2 Computational Models

2.1 The notion of computational model

2.1.1 Introduction

The abundance of different programming languages and computer architectures can best be overview and classified using a powerful high level abstraction, called computational model.

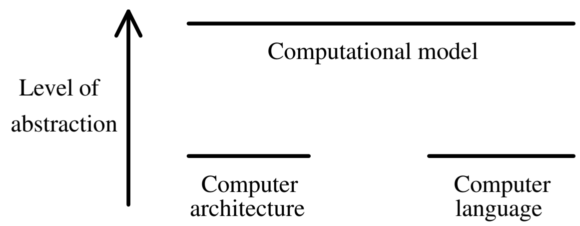


Figure 2-1: Interpretation of the notion computational model as a higher level abstraction

This notion is a *higher level abstraction* than the notions of computer architecture or computer language, since it represents the underlying concept of both (Figure 2-1).

The most complete interpretation of this notion was formulated by Treleaven [Treleaven87, Treleaven90]. In his interpretation the concept of computational model covers the underlying principles for both computer architectures and the corresponding programming languages. He identified the following eight parallel models of computation and the associated programming styles as well as the programming languages (see Table 2-1).

Programming Style	**Programming Languages**	**Models of Computation**
Procedural	C, OCCAM	Control Flow
Object-Oriented	C++, Java	Object-Oriented
Single-Assignment	ID,SISAL	Data Flow
Applicative	Pure Lisp, MIRINDA	Reduction
Predicate Logic	Prolog, GHC	Logic
Production System	OPS5	Rule-Based
Semantic Network	NETL	Cellular Array
Neural Network	SAIC ANSpec	Neurocomputer

Table 2-1: Parallel models of computation identified by Treleaven (1990).

2.1.2 Proposed interpretation of the notion of computational model

It is widely noticed that certain architecture and programming language classes correspond to each other. Examples are the class of von Neumann architectures and the imperative languages (see 2.3) or reduction architectures and functional languages (see 2.5). It is quite obvious that each corresponding architecture and language class must have a common foundation. This *common foundation* for a pair of corresponding architecture and language classes is called *computational model*. For instance, von Neumann architectures and imperative languages have a common foundation which should be called von Neumann computational model (see 2.3). In the other case mentioned, reduction architectures and functional languages are based on the applicative computational model (see 2.5). It is a fact that most existing architectures and languages can be associated with a few computational models.

The concept of computational model is obviously a *higher level abstraction* than either the concept of computer architecture or the programming language, and covers both [Treleaven90]. In our interpretation the notion of **computational model** comprises the *set of* the following *three abstractions*:

- the basic items of the computation,
- the model of the problem description and
- the model of the execution (see Figure 2-2).

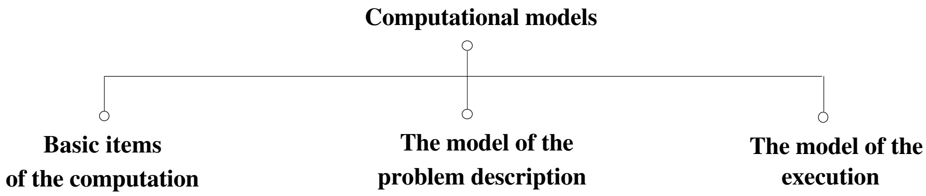


Figure 2-2: Interpretation of the notion of computational model as, convering three abstractions

The first abstraction identifies the **basic items of the computation.** This is a specification of the *items* the computation is declared on and the *kind of computations* (operations) which may be performed on them.

For instance, in the von Neumann computational model the basic items of the computation are data. Data will be typically represented by named (i.e. identifiable) entities in order to be able to distinguish between a number of different data items in the course of a computation. These named entities are commonly called variables in programming languages and are implemented by memory or register addresses in architectures.

All the best known computational models, namely:

- the Turing model,
- the von Neumann model and
- the dataflow model

are based on the *concept of data*.

In contrast, there is a number of models which are *not data based*. In these models the basic items of the computation are:

- either objects as well as messages sent to them requiring an associated manipulation (as in case of the object based model, see 2.7),
- or arguments as well as functions applied on them (applicative model, see 2.5),

- or elements of sets as well as predicates declared on them (predicate-logic based model, see 2.6).

The **model of problem description** refers to both, the type and the method of problem description (Figure 2-3)

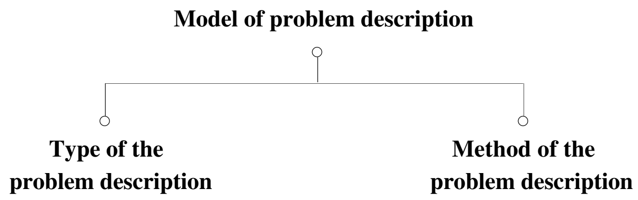


Figure 2-3: Interpretation of the model of problem description

Usually the **type of the problem description** is either procedural or declarative (Figure 2-4).

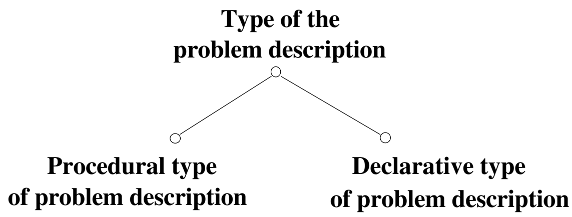


Figure 2-4: Types of the problem description

In a **procedural type** of problem description the *algorithm* of the problem solving is stated. It is declared how the problem can be solved by specifying a particular solution in form of an algorithm. For instance, an algorithm to calculate 'n' factorial could be given as in Figure 2-5 using a Pascal-like syntax. For sake of simplicity the declaration part is omitted.

Procedural style

```
if n = 0 then
     fac = 1
     else fac: = 1;
          for i := 1 to n do
               fac = i * fac;
end;
```

Declarative style

```
fac(0):= 1
fac( n>0 ):= n * fac( n-1 );
```

Figure 2-5: Contrasting the procedural and the declarative style of problem description

On the other hand, if a **declarative style** of problem description is used, all the facts and relationships relevant to the given problem, have to be stated.

There are two fundamental methods for expressing relationships and facts (see Figure 2-6). The first method is using *functions*, showing the way how in the applicative model of computation (section 2.5), problems are described. The other method is to state the relationships and facts in form of *predicates*, which in employed is the predicate logic based computational model (section 2.6).

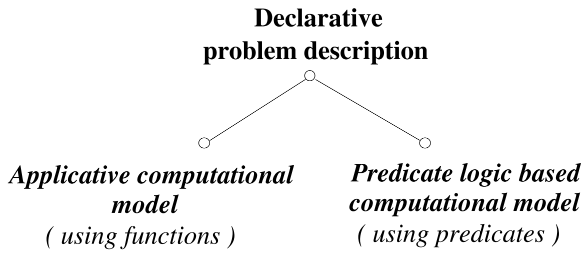


Figure 2-6: Fundamental methods of declarative problem description

In order to illustrate the declarative style of problem description, let us recall the problem of calculating 'n' factorial using functions, i.e. the applicative way of description, (see Figure 2-5).

If, in this case, we would like to calculate, say, 3 factorials, that is fac(3), then first we have to look for the appropriate function definition and substitute into its right side the argument. Next we have to continue substituting repeatedly as long as possible. Finally, we have to evaluate (reduce) numerically the result: Thus, we get:

fac(3) = ?
fac(3) := 3 * fac(2) = 3 * 2 * fac(l) = 3 * 2 * 1 = 6

Note that in the declarative style of problem description the sequence of the statements has no significance, in contrast to the procedural style, where it does.

While stating the relevant relationships and facts that specify a particular problem, the *declarative style* expresses *what the problem is*. This is in sharp contrast to the *procedural style* which articulates *how the problem can be solved* and reveals one specific solution to it. Obviously, the declarative style is a more elegant, concise and abstract alternative for problem description than the procedural style.

The other component of the problem description model is the **method of problem description.** It specifies *how a problem has to be described* in a given computational model.

For instance, in the von Neumann computational model a problem will actually be described as a sequence of instructions expressing the algorithm of the problem. In the applicative model, this is done by stating a particular set of functions which expresses the relationships and facts which are characteristic of the given problem. The specific ways how problems are described in each of the basic computational models will be presented along with the discussion of the basic computational models in sections 2.2 - 2.7.

The third, and the last, part of the computational model outlines the **model of the execution**. It consists of three components, (Figure 2-7).

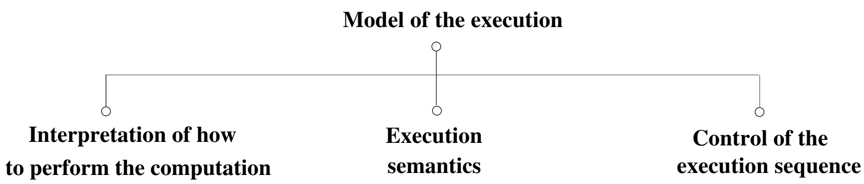


Figure 2-7: Interpretation of the model of execution

The first component of the execution model declares the **interpretation of the computation.** This component of the computational model is strongly related to the method of problem description.

Obviously, the choice of problem description and the model of execution mutually determine and presume each other. Therefore, the interpretation of computation is specific to the computational model concerned. Let us consider one straightforward example. In the case of the von Neumann computational model the problem is described as a sequence of instructions specifying data and control manipulations. Thus, the computation has to be interpreted as the execution of the given sequence of the instructions. This component will also be detailed by the description of the basic computational models in sections 2.2-2.7.

The next component of the execution model specifies the **execution semantics.** It can be interpreted as a *rule* that prescribes *how a single execution step* is to be performed. This rule is, of course, associated with the chosen approach of how problems are described and the execution of the computation interpreted. The different kinds of execution semantics applied by the basic computational models are summarised in Figure 2-8. The *state transition semantics,* is used in the Turing and von Neumann as well as object based models. The *dataflow semantics*, is applied in the dataflow model. Further on the *reduction semantics* is utilised in the applicative model. Lastly, the *SLD-resolution,* is the execution rule of the predicate logic based computational model. These different types of execution semantics will be described in the framework of the basic models.

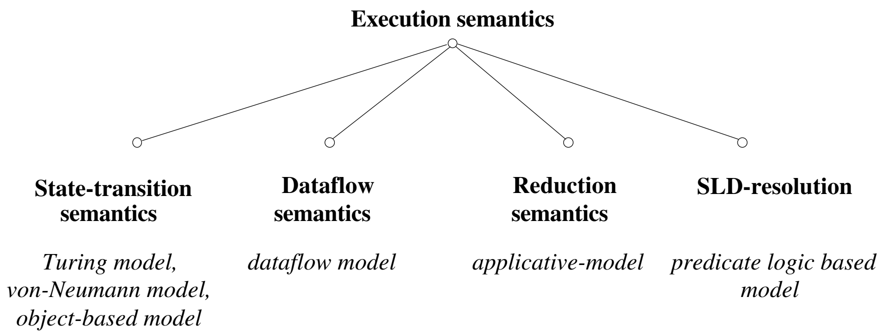


Figure 2-8: Main types of execution semantics

The last component of the model specifies the **control of the execution sequence**.

In the basic models the execution is predominantly control driven, data driven, or demand driven (Figure 2-9).

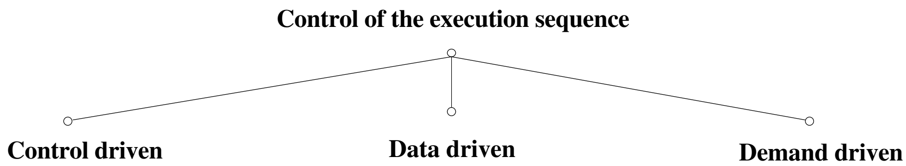


Figure 2-9: Basic ways for controlling the execution sequence

For a **control driven execution** it is assumed that there exists a program consisting of a sequence of instructions. The *execution sequence* is then *implicitly* given by the order of the instructions. However, *explicit control instructions* also exist in order to be able to specify a departure from the implied execution sequence. For instance, in the von Neumann model or the object based model of computation, a control driven execution is taken for granted.

A **data driven execution** is characterised by the rule, that an operation is activated *as soon as all the needed input data are available*. This mode of sequence control is also called **eager evaluation** for apparent reasons. The data driven execution control is characteristic for the dataflow model of computation.

By the **demand driven mode** of execution the operations will be activated only when their *execution is needed* to achieve the final result. This mode of execution control is also termed as **lazy evaluation** because the "delayed until needed" philosophy is applied. The demand driven execution control is usually applied for the applicative model of computation.

Comparing the data and demand driven modes of execution control, it can be stated that data driven computation activates a great amount of unnecessary computation by evaluating all conditional operations. The demand driven computation avoids such computation. However, demand driven operation usually incurs a high overhead to process the propagation and cancellation of demands, and to perform the related additional operations.

2.1.3 Relationships between the notions of computational model, programming language and architecture

As we have seen in the foregoing paragraphs, the notion of computational model is a higher level abstraction than the notions of programming language or computer architecture. In addition, there is the following relationship between these notions: A *programming language* can be thought of as a *specification tool* making possible the formulation of a computational task whereby a particular computational model is presumed. On the other hand, a *computer architecture* may be looked upon as a *tool to implement a computational model*, or to execute a given computational task expressed in a programming language, whereby a particular computational model is given (see Figure 2-10).

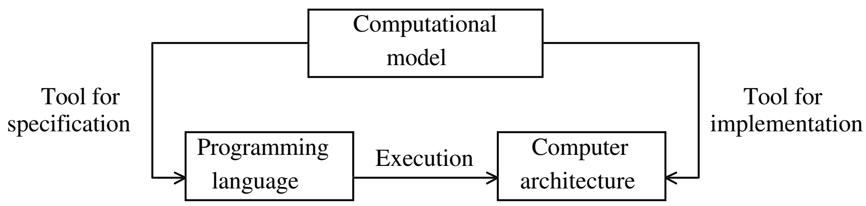


Figure 2-10: Interrelationships between the notions computational model, programming language and computer architecture

In order to illustrate these relationships let us choose the *von Neumann computational model* and examine the basic features of those programming languages and computer architectures which correspond to this model.

A programming language corresponding to the von Neumann model should allow to declare variables and to manipulate (alter) their values by means of a proper set of instructions during computation as many times as required. Furthermore, the language should provide control instructions to allow the explicit control of the execution sequence. Languages fulfilling these requirements are called **imperative languages**. The most widely used languages such as C, Pascal, FORTRAN, etc., are all imperative languages.

It can easily be shown that **von Neumann architectures** suit the above mentioned computational model. As far as the variables are concerned, they are implemented as addressable locations in the memory, or registers, and, therefore, their values can be altered as many times as required. Sequence control is implemented by storing the instructions in the memory and declaring a pointer, called Program Counter, which indicates the particular instruction which is to be executed next.

Finally, it should be pointed out that new computational models, and the corresponding programming languages as well as the architectures usually evolve in the following sequence: computational model, programming language and computer architecture (see Figure 2-11).

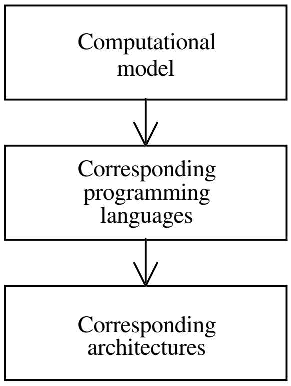


Figure 2-11: Typical sequence of the evolution of a computational model and the corresponding languages and architectures

Summing up, we can state that a number of basic computational models can be declared each covering a pair of corresponding language and architecture classes. In the next section the basic computational models will be overviewed along with the corresponding language and architecture classes.

2.1.4 Basic computational models

In the following sections 2.2-2.7 six basic computational models will be described informally using the common framework introduced in the foregoing section.

The models discussed are:

- Turing,
- von Neumann,
- dataflow,
- applicative,
- object based and
- predicate-logic based model.

These models are called *basic models* in the sense that they may be declared by using a minimal set of abstractions.

For each of these models a number of *submodels* may be distinguished by considering additional abstractions. If, for instance, in case of the von Neumann model, the process abstraction is introduced, new subclasses may be defined, by considering whether all processes have access to a global data-space (*shared memory subclass*) or, alternatively, the processes have local data spaces, and they can access remote data spaces by sending messages *(message-passing subclass)*

After discussing the basic models, we mention in section 2.8 two further real computational models (knowledge-based as well as the neural computational model). Finally, two abstract models will be introduced, the spatial computational model and the PRAM model of computation.

2.2 The Turing computational model

In addition to the realistic computational models, which represent the underlying concept of well known language and architecture classes, we also briefly include into our overview the abstract *Turing model* [Turing36] too for two reasons:

a) this model is of outstanding theoretical importance constituting the basis of notions like "algorithm", "computability", "Church's hypothesis", or "complexity of algorithms",

b) we intend to contribute to bridging the gap between the purely theoretical and pragmatic only approaches.

First, the key features of the Turing machine will be introduced using the formalism summarized below in Figure 2-12.

A **Turing Machine** is considered as consisting of two components; a *tape* and a *control unit*.

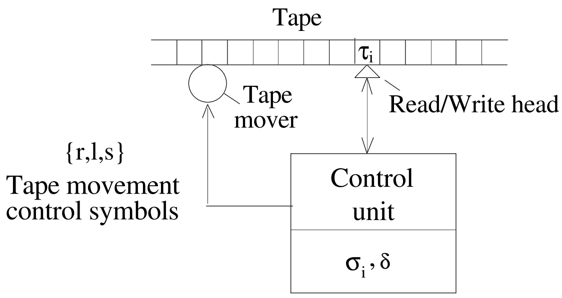


Figure 2-12: Components of a Turing Machine

The **tape**

- is regarded as an endless tape, subdivided into fields, each of which is capable of storing one *tape symbol* t_i , of the set of the tape symbols τ, $t_i \in \tau$
- can be moved by the *tape movement control symbols* {r, l, s} issued by the control unit, one field to the right {r}, to the left {l}, or it can be commanded to remain in the same position {s},
- contains the string of the input symbols ι_i when beginning the computation, where the input symbols are elements of the set of ι, $\iota_i \in \iota \subseteq \tau$,
- contains the string of the output symbols at the end of the computation.

The **control unit**

- possesses a read/write head, which enables it to read or write one symbol from/to the actual field,
- issues the tape movement control symbols {r, l, s} to the tape,
- is in one of a number of *finite states* σ_i, of the set of states σ, $\sigma_i \in \sigma$,
- operates according to a *state transition function* δ described later in this section.

Operation of the Turing machine

When the machine commences the computation it is in its *initial state* σ_0, the input data are stored on the tape as a string of the input symbols ι_i and the read/write head is assumed to be positioned at the first symbol of the input string.

The execution is performed as a sequence of *execution steps*, each of which includes *three actions*. Let us consider the j-th step, which consists of the following actions:

a) Reading in a tape symbol.

b) Determining

b_1) the next output symbol,

b_2) the next state,

b_3) the next tape control symbol.

c) Accomplishing the associated operations of

c_1) writing the symbol on the tape,

c_2) changing the state,

c_3) moving the tape.

However, this well known Turing Machine may also be considered *as a computational model*. Its key features will be summarised as follows (Figure 2-13).

The *basic items of the computation* are the elements of the symbol set τ.

The operations which may be performed on the symbols are the mappings specified by the partial transition function $\delta_\tau : \tau \times \sigma \rightarrow \tau$. Here δ_τ assigns to each pair (τ_i, σ_j) the symbol τ_k that is to be written on the tape, overwriting the symbol in the actual position.

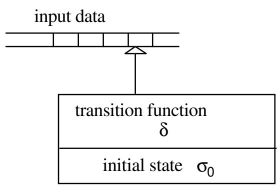
Computational model		**Turing**
Basic items of computation		· elements of the tape symbol set, · mappings defined on them
Model of problem description	**type**	procedural
	Way of description	
Assumed model of execution	**Interpretation of how to perform the computation**	starting with the initial state a sequence of subsequent states will be performed until a final state is reached
	Execution semantics	state transition semantics
	Control of the execution sequence	control driven

Figure 2-13: Key features of the Turing computational model

It should be pointed out, however, that the input symbols are usually considered being elements of one or more input words, in the same way as the string of the output symbols is regarded as constituting one or more output words.

The *problem to be solved* is procedurally described as follows (see Figure 2-13). The input data are represented by the string of the input symbols ι, the program is expressed by the transition function δ and the output data are given by the string of the output symbols (symbols which are on the tape after the Turing machine has stopped).

When the computation commences, the control unit is in its initial state σ_0, and when it stops, the control unit is at the halt state σ_f.

According to the aforesaid the *execution semantics* may be considered as being basically a state transition semantics.

In this model the *execution sequence* i.e. the sequence of the internal states σ^i is determined by the "control part" $\delta_s : \tau \times \sigma \rightarrow s$ of the transition function δ. Thus, the execution sequence may be called control driven.

Because of the operational limitations of the Turing model arising from a lack of an operative memory and from the complexity of its programming, there have been no programming languages or architectures developed which directly implements this theoretical model.

However *type 0 grammars* are equivalent to Turing machines, and thus, *type 0 languages* generated by type 0 grammars, would be the corresponding language class. This correspondence is only of theoretical significance, since there are no practical implementations at all.

2.3 The von Neumann computational model

We designate the underlying model of von Neumann computers and the corresponding languages as **von Neumann computational model.** The key features of this oldest and most widely used data-based model may be summarised as follows (see Figure 2-14):

Computational model		**von Neumann**
Basic items of computation		· data assigned to named date entities (variables) · operations performed on data, where multiple assignments are allowed
Model of problem description	**Type**	procedural
	Way of description	input data ⇓ sequence of instructions ⇓ output data
Assumed model of execution	**Interpretation of performing the computation**	with the input data the given sequence of instructions will be executed
	Execution semantics	state transition semantics
	Control of the execution sequence	control driven

Figure 2-14: Key features of the von Neumann computational model

The *basic items of the computation* are data, where computation means operations performed on data.

The *data items* are identified by names in order to distinguish between different data items used in the same computation. These **named data entities** are called *variables* in programming languages. On the other hand, in architectures, during a computation they are associated with selected *memory or register locations* whose addresses correspond to the names of the variables. The mapping of the memory or register addresses to the variable names is usually performed by the compiler. In the following, instead of the term named data entity we will use the term variable, for ease of understanding, neglecting the fact, that in architectures named data entities are implemented by distinguished memory or register locations rather than by variables.

The peculiarity of the named data entities is that they behave as *data containers*, i.e. during a computation they can be assigned new values as many times as required. They can keep the value which they already have until a new value is assigned to them. This behaviour will be expressed by saying that **multiple assignments** of data to the variables are allowed, or briefly, that multiple data assignments are allowed.

The *problem description* is procedural, whereas the computational task is specified as a sequence of instructions.

Performing the computation means simply that the given sequence of instructions is to be be executed.

The *execution* of the instructions follows a state transition semantics, i.e. as far as the execution is concerned the *von Neumann model* behaves like a *finite state machine (FSM).* In order to demonstrate this, let us first introduce the notion of *dual view of variables.* Then, based on this concept the data-, control-, status- and global-state spaces can be interpreted as well as the execution behaviour of the von Neumann model can be discussed.

2.3.1 Basic characteristics of the von Neumann model

The main attributes of the model follow from the possibility of multiple assignments of data as well as from the control driven execution.

a.) **Consequences of the multiple assignments of data**

A consequence of the possibility of multiple assignments is, that the values of the variables may change during the execution of each data manipulation instruction. This means that the actual values of the variables are **history sensitive**, i.e. they are dependent on the actual sequence of the executed instructions.

Obvious examples for history sensitivity are loop variables. Another way of expressing the history sensitivity of variables is to say that the variables **are not referentially transparent**. This expression stresses the fact that the actual values of the variables are not independent of actual point of the execution.

There exists a special case of history sensitivity called **side effects**. In the context of the von Neumann model of computation, it is usually interpreted as an unwanted, marginal modification of the state, e.g. of global variables. As an example, consider the case when a procedure invocation modifies the value of a variable which is out of the scope of the programmer.

An illustrative example of side effects is given by Field & Harrison [FieldHarr88]; which is shown below. Let us consider the following short program using Pascal syntax (Listing 2-1):

```
program example(output)
var  flag : boolean;
function f( n : integer ):integer;
begin
if flag
     then f:=n
     else f:=2*n;
flag:=not flag;
end;
begin
     flag := true;
     writeln( f( 1 ) + f( 2 ) );
     writeln( f( 2 ) + f( 1 ) );
end.
```

Listing 2-1: A program exposing side effects [FieldHarr88]

The execution of this program results in the print out:

f(1) + f(2) = 5

f(2) + f(1) = 4

In everyday mathematical reasoning we would expect that

f(1) + f(2) = f(2) + f(1)

because of the commutativity of the addition. However, the marginal operation

flag := **not** flag;

causes the side effect as indicated in Listing 2-1.

History sensitivity and, especially, side effects put a heavy burden on the programmer, particularly during program testing.

Application of the state transition semantics presumes the existence of **state**. It means that the variables have a *dynamic* nature, they can take on new values as often as required which they preserve until new values are assigned. The execution of each instruction results in taking on a new state, i.e. one or more of the variables are given a new value and/or the control state is changed. The state transition semantics also presumes the possibility of multiple assignment of data to the variables (variables, placeholders). To put it in another way, state transition semantics and multiple assignments are interchangeable notions.

b.) **Consequences of the control-driven execution**

Because of the fact that the computational task is specified as an ordered sequence of instructions and the execution sequence of these instructions is control-driven, this model of computation is basically a *sequential* one. Such an essentially sequential nature of problem description forces a serialisation also in those cases also where the problem is inherently parallel, like in the example:

$$z = (a + b) * (c + d) .$$

In this case the terms (a+b) resp. (c+d) could obviously be calculated in parallel. Nevertheless, the computational model enforces a serialisation which could be expressed something like:

$$z_1 = a + b \qquad z_2 = c + d \qquad z = z_1 * z_2$$

This imposed serialisation often covers up potential parallelism inherent in the problem and gives rise to serial and, therefore, time-consuming execution. The serial way of execution is often intolerable, in particular for problems with massively parallel nature such as numerical problems using vector or matrix calculus.

2.3.2 Corresponding languages and architectures

Languages implementing the von Neumann model of computation should allow the declaration of variables with multiple assignments of data values, offer an adequate set of operations on them, as well as provide a proper set of control statements to implement the control-driven mode of execution. These languages are often called **imperative languages** emphasising the possibility of multiple assignments, or sometimes **procedural languages**, pinpointing their procedural nature of problem description. Because of the fact, that the procedural style of problem description is used in more than one computational model, in the following, we would prefer using the term *imperative languages* to denote the language class corresponding to the von Neumann computational model. Examples of imperative languages are C, Pascal, ALGOL, FORTRAN, etc.

Architectures with the underlining von Neumann model of computation are called **von Neumann architectures**. They should provide a memory with addressable and rewritable locations to implement the variables with multiple assignment and preserve their states. Furthermore they should maintain a pointer, called Program Counter, which points to the instruction to be executed to implement the control driven mode of execution. Lastly they should contain adequate data and control manipulation facilities (processing unit, control unit) to implement the state transition semantics.

Neumann architectures are, for the time being, the most widely used class of architectures because of their relative simplicity and effectiveness for general type of computational tasks. Examples are all the well known architecture families like the IBM 360/370 series, DEC PDP/11 and VAX families, or all the widely used microprocessors such as the Intel 80x86 and Pentium processors, Motorola 68OOO and 88000 families, the MIPS R series, the Sparc processors, members of the PowerPC family and others.

Imperative languages as well as von Neumann architectures inherit the fundamental *drawbacks and limitations* of their underlying model of computation, namely

- history sensitivity (s*ide* effects, lack of referential transparency)
- basically sequential nature of execution (word-at-a-time execution).

However, this model has, from the architectural point of view, a significant *advantage,* namely; it can easily be implemented. This seems to be the point responsible for the clear dominance of the von Neumann model of computation at present, in spite of fierce criticism and a huge amount of research and development work carried out on alternative models.

2.3.3 Extensions of the von Neumann computational model

The basic von Neumann model may be enhanced along many directions.

By extending the set of the abstractions already taken into consideration, subclasses, sub-subclasses etc. may be defined.

E.g. the fundamental constraint of the sequentiality can be relaxed by introducing a new abstraction of *parallelly executable pieces of computation*, usually called *processes* or *threads*, which provide the necessary mechanisms for synchronisation, communication and data sharing.

The **communication mechanism** offers means for the transfer of data (parameter) among executable units. The most widely used mechanisms are (see [AndrSchne83]):

- unprotected shared (global) variables,
- shared variables protected by modules or monitors,
- message passing and,
- rendezvous.

The **synchronisation mechanism** allows us to impose restrictions upon the execution sequence of the executable units.

Commonly used methods are (again according to [AndrSchne83]):

- semaphores,
- signals,
- events,
- queues,
- barrier synchronisation etc.

A **data sharing mechanism** is needed to declare whether parallel executable units (processes) are allowed to access a common data space (**shared variable concept**), or whether distinct processes have access only to their local data spaces. Also a communication mechanism should be provided for accessing data pertaining to other processes (often termed as **message passing concept**).

Thus, *imperative languages* may be subdivided into two subclasses according to whether they provide means for parallel execution or not (see Figure 2-15).

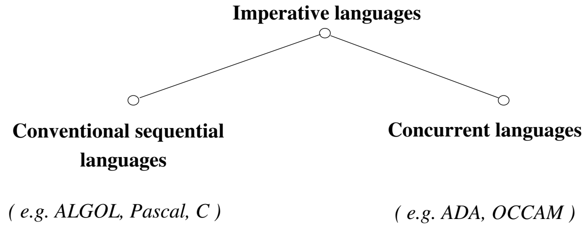


Figure 2-15: Subdivision of imperative languages according to their support of parallel execution

Concurrent imperative languages may be classified further in accordance with their basic features. For example, as far as the *data sharing mechanism* is concerned, the most important subclasses of concurrent languages are as follows (see Figure 2-16):

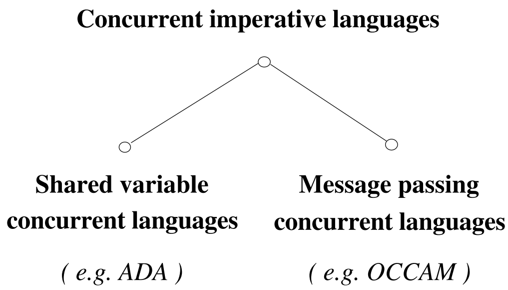


Figure 2-16: Subdivision of concurrent imperative languages according to their data sharing mechanism

2.4 The dataflow model of computation

The ultimate goal of this computational model is to speed up computation utilising available parallelism on the level of elementary operations inherent in most problems.

In the dataflow model, which has its origins in the late 60s ([KarpMill66], [Adams68]), this is achieved by providing a means for the specification of parallel executable tasks in the form of a dataflow graph (**DFG**) and by making use of a data-driven execution control as well as of the dataflow execution semantics, explained later in this section. The term dataflow is considered as having been conceived in 1968 [Adams68].

The key features of this model may be summarised as given in Figure 2-17.

The *basic items of computation* are data and the operations performed on them. Data are named, i.e. they are identifiable entities. However, in contrast to the von Neumann model of computation, a value may be assigned to a named data entity only once, and its value is retained merely until the related operation is performed. This will be usually expressed by saying that only **single data assignments** are permitted.

The *computational task* is stated procedurally in the form of a dataflow graph or in an equivalent way (for instance in a dataflow language).

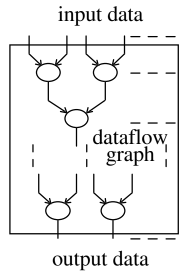
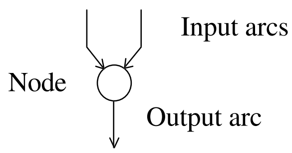
<table>
<tr><th colspan="2">Computational model</th><th>Dataflow</th></tr>
<tr><td colspan="2">Basic items of computation</td><td>· data assigned to named entities
· operations performed on data</td></tr>
<tr><td rowspan="2">Model of problem description</td><td>Type</td><td>procedural</td></tr>
<tr><td>Way of description</td><td>
</td></tr>
<tr><td rowspan="3">Model of execution</td><td>Interpretation of how to perform the computation</td><td>the input data will flow through the dataflow graph and as a consequence output data will be produced</td></tr>
<tr><td>Execution semantics</td><td>dataflow semantics</td></tr>
<tr><td>Control of the execution sequence</td><td>data driven
(eager evaluation)</td></tr>
</table>

Figure 2-17: Key features of the dataflow computational model

Dataflow graphs are a possible means for expressing parallel executable operations, already in the problem description phase. A **dataflow graph** is a directed graph, which consists of nodes and arcs.



Nodes (sometimes called **actors**) represent the *operations* to be performed. On the other hand, the directed **arcs** specify the *input and output data* of the nodes.

For example, the computational task

$$z = (a + b) * (c - d)$$

may be represented by the dataflow graph shown in Figure 2-18 as follows:

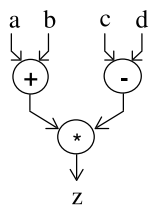
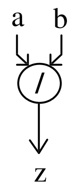


Figure 2-18: The dataflow graph representation of
*z = (a+b) * (c-d)*

For *non-commutative* operators (like -, /), however, some provisions are needed to avoid possible ambiguities in the dataflow graph representation. This can easily be achieved by requesting that for non-commutative operations the order of the input arcs must remain the same as the order of the operands. For example, the non-commutative operation

$$z = a / b$$

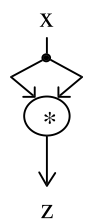
will be represented by:



Further provisions are required for copying the operands or results. For example, in case of

$$z = x * x$$

x is to be copied. The simplest solution to achieve this is to let the arc branch, whereby the branch symbolises the copying. Thus, the above expression can be represented by:



Using these extensions of the basic representation rules, the data flow graphs for simple problems may be drawn.

Further extensions are needed to specify conditional operations, cycles, function calls, etc., which are not discussed here. For details see [Gurd87].

The specified task will be performed by executing the operations indicated by the DFG using the given input data (Figure 2-19).

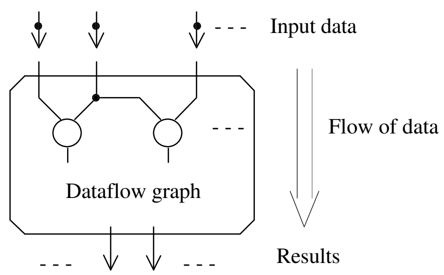
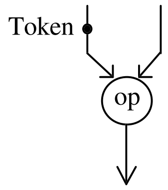


Figure 2-19: Performing the computation in the dataflow computational model

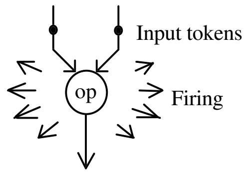
The *execution* is data-driven, i.e. an operation will be scheduled for execution when all necessary input data are available.

The *semantics* of the operations may be called *dataflow semantics*, which will be interpreted in the context of data flow graphs and data-driven execution as follows.

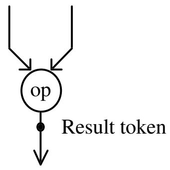
Let us indicate the presence of a data item on an arc by saying that a **data token** is present and illustrate it by drawing a dot on the arc:



The condition for activating an operation is that all data tokens needed, should be available. In data flow terms the activation is often called **firing** and the condition for firing an actor is that all required tokens are present:



After firing, the result will be generated and the input tokens will be "consumed".



Thus, the input values vanish, no intermediate result will be preserved. Therefore, the execution has a *single-assignment* character.

From the user's point of view, the **essential characteristics** of the dataflow model of computation are:

a) It is an inherently parallel model of execution,

b) Variables play quite a different role from that in the von Neumann model of computation, hence they are used only for notational convenience. This means that values may be assigned to the variables only once and they preserve their values only until the operation is performed (since in the course of execution the tokens will be consumed). This fact is expressed saying that only **single assignments** of values to variables are allowed. As a consequence, there does not exists the notion of state (as the set of actual values of the data variables), and, therefore, there is no history sensitivity, neither can side-effects occur.

c) On the other hand, in the absence of multiple assignments and states the implementation of this model is demanding. The reason for this lies, first of all, in the high communication and synchronisation needs for implementing this model of computation.

In order to asses the difficulties of the implementation, let us discuss the execution of a simple computational task expressed in form of a DFG (see Figure 2-20). For sake of simplicity, we restrict ourselves to two operand operations. Permitting one, or more than two operands would not affect the conclusion which we shall reach.

The most straightforward way to perform a prescribed computation in the "*dataflow style*" would obviously be by using for each node "dataflow style" functional units and interconnecting them according to the given DFG (Figure 2-21). In this case the functional units would be interconnected to perform the computation in much the same way as for analogue computers.

The term "*dataflow style*" is used here to emphasize the ability of the functional units to monitor their inputs (for instance, by making use of data ready signals) and to operate along the lines of dataflow semantics. Unfortunately, this simple way of implementation is not practicable for a programmable processor. As a consequence, it is not possible to detect directly at the node level whether the corresponding input data are available or not. Instead, the following principle can be utilised (see Figure 2-22).

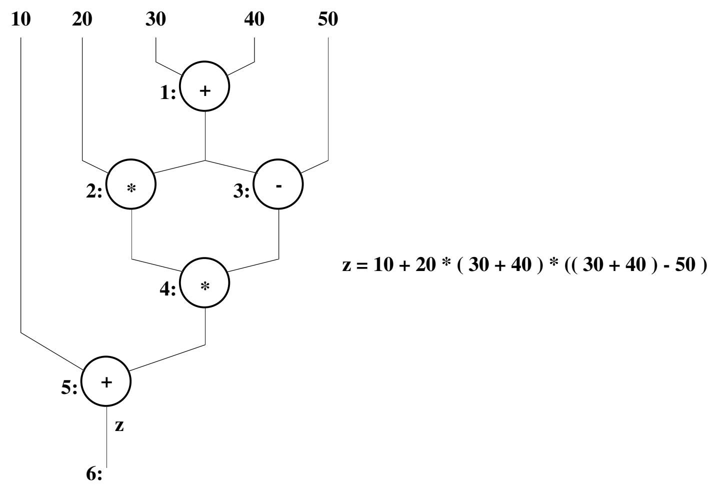


Figure 2-20: DFG of a straightforward computational task with the given input data

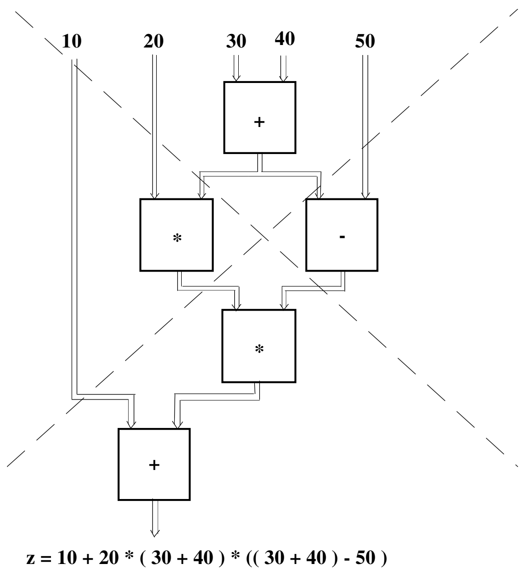


Figure 2-21: The simplest way to execute the specified DFG of Figure 2-20 using dataflow style functional units

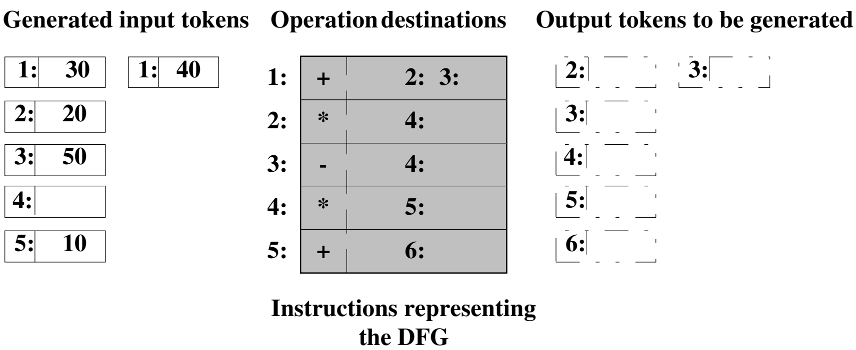


Figure 2-22: Mapping DFG and input data into a dataflow program together with a set of generated input tokens

Let us first describe the DFG in the form of a *dataflow program*, so that each node will be represented by a dataflow instruction. In addition, let us assign to each instruction (i.e. to each node) *two input tokens* and as many *output tokens* as there are nodes connected to the node under consideration. The presence of input tokens indicates the availability of input data. The execution of each instruction results in the generation of one or more output tokens, depending on how many instructions are supplied with the result. A token contains the data value and carries the same identifier as the associated node, that is the same as of the corresponding program line (Figure 2-23).

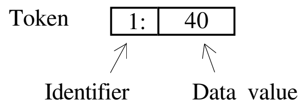


Figure 2-23: Basic components of a token

In this case the dataflow semantics will be implemented by using a *centralised token matching unit* (Figure 2-24).

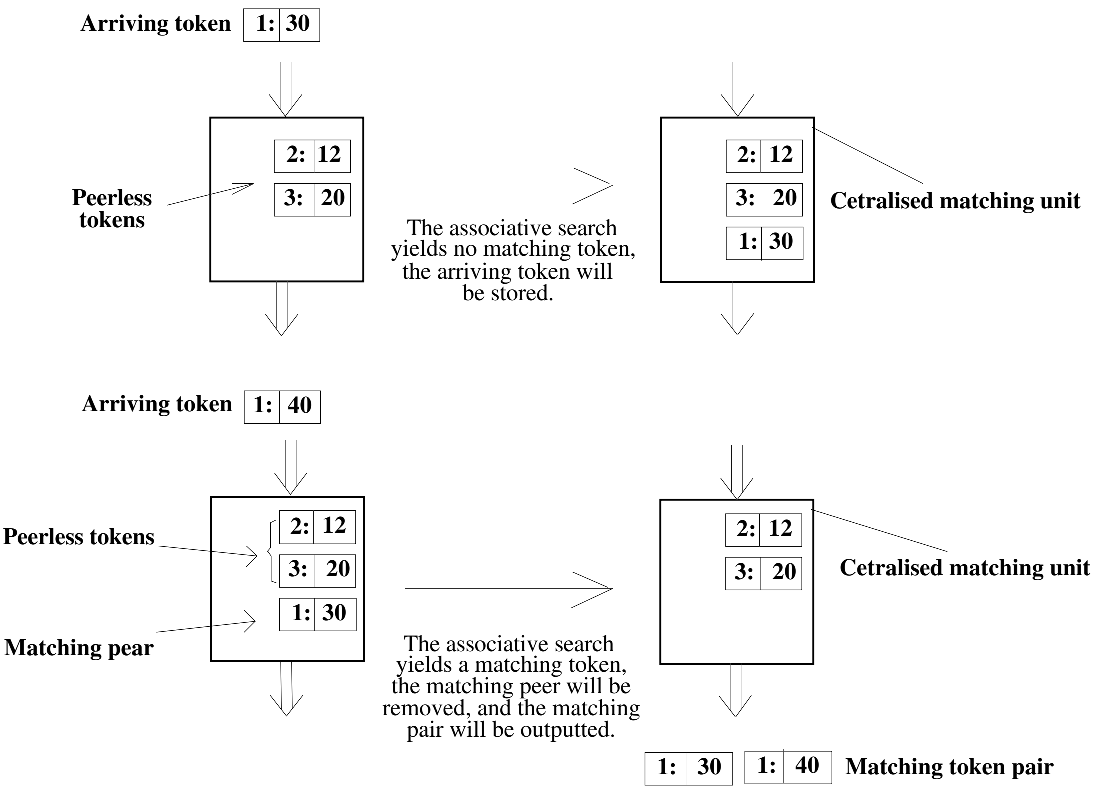


Figure 2-24: Operation principle of a centralised matching unit

The matching unit operates like an associative memory as follows: First, the generated input tokens and subsequently all generated output tokens are directed to the matching unit. The first token of a corresponding couple, called peerless token, will always be stored temporarily here. If, then, the missing member of a token pair arrives, the matching peerless token will be read and removed form the matching unit.

A typical dataflow architecture is constructed basically from a centralised matching unit, an instruction memory and a number of functional units (see Figure 2-25).

When the matching unit outputs a matching token pair, the corresponding instruction will be fetched from the instruction memory, and an execution package will be assembled which contains the operation code and the required operands. This package is then sent to one of the available functional units. The execution leads to the generation of one or more output tokens each carrying the resulting data as well as the identifier of the instruction which will use this result as an input. The generated output token(s) are passed on to the matching unit.

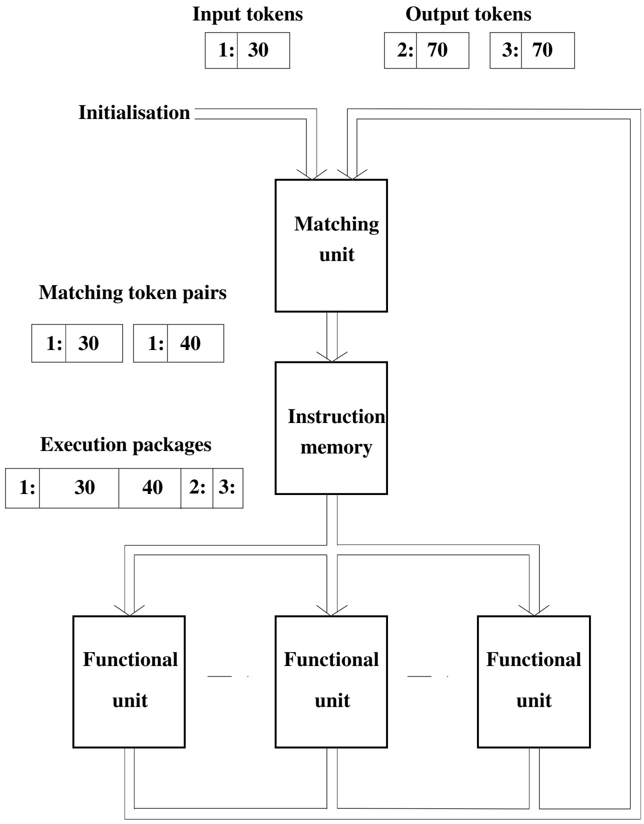


Figure 2-25: Basic structure and operation of a dataflow architecture

However, the real problem is, *how to find the companion* token. There are two possible principles to be used, an associative search and a direct match (see Figure 2-26).

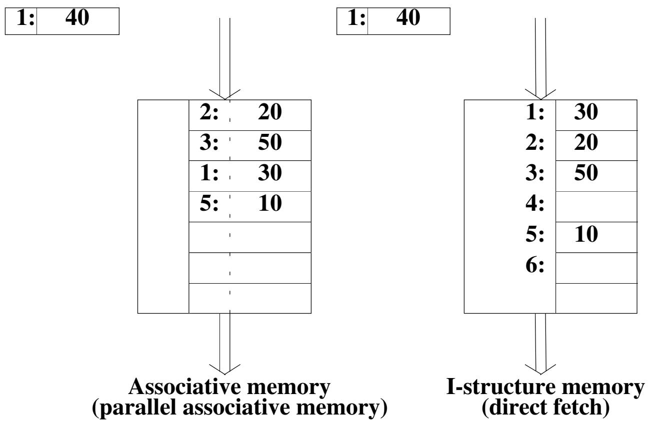


Figure 2-26: Basic alternatives of centralized matching unit

If the peerless tokens are stored *without ordering*, then an *associative search* is to be performed. This would result in a rather demanding implementation concerning the fact that real systems may require to store tokens of the order of 10^5 to 10^6 [Gurd87]. In fact, a direct implementation using an associative memory of such a large capacity is presently not feasible. Practical solutions approximate associative searching by parallel hashing (see again [Gurd87]). The alternative solution is to store the peerless tokens using an *ordering*, for instance to locations that correspond to their identifiers (termed as *explicit token store*, introduced by [CullPapa90]. Then, an I-structure memory (incomplete memory) can be used, where an access to an empty location results in writing (storing) and an access to a written location causes a 'read' and 'remove'.

In any case, the difficulty is that a centralised token matching is either too demanding, if a high enough capacity and throughput is required, or it is too slow to fit the requirements of a highly parallel computation, if an acceptable implementation cost is presumed. Because of this shortcoming all general purpose dataflow computers built until now, are of experimental or prototype character. Thus, the lack of a suitable implementation of the dataflow semantics restricts the pure dataflow model of computation to be exploited on a large scale. As a further consequence, the focus of the related research and development work has recently shifted more in the direction of reduction and hybrid architectures (see section 2.5 and 2.10).

Language		Developed at	Reference
LAU	(Langue de l'assignation unique)	ONERA-CERT Toulouse	Syre & al. 1977
Id	(Irvine Dataflow)	U. of Calif. Irvine	Arvind & al. 1978
VAL	(A Value Oriented Algorithmic Language)	MIT	Ackerman, Dennis 1979
SISAL	(Streams and Iteration in a Single-Assignment Language)	LLNL	McGrow & al. 1983

LLNL : Lawrence Livermore National Laboratory

Table 2-2: Best known dataflow languages

Languages supporting this model are called **dataflow**- or **single-assignment languages** for obvious reasons. The term single assignment language was first used by Tesler and Enea [TeslerEnea68]. The best known dataflow languages are summarised in Table 2-2. A comprehensive discussion of the origins of dataflow architectures may be found in [Gurd87].

2.5 The applicative model of computation

Like the dataflow model, this one also aims at parallel execution at the level of elementary operations. Its key features are summarised in Figure 2-27.

In this model, *computation* means the application of functions to arguments, which are typically data or functions (Figure 2-28).

An outstanding feature of this model of computation is that it is *declarative*, the same as the predicate-logic based computational model; in contrast with the procedural models such as the von Neumann or the dataflow model.

While in a procedural model a given problem is solved by writing an appropriate procedure, in a declarative model all the facts and relations, which are relevant for solving a given problem are declared. Furthermore, an execution mechanism for providing the result, is assumed.

In the applicative model of computation the function 'f' is a pure mathematical function which may be either a first-order or a higher-order function.

First-order functions are functions which accept data values as arguments and produce data values as results (see Figure 2-29).

Higher-order functions, however, can accept functions as arguments and return functions as the result (see Figure 2-29).

Computational model		**Applicative**
Basic items of computation		· arguments · application of functions to their arguments
Model of problem description	**Type**	declarative
	Way of description	arguments of the function to be evaluated ⇓ set of function definitions involved ⇓ evaluated function
Assumed model of execution	**Interpretation of how to perform the computation**	the function with the given arguments will be subsequently rewritten by substituting the involved function definitions and performing the possible reductions until no more subtitutions or reductions are feasible
	Execution semantics	reduction semantics
	Control of the execution sequence	demand driven (lazy evaluation)

Figure 2-27: Key features of the applicative model of computation

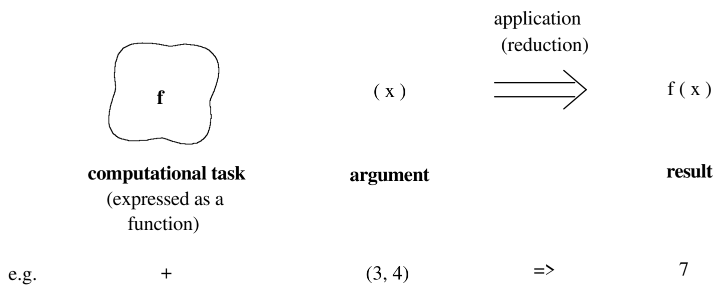


Figure 2-28: Basic interpretation of the applicative model of computation

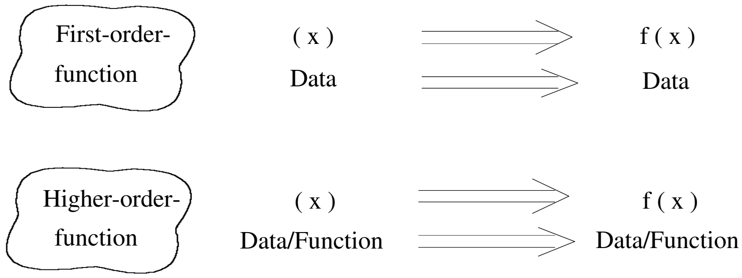


Figure 2-29: The notion of first and higher-order functions

In the general case, the function 'f ' that specifies the computational task is given as a rather complex definition involving a set of composite functions f_i, (i = 1, ..., n):

$$
\begin{aligned}
&f(x) = f(f_1, \dots, f_i, \dots, f_n, x)\\
&f_1(x_1) = \dots\\
&.\\
&.\\
&f_i(x_i) = \dots\\
&.\\
&.\\
&f_n(x_n) = \dots
\end{aligned}
\qquad (2.5.1)
$$

with additional function definitions, say $f_{n+1} \dots f_{n+k}$, relating to functions occurring on the right sides of the definitions:

$$
\begin{aligned}
&f_{n+1}(x_{n+1}) = \dots\\
&f_{n+k}(x_{n+k}) = \dots\\
&.\\
&.
\end{aligned}
\qquad (2.5.2)
$$

Furthermore, each function definition $f_i(x_i)$ may be given as a set of function definitions differing in their *parameter patterns*. Thus, for example, $f_i(x_i)$ may be defined as

$$
\begin{aligned}
&f_i(x_{i1}) = \dots\\
&f_i(x_{i1}, x_{i2}) = \dots\\
&.\\
&.
\end{aligned}
\qquad (2.5.3)
$$

An illustrative example of a complete function definition, using, for simplicity, only first-order functions, is taken from [FieldHarr88]. In this example (see Listing 2-2) the function 'TreeSort' sorts a list of unsorted numbers into ascending order. This will be done by constructing, first, an ordered binary tree, using the function 'MakeTree', and then flattening (transforming) this tree into a sorted list by means of the function Flatten (see Figure 2-30).

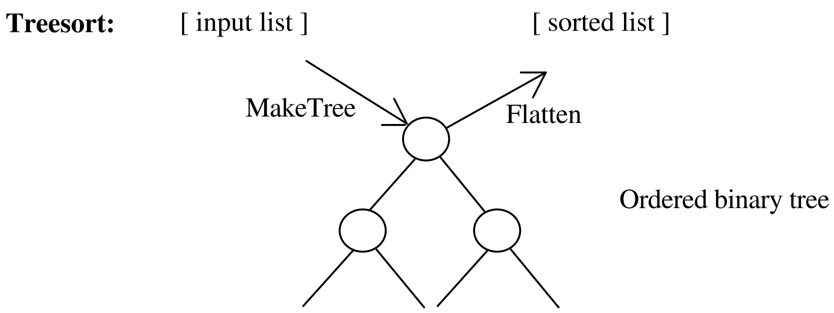


Figure 2-30: Functional principle of TreeSort

Since our aim is only to illustrate the style of function definitions used in the applicative model, we do not want to go too far into details. Instead, we list first the function TreeSort, written in the functional language HOPE, and then we explain the functions used.

```
dec TreeSort: list( num ) → list( num );
dec MakeTree: list( num ) → tree;
dec Flatten: tree → list( num );
```

```
dec Insert: num # tree → tree;
--- TreeSort( UnsortedList ) <=
   Flatten( MakeTree( UnsortedList ) );
--- MakeTree( nil )<= empty;
--- MakeTree( n::rest )<= Insert( n, MakeTree( rest ) );
--- Insert( n, empty ) <= leaf( n );
--- Insert( n, leaf(i))<=
   if n <= i
   then node( leaf(n), i,empty )
   else node( leaf(i), n, empty );
-- Insert( n, node( left, value, right ) )<=
   if n=< value
   then node( Insert( n, left ), value, right )
   else node( left, value, Insert( n, right ) );
--- Flatten( empty )<= nil;
--- Flatten( leaf( n ) ) <= [n];
--- Flatten( node( left( value, right ) ) <=
   Flatten( left )<>( value:: Flatten( right ) );
```

Listing 2-2: Program listing of TreeSort [FieldHarr88]

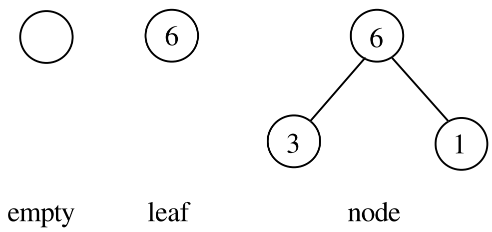
The program begins with the *declarations* of the functions used, specifying the names of the functions as well as the data types of the arguments and of the function values (results).

There are four functions declared, the main function TreeSort and the auxiliary functions MakeTree, Flatten and Insert. For example, TreeSort accepts as input a list, consisting of integer numbers, which is the unsorted list, and produces as output a list of integers, which is the sorted list.

As far as the data types are concerned, there are two data types occurring in the declarations and subsequently in the definitions, the data types list and tree. The data type *list* is predefined; whereas the data type *tree* is a user-defined data structure. A tree can be declared by:

data tree == empty++leaf(num)++node(tree#num#tree);

This means that a tree is either empty or it is a leaf containing an integer number or, it is a node containing an integer number and two subtrees:



In the declaration the hash symbol (#) denotes the Cartesian product of types.

Subsequently, the *function definitions* follow (identified by the sign ---) in the TreeSort program. Note that *argument names* are only valid *within* the same function definition. Actually they have an identical role in passing argument values like dummy parameters have in subroutine calls.

In this example, the main function TreeSort (UnsortedList) is defined using the functions 'MakeTree' as well as Flatten. 'MakeTree' takes as input argument 'UnsortedList' and creates, as output, an ordered binary tree. In an **ordered binary tree** all the values of the left subtree are less than or equal to the value of the node, concerned. Figure 2-31 shows an example of an ordered binary tree.

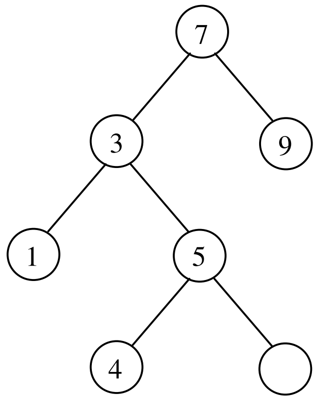


Figure 2-31: An example of an ordered binary tree

'MakeTree' is defined for two different argument patterns. First, if the argument list is *empty* (nil) it produces an empty tree.

```
--- MakeTree( nil )<= empty;
```

In the other case, when the input list is *not empty*, say, it contains a head 'n' and a tail rest, written in the form (n::rest), an auxiliary function 'Insert' is invoked. 'Insert' places a list item (head n) into the proper place of the tree built up of the rest of the elements of the list.

```
--- MakeTree( n::rest )<= Insert( n, MakeTree( rest ) );
```

From the definition of 'Insert', three cases are to be distinguished, in accordance with the three cases of the definition of the data type tree; a) when the tree is empty, b) when it has already one leaf, and c) when it is a true tree.

.

a) If the tree is *empty*, the Insert creates a leaf node:

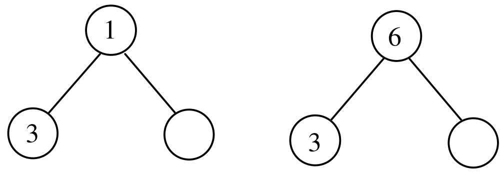
```
--- Insert(n, empty) <= leaf(n);
```

b) If the tree is already *a leaf*, say, leaf(i), then there is a tree to be built consisting of three nodes such that if n ≤ i, i becomes the root, n will be the left leaf and the right leaf is empty (see the figure below). Conversely, if n > i, n turns into the root node, i becomes the left leaf and the right one is empty (see the next figure).

```
--- Insert( n, leaf(i))<=
        if n <= i
        then node( leaf(n), i,empty )
        else node( leaf(i), n, empty );
```

For example, in the two cases when n=1 and i=3 as well as n=6 and i=3 the resulting tree will look like:

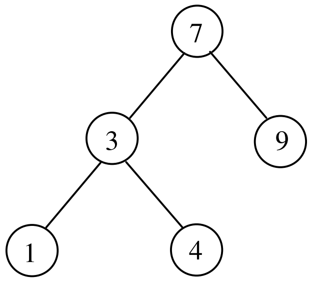
n = 1, i = 3 n = 6 , i = 3



c) Finally, if the tree, by now, consists of *at least one node with leaves* (i.e. it is already a real tree), then the new element 'n' is to be placed either in the left subtree (if n <= node value) or in the right subtree (if n > node value) repeatedly until the insertion is completed:

```
--- Insert(n, node(left, value, right) <=
       if n <= value
       then node(Insert(n, left),value, right)
       else (node(left, value, Insert(n, right));
```

For example if n = 5 and the tree at present looks like the one below:



then the application of 'Insert' to n = 5 and the tree, shown, yields:

```
Insert(5,node(node(leaf(1),3,leaf(4)),7,leaf(9)))
→node(Insert(5,node(leaf(1),3,leaf(4)),7,leaf(9)))
→node(node(leaf(1),3,Insert(5,leaf(4))),7,leaf(9))
→node(node(leaf(1),3,node(leaf(4),5,empty),7,leaf(9)))
```

Finally, the function 'Flatten' is used to convert the ordered binary tree into a sorted list. In the definition of 'Flatten', again, the same three cases are to be distinguished as for 'Insert'.

If the tree is empty, then the resulting list should be nil:

```
--- Flatten(empty)<= nil;
```

If the tree consists only of a leaf, then the resulting list will have only one element corresponding to the leaf:

```
--- Flatten(leaf(n))<=[n];
```

In the last case, when the tree is a "real" one, let us bear in mind that for an ordered tree all elements in the left subtree are less than or equal to the node value. Therefore, 'Flatten' has to put the node value into a list and append to it the values arising from the left and from the right subtrees:

```
--- Flatten(node(left, value, right))<=
       Flatten(left)<>(value:: Flatten(right));
```

where <> denotes the infix operator append.

All in all, the function 'Flatten' is defined for three different argument patterns, whereas 'MakeTree' is defined for two cases. In addition, 'MakeTree' uses the auxiliary function 'Insert', which is also given for three different argument variants.

The computational task will be stated as the application of the function 'TreeSort' to the given argument list [UnsortedList], like 'TreeSort' [5.2.3].

In the general case, the *execution* is accomplished as a series of *rewritings* where the occurring functions will be substituted into the right sides of the matching function definitions, until no further substitutions are required. Then, the numerical computations will be performed and the expressions with the results will be substituted (called reductions).

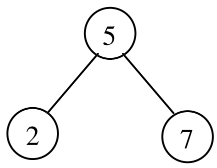
The following example illustrates how the execution will actually be performed. In the example let us sort the list [7, 5, 2] using 'TreeSort'.
Now the application of 'TreeSort' to the input list, say [7, 5, 2] results in:

```
 TreeSort[ 7, 5, 2]
→Flatten(MakeTree([ 7, 5, 2]))
```

First, the corresponding ordered binary tree is to be built applying 'MakeTree'.

```
→Flatten(Insert(7,MakeTree([5,2])))
→Flatten(Insert(7,Insert(5,MakeTree([2]))))
→Flatten(Insert(7,(Insert(5,(Insert(2,MakeTree(nil)))))
→Flatten(Insert(,7,Insert(5,Insert(2,empty))))
→Flatten(Insert(7,Insert(5,leaf(2))))
→Flatten(Insert(7,node(leaf(2),5,empty)))
→Flatten(node(leaf(2),5,Insert(7,empty)))
→Flatten(node(leaf(2),5,leaf(7)))
```

Now we obtain the tree:



Next, using Flatten we read out the sorted list from the ordered binary tree.

```
→Flatten(2)<>(5::Flatten(7))
→[2,<>(5::7)]
→[2,5,7]
```

This example shows very clearly some *important features* of the applicative computational model:

a) The computational task is given as a complex function which consists of a set of function definitions. These definitions can be seen as *rewrite rules*.

b) Each function definition is usually given as a set of distinct definitions, differing in the pattern of their parameters (arguments). Therefore, in order to select the proper version of an invoked function definition typically some form of a *pattern matching* is needed.

c) Each rewriting preserves the value which the expression represents (*referential transparency*). Referential transparency may easily be checked for the following simple example.

In this example, written again in HOPE, the factorial of a non-negative number 'n' is calculated using the function fac:

```
dec fac: num → num;
    fac( n ) <= if n = 0 then 1 else n*fac( n-1 )
```

Consider e.g. the evaluation of fac for, say n = 3 :

```
  fac( 3 )
→3 *  fac( 2 );
→3 * 2 * fac( 1 );
→3 * 2  * 1 * fac( 0 );
→6
```

Here, each rewriting obviously preserves the same value of 6.

The *execution semantics* used is the reduction semantics. It is based either on the *lambda-calculus* [Church41] or on *combinators* introduced in [Turner79]. The corresponding rules are called *beta-reduction* and *combinator-style reduction* respectively. A good overview on related issues may be found [FieldHarr88].

The *execution sequence* is inherently demand driven.

The *advantages* and *disadvantages* of this model of execution are very similar to those stated for the dataflow model of computation. It is a parallel model, there is no history sensitivity, and there are no side-effects.

At this point it should be noted, that both the applicative and the predicate logic based computational models are declarative. In the latter, clauses are used to state the problem instead of functions. How the computation is accomplished, is a further difference. Whereas in the *applicative model* the computation is performed using *reduction rules*, the *logic based model* uses some *theorem-proving concepts* such as *resolution*. As a consequence of the **Church-Rosser theorem** the applicative model is deterministic while the logic based model using search based computation like backtracking, is inherently non-deterministic.

For the applicative model of computation the matching programming language class is designated as **functional languages** - e.g. FP [Backus78], ML [Gordon79], HOPE [Burstall80, Miranda [Turner85].

The corresponding architecture class is called **reduction architectures**. Examples are Magos Tree Machine [Mago80], SKIM [Clarke80], ALICE [HarrRee87], Flagship [Townsend87], GRIP [PJones87], COBWEB [Anderson90].

2.6 Predicate logic based computational model

A new computational model was born about 1970 when the predicate logic, or more precisely the Horn clause subset of the first-order predicate logic[1] was first interpreted as a programming language [Green69], [Colmerauer72], [Sanderwall73], [Kowalski74].

From the papers mentioned, probably Kowalski's excellent publication was the most influential one to the development of the *theory of logic programming*. However, the achievements of Colmerauer and his co-workers in creating the logic programming language *Prolog* were the most seminal for the advance of the application side of this new discipline.

In order to become acquainted with the idea of logic programming, first a few basic notions such as, atom, clause and Horn clause should be introduced.

Atoms, or **predicates**, are statements expressed in a formalised way, such as

```
is_a_hungry_kid(steve),
is_some_kind_of_food(lemon),
likes(steve,beer) or
is_a_hungry_kid (X),
is_some_kind_of_food(Y),
likes(X,Y).
```

These examples are taken from the Turbo Prolog program, listed below in this section (Listing 2-3).

[1] *First-order predicate logic* has the constraint, that variables represent elements of sets, which are interpreted as individual entities, whereby no functional variables and no predicate variables are allowed.

The natural interpretations of the above atoms are:

```
"Steve is a hungry kid",
"lemon is a kind of food",
"Steve likes beer",
"X is a hungry kid" etc.
```

Syntactically, atoms consist of predicate symbols applied to constants, variables or functional constants. They have the general form of

$$P_i(t_1, \dots t_i, \dots t_k), \; k \geq 0,$$

with P_i are **k-ary predicate symbols**,
t_i are **terms** i.e.:

- *constants* (c_i),
- *variables* (X_i) or
- *functional constants* (expressions) of the form $f(t_1,\dots t_i,\dots t_l)$, where f is a function applied to the l-tuple of terms.

In the examples, listed above, is_a_hungry_kid, is_some_kind_of_food, likes etc. are predicate symbols, steve, lemon, beer etc. are constants, X,Y are variables (in Turbo Prolog, variables start with a capital, while constants start with a small letter),

Functional constants are functions applied to a set of terms. It is important to note that functions map into the same domain, and thus they are not to be confused with predicates.

To illustrate functional constants, let us assume that we want to extend the names "steve" or "jane" to full names, say "steve morgan" and "jane smith". In addition, if we want to handle both the first and second names separately, it is useful to introduce the functional constant name (in Turbo Prolog functional constants are called compound variables). Then, name(steve, morgan) means "steve morgan". The advantage of this is that using functional constants, the terms used may be structured.

Thus, instead of the atom

likes (steve, morgan, beer)

we can write

likes(name(steve, morgan), beer)

This can provide additional semantic information and allow for additional possibilities in handling. For example the goal hungry_kid_buys_food (name (steve,X), WHAT) expresses the question " WHAT food is bought by steve and what is his second name ?"

Based on the notion of atoms, we are now in a position to introduce *clauses*. A clause is a set of disjunctively connected positive or negative (negated) atoms. Formally, a clause can be expressed as

$$A_1 \vee \dots A_i \vee \dots \vee A_m \vee \neg B_1 \; \dots \; \vee \neg B_i \; \vee \dots \vee \neg B_n \tag{2.6.1}$$

where $\neg$ is the negation sign,
A_i is a positive atom and
B_i is a negative atom.

In the general case clauses may obviously contain *variables* as well. It is to be emphasized that variables are considered to be *universally quantified*. This means that all variables are interpreted locally in the clause where they occur like local variables in conventional subroutines. Thus **clauses** have the general form

$$\forall x_1 \ldots \forall x_i \ldots \forall x_s \, (A_1 \vee \ldots A_i \vee \ldots \vee A_m \vee \neg B_1 \vee \ldots \vee \neg B_i \vee \ldots \vee \neg B_n) \quad (2.6.2)$$

where $\forall$ is the *universal quantifier* indicating for all.

Obviously clauses may also be written in the form:

$$\forall x_1 \ldots \forall x_i \ldots \forall x_s \, (A_1 \vee \ldots \vee A_i \vee \ldots \vee A_m \vee \neg (B_1 \wedge \ldots \wedge B_i \wedge \ldots \wedge B_n)) \quad (2.6.3)$$

or using the implication ($A \leftarrow B = A \vee \neg B$): (2.6.4)

$$\forall x_1 \ldots \forall x_i \ldots \forall x_s \, (A_1 \vee \ldots \vee A_i \vee \ldots \vee A_n \leftarrow (B_1 \wedge \ldots \wedge B_i \wedge \ldots \wedge B_m)) \quad (2.6.5)$$

This will usually be written in the concise notation:

$$A_1, \ldots A_i, \ldots A_m \leftarrow B_1, \ldots B_i, \ldots B_n \quad (2.6.6)$$

where the comma means $\vee$, on the l.h.s., and $\wedge$, on the r.h.s., of the implication, all variables are understood as universally quantified.

Horn-clauses [Horn51] are clauses with $m \leq 1$.

Thus the general form of a **Horn clause** is:

$$A \leftarrow B_1, \ldots B_i, \ldots B_n \quad (2.6.7)$$

The interpretation of the Horn clauses as a "programming language" is based on the interpretation of the implication as a procedure declaration where A is the procedure name and the B_i-s are procedure calls constituting the procedure body [Kowalski74].

There are three types of Horn clauses:

- **Definite Horn clauses with n ≥ 1**

$$A \leftarrow B_1, \ldots B_i, \ldots B_n \quad (2.6.8)$$

They represent rules (procedures), where A is called the **head,** or conclusion, B_i are the **conditions** (calls), and $\{B_1 \ldots B_n\}$ is the **body**.

An example for a definite Horn clause (in everyday language a rule) is

```
hungry_kid_buys_food(X,Y) :-
is_a_hungry_kid(X),
is_some_kind_of_food(Y),
likes(X,Y),
is_for_sale(Y).
```

This can be interpreted as "X will buy Y if X is a hungry kid, and Y is a kind of food, and X likes Y and moreover Y is for sale".

- **Definite Horn clauses with n=0**

They have the form

$$A \quad \leftarrow \tag{2.6.9}$$

and represent facts (assertions).

For instance, is_a_hungry_kid(steve), expresses that steve is a hungry kid, or likes(steve,pizza) means that steve likes pizza.

- **Negative Horn clauses**

They are Horn clauses of the type

$$\leftarrow \quad B_1, \dots B_i, \dots B_n \tag{2.6.10}$$

These clauses are interpreted as goals (queries).

For example, hungry_kid_buys_food(steve, WHAT) is a goal (query) asking what kind of food is bought by Steve.

- **The null clause**

$$\leftarrow \tag{2.6.11}$$

represents the satisfied goal statement (halt statement).

Referring to the foregoing notation, the *predicate logic based computational* model may be outlined as follows (Figure 2-32).

<table>
<tr><th colspan="2">Computational model</th><th>Predicate logic-based</th></tr>
<tr><td colspan="2">Basic items of computation</td><td>· elements of sets
· predicates applied to them</td></tr>
<tr><td rowspan="2">Model of problem description</td><td>Type</td><td>declarative</td></tr>
<tr><td>Way of description</td><td>goal statment
⇓
set of definite Horn clauses expressing rules and facts
⇓
logical inference</td></tr>
<tr><td rowspan="3">Model of execution</td><td>Interpretation of how to perform the computation</td><td>the goal statement will be subsequently rewritten using resolution until a logical interference is found or no more unifications are feasible</td></tr>
<tr><td>Execution semantics</td><td>SLD - resolution</td></tr>
<tr><td>Control of the execution sequence</td><td>defined by both the
· goal processing rule and
· search rule</td></tr>
</table>

Figure 2-32: Key features of the predicate logic based computational model

The *basic items of the computation* are declared as elements of sets (which can be either constants, variables or functional constant) and predicates declared upon them.

The *problem description* is declarative, where a problem is described as a set of definite Horn clauses with $n \geq 1$ representing rules (relations) as well as definite Horn clauses with $n = 0$ denoting facts:

$$
\begin{array}{l}
A_1 \quad \leftarrow \quad B_{11}, \dots B_{1i}, \dots B_{1n} \\
. \\
. \\
A_i \quad \leftarrow \quad B_{1i}, \dots B_{ii} \dots B_{in} \\
. \\
. \\
A_u \quad \leftarrow \quad B_{u1}, \dots B_{ui}, \dots B_{un} \\
\\
A_{u+1} \leftarrow \\
. \\
A_{u+v} \leftarrow \; .
\end{array}
\tag{2.6.12}
$$

A simple example follows (Listing 2-3) describing rules and facts, of a particular problem, using the Turbo Prolog language :

```
predicates
hungry_kid_buys_food(symbol,symbol)
is_a_hungry_kid(symbol)
is_some_kind_of_food(symbol)
likes(symbol,symbol)
is_for_sale(symbol)
clauses
hungry_kid_buys_food(X,Y):-
    is_a_hungry_kid(X),
    is_some_kind_of_food(Y),
    likes(X,Y),
    is_for_sale(Y).
is_a_hungry_kid(steve).
is_a_hungry_kid(jane).
is_some_kind_of_food(lemon).
is_some_kind_of_food(hot_dog).
is_some_kind_of_food(pizza).
is_some_kind_of_food(beer).
likes(kelly,hot_dog).
likes(steve,pizza).
likes(judy,pizza).
likes(steve,beer).
is_for_sale(pizza).
is_for_sale(lemon).
is_for_sale(hot_dog).
is_for_sale(beer).
```

Listing 2-3: A simple demonstration program written in Turbo Prolog

In general, the *task* is stated as a goal statement:

$$\leftarrow B_1, \dots B_i, \dots B_n \tag{2.6.13}$$

In our simple example, the goal should have only one atom B_1 with

$$B_1 = \texttt{hungry\_kid\_buys\_food(steve, WHAT)}$$

This goal can be naturally interpreted as asking "What kind of food does the hungry kid Steve buy ?".

The *computation* is aimed at achieving an inference (logical conclusion). It will be performed as a sequence of rewritings of the goal statement using SLD-resolution [AptEmden82].

SLD-resolution (Linear Resolution for Definite Clauses with Selection) is a non deterministic goal rewriting rule. **Resolution** will be performed by repeating the following steps:

a.) *unification* of an atom in the goal, say B_i with the head of a definite clause say A_j, which results in the most general unifier Θ, and

b.) *rewriting of* the goal by substituting the body of the definite clause (B_{j1},...B_{ji},...B_{jn}) instead of B_i and applying Θ, which yields a new goal statement [Lloyd84],

this is carried out in a prescribed sequence until at least one solution (conclusion) is found or no further unification is possible.

Let us demonstrate one resolution step in our example. Here, the question is: what does the hungry kid Steve buy? The associated goal

```
hungry_kid_buys_food(steve, WHAT)
```
(2.6.14)

consists of merely one atom.

If this atom is unified with the head of the definite clause,

```
hungry_kid_buys_food(X,Y):-
        is_a_hungry_kid(X),
        is_some_kind_of_food(Y),
        likes(X,Y),
        is_for_sale(Y).
```
(2.6.15)

then, the arguments are matched against each other. As a result of the unification we get the most general unifier Θ :

$$\Theta = \{ X/\text{steve}, Y \} \qquad (2.6.16)$$

i.e. X will be bound to steve.

(For a full explanation of the concept of unification, see e.g. [Lloyd84]).

Subsequently, the goal will be rewritten by substituting the body of the definite clause (2.6.15) and applying steve to X. The result of this unification step is the new goal:

$$\leftarrow B_{j1}, B_{j2}, B_{j3}, B_{j4}$$

where

$$B_{j1} = \texttt{is\_a\_hungry\_kid(steve)}$$

$$B_{j2} = \texttt{is\_some\_kind\_of\_food(Y)} \qquad (2.6.17)$$

$$B_{j3} = \texttt{likes(steve, Y)}$$

$$B_{j4} = \texttt{is\_for\_sale(Y).}$$

It should be stressed that this is only one single step in unification and rewriting.

The *execution sequence* is controlled by a so called **computational strategy** specifying

- a goal selection rule and
- a search rule.

The **goal selection rule** states which atom in the goal should be selected first (for example, either the left most or the rightmost atom).

The **search rule** specifies the exploration sequence of the atoms in the body of the definite clauses and states how to handle non determinism. Rules which are used are the *backtracking* based *depth-first rule* and the *breadth-first rule* [Levi86].

In Prolog choosing the leftmost atom and the depth-first rule are used as the goal selection rule and the search rule respectively.

Backtracking is an important technique applied in Prolog programs. While working, Prolog tries to substitute a predicate in the body of a rule for the body of another rule, whose head unifies with the predicate concerned (see Listing 2-4 as an example). When this cannot be carried out any longer, Prolog goes back one step (this is backtracking) and tries another rule, the head of which unifies with the predicate in question. So Prolog does not leave out any possibility to prove the query.

A *depth-first approach* explores each possible path to its conclusion before another one is tried. For an illustration assume that we add another rule to the example; so the definition of 'hungry_kid_buys_food' looks like

```
⟨Rule1⟩ hungry_kid_buys_food(X,Y):-
                    is_a_hungry_kid(X),
                    some_kind_of_is_food(Y),
                    likes(X,Y),
                    is_for_sale(Y).
⟨Rule2⟩ hungry_kid_buys_food(X,Y):-
                    has_money(X),
                    wants_to_buy_food(X).
```

Let the goal be, as before, hungry_kid_buys_food(steve,X).

Then, according to the depth-first rule, ⟨Rule 2⟩ is not used until all possibilities arising from ⟨ Rule 1⟩ have been exhausted.

Now, it is instructive to see how our example Prolog program is executed (see Listing 2-4).

In the following trace list, RETURN means that the unification was successful, REDO means that it was not and backtracking is interpreted as introduced before.

The trace list is as follows:

```
CALL      hungry_kid_buys_food("steve",WHAT)
CALL      is_a_hungry_kid("steve")
RETURN    is_a_hungry_kid("steve")
CALL      is_some_kind_of_food(_)
RETURN    is_some_kind_of_food(lemon)
CALL      likes("steve","lemon")
REDO      likes("steve","lemon")
REDO      likes("steve","lemon")
REDO      likes("steve","lemon")
REDO      is_some_kind_of_food(_)
RETURN    *is_some_kind_of_food(hot_dog)
CALL      likes("steve","hot_dog")
REDO      likes("steve","hot_dog")
REDO      likes("steve","hot_dog")
REDO      likes("steve","hot_dog")
REDO      is_some_kind_of_food(_)
RETURN    *is_some_kind_of_food("pizza")
```

```
   CALL      likes("steve","pizza")
   REDO      likes("steve","pizza")
   RETURN    likes("steve","pizza")
   CALL      is_for_sale("pizza")
   RETURN    is_for_sale("pizza")
   RETURN    hungry_kid_buys_food("pizza",WHAT)
WHAT=pizza
   REDO      is_some_kind_of_food(_)
   RETURN    is_some_kind_of_food("beer")
   CALL      likes("steve","beer")
   REDO      likes("steve","beer")
   REDO      likes("steve","beer")
   RETURN    likes("steve","beer")
   CALL      is_for_sale("beer")
   REDO      is_for_sale("beer")
   REDO      is_for_sale("beer")
   REDO      is_for_sale("beer")
   RETURN is_for_sale("beer")
   hungry_kid_buys_food("steve","beer")
WHAT=beer
```

Listing 2-4: Execution of the example Turbo Prolog program

Since the SLD resolution is a non deterministic rule, as a consequence, the predicate logic based computational model is inherently *non deterministic*.

Languages used in this model of computation are called **logic programming languages**. Their most popular representative is Prolog which was first implemented in 1972 by Colmerauer and Roussel at the University of Marseilles.

A recent logic programming language is LOP (LOgic Programming) [Kurfeß90] developed as part of the ESPRIT 415 project.

At present, there are only *system proposals* and *experimental implementations* of architectures which support or implement the predicate-logic-based computational model, such as: ZMOB [Chakra82], DADO [Taylor84], Warren's abstract Prolog machine [Warren83], SETHEO abstract machine, SAM [Kurfeß90], the abstract PARTHEO machine or the APM [Kurfeß90]. SAM and APM were developed within the framework of ESPRIT 415.

Of course there are numerous modifications, and extensions to the basic model as far as the computational strategy or supporting concurrent programming mechanism is concerned. Examples of concurrent extensions are PARLOG [ClarkGreg83] or Concurrent Prolog [Shapiro83].

2.7 The object based computational model

There is an excellent summary of the *roots* of object based computational model in [YoneToko87]. Their well-considered and concise review in quoted as follows.[1]

"...The term 'object' emerged almost independently in various fields in computer science, almost simultaneously in the early 1970s, to refer to notions that were different in their appearance, yet mutually related. All of these notions that were invented to manage the complexity of software systems in such a way that objects represent components of a modularly decomposed system or modular units of knowledge representation. Typical notions of objects that have emerged in various field are

[1] The quotation is supplemented by references for more appreciation of when the object based model emerged.

- computational agents that carry out their actions in response to a message, namely Hewitt's actors [Hewitt73], in parallel computational models and AI programming;
- packages of information with class/instance and super-class/sub-class hierarchies, such as objects in SIMULA [Dahl68] and Smalltalk [GoldKay76], in simulation, AI, and more general programming;
- abstract data types in programming languages [LiskZill74], [JonesLisk76], [Wulf76], [Liskov76];
- modules or units for knowledge and expertise, such as Minsky's frame [Minsky75], in knowledge representation; and
- protected resources in operating systems [Wulf74].

The common characteristics of these notions are that an *object* is a logical, or physical entity that is self-contained and provided with a unified communication protocol. ..."

First the basic terms used in object-based computational model are introduced.

Objects are self-contained entities with an internal and external view (see Figure 2-33).

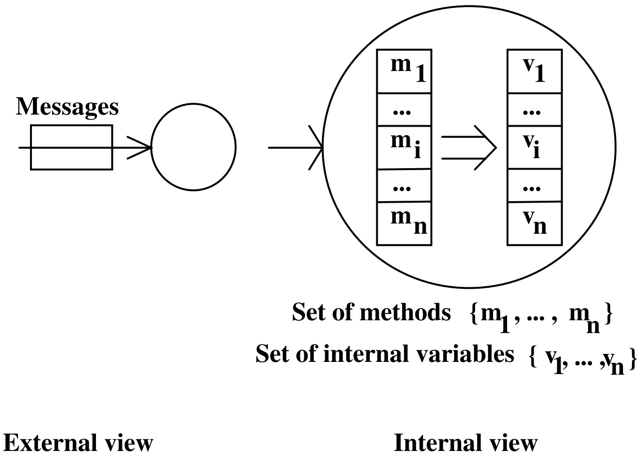


Figure 2-33: The external and internal view of objects

The **external view** of an object is that it is an entity which accepts a *set of messages* and *responds to* them in a specified way. Sending messages is the only way to manipulate objects. Therefore, objects may be viewed as encapsulated entities, whose internal details are not known and are not exposed to the sender of a message. Messages are similar to procedure calls used in conventional programming.

The set of messages, which can be sent to an object along with their syntax and semantics, is called the **protocol** or **unified communication protocol** of the object.

Consider the following example of an object representing a dog, called Benji, which accepts two messages: [1]

```
Object Benji
```

Messages:

```
name: aString    "This message requires a string as parameter and
                 aims at renaming Benji to the given string
                 aString"
```

[1] The example shown throughout in this section is based on an example program listed in Digitalk (1988).

```
bark            "This message results in a screen-message on the
                transcript window saying 'Dog Benji: Wov, Wov".
```

It should be stressed that the specification and the implementation of messages is separated. The *specification* of a message is *public* to its user. On the other hand, its *implementation,* in form of a method, is *hidden* from him. This is **in** sharp contrast to the procedure abstraction of conventional programming where the procedure specification is public.

Obviously, the separation of specification and implementation contributes to easier software maintenance and to enhanced reusability. Thus, the implementation of a message may be changed without notice assuming the specification remains unaltered.

The **internal view** of an object is, that it has

- a set of internal variables and
- a set of methods.

The set of **internal variables** is used by the object to maintain an object state.
The **methods** will be invoked by the corresponding messages. They are similar to procedure specifications in that, they are descriptions of actions to be performed when the corresponding message is received.

In the example above, Benji has one internal variable called 'name'.

The defined methods are 'name:' and 'bark', they are implemented in Smalltalk as follows:

```
name: aString

      "Change the dog's name to aString".

      name := aString!

bark

      "Have the receiver dog bark"

      Transcript nextPutAll:

      self class name, ' ', name, ': Wov, Wov ';

      cr!
```

The implementation of the method 'name:' is self-explanatory. On the other hand, the method 'bark' consists of displaying the following message parts on the transcript window (by **`Transcript nextPutAll:`**): the class name, the instance name, and the string 'Wov, Wov'. Finally, a carriage return should be executed.
In a sense, objects are *self-contained* since they embody both variables to maintain a state as well as methods to perform computations specified by the methods. This feature allows a more convenient and effective problem solving than would occur when more primitive items are used.
This self-containedness allows objects of the problem world to be mapped directly into objects of the computation world by creating objects which behave or may be manipulated much like real world objects.
As an example, consider the problem when two-dimensional graphical objects (like triangles, squares, circles, etc.) are to be manipulated. An object-based problem solution requires the creation of corresponding graphical objects, such that they can be easily shifted, rotated, enlarged, etc. by sending them the appropriate messages. Obviously, this is a much more lucid and reliable approach than using only elementary data types as in the data-based computational models and writing intricate programs using these elementary data. However, by using elementary data, an approach similar to the object based model would be to define and to make use of subroutines for the manipulation of the graphical

objects, as was pointed out previously. By comparison, object based programming enforces and supports modularisation.

In conventional languages, data items with the same behaviour i.e. with the same domain and with the same set of available operations, are considered as belonging to a specific data type. Similarly, objects with the same external behaviour i.e. with the same protocol, belong to the same specific **object class** (see Figure 2-34).

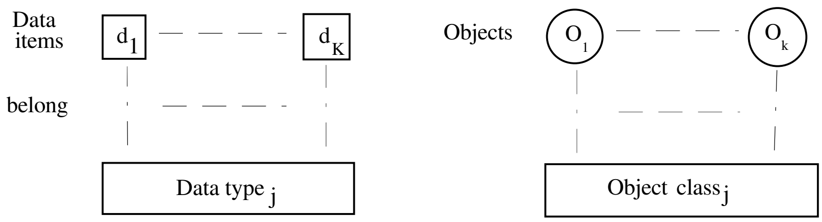


Figure 2-34: Contrasting the notions of data type and objects class

Now, we are in a position to formulate the *key features* of the object based computational model (see Figure 2-35).

Computational model		**Object-based**
Basics items of computation		· objects, which can be manipulated by a set of messages
Model of problem description	Type	procedural
	Way of description	initial states of objects ⇓ sequence of messages sent to created objects
Model of execution	Interpretation of how to perform the computation	the given sequence of messages will be executed
	Execution semantics	state transition semantics
	Control of the execution sequence	control driven

Figure 2-35: Key features of the object based computational model

The *basic items of the computation* are objects. Objects may be manipulated by sending them messages, whereby each object accepts a particular set of messages.

The *problem* will be procedurally described. First, a proper set of objects will be created. Usually this is achieved by using the set of available object classes in much the same way as available data types are used for declaring variables in conventional languages. In cases, however, when the required object classes are not available, they have to be established as for user defined data types in conventional languages.

For instance, in the above example an object class called Dog will be declared by:

```
" Dog class definition"
      Object subclass: #Dog
      instanceVariableNames: 'name '
      classVariableNames: ''
      poolDictionaries: '' !

! Dog methods !

         name: aString   "Change the receiver dog's name to
      aString."

             name := aString !

         talk   "Have the receiver dog talk"

             Transcript nextPutAll:
                 self class name, ' ', name, ': Wov, Wov ';
                 cr !
```

The most salient feature of the object based model is related to the way a problem is described. That is, it allows the description of a problem solution at a *higher abstraction level* than by using conventional data based models. This is possible, since this model makes it feasible to create objects such that they are direct mappings of the problem world objects. The created objects can then be manipulated with the same high abstraction level commands, called messages, as the real problem world objects .

The *computation* will be performed by executing the given sequence of messages.

For instance, a simple program is outlined as follows: let us create two dogs with the names Benji and Rintintin. Then, let both dogs bark, in a way as described before, by sending them the message 'bark' (see Listing 2-5).

```
 "creating two dogs with the names Benji and Rintintin"
Benji := Dog new.
Benji  name: 'Benji'.
Rintintin := Dog new.
Rintintin name: 'Rintintin'.
 "sending a message to each dog to let him talk"
Benji talk.
Rintintin talk.
```

Listing 2-5: An example program written in Smalltalk/V 286

Basically, the *execution* is accomplished according to a state transition semantics since objects preserve their state, and messages typically modify the state of the receiving object.

Execution of the example program would result in the screen-message appearing in the Transcript window of the Smalltalk user interface (Listing 2-6).

```
Dog Benji: Wov, Wov
Dog Rintintin: Wov, Wov
```

Listing 2-6: Screen-messages appearing as a result of execution of the example program

Finally, it should be emphasised that the pure object-based computational model is *sequential*. However, it can easily be extended to a concurrent one as discussed later.

The particular class of languages which allows to create and to manipulate objects, as mentioned before, is called **object-oriented**[1] or **object-based languages.** (Although some authors make a distinction between these notions, interpreting object-based languages as more strongly object-oriented languages, we shall use the notion 'object-based' to denote the entire class.)

The best known examples of *conventional* (sequential) *object-oriented languages* are SIMULA [DahlNyg66], Smalltalk with a series of implementations such as Smalltalk-72, -74, -76, -80 [GoldKay76] [Ingalls78] [GoldRob83], Smalltalk/V 286, C++ [Stroustrup86], and ACTOR [Duff90].

Architectures implementing the above mentioned ideas are designated *object-oriented* or *object-based* (like TOPSTAR).

So far, we have discussed so far the basic model. However there is a great number of possible *sub models* taking into account particular criteria and extensions of the basic model. Again, Yonezawa and Tokoro [YoneToko87] give an excellent presentation of these, well worth of quotation.

"...An object may be activated by a message arrival and become inactive after a series of actions, or it may always be activate and receive messages by executing certain commands. What kind of intra-object actions are allowed and how they are characterised, in either the imperative, functional, or logical styles? Should every action be expressed in terms of message passing? Should all the entities involved in computation be objects? More than one chain of action may or may not take place within an object simultaneously.

Message transmission may be point to point, or broadcasting may be allowed. Message passing between objects may be synchronous or asynchronous. The order of message transmissions may or may not preserved in the order of message arrivals. What kind of information may be contained in messages? What types of synchronisation mechanism is provided? Is message processing fair or may some messages be left unprocessed? Are relativistic assumption on time and state imposed or not? ..."

Based on the unified communication protocol, it is quite easy to extend the basically sequential object-based model to concurrent object-based one. A *concurrent extension* provides at least a mechanism for the concurrent execution of objects and a synchronisation mechanism. Examples of concurrent object-based languages are: Concurrent Smalltalk [YokoToko87], Act 1 [Liebermann87] and Pool [AnnotHaan90].

An example of a **concurrent object-based architecture** is DOOM which is being developed in the framework of the ESPRIT Project 415 by Philip, Eindhoven [AnnotHaan90].

2.8 A summary of the basic computational models

In order to give an overview and allow comparisons the key features of the basic computational models discussed in the sections 2.2-2.7 are condensed form in Figure 2-36. This table is based on the figures presented in the foregoing sections. However, the sequence of the basic models has been slightly rearranged and additionally, the point of concurrency has been included.

It is interesting to note that most basic models using a procedural problem description, are essentially sequential, others which use a declarative method of problem description, are parallel. It is worth noting that the indicated modes of concurrency is the inherently fitting mode of operation which is associated with a particular basic model. As already mentioned, all sequential models may be properly extended to provide concurrent execution. On the other hand, an inherently parallel model may also be implemented sequentially to reduce complexity (as, for example, in the early implementations of reduction architectures e.g. in SKIM).

In Table 2-3 the corresponding computational models, languages and architecture classes are reviewed.

[1] The term object oriented language was probably first introduced by Jones and Liskov (1976).

Computational model	Language class	Architecture class
Turing	"type 0 " languages	-
von Neumann	imperative	von Neumann
dataflow	single assignment (dataflow)	dataflow
applicative	functional	reduction
object-based	object-oriented	object-oriented
predicate login based	logic programming	till yet unnamed

Table 2-3: Corresponding computational models, languages and architecture classes

Computational model		Turing	von Neumann	Object-based	Dataflow	Applicative	Predicate logic-based
Basic items of computation		· elements of the tape symbol set, · mappings defined on them	· data assigned to named entities (variables) · operations performed on data	· objects, which can be manipulated by a set of messages · messages sent to manipulate objects	· data assigned to named entities (variables) · operations performed on data	· arguments · application of functions to their arguments	· elements of sets · predicates declared to them
Model of problem description	Type	procedural				declarative	
	Way of description	input data transition function δ initial state σ_0	input data sequence of instructions output data	initial states of objects sequence of messages sent to created objects final states of objects	input data dataflow graph output data	arguments of the function to be evaluated set of function definitions involved evaluated function	goal statment set of definite Horn clauses expressing rules and facts logical inference
Assumed model of execution	Interpretation of how to perform the computation	starting with the initial state a sequence of subsequent states will be performed until a final state is reached	with the input data the given sequence of instructions will be executed	the given sequence of messages will be executed	the input data flow through the dataflow graph and, as a consequence, output data are produced	the function with the given arguments in subsequently rewritten by substituting the involved function definitions and performing the possible reductions until no more subtitutions or reductions are feasible	the goal stattment in subsequently rewritten using resolution until a logical interference is found or no more unifications are possible
	Execution semantics	state transition semantics	state transition semantics	state transition semantics	dataflow semantics	reduction semantics	SLD - resolution
	Control of the execution sequence	control driven	control driven	control driven	data driven (eager evaluation)	demand driven (lazy evaluation)	defined by both the · goal processing rule and · search rule
Concurency		sequential			parallel		

Figure 2-36: A summary of the key features of the basic computational models