the term flatten(3) corresponds to the function flatten, and the term flatten(3, Tree) to the function (flatten tree) (where Tree and tree are equivalent trees). A corresponding agent

```
apply(flatten(3), [X,Y,Z]) :-
    → flatten(X, Y, Z).
apply(flatten(3,X), [Y,Z]) :-
    → flatten(X, Y, Z).
apply(flatten(3,X,Y), [Z]) :-
    → flatten(X, Y, Z).
apply(flatten(3,X,Y,Z), []) :-
    → flatten(X, Y, Z).
```

is also defined. In practice, it is convenient to regard apply as being defined implicitly for all definitions in a program, which is also easily achieved.

This functionality may now be used as in functional programs as follows. We define an agent map / 3, which maps a list to another list.

```
map(P, [], Ys) :-
    → Ys = [].
map(P, [X | Xs], Ys0) :-
    → Ys0 = [Y | Ys],
      apply(P, [X, Y]),
      map(P, Xs, Ys).
```

and may then call it with, e.g., map(append(3,[a]), [[b],[c]], Ys) and get the result Ys = [[a,b],[a,c]].

Although by no means necessary, expressions corresponding to lambda expressions can also be introduced. Let an expression

$(X_1, \ldots, X_k) \backslash A$

where A is a statement with free variables $Y_1, \ldots, Y_m$, stand for a term

$p((m+k), Y_1, \ldots, Y_m)$

where $p/(m+k)$ is a new agent defined as

$p(Y_1, \ldots, Y_m, X_1, \ldots, X_k) := A.$

We may now write, e.g., map((X,Y)\append(X, Z, Y), [[b],[c]], Ys) and get the result Ys = [[b | Z],[c | Z]]. Finally, the syntactic gap can be closed even further by introducing the notation

$P(X_1, \ldots, X_k)$

standing for

$apply(P, [X_1, \ldots, X_k])$

Obviously, the terms corresponding to functional closures may be given more efficient representations in an implementation.

### *3.4.2 Relations*

The relational paradigm is known from logic programming as well as from database query languages. Most prominent of logic programming languages is Prolog (see, e.g, [Clocksin and Mellish 1987; Sterling and Shapiro 1986; O'Keefe 1990]), which is entirely based on the relational paradigm. A large number of powerful programming techniques have been developed. Prolog and its derivatives are used for data and knowledge base applications, constraint satisfaction, and general symbolic processing. AKL supports Prolog-style programming. This relation is discussed in Section 8.1.

Characteristic of the relational paradigm is the idea that programs interpreted as defining relations should be capable of answering queries involving these relations. Thus, if a parent relation is defined, the program should be able to produce all parents for given children and all children for given parents, enumerate all parents and corresponding children, and verify given parents and children.

The following definition clearly satisfies this condition.

```
parent(sverker, adam).
parent(kia, adam).
parent(sverker, axel).
parent(kia, axel).
parent(jan_christer, sverker).
parent(hillevi, sverker).
```

It is also satisfied by any non-recursive AKL program that does not use conditional choice, committed choice, or bagof.

Maybe less intuitive, but just as appealing, is the following: a simple parser of a fragment of the English language. The creation of a parse-tree is omitted.

$s(S_0, S) := np(S_0, S_1), vp(S_1, S).$

$np(S_0, S) := article(S_0, S_1), noun(S_1, S).$

article([a | S], S).
article([the | S], S).

noun([dog | S], S).
noun([cat | S], S).

$vp(S_0, S) := intransitive\_verb(S_0, S).$

intransitive_verb([sleeps | S], S).
intransitive_verb([eats | S], S).

The two arguments of each atom represent a string of tokens to be parsed as the difference between the first and the second argument. The following is a sample execution.

$s([a, dog, sleeps], S)$

$np([a, dog, sleeps], S_2), vp(S_2, S)$

$article([a, dog, sleeps], S_1), noun(S_1, S_2), vp(S_2, S)$

$noun([dog, sleeps], S_2), vp(S_2, S)$

$vp([sleeps], S)$

$intransitive\_verb([sleeps], S)$

$S = []$

The relation defined by s is

| | |
|---|---|
| s([a, dog, sleeps \| S], S) | s([a, dog, eats \| S], S) |
| s([a, cat, sleeps \| S], S) | s([a, cat, eats \| S], S) |
| s([the, dog, sleeps \| S], S) | s([the, dog, eats \| S], S) |
| s([the, cat, sleeps \| S], S) | s([the, cat, eats \| S], S) |

for all S, and will be generated as (don't know nondeterministic) alternative results from

$s(S_0, S)$

The idea of a pair of arguments representing the difference between lists is important enough to warrant syntactic support in Prolog, the DCG syntax, which allows the above definitions to be rendered as follows.

s --> np, vp.

np --> article, noun.

article --> [a].
article --> [the].

and so on.

The example is naive, since real examples would be unwieldy, but the state of the art is well advanced, and the literature on *unification grammars* based on the above simple idea is rich and flourishing.

## 3.5 CONSTRAINT PROGRAMMING

Many interesting problems in computer science and neighbouring areas can be formulated as *constraint satisfaction problems* (*CSPs*). To these belong, for example, Boolean satisfiability, graph colouring, and a number of logical puzzles (a couple of which will be used as examples). Other, more application oriented, problems can usually be mapped to a standard problem, e.g., register allocation to graph colouring. In general, these problems are NP-complete; any known general algorithm will require exponential time in the worst case. Our task is to write programs that perform well in as many cases as possible.

A CSP can be defined in the following way. A *(finite) constraint satisfaction problem* is given by a sequence of variables $X_1, \ldots, X_n$; a corresponding sequence of (finite) domains of values $D_1, \ldots, D_n$; and a set of *constraints* $c(X_{i_1}, \ldots, X_{i_k})$. A *solution* is an assignment of values to the variables, from their corresponding domains, which satisfies all the constraints.

For our purposes, a constraint can be regarded as a logical formula, where satisfaction corresponds to the usual logical notion, but formalism will not be pressed here. Instead, AKL programs are used to describe CSPs, and their intuitive logical reading provides us with the corresponding constraints. Each agent is regarded as a (user-defined) constraint, and will be referred to as such. These agents are typically don't know nondeterministic, and those assignments for which the composition of these agents does not fail are the solutions of the CSP.

The example to be used in this section is the *n*-queens problem: how to place *n* queens on an *n* by *n* chess board in such a way that no queen threatens another. The problem is very well known, and no new algorithm will be presented. The novelty, compared to solutions in conventional languages, lies in the way the algorithm is expressed. The technique used is due to Saraswat [1987b], and was also used by Bahgat and Gregory [1989].

Each square of the board is a variable V, which takes the value 0 (meaning that there is no queen on the square) or 1 (meaning that there is a queen on the square).

The basic constraint is that there may be *at most one* queen in each row, column, and diagonal. Given that *n* queens are to be placed on an *n* by *n* board, a derived constraint, which we will use, is that there must be *exactly one* queen in each row and column. Note that the exactly-one constraint can be decomposed into an at-least-one and an at-most-one constraint. We now proceed to define these constraints in terms of smaller components. The problem is not only to express the constraint, which is easy, but to express it in such a way that an appropriate level of propagation will occur, which will reduce the search space dramatically.