0

The Wild LIFE Handbook
(prepublication edition)

digital
PARIS RESEARCH LABORATORY

March 1994

Hassan Aït-Kaci
Bruno Dumant
Richard Meyer
Andreas Podelski
Peter Van Roy

0

The Wild LIFE Handbook
(prepublication edition)

Hassan Aït-Kaci

Bruno Dumant

Richard Meyer

Andreas Podelski

Peter Van Roy

March 1994

Abstract

This handbook provides a tutorial of the LIFE programming language as well as a complete description of the capabilities of the Wild LIFE 1.0 system. Although we have attempted to make the tutorial self-contained, it is preferable that the reader be familiar with Prolog. The tutorial exposes gradually the main components of LIFE in a synthetic approach: its original data structure—ψ-term—and its use in predicates, functions, and sort (type) definitions. Along the way, many useful examples are provided and some common pitfalls are discussed and illustrated.

Résumé

Ce manuel fournit un cours d'initiation au langage de programmation LIFE, ainsi qu'une description détaillée des prédicats et fonctions prédéfinis dans l'interprète Wild LIFE 1.0. C'est la première fois que nous tentons d'écrire une introduction pratique à la programmation en LIFE. Bien que nous fassions l'effort de n'utiliser que des notions élémentaires ou prédefinies, il est préférable que le lecteur ait déjà une connaissance de base de Prolog. Le cours expose graduellement et de façon synthétique les principales composantes de LIFE: sa structure de données originale—le ψ-terme—et son emploi dans les clauses, les fonctions et les définitions de sortes (types). Beaucoup d'exemples utiles sont donnés tout au long du manuel et quelques pièges courants sont illustrés et discutés.

Keywords

Constraint programming, logic programming, functional programming, object-oriented programming, rapid prototyping, inheritance.

Acknowledgements

The Wild LIFE project has been very much a team effort. The low-level X Window interface was implemented by Jean-Claude Hervé. The graphical interface toolkit and accumulator preprocessor were implemented by Bruno Dumant. We thank the other, past and present, members and visitors of the Paradise project at PRL and its follow-on, the Proteus project, for their suffering along with the final debugging of the design and delivery of Wild LIFE. We appreciate their active and continuing participation giving many examples, bug reports, and constructive criticisms. For the final haul on version 1.0 we especially thank Arnaud Venet.

Contents

> ... l'élément ne préexiste pas à l'ensemble, il n'est ni plus immédiat ni plus ancien, ce ne sont pas les éléments qui déterminent l'ensemble, mais l'ensemble qui détermine les éléments.
>
> GEORGES PEREC *La vie, mode d'emploi.*[1]

1 Introduction

LIFE is a programming language originally conceived by Hassan Aït-Kaci and his colleagues at MCC, in Austin, Texas [4, 5, 3]. It is a synthesis of three different programming paradigms: logic programming, functional programming and object-oriented programming.[2] LIFE is a declarative logic-based language that can be seen as a constraint language. It derives its syntax and resolution method from Prolog. Except for differences in built-ins, Prolog programs can run unaltered in LIFE if they follow the syntactic convention that each predicate and functor symbol is used with one arity only. However, the addition of functions, approximation structures (ψ-terms) and inheritance greatly enriches the language and allows one to formulate efficient programs more easily, more concisely, and—in our opinion—more naturally [7, 8].

Wild LIFE 1.0 is an interpreter for LIFE written in C. The main design goals for Wild LIFE are functionality and robustness. The interpreter implements most of the LIFE language and is robust enough to support serious program development. To increase the system's general usefulness, we have added a large number of useful tools and example programs.

As a general rule, users of Wild LIFE are encouraged to use LIFE's unique features wherever possible. They should not worry about low-level efficiency issues, but think only of using the expressive power of the language. We are using Wild LIFE to develop a LIFE compiler. We are making every effort to ensure that the compiler generates efficient code for well-written programs. Its design goals are to be competitive in speed with the best implementations of Prolog and to be usable for arbitrary large programs and data.

2 Road map

The purpose of this document is twofold: it is intended as a tutorial to the LIFE language as well as a programmer's reference to the Wild LIFE system. Table 1 gives an outline of its contents.

To aid in understanding Wild LIFE, we shall use a special "watchful eyes" sign, ◔◔, to highlight specific points where the behavior of Wild LIFE may run counter to conventions or expectations. It will act as a conspicuous marking device at the outset of a paragraph to indicate that you should watch out for the observation to follow to avert potential trouble.

The LIFE compiler will be as compatible as possible with Wild LIFE 1.0. To ensure that your programs will be portable to the compiler, they should only make use of built-ins documented in this handbook. There exist other, undocumented built-ins. We strongly recommend that you do *not* use undocumented built-ins. We do not make any guarantees that they will continue to exist in the compiler.

1 Life, a user manual.

2 Or rather, a particular view of object-oriented programming dealing essentially with inheritance.

Table 1: Road map of the Wild LIFE handbook

3 Running Wild LIFE

3.1 Getting started

To start running the interpreter at the shell level, simply invoke the command `wild_life`. The program responds with the following message of identification, and the prompt > when it is ready to accept input:

```
Wild_Life Interpreter Version 1.0
Copyright (C) 1991-93 DEC Paris Research Laboratory
No customizing file loaded.
>
```

At this point you may either make assertions, submit queries to be solved or exit by typing the key sequence CTRL-D which closes the input stream. The query `halt?` can also be used to terminate Wild LIFE. (**Note:** throughout the handbook `anything written like this` is simulated input or output from the interpreter.)

Upon startup, Wild LIFE automatically loads the file `.wild_life`. This is a convenience to allow you to customize your environment at each session. The customization file is first looked for in the current directory, and if not found there, in your home directory. If none is found, then Wild LIFE prints the message `No customizing file loaded` as is done above.

3.2 Input syntax

Wild LIFE uses essentially the same syntax as ISO Standard Prolog, which is very close to the Edinburgh syntax [10, 15, 19]. Unless specifically indicated to be different, the same syntactic conventions apply. In particular, variables are capitalized (or start with an underscore "_") whereas everything else is not, = is the unification predicate, `:-` defines a clause, `!` is the cut predicate, *etc*. When we depart from the familiar syntax, we will comment on the changes, giving our justification for adopting them.

There are two kinds of user input to Wild LIFE: declarations and queries. A declaration becomes part of the program and is not executed. A query is a question asked of the system. The syntactic difference between the two is how they terminate:

- Declarations are terminated by a period: "`.`".
- Queries are terminated by a question mark: "`?`".

Example 3.1

- Assert that chaplin is funny: type `funny(chaplin).` then ⟨`CR`⟩.[3]
- Ask who is funny: type `funny(X)?` followed by ⟨`CR`⟩; this will yield the answer `X=chaplin`.
- Ask for another answer: type `;` followed by ⟨`CR`⟩; there is no more answer since chaplin is the only funny thing around.

---

[3]We will use the notation ⟨`CR`⟩ to denote a carriage return.

Spaces, tabs, and line feeds appearing between tokens are ignored, so feel free to indent things as you wish or give input over several lines. When you do this, the prompt becomes "| " after the first line. The system knows when the query or declaration has ended. Comments are introduced by `%` and terminate at the end of the line.

System messages, warnings, and error messages all start with three stars `***`, which make them easy to recognize. If the query can be satisfied then Wild LIFE will print the bindings of the query variables followed by the message `*** Yes`. If the query cannot be satisfied, the message `*** No` will be printed.

Example 3.2 Let us assert a few facts defining a paternity relation:

```
> father(john,harry).

*** Yes
> father(john,charles).

*** Yes
> father(harry,michael).

*** Yes
```

We assert that X is the grandfather of Y if X is the father of Z and Z is the father of Y:

```
> grandfather(X,Y) :- father(X,Z),father(Z,Y).

*** Yes
```

We try a query:

```
> grandfather(A,B)?

*** Yes
A = john, B = michael.
```

3.3 Incremental query extension

Upon success of a query, the user is offered the possibility of extending it *using the resulting context*. This can be continued to any level of nesting. The query level is printed in the form of a numbered and indented prompt. At the prompt the user can take one of the following actions:

- Type a goal followed by `?` to extend the query.
- Type ⟨`CR`⟩ to abandon the last query increment and go back to the previous level.
- Type `;` to force backtracking and look for another answer.
- Type a period `.` to pop up to the top-level prompt from any depth.

- Make an assertion that uses the variable-binding context by typing a declaration followed by a period "`.`".

Example 3.3 Continuing on the previous example, now enquiring about A's son C (where A=`john`):

```
--1> father(A,C)?

*** Yes
A = john, B = michael, C = harry.
```

Let us check whether an alternative solution exists that is compatible with the context of the previous level by forcing backtracking:

```
----2> ;

*** Yes
A = john, B = michael, C = charles.
```

We are again at query level 2, but one that is independent from the previous level 2. Let us again force backtracking:

```
----2> ;

*** No
A = john, B = michael.
```

This query extension fails and brings the level back to 1; *i.e.*, back to the answer to goal `grandfather(A,B)?`. Observe that in this context, variable C no longer exists. Let us proceed, asking for the father C of B (michael):

```
--1> father(C,B)?

*** Yes
A = john, B = michael, C = harry.
```

And further, whether A (john) is the father of C (harry):

```
----2> father(A,C)?

*** Yes
A = john, B = michael, C = harry.
```

To go back to level 2, we type ⟨`CR`⟩:

```
------3>

*** No
A = john, B = michael, C = harry.
```

We now give up with `.` to go back to top level:

```
----2> .
>
```

We find incremental querying to be a powerful means of user interaction. It is particularly useful for exploring solutions step by step or seeking further information. Fine points about incremental querying are given in Section 13.2.

3.4 Loading files

A program file is a text file composed of definitions and queries. A query in a file behaves as a *directive* does in Prolog. Any combination of definitions and queries that are entered by the user may appear in a program file. To load a program file simply type the query `load("filename")?`. The filename must be a string, *i.e.*, enclosed in double quotes. The suffix "`.lf`" is added automatically if necessary. If a syntax error occurs in a file being loaded, it is reported along with the line number at which it was detected, and it causes Wild LIFE to abort further processing and go back to the top level.

For serious development work, it is recommended that you put your programs in modules and load them with the command `import("filename")?`. The Wild LIFE system comes with a large number of example programs, each of which is defined in a separate module. This allows example programs to be loaded in the same session without affecting each other or any other programs that may be loaded. See for example the cryptarithmetic program of Section 12.3.

`load` is neither Prolog's `consult` nor `reconsult` (although it comes closest to `consult`). If a file being loaded attempts to redefine an already defined object whose definition may not be extended, then an error is reported. As a result, you will have to exit Wild LIFE and restart it when you need to reload a file.

3.5 Interrupting execution

If execution of a query goes into an infinite loop or takes too long to compute, you may interrupt it by typing CTRL-C. There might be a slight delay if a garbage collection was taking place. When the interrupt is dealt with, the following prompt appears:

```
*** Command (q,c,a,s,t,h)?
```

Each letter is a mnemonic shorthand for a command corresponding to a specific action to be taken for Wild LIFE at this point. Hence, typing `h` (or `?`) in response to this prompt will yield the following to be printed:

```
*** [Quit (q), Continue (c), Abort (a), Step (s,RETURN),
     Trace (t), Help (h,?)]
*** Command (q,c,a,s,t,h)?
```

Typing `q` will make Wild LIFE exit the session; `c` will make execution resume from where it was interrupted; `a` will cause it to abort execution and return to top level; `h` will make Wild

LIFE print out a reminder of the meaning of these abbreviations; `t` will turn trace mode on; `s` (resp., ⟨CR⟩) will turn single-stepping on. Both tracing and single-stepping are automatically turned off by `a` and `c`.

4 The basic data structure: ψ-terms

Just as Prolog is based on first-order (Herbrand) terms, LIFE is based on ψ-terms.[4] In order to compare the two, one can say that ψ-terms are to Prolog terms what flexible records are to static arrays. Namely, instead of selecting subterms just by numeric positions, we can use *labels* (*i.e.*, symbolic keywords, also called *features*), and instead of fixing the number of subterms beforehand, we can add more subterms to a term at any time (even by user-input at run time). We call a subterm together with its label an *attribute* of the ψ-term. A label may be any natural number or any symbol. The symbol must be single-quoted if it does not start with a lower-case letter or if it contains non-alphanumeric characters other than underscore.

Furthermore, instead of Prolog's mutually incompatible functor symbols we use *sort* symbols that the programmer puts into a hierarchy (expressing hereby their relation, *i.e.*, their "degree of compatibility").

Finally, in order to express coreferences ("aliasing") not just between leaves of the term, but between *any* subterms, we may attach *tags* (*i.e.*, variables) to any subterms of a ψ-term. In particular, this allows us to describe cyclic structures.

Example 4.1 Here are a few sample ψ-terms:

- `42`, a specific integer;
- `int`, the sort that denotes all integers;
- `-5.66`, a specific floating point number;
- `2.10776e-8`, a specific floating point number in exponential notation;
- `real`, the sort that denotes all floating point numbers;
- `"a small piece of rope"`, a specific string;
- `string`, the sort that denotes all strings;
- `abc_def1`, a sort;
- `'%* strange characters!'`, a sort whose name contains strange characters (note the single quotes);
- `date(friday,"XIII")`, a ψ-term with implicit numeric labels;
- `date(1=>friday, 2=>"XIII")`, the same ψ-term with explicit numeric labels;
- `date(2=>"XIII", 1=>friday)`, the same ψ-term yet again, showing that the order of explicit labels is irrelevant;
- `freddy(nails => long, face => ugly, films => 5)`, a ψ-term with keyword labels;
- `rectangle(width => S:int, length => S)`, a ψ-term with two attributes that are aliased together, as indicated by the tag S;
- `X:person(home => address(occupants => [X]))`, a ψ-term with a cyclic reference, as indicated by the tag X.

[4] If you substitute Prolog terms by LIFE's ψ-terms the resulting language is called Login [4].

A ψ-term describes a set of objects, which themselves may be represented as records. Hence, the ψ-term may be construed as a record type. A ψ-term with given attributes describes records with at least the fields specified by the attributes, but possibly others. The values in the fields are themselves records which are described by ψ-terms. Furthermore, the records must satisfy the coreferences expressed by the tags of the ψ-term.

4.1 Sorts

Sorts are the syntactic entities attached to the root of every ψ-term. In this sense, they take over the role of type names in C and functors in Prolog. Sorts by themselves constitute the most basic ψ-terms. The sort attached to the root of a ψ-term is called its *principal* sort (or *root* sort).

Any symbol that is introduced in the text of a program or query is considered a sort unless it is declared otherwise or it is a label. A sort may be any integer or floating point number or character sequence. The character sequence must be surrounded by single quotes if it does not start with a lower-case letter or if it contains non-alphanumeric characters other than underscore. A character sequence surrounded by double quotes is also allowed; it is an explicit string.

No conceptual difference is made between values and sorts. This means that, for example, the value 1 is specified by the sort `1`. The fact that this is an integer value is specified by saying that `1` is a subsort of the sort of all integers, which is written `int`.

4.1.1 Defining sort inheritance

Sorts are put in a hierarchy by specifying a partial order relation between sorts. The order relation is declared by means of *sort inheritance definitions*, which are written as `s1 <| s2`. This declaration can be read as "`s1` is-a `s2`" or "`s1` is a subsort of `s2`." [5] The text `s1 <| s2` is the programmer equivalent of the mathematical notation $s_1 \lhd s_2$. Through this declaration, `s1` inherits all properties of `s2`. A sort may occur in several inheritance definitions if it has more than one direct relative (*i.e.*, parent or child).

Sorts denote sets, and the partial order between them amounts to the subset relation. Hence, a sort may be viewed as a *type* of records and the partial order as type *inheritance*. We will later see (in Section 7, page 29) that we can declare a definition for a sort; it then becomes a *class* in the object-oriented sense. The subsorts of a sort inherit all the properties defined for the parent sort. Different sorts may have common subsorts, thus allowing *multiple inheritance*.

Example 4.2 Let us declare a small hierarchy that describes the relationships between trucks and vehicles. We assume that any truck is also a vehicle and that any property pertaining to vehicles also applies to trucks. That is, trucks inherit all properties of vehicles. This is specified in Wild LIFE by typing `truck <| vehicle`. Subsequently, Wild LIFE's unification will take the inheritance information into account.

```
> truck <| vehicle.     % A truck is a vehicle
```

[5] This must not be confused with the predicate `subsort` that *tests* whether two sorts are related (see Section 8.4).

```
*** Yes
> mobile(vehicle).      % A vehicle is mobile

*** Yes
> useful(truck).        % A truck is useful

*** Yes
> mobile(X),useful(X)?  % What is mobile and useful?

*** Yes
X = truck.
```

A cycle within the hierarchy (such as `a <| b. b <| c. c <| a.`) is reported as an error. This enforces consistency of the sort ordering as it is specified by the user. Consistency checking becomes more complex when properties are attached to sorts (in Section 7). Wild LIFE does not check consistency of properties attached to sorts.

4.1.2 Built-in sorts

At the summit of the hierarchy of sorts, there is a greatest sort, denoted by $\top$, pronounced "top." That is, between any sort s and the sort $\top$ the order relation $s <| \top$ is always specified. In Wild LIFE the symbol `@` is used to represent $\top$.[6]

Likewise, at the base of this hierarchy we find the sort $\perp$, pronounced "bottom". That is, between any sort s and the sort $\perp$ the order relation $\perp <| s$ is always specified. In Wild LIFE the symbol $\{\}$ is used to represent $\perp$.

From now on, we will write `@` and $\{\}$ to denote $\top$ and $\perp$. The sort `@` denotes the set of all records. The sort $\{\}$ denotes the empty set.

Besides `@` and $\{\}$, the other built-in sorts available to the user are:

- All integers, floating point numbers and the symbols `int` and `real`, with the order relations n <| `int` (for all integers n), r <| `real` (for all non integral floating point numbers r), and `int <| real`. For example, `0`,`-5`, and `3.0` are subsorts of `int` and thus of `real`; `2.5` and `26.77` are subsorts of `real` but not of `int`.
- `list`, `[]` (empty list, which may be written `nil`) and `cons` (the list constructor), with the order relations `[] <| list` and `cons <| list`. Lists may be written with the same syntax as Prolog; *e.g.*:
  `[a,b,c] = [a,b|[c]] = [a,b,c|[]] = [a|[b|[c|[]]]].`
- All strings s and the symbol `string`, with the order relations s <| `string`. Specific strings appear within double quotes as in `"this is a string"`.[7]
- `bool`, `true` and `false` with the order relations `true <| bool` and `false <| bool`.

[6]Because $\top$ is not a standard ASCII character, we need an ASCII symbol to stand for it. Besides resembling a looped-around `a` that could stand for `anything`, the symbol `@` has a shape that is reminiscent of an embryo—a perfect ideogram to denote the most primeval sort in LIFE!

[7] As opposed to other non-numeric sorts, strings are *non-interned* symbols, *i.e.*, they are not put in the symbol table.

- `built_in`, a sort which is a supersort of all built-in sorts, with the order relations `list <| built_in`, `string <| built_in`, `real <| built_in` and `bool <| built_in`.

To summarize, the system contains the following predefined declarations.

```
built_in <| @.

list <| built_in.
string <| built_in.
real <| built_in.
bool <| built_in.

cons <| list.
[] <| list.

int <| real.

true <| bool.
false <| bool.
```

When extending built-in sorts it is recommended to respect the following elementary properties.

- `int <| real`
- glb(`list`, `real`) = {}
- glb(`string`, `real`) = {}
- glb(`list`, `string`) = {}
- glb(`true`, `false`) = {}

If these properties are not respected then the interpreter may behave in a strange manner: it may crash or it may not recognize the built-in sorts for what they are.

4.1.3 Greatest lower bound (glb)

The *glb* of two sorts r and s is their largest common subsort. Since sorts denote sets, the glb corresponds to set intersection.

There always exists at least one common subsort, namely $\{\}$. If $\{\}$ is the largest common subsort, then we say that r and s are *incompatible*. We note that incompatibility is always declared implicitly, *i.e.*, two sorts are incompatible because the user has not declared any common subsort, nor is there is a common built-in sort (except for $\{\}$, of course). This is in contrast to other hierarchies used for knowledge representation or computational linguistics where the incompatibility has to be declared explicitly.

The glb of two sorts r and s is not always given by *one* sort which is declared in the hierarchy. Namely, if there are several sorts $s_1 ; \ldots ; s_n$ which are common subsorts of r and s and which are not in an order relation among themselves, then the glb of r and s is given by the *disjunctive sort* $\{s_1; \ldots ; s_n\}$.

In Wild LIFE, if during execution a disjunctive sort is computed, then it is not created or printed out as such. Instead, operationally it introduces a disjunction, just as a predicate definition with several clauses introduces a disjunction.

Example 4.3 Say we define a hierarchy containing the sorts `two_wheels` and `four_wheels` that represent classes of objects having respectively two and four wheels.

- `bike <| two_wheels.`
- `bike <| vehicle.`
- `truck <| four_wheels.`
- `truck <| vehicle.`
- `car <| four_wheels.`
- `car <| vehicle.`

From this hierarchy we can deduce the following values for the glb:

- glb(`two_wheels`, `vehicle`) = `bike`
- glb(`four_wheels`, `vehicle`) = {`car;truck`}
- glb(`two_wheels`, `four_wheels`) = {}
- glb(`bike`, `vehicle`) = `bike`
- glb(`bike`, `@`) = `bike`

The disjunctive sort {`car;truck`} is immediately enumerated.

4.2 Attributes

A ψ-term in its basic form is simply a sort. A more complicated ψ-term is constructed by taking a sort and attaching attributes to it. An attribute is a pair consisting of a label and an associated ψ-term. For example, consider the ψ-term:

```
show(title => "Bewitched",
     genre => sitcom,
     where => television,
     mother_in_law => "Agnes Moorehead")
```

This ψ-term describes the set of records with at least the four fields `title`, `genre`, `where` and `mother_in_law`. This set includes records with other fields as well. The ψ-term makes no statement regarding the other fields, not even whether they exist or not.

Ψ-terms are *extensible* record descriptions. This means that attributes may be added at will at run-time, even by user-input. For example, a computation may start with the value `circle(origin => P)`, and this value can later be refined to `circle(origin => P, radius => R)`. An important application of this extensibility is the use of ψ-terms as hash tables.

For ease of use, Wild LIFE allows consecutive numeric labels to be implicit. For example, `thing(a,b,c)` is equivalent to `thing(1 => a, 2 => b, 3 => c)`. The order of attributes is completely irrelevant and so this ψ-term can also be written as `thing(2 => b, 3 => c, 1 => a)`.

4.3 Variables and tags

Variables start with _ (underscore) or an upper case letter. An anonymous variable may be written with a single _. This is in fact equivalent to `@`, the top sort. Variables can be used to *tag* a ψ-term and then used as explicit handles for referencing the ψ-term. The syntax used to express the tagging of a ψ-term *t* by a variable `X` is `X` : *t*. These references may be cyclic; *i.e.*, a variable may occur within a ψ-term tagged by it. For example, `X:[42|X]` represents a cyclic list with single element 42. If a variable occurs by itself, not tagging a ψ-term, then it is implicitly considered to be tagging `@`, exactly as if it had been written `X:@`.[8]

◑◑ The terms `s(X,X:t)` and `s(X:t,X)` are identical, *i.e.*, the tagging construct "`:`" is simply a syntactic device used to represent sharing and cycles in the textual representation of a term. It is possible to write any directed graph structure as linear text using variables and tags. For example, when Wild LIFE sees `X:foo(bar => X)`, it creates a single cyclic structure with name `X`.[9]

The tagging construct `:` is not exclusively reserved to be used in the form `X:t`. Other sensible forms are `t:X` (which is equivalent to `X:t`), and more generally, $X_1:X_2: \ldots :X_n$, (which is equivalent to $X_1:X_2, X_2:X_3, \ldots, X_{(n-1)}:X_n$, for *n* variables X_1 through X_n). At most one of the X_i may be a term whose value is different from `@`.

Example 4.4 Here are four examples of the use of tags.

- `father(name => N:string,son => boy(name => N))`
  represents a father who has a son, such that the son shares the same name as his father. The tag N represents the name.
- `[A,A]`
  is a list whose first and second elements are identical.
- `L:[int,int,int|L]`
  is a cyclic list of length three whose three elements are integers.
- `write(A),happy(A:person)?`
  prints the output `person`, then go on to prove the goal `happy(person)`, because the tag A was bound to person globally for the whole query at parse time. This query is equivalent in all respects to `write(A:person),happy(A)?`.

4.4 Unification

Unifying two ψ-terms consists in (1) computing the greatest lower bound *glb* of their root sorts, (2) binding the root variables together, (3) attaching to them all the attributes of the two parent ψ-terms, and (4) unifying recursively the ψ-terms in corresponding attributes. If during this procedure a glb is found to be $\{\}$, then the unification is said to *fail*. Otherwise it is said to *succeed*.

Example 4.5 Here is a series of sample unifications that progressively illustrate the properties of ψ-term unification. The last example is particularly interesting as it shows the unification of two cyclic ψ-terms. Note that $\{\}$ causes an immediate failure when executed.

[8] The term `X:@(1)` is *not* the same as `X(1)`. The latter is a function application. See Section 6.4.

[9] The `:` is different from the unification predicate "=". The latter performs a calculation at run-time. For example, in `X = foo(bar => X)` the term on the right-hand side of = is not bound to X until run-time.

Unifying	with	results in
`U:@`	`V:@`	`U:V:@`
`U:100`	`V:int`	`U:V:100`
`5.6`	`int`	`{}`
`car(wheels=>4)`	`vehicle(wheels=>N:int)`	`car(wheels=>N:4)`
`int(luck=>bad)`	`13(roman=>"XIII")`	`13(roman=>"XIII",luck=>bad)`
`[A,B,C]`	`[1,2,3]`	`[A:1,B:2,C:3]`
`[H|T]`	`[a,b,c,d]`	`[H:a|T:[b,c,d]]`
`X:s(s(X))`	`Y:s(s(s(Y)))`	`X:Y:s(X)`

The unification of two ψ-terms corresponds to testing whether the intersection of the two sets of records described by the two ψ-terms is empty. If it is nonempty, then the unification yields a ψ-term that describes the intersection. The records in the intersection have all the fields of both sets. This explains why the attributes of the unifier ψ-term form the "union" of the ones of the two parent ψ-terms. Here union means the set of attributes that appear in at least one of the two ψ-terms. The attribute for a common label has a ψ-term describing the intersection of the two corresponding ψ-terms.

4.4.1 A step-by-step comparison with Prolog unification

Unification of ψ-terms generalizes unification of Prolog terms. This is progressively illustrated in the following cases.

1. Unifying two terms with same root sorts and arguments, *e.g.*, $X : \texttt{foo}(s_1, \ldots, s_n)$ and $Y : \texttt{foo}(t_1, \ldots, t_n)$, is done as in Prolog.

2. Unifying $X : \texttt{foo}(s_1, \ldots, s_n)$ and $Y : \texttt{bar}(t_1, \ldots, t_m)$ leads to failure exactly as in Prolog if `foo` and `bar` are incompatible in the sort hierarchy. This means that the user has not declared any subsort which is below both `foo` and `bar`. Whether $n = m$ or $n \neq m$ has no effect on success or failure.

3. Unifying two ψ-terms which have exactly the same set of labels and the same sort, $X : \texttt{foo}(l_1 \texttt{=>} s_1, \ldots, l_n \texttt{=>} s_n)$ and $Y : \texttt{foo}(l_1 \texttt{=>} t_1, \ldots, l_n \texttt{=>} t_n)$, is a slight generalization of case (1); there the numeric features $l_1 = 1, \ldots, l_n = n$ are left implicit. The unification proceeds in direct extension of case (1).

4. Unifying two ψ-terms which have the same set of labels and two compatible sorts `s1` and `s2`, $X : \texttt{s1}(l_1 \texttt{=>} s_1, \ldots, l_n \texttt{=>} s_n)$ and $Y : \texttt{s2}(l_1 \texttt{=>} t_1, \ldots, l_n \texttt{=>} t_n)$, proceeds like in case (3) with the only difference that the unifier has as root sort the *glb* of the sorts `s1` and `s2`.

5. Unifying two ψ-terms which have *any* sets of labels and two compatible root sorts `s1` and `s2`. Some of the labels are common to both terms and others exist in only one term. This proceeds as in case (4); the only difference is that the unifier ψ-term has the attributes obtained by unifying the terms at the common labels *plus* all the attributes that exist in only one term.[10]

[10] This allows one to program with hash tables in LIFE.

6. Unifying two ψ-terms which have any sets of labels and two incompatible root sorts *r* and *s* leads to failure as in case (2).

5 Predicates

Predicates are defined and executed in Wild LIFE in the same manner as they are in Prolog, only ψ-terms replace Herbrand terms. This is not a tutorial on Prolog, so—for information going further than the explanations given below—please consult your local library [10, 15, 19].

Except for differences in built-ins, Prolog programs can run unaltered in LIFE if they follow the syntactic convention that each predicate and functor symbol is used with one arity only. See Section 15 for a complete list of differences between LIFE and Prolog.

5.1 Defining predicates

A predicate definition consists of one or several definite clauses. Clauses are stored in the assertion base in the same order as they will be entered during execution. A definite clause is written in the form `Head :- Body.` where `Head` is an atomic goal and `Body` is a non-empty sequence of atomic goals. An atomic goal is a ψ-term whose root sort is defined as a predicate in the program. The clause `Head :- succeed.` may be abbreviated as `Head.`. It is called a *fact*.[11]

Once defined as a predicate name, an identifier may not be declared as a sort. However, a predicate name may be used as data. It behaves as if it were a sort with single parent `@` and single child $\{\}$.

5.2 Executing predicates

As in Prolog, Wild LIFE uses top-down/left-right SLD-resolution to execute a query. A query (or resolvent) is a sequence of atomic goals separated by a comma "`,`", exactly as in the body of a clause.

Operationally, execution of a query proceeds roughly in the following manner. The resolvent is set initially to the query, and it behaves as a stack of predicate calls:

1. If the resolvent is not empty, an atomic goal *P* is popped from the resolvent. If the resolvent is empty, then the query succeeds. The result of the computation is given in the variable bindings.

2. The first clause `Head :- Body` defining *P* is chosen. If there are no more clauses, then the query fails. A choice point is created to allow a return to this execution state on backtracking. The choice point contains the next clause defining *P*.

3. `Head` is unified with *P* and `Body` is pushed on the resolvent. The new resolvent is the body `Body` followed by the remainder of the old resolvent. Execution continues at step (1).

4. If the unification in step (3) leads to failure, then execution backtracks to the most recent choice point, and continues forward from there.

[11] LIFE uses `succeed` and `fail` as the predicates that are always satisfied and never satisfied, respectively. LIFE uses `true` and `false` as sorts that mean true and false.

The above execution is similar to that of an imperative language such as C except that parameter passing is done by unification and execution may return to previous states by backtracking.

Example 5.1 Here is a two-clause program which will print the items in a list one per line:

```
print_list([]) :- !.
print_list([H|T]) :-
    write(H),       % Print the first element
    nl,             % Start a new line
    print_list(T). % Print the rest of the list
```

This translates in English to something like this:

- To print the empty list don't do anything.
- To print a list whose first element is H (the rest being T) write H and start a new line and print the list T.

And now we can try it out:

```
> print_list([a,b,c])?
a
b
c

*** Yes
```

5.3 Pruning the search tree with cut

The built-in predicate "`!`" in Prolog or Wild LIFE, pronounced "cut", allows the user to control the searching behavior of a program by cutting out alternatives from the search tree. Executing a cut is like executing `succeed` but with a side-effect: it removes all the alternatives which occurred from the moment the *rule was chosen* to the moment the *cut is reached*. This means that the next rules in the predicate's definition will not be tried and any alternatives that may have arisen during the proof of the current rule will not be explored either.

The Wild LIFE 1.0 interpreter does no clause indexing. This means that each clause of a deterministic predicate should contain a cut. The cut must be placed at the earliest point in the clause body where you are sure that if execution reaches that point, that the correct clause has been selected. No bindings to variables visible outside of the predicate should be done before the cut. This condition ensures that the cut does not change the predicate's semantics, *i.e.*, it is a *green* cut [15]. The `print_list` example given above contains a correct use of cut.

The cut predicate is useful because it allows one to reduce the number of alternatives, especially if you know for certain that all alternatives would fail. For the interaction of cut and functional evaluation (including residuation and coroutining), see Section 6.

5.3.1 The scope of cut

The choice-point associated to a cut is linked to the environment that existed when the cut was first encountered by the interpreter. If a goal passed as an argument contains a cut, that cut will remove choice points much further than expected. This behavior of cut should not be relied upon, since it probably will change in the compiler. Cut should be used in Wild LIFE only to guarantee that a predicate is deterministic, without changing its semantics.

5.3.2 Disjunctive terms and cut

If a goal contains disjunctive terms in its arguments, then the choice-points for those are created *before* a rule to prove it is chosen. Hence, they do not lie in the scope of a cut in the body of the rule.

Example 5.2 Suppose that the rule:

```
p(A) :- ! , write(A).
```

has been asserted. The query `p(X:{1;2;3})?` will generate three solutions: `X = 1` and the output `1`, then (upon backtracking with `;`) `X = 2` and the output `2`, and then `X = 3` and the output `3`.

Since executing the cut removes alternatives relative to disjunctive terms in the head of the rule, the query `q(X)?` will only generate one solution: `X=1` and write `1` if the rule:

```
q(A:{1;2;3}) :- !, write(A).
```

has been asserted.

5.3.3 Negation-as-failure

An interesting application of cut is negation-as-failure,[12] just as in Prolog. It is written "\+". The symbol \+ was chosen in Prolog because it resembles a tilted version of the mathematical symbol for "not provable", namely "$\not\vdash$".

Example 5.3

```
\+(G) :- G,!,fail.
\+ .
```

The first rule uses the fact that goals use the same data structure as ψ-terms. It first tries to prove the goal G, then if there was a solution, cuts out all alternatives and fails. If there are no solutions to G then the next rule is used for *not* and this always succeeds.

This does not quite do what you expect because all the parameters of X have been evaluated *before* they were passed. In other words, functions, disjunctions and sorts that needed evaluating have been dealt with, so "`\+(write({a;b;c}))?`" for example will print `abc` before failing.

Example 5.4 This example shows the use of a variable name in a query to replace a goal.

[12] This form of negation is often unsound. Functions in LIFE provide sound negation for equalities.

```
> X=write("Here I am!")?

*** Yes
X = write("Here I am!").
--1> X?
Here I am!
*** Yes
X = write("Here I am!").
```

For more information on using variables as predicates, see Section 13.3 (page 90).

6 Functions

In LIFE, a function is a routine that is called by matching and that returns a result. Functional computations are determinate, *i.e.*, a function call only fires once and is never backtracked to. A function may be called before the values of its arguments are known. In that case, the function will *suspend*. Technically, a function that suspends is said to *residuate* and the suspension is called a *residual equation* or *residuation*. The function will execute as soon as its arguments are known. A function may also be *curried*, *i.e.*, called with missing arguments. Residuation and currying provide implicit coroutining.

An *order-independent* routine is one whose result depends only on the value of its arguments, and not on the order in which the arguments were bound relative to when the routine was called. If all built-in routines called during the execution of a program are order-independent, then the value of the final result is independent of the order in which the program's functions and predicates are executed. The order affects only the execution efficiency. See Section 13.3 (page 90) for more information on order-independence when functions and predicates are used together.

Since functions can be called before their argument values are known, this frees the programmer from having to know what the data dependencies are. It provides a powerful search-space pruning facility by letting "generate-and-test" search be changed into daemon-controlled "test-and-generate" search. A residuated function acts like a daemon: it continuously checks whether its arguments are sufficiently instantiated. These checks implement a form of "data-driven" synchronization. That is, the ability of functions to residuate yields a form of concurrent programming.

Moreover, such non-declarative heresies as the *is/2* predicate in Prolog and the *freeze* meta-predicate in Prolog II are not needed. Arithmetic functions in Wild LIFE residuate when necessary. Functional residuation provides most of the abilities of MU-Prolog's *wait* declarations [13] and NU-Prolog's *when* declarations [14].

6.1 Defining functions

A function definition consists of one or more *function rules*. These are stored in the assertion base in the same order as they will be tried during execution. A function rule is written in the form `Head -> Expr.` where `Head` is a ψ-term whose root-sort name f is the name of the function being defined, and `Expr` is a ψ-term. The rule is read as "`Head` evaluates to `Expr`".

`Head` may not contain any function calls.

Functions, declared sorts, and predicates share the same name space. That is, a given identifier may only be used for one of the three. Attribute labels (features), however, have their own independent name space.

Example 6.1 This example defines the factorial function $n! = n(n-1)(n-2)\ldots1$. The result expression of the second rule contains a call to the built-in function `*`, which is written as an infix operator.

```
> fact(0) -> 1.
> fact(N:int) -> N*fact(N-1).
> write(fact(5))?
120
*** Yes
```

Example 6.2 This example defines a function `listlen(L)` that calculates the length of a list. It uses ψ-terms to represent the natural numbers, `0` for 0, and `s(T)` for $n+1$ if T represents n. The call `listlen(L)` will residuate if L's value is unknown (usually `@`). For example, the result of `listlen([_,_|_])` is the ψ-term `s(s(@))` which represents any $n \geq 2$. This correctly represents what is known of the length of `[_,_|_]`.

```
> listlen([]) -> 0.
> listlen([_|L]) -> s(listlen(L)).
> write(listlen([a,b,c]))?  % A list of length 3
s(s(s(0)))
*** Yes
> write(listlen([_,_|_]))?  % A list of length at least 2
s(s(@))
*** Yes
```

The result of a function can be a ψ-term that represents a predicate. For this result to be executed as a predicate, it must be in a *predicate position*, *i.e.*, in a place where a predicate is expected.

The result of a function can be a boolean, *i.e.*, a ψ-term whose root sort is `true` or `false`. Such a function may be used in a predicate position. The result `true` is executed as success and the result `false` is executed as failure.

A *functional expression* is a ψ-term whose root-sort name is a function name, say `f`. The attributes of the ψ-term are its arguments. A special case is the functional expression that constitutes the head of a function rule defining f; its attributes are called the *formal* arguments. Otherwise, the functional expression represents a call to the function f; then, its attributes are called the *actual* arguments.

6.2 Executing functions

A functional expression *F* is evaluated in the following manner. All arguments of a function are evaluated before the function is evaluated. In what follows we assume that *F* contains no evaluable expressions in its subterms.

1. Choose the first rule `Head -> Expr` defining *F*. If no such rule exists, then return {}. Returning {} causes an instant failure to take place. Failure causes backtracking to the most recent choice point.

2. Match *F* with `Head`.

3. If *F* matches `Head` then the rule fires and the body is evaluated and returned. We say *F* matches `Head` if to make them equal, refinements have to be made only to `Head` and not to *F*. That is, `Head` is more general than *F*.

4. If *F* and `Head` are non-unifiable, then the rule fails. Pick the next rule in the definition of *F* and go to step (2). If there are no more rules then return {}.

5. If neither step (3) nor step (4)'s conditions are true then residuate. This suspends the execution of *F* and returns the temporary result `@`. If any of *F*'s arguments are refined then resume execution at step (2). When the function fires, the actual result will be unified with the temporary result. Therefore the temporary result may be used in further function calls as if it were the actual result.

This execution mechanism corresponds to a simple formal logical specification [6]. Step (3) corresponds to testing whether *F* implies `Head`. Step (4) corresponds to testing whether *F* implies the negation of `Head`.

Operationally, the residuation is attached to each of the *residuation variables*. These are the variables of the calling functional expression on which its comparison with the formal one is still pending. More precisely, two conditions are guaranteed to hold for the residuation variables. It is possible to instantiate the set of residuation variables so that the rule fires. It is possible to instantiate any one residuation variable so that the rule fails.

Multiple residuations can be attached to a single variable if there are multiple function calls with this variable in their arguments. If the variable is unified then all the residuations are resumed. The order in which the functions are resumed is deliberately left unspecified.

As with clauses, the rules defining a function are looked up in the order they are entered. The important difference is that *functions are deterministic*, *i.e.*, functional computations are determinate. That is, there is no backtracking once a rule has fired, so the first rule to fire hides all those following it. In the jargon of committed-choice languages, matching is a commit-condition; if the head of a rule is matched, the execution is committed to take that rule.

Another way of seeing this is that functional evaluation does not allow argument guessing as would be non-deterministically possible by *narrowing* (*i.e.*, using unification instead of matching when calling functions). In order to unify the variables in the calling functional expression, one has to use the built-in function "|" (such-that).

The built-in arithmetic functions extend the above scheme in an important way: they perform all deterministic local propagations. That is, they infer the values of one or more arguments, if that inference is unique. For example, the goal `0=B-C` unifies B and C. The goal `A=A*B` does nothing, since there are *two* possible solutions: A=0 or B=1. This is a form of narrowing.

Example 6.3 Here is an example of residuation using the previously defined function `fact`. First, we impose the constraint $A = B!$, *i.e.*, A is the factorial of B, using the `fact` function given earlier.

```
> A=fact(B)?

*** Yes
A = @, B = @~.
```

The expression `fact(B)` residuates, yielding `@` as a temporary result. The tilde `~` after `@` means that B is a residuation variable, *i.e.*, a variable which, if its sort is made more precise (more information is known), will cause the residuated function to be re-evaluated.

```
--1> B=real?

*** Yes
A = @, B = real~.
```

The function `fact` still residuates because `int <| real`.

```
----2> B=5?

*** Yes
A = 120, B = 5.
```

`5 <| int` so `fact(B:5)` can be calculated. Let us now go back to the previous query level by typing ⟨CR⟩:

```
------3>

*** No
A = @, B = real~.
----2> A=123, B=6?

*** No
A = 123, B = real~.
```

We have now strengthened the constraint to $6! = 123$, and of course this constraint always fails.

Example 6.4 This example illustrates the execution order of a residuating functional evaluation in interaction with a calling predicate.

```
> p(X) :- write("From p: ",X).
> f(X:int) -> X | write("From f:",X).

*** Yes
> p(f(X))?   % p's body is executed, f's is not
From p: @
*** Yes
X = @~.
--1> X=1?    % f's body is now executed
From f: 1
*** Yes
X = 1.
```

Example 6.5 This example illustrates the execution order of functions in interaction with a calling predicate (namely, unification =) and with predicates called by such-that.

```
> fact2(0) -> 1.
> fact2(N) -> N*fact2(N-1) | write(N," ").

*** Yes
> 7=fact2(3)?
1 2 3
*** No
```

6.3 Matching

We describe the problem of matching of two ψ-terms as the problem whether the "actual ψ-term" (the caller) matches the "formal ψ-term" (the definition). This avoids the potential confusion about the direction of matching. The actual ψ-term matches the formal one if it describes a *subset* of the set described by the formal ψ-term.

This yields the following conditions for the actual ψ-term to match the formal ψ-term:

1. The root sort of the actual ψ-term is a subsort of the root sort of the formal ψ-term.

2. The ψ-terms in the attributes of the formal ψ-term are matched by the ψ-terms in the corresponding attributes of the actual ψ-term (which must exist).

3. The coreferences (the variable aliases) of the actual ψ-term express (at least) the coreferences of the formal ψ-term.

The first condition is clear from the sort hierarchy. The two root sorts can be equal, since a sort is a subsort of itself. The second condition means that if a label together with its value t forms an attribute of the formal ψ-term, then an attribute with the same label and a value t' has to be present in the actual ψ-term, and t' has to match t. The third condition means that if two occurrences in the formal ψ-term are aliased (tagged with the same variable), then this has to be true for the corresponding occurrences in the actual ψ-term.

A successful match is possible if the actual and the formal ψ-terms can be unified so that only formal variables are bound; *i.e.*, the actual ψ-term remains untouched. To be precise, this means that three conditions have to be met after the unification. First, a sort in the actual ψ-term is intersected only with a supersort (higher in the hierarchy). Second, no subterm in the actual ψ-term gets attributes with new labels. Third, an actual variable is not bound to another actual variable.

Example 6.6 A call to the following function `diff(X,Y)` will fail if X and Y have been unified and succeed if X and Y are non-unifiable. Otherwise it will residuate. Note that it residuates on *all common subterms* of X and Y. This is necessary to guarantee that non-unifiability is seen immediately.

```
> diff(X,X) -> fail.
> diff(X,Y) -> succeed.

> diff(X,Y)?                          % This residuates

*** Yes
X = @~, Y = @~.
--1> X=1, Y=1?                        % This also residuates

*** Yes
X = 1~, Y = 1~.
----2> X=@(int,foo), Y=@(42,foo)?     % This also residuates

*** Yes
X = 1(int~,foo~)~, Y = 1(42~,foo~)~.
------3> X=Y?                         % This fails

*** No
------3> X.1=23?                      % This succeeds

*** Yes
X = 1(23,foo), Y = 1(42,foo).
```

6.4 Currying

A functional expression is *curried* if it has one or more missing arguments. A function may be curried on any subset of its arguments, and the missing arguments may be applied in any order. A curried functional expression is itself a function. It can be passed around and applied several times.

Example 6.7 We define a function `f` on three arguments which are selected by the labels `eggs`, `bacon` and `toast`. It returns the list of its three arguments. We call it with only two arguments. The function `f` returns a curried form which is bound to A. We can apply A any number of times, by specifying a third argument (which is indexed by the label `toast`).

```
> f(eggs=>X, bacon=>Y, toast=>Z) -> [X,Y,Z].

> A=f(eggs=>a, bacon=>b), write(A)?
f(eggs => a,bacon=> b)
*** Yes
--1> R1=A(toast=>c),R2=A(toast=>d),write(R1,R2),nl,write(A)?
[a,b,c][a,b,d]
f(eggs => a,bacon=> b)
```

The curried expression A is unchanged.

Currying is different from residuation. A residuated functional expression returns its *result*, which has the value `@` until the function fires. The result may be used in further calculations before the function has fired. A curried functional expression A may only be used in functional applications of the form $\mathtt{A}(l_1{=}{>}X_1, ..., l_n{=}{>}X_n)$.

Example 6.8 This example shows the difference between currying and residuation. We define a function in *one* argument, which is a ψ-term with three attributes. We call it with a ψ-term with only two attributes. Hence, the functional expression residuates and returns `@` as its value. If its argument gets more instantiated by adding the missing attribute, the function fires. This is possible only once.

```
> g(@(eggs=>X, bacon=>Y, toast=>Z)) -> [X,Y,Z].

> A=g(B:@(eggs=>a, bacon=>b)), write(A)?
@
*** Yes
--1> B=@(toast=>c), write(A)?  % Fire the function g
[a,b,c]
```

Example 6.9 We define the function `h` with labels `1`, `2` and `3`. Since these are consecutive numeric labels, they do not have to be explicit. If the function is curried in the first two arguments then we need to give the label `3` explicitly. If the function is curried in the third argument then we can leave the labels `1` and `2` implicit since they are consecutively numbered from `1`.

```
> h(A,B,C) -> [A,B,C].

> X1=h(a,b), X2=h(3=>f)?

*** Yes
X1 = h(a,b), X2 = h(3 => f).
--1> Y1=X1(3=>c), Y2=X2(d,e), write(Y1), write(Y2)?
[a,b,c][d,e,f]
```

◑◑ Currying in Wild LIFE is not defined in the usual way because the list of required arguments is indexed by labels which are not necessarily consecutive natural numbers. Therefore, arguments are consumed by *name*, and not by *position* as in the λ-calculus.[13] The usual way of defining currying would be to say: $f(X, Y) = f(X)(Y)$. The definition imposed by labels in Wild LIFE is: $f(X_1, X_2) = f(X_1)(2 => X_2)$. We get the same result with: $f(X_1, X_2) = f(2 => X_2)(X_1)$.

Wild LIFE 1.0 cannot parse an expression of the form $f(X_1)(2 => X_2)$. It must be written in two parts as $F(2 => X_2)$ and $F = f(X_1)$. A curried functional expression F looks like a ψ-term, and indeed has the same syntax as a ψ-term. However, it is illegal to unify F with another term. It is possible to inspect the arguments of F with the `.` (projection) or `has_feature` functions. More information about the implementation of currying in Wild LIFE may be found in Section 13.1 (page 88).

Example 6.10 In this example, the operator `/` (real division) is curried and inverted.

```
> A = F(B), F = /(2=>A)?

*** Yes
A = real~, B = real~, F = /(2 => A).
--1> A = 5?

*** Yes
A = 5, B = 25, F = /(2 => A).
```

Example 6.11 The standard example of a higher-order function is the built-in function `map`. It is defined as follows:

```
map(F,[])->[].
map(F,[H|T])->[F(H)|map(F,T)].
```

It takes a function, or a variable that will be instantiated to a function (*e.g.*, a curried functional expression), and a list and applies the function to every element of the list. The following execution fragment shows how to couple the function `map` with residuation.

[13] A calculus handling currying with named arguments and consumption by position is presented in [2].

```
> fact(0) -> 1.
> fact(N) -> N*fact(N-1).
> R=map(F,[4,5,6,7])?

*** Yes
F = @~~~~, R = [@,@,@,@].
--1> F=fact?

*** Yes
F = fact, R = [24,120,720,5040].
```

Example 6.12 This example defines a function with two rules where each rule has an argument with a different label. The function will curry if arguments are missing relative to the *rule being matched against*. Taking advantage of this behavior is bad programming style in Wild LIFE and will probably be forbidden in the compiler.

```
> foo(a=>int) -> 1.
> foo(b=>int) -> 2.

> X=foo(b=>0)?

*** Yes
X = foo(b => 0).
--1> Y=X(a=>string)?

*** Yes
X = foo(b => 0), Y = 2.
```

Example 6.13 This example shows how residuation, currying, and functional variables can be combined together: a constraint is generated which binds the variable R to the result of applying F to A, where at that point both F and A are unknown. Later the argument A is chosen by the predicate `pick_arg` and the function F by the predicate `pick_function`. Note that `*(2 => 4)` is a curried function which multiplies its first argument by 4. Define the following predicates in addition to `fact` seen before.

```
pick_arg({5;3;7}).

pick_func( *(2=>4) ).
pick_func( fact ).

test :-
```

```
      R=F(A),         % Apply an unknown function
                      % to an unknown argument
      pick_arg(A),    % Pick an argument
      pick_func(F),   % Pick a function
      write("Function ",F,
            " applied to argument ",A,
            " is ",R),
      nl,
      fail.           % Force backtracking
```

and let us try it.

```
> test?
Function *(2 => 4) applied to argument 5 is 20
Function fact applied to argument 5 is 120
Function *(2 => 4) applied to argument 3 is 12
Function fact applied to argument 3 is 6
Function *(2 => 4) applied to argument 7 is 28
Function fact applied to argument 7 is 5040

*** No
```

6.5 Quote and eval

Since terms are manipulated as data in Wild LIFE, and since functions are evaluated eagerly, it is often necessary to prevent such an evaluation to happen. This is done, quite similarly to Lisp, by using a quoting operator. It is written as a prefix backquote ` . This distinguishes it from the single quote ' already used to write symbols that contain non-alphanumeric characters. A backquoted expression is a ψ-term and may be manipulated like any ψ-term.

Example 6.14 This is an example of quoting a functional expression.

```
> X=1+2?

*** Yes
X = 3.
--1> Y=`(1+2)?

*** Yes
X = 3, Y = 1 + 2
```

The following built-ins are related to quote.

- The function `eval(E)` evaluates the result of the quoted expression E. It does not modify E. For example, continuing the interaction above:

Example 6.15

```
----2> Z=eval(Y)?

*** Yes
X = 3, Y = 1 + 2, Z = 3.
```

- The function `evalin(E)` evaluates in-place the quoted expression E. The quoted expression E is replaced by its result.
- The predicate `non_strict(P)` declares that the arguments of P are quoted, not evaluated, when the routine is called. This predicate should be used only as a declaration, *i.e.*, in a query and not in a definition. This predicate is non-strict, *i.e.*, it does not evaluate its argument.

Example 6.16 This is an example of a non-strict predicate.

```
> non_strict(p)?
> p(X) :- write(X).

> p(1+2)?
1 + 2
*** Yes
```

Calling `p(1+2)` prints the quoted term `1+2`.

If an argument is shared between a strict and a non-strict routine, then it is considered non-strict. That is, non-strict dominates over strict.

6.6 Choosing between matching and unification

How does a programmer choose between matching and unification as calling mechanisms? Functions are called by matching and predicates are called by unification. Matching is one-way, *i.e.*, it does not modify its arguments, so it can act only as an input passing mechanism. Unification is symmetric, *i.e.*, it can act as both an input and an output passing mechanism.

To modify a function's arguments, the built-in function `E|G` (pronounced "E *such that* G") can be used. It evaluates and returns the expression E, and then proves the predicate G. The latter is called by unification, and hence if it is given one of the function's arguments, that argument will be unified.

We illustrate the differences between unification and matching as calling mechanisms by means of a routine to append two lists. We give three definitions of this routine: as a function, a predicate, and using such-that. Here are the definitions:

```
% The standard functional version of append:
append1([],L) -> L.
append1([X|L1],L2) -> [X|append1(L1,L2)].

% The standard predicate version of append:
append2([],L,L).
append2([X|L1],L2,[X|L3]) :- append2(L1,L2,L3).

% A variant that matches its first argument
% and unifies its third:
append3([],L,R) -> true | R=L.
append3([X|L1],L2,R) -> true | R=[X|L3], append3(L1,L2,L3).
```

The `append1` function waits until its first argument is a list before returning its result. The `append2` predicate executes to completion immediately and unifies its result with its third argument. Backtracking will provide alternate results. The `append3` function uses such-that. It waits until its first argument is a list (just like `append1`) and unifies its result with its third argument (just like `append2`). The advantage of `append3` over `append1` is that tail recursion optimization is regained because the variable L3 is used as a place-holder for the result. Here is an example execution showing that `append1` and `append2` behave the same if their arguments are instantiated:

```
> A=append1([1,2],[3,4]), append2([1,2],[3,4],B)?

*** Yes
A = [1,2,3,4], B = [1,2,3,4].
```

In the following example execution of `append1`, the arguments are not instantiated at the call:

```
> A=append1(B,C)?

*** Yes
A = @, B = @~, C = @.
--1> B=[1,2]?

*** Yes
A = [_A: 1,_B: 2|C], B = [_A,_B], C = @.
```

The call residuates (which is marked by the tilde ~) until B is sufficiently instantiated. In the following example execution of `append2`, the arguments are not instantiated at the call:

```
> append2(B,C,A)?

*** Yes
A = @, B = [], C = A.
--1> ;

*** Yes
A = [_A|C], B = [_A], C = @.
--1> ;
```

```
*** Yes
A = [_A,_B|C], B = [_A,_B], C = @.
```

In contrast to `append1`, `append2` attempts to guess the right solution. It returns longer and longer possible solutions on backtracking.

7 Constrained sorts

In Section 4.1.1 (page 8) we saw how to define a simple sort hierarchy. In practice, LIFE has more to offer: it is possible to attach properties (attributes or arbitrary constraints) to sorts. These properties are verified during execution and are inherited by subsorts.

Example 7.1 This example ensures that all terms with sort `person` have an age field whose value is an integer.

```
> :: person(age => int).
> man <| person.
> A=man?

*** Yes
A = man(age => int).
```

The sort `man` is a subsort of `person`. It inherits all attributes of `person`.

Sorts, properties, and inheritance in LIFE resemble classes, class data, and inheritance in a mainstream object-oriented language like C++. See Section 12.6 (page 82) for more details.

7.1 Defining constrained sorts

A sort that is given a sort declaration is called a *declared sort*. There are two independent ways to declare a sort. First, it can be given a place in a hierarchy. Second, it can be given properties to check.

By default, an uninterpreted identifier, *i.e.*, a symbol that does not occur in a predicate, function, or sort declaration, behaves like a sort whose only parent is `@` and whose only child is $\{\}$. It is also known as an *undeclared sort*. Predicates and quoted functions also behave in this way.

7.1.1 Sort attributes

A sort is given a ψ-term property with a *sort attribute declaration*. This is written as `:: Head` where `Head` is an arbitrary ψ-term. This attaches the property `Head` to the root sort of `Head`. For example, `:: person(age=>int)` ensures that all instances of `person` have the feature `age` whose value is an `int`.

Example 7.2 This example defines two sorts with attributes. All vehicles have a make which is a string, and a number of wheels which is an integer:

```
:: vehicle(make => string, wheels => int).
```

All cars have 4 wheels:

```
:: car(wheels => 4).
```

If the relation `car <| vehicle` is asserted then any properties attached to `car` must be compatible with those attached to `vehicle`.

7.1.2 Constrained sorts

A sort is given an arbitrary predicate as property with a *constrained sort declaration*. This is written as `:: Head | Goal`. For example, `:: X:person | X.age=int` adds the property `X.age=int` to all instances of `person`. This has the same effect as `:: person(age=>int)`.

A sort attribute declaration is a special case of a constrained sort declaration. Both attribute declarations and constrained sort declarations may contain functions. The resemblance between a constrained sort definition and the built-in function such-that is intentional. Such-that attaches a predicate to a function, and a constrained sort definition attaches a predicate to a sort.

Example 7.3 A rectangle has a length, width and area. A square is a rectangle with identical length and width.

```
:: rectangle(length=>L:real, width=>W:real, area=>L*W).
:: square(length=>S, width=>S, side=>S).
square <| rectangle.
```

Here is an example query:

```
> R=rectangle(area => 16, width => 4)?

*** Yes
R = rectangle(area => 16,
              length => 4,
              width => 4).
--1> R=square?

*** Yes
R = square(area => 16,
           length => _A: 4,
           side => _A,
           width => _A).
```

The system makes a distinction between a square and a rectangle with identical length and width. The latter can be refined into the former.

Example 7.4 This example defines a subtype of list, `int_list`, which defines lists that contain only integers. This property is guaranteed to be true for an `int_list`. This definition does not interfere in any way with existing code using lists. An `int_list` will work with existing list-based routines.

```
int_list <| list.
int_cons <| cons.
int_nil  <| nil.

:: int_cons(int,int_list).
int_cons <| int_list.
int_nil  <| int_list.
```

Here is an example query:

```
> X=[1,2,3]?          % X is a standard list

*** Yes
X = [1,2,3].
--1> X=int_list?      % X is an int_list

*** Yes
X = int_cons(1,int_cons(2,int_cons(3,int_nil))).
----2> Y=length(X)?  % Calculate the length of X

*** Yes
X = int_cons(1,int_cons(2,int_cons(3,int_nil))), Y = 3.
```

7.2 Executing constrained sorts

The Wild LIFE system guarantees that all instances of a sort are consistent with all declarations of that sort. For the inheritance hierarchy this is guaranteed by ψ-term unification (see Section 4.1.3, page 10). Consistency with properties is guaranteed by a mechanism called *sort unfolding*.

Sort unfolding makes a copy of the declaration and unifies it with the term. Each declaration is checked at most once for a given sort instance. This performs a kind of *memoization*, *i.e.*, the solution of an intermediate problem is remembered and reused when encountered again. An arbitrarily complex proof can be attached to a sort. For a given variable, the proof will only be executed once: when the variable is refined to have that sort or a lower sort. The fact that the proof has been executed is immediately available by inspecting the value of the variable's root sort.

A *recursive* sort declaration is a declaration that has a compatible sort in a subterm. That is, a declaration of `s1` is recursive if it contains `s2` such that glb(`s1`,`s2`)$\neq\{\}$.

In the current implementation, a recursive sort definition such as `:: person(spouse => person)` will go into an infinite loop because the definition of `person` is expanded indefinitely. To cope with such definitions, Wild LIFE uses the declaration `delay_check(`A_1,

A_2, ...) where A_1, A_2, *etc.*, are sorts. This predicate is non-strict, *i.e.*, it does not evaluate its argument. `delay_check` delays expanding a term until it has at least one attribute.

With this declaration, the above definition of `person` is:

```
delay_check(person)?
:: person(spouse => person).
```

This delays the expansion of a term with sort `person` until that term has at least one attribute. The `delay_check` declaration is propagated to all subsorts of `person`.

Example 7.5 This is an example of how to use `delay_check`. Given the following declarations:

```
:: P:person(best_friend => Q:person) | get_along(P,Q).
delay_check(person)?

cleopatra <| person.
:: cleopatra(nose => pretty, occupation => queen).

julius <| person.
:: julius(nose => big, last_name => caesar).

get_along(cleopatra,julius).
get_along(julius,cleopatra).
```

This explains the following behavior:

```
> A=person?

*** Yes
A = person.
--1> A=@(nose => pretty)?

*** Yes
A = cleopatra(best_friend => julius,
              nose => pretty,
              occupation => queen).
```

The constraints attached to `cleopatra`, by inheritance from `person`, are not checked until an attribute becomes present. Having appeared as part of the structure of `cleopatra` as a result of checking `cleopatra`'s constraints, `julius`, who is also a `person`, is left unconstrained for the same reason.

Unfortunately, the `delay_check` mechanism is not sufficient to guarantee completeness and convergence in all cases. The LIFE compiler will have a consistent handling of recursive sorts that does not need `delay_check`. A complete and consistent algorithm to do lazy attribute inheritance has been developed [9]. This algorithm takes an attribute constrained in a sort definition into consideration only if it appears in the resolvent.

7.3 Constrained sorts as daemons

A constraint that is attached to a sort declaration is checked at run-time during unification. It can be used very effectively as a data-driven daemon–code whose execution is triggered upon access to an object.

Constrained sorts can be used to help debugging, for example by printing all ψ-terms of a given sort each time they have their constraints checked.

Example 7.6 This example shows how attaching a constraint to `int` lets one trace all integer calculations.

```
> :: I:int | write(I," ").

*** Yes
> A=5*7?
5 7 35
*** Yes
A = 35.
--1> B=fact(5)?
5 1 4 1 3 1 2 1 1 1 0 1 1 2 6 24 120
*** Yes
A = 35, B = 120.
```

Example 7.7 This example shows how attaching a constraint to `cons` lets one trace all list calculations.

```
> :: C:cons | write(C.1), nl.

*** Yes
> A=[a,b,c,d]?
d
c
b
a

*** Yes
A = [a,b,c,d].
```

---

8 Basic built-in routines

This section summarizes the basic built-in functions and predicates of Wild LIFE.

8.1 Control flow

The following built-ins provide the ability to modify the execution flow of a program.

- The predicate `succeed` always succeeds.
- The predicate `fail` always fails.
- The predicate `(A,B)` executes predicate A and then predicate B. It is a logical conjunction.
- The predicate `(A;B)` creates a choice point and then executes predicate A. If on backtracking execution returns to the disjunction, then predicate B is executed. `(A;B)` is a logical disjunction.
- The function `{A;B}` creates a choice point and then returns A as its result. If on backtracking execution returns to the disjunction, then B is returned as its result. `{A;B}` is called a *type disjunction* or a *disjunctive term*. A "singleton" disjunctive term `{t}` is equivalent to `t`. An empty disjunctive term {} is equivalent to the bottom sort and causes an immediate failure.

Example 8.1 This example shows how disjunctive terms result in more compact code.

- `A={1;2;3;4}?` is equivalent to `A=1;A=2;A=3;A=4?`.
- `p({a;b;c}).` is equivalent to asserting `p(a). p(b). p(c).`

Unifying `U:{1;2;3;4;5}` with `V:{int;2;4;6;8}` results in the disjunctive term `U:V:{1;2;2;3;4;4;5}`. In theory, the above ought to be `U:V:{1;2;3;4;5}` rather than `U:V:{1;2;2;3;4;4;5}` since a unifier does not contain redundant information. To perform this correctly, unification of disjunctive terms should remove any term in the resulting disjunction that is subsumed by another disjunct. Since this is costly, Wild LIFE does not perform the complete operation.

- The predicate "`!`" (pronounced "cut") removes all choice points up to and including the choice point that was created when entering the predicate in which the cut occurs. See Section 5.3 for a discussion of cut.
- The function `cond(B,T,F)` is a conditional that returns T or F depending on the result of B. The conditional returns T if B evaluates to `true` and F if B evaluates to `false`. If B is a predicate, then B is executed and the conditional returns T if B succeeds and F if B fails. The conditional residuates otherwise. It does not evaluate its second or third arguments until the first is known. At most one of the second and third arguments are evaluated.

Example 8.2 This example defines an absolute value function using `cond`.

```
> absolute(V:real) -> cond(V>=0,V,-V).
> A=absolute(X)?

*** Yes
A = @, X = @~.
--1> X= -34?

*** Yes
A = 34, X = -34.
```

- The function `E|P` (called "such that") takes an expression E and a goal P. It evaluates E and proves P, and then returns the value of E as its result. No implicit cut is performed, *i.e.*, if either E or P returns more than one solution, then so will `E|P`. This is a bridge between functions and predicates.

Example 8.3 This example shows how the concatenation of two difference lists may be written as a function. We use the sort `--`, declared as an operator, to represent a difference list.

```
> op(500,xfy,--)?
> diffappend(A1--A2,B1--B2) -> A1--B2 | A2=B1.

> A=[1,2|X]--X, B=[3,4|Y]--Y?

*** Yes
A = [1,2|X]--X, B = [3,4|Y]--Y, X = @, Y = @.
--1> C=diffappend(A,B)?

*** Yes
A = _A:[1,2|X]--X, B = X--Y, C = _A--Y, X = [3,4|Y],
Y = @.
```

- The function `call_once(P)` yields `true` if the goal P can be proved (*i.e.*, P succeeds), and `false` if it cannot (P fails). The call `call_once(P)` always succeeds. This is a bridge between functions and predicates. This function returns exactly one solution, *i.e.*, it performs an implicit cut to remove all choice points created during the execution of P. This function residuates until P is different from `@`.

Example 8.4 This example illustrates `call_once`.

```
> p(1).
> p(2).
> A=call_once(p(X))?
*** Yes
A = true, X = 1.
--1> ;

*** No
```

The second solution, `X=2`, is not returned.

- The function `bagof(X,P)` returns a list containing as many elements as there are solutions to P. This function is non-strict, *i.e.*, it does not evaluate its arguments. Each element of the list is the value of X for one solution of P. Calls to bagof may be nested to any level. There is no existential quantification such as in Prolog's bagof.

Example 8.5 This example illustrates `bagof`.

```
> p(a).
> p(b).
> A=bagof(s(X),p(X))?

*** Yes
A = [s(b),s(a)], X = @.
```

- The function `bestof(X,Q,P)` returns the largest value of X for all solutions of P according to the total order Q. This function is non-strict, *i.e.*, it does not evaluate its arguments. Calls to bagof may be nested to any level. There is no existential quantification.

Example 8.6 This example illustrates `bestof`.

```
> p({1;-2;3;-4;5}).

> A=bestof(X,>,p(X))?

*** Yes
A = 5, X = @.
--1> B=bestof(X,<,p(X))?

*** Yes
A = 5, B = -4, X = @.
```

- The predicate `residuate(X,P)` attaches the predicate P to the term X. If X is touched then P will be executed. P is executed only once.

Example 8.7 This example illustrates `residuate`.

```
> p(A) :- write(A), nl.
> residuate(A,p(A))?

*** Yes
A = @~.
--1> A=B?   % Touch A
@           % Write the value of A

*** Yes
```

The predicate `residuate` is helpful to set breakpoints on variables when debugging, *i.e.*, to perform an action when a variable is touched.

- The predicate `mresiduate(L:[X1,X2,...],P)` attaches the predicate P to all the terms in the list L. If one of the terms is touched, then P is executed. P is executed only once, even if more than one of the terms is touched.

Example 8.8 This example illustrates `mresiduate`.

```
> p(A) :- write(A), nl.
> mresiduate([A,B],(p(A),p(B)))?

*** Yes
A = @~, B = @~.
--1> A=2?       % Touch A
2
@

*** Yes
A = 2, B = @.   % All residuations are gone
----2> <CR>     % Go back to previous level

*** No
A = @~, B = @~. % Residuations reappear
--1> B=6?       % Touch B
@
6
```

```
*** Yes
A = @, B = 6.   % All residuations are gone
```

- The predicate `implies(G)` calls the goal G using matching on the heads of the clauses defining G. If the calling argument implies the head of the clause, then the match succeeds and the clause body is called. Otherwise, the match fails, and the next clause is called. Backtracking to a successful `implies` call will fall through to the following clauses, just like a standard predicate call. Execution is identical to that of a standard predicate except that matching is used instead of unification to see whether a clause is entered. See Section 12.10 for an illustration of the use of `implies` in the graphical interface toolkit.

Example 8.9 This example illustrates `implies`.

```
> p(X:int) :- write('X=',X),nl.

*** Yes
> X={1;int;real}, implies(p(X)), fail?
X=1
X=int

*** No
```

8.2 Ψ-term manipulation

These functions are useful for explicitly building and taking apart ψ-terms.

- The predicate `A=B` unifies the two ψ-terms A and B. It succeeds if the terms are unifiable and fails if they are not. Writing `X:s(...)=Y:s(...)` amounts to the same as writing the predicate call `eq(X:s(...),Y:s(...))` if the predicate `eq` is declared by the clause `eq(U,U)`.
- The function `A&B` unifies the two ψ-terms A and B. It returns the unified value. It succeeds if the terms are unifiable and fails if they are not. Writing `X:s(...)&Y:s(...)` amounts to the same as writing the functional expression `eqfun(X:s(...),Y:s(...))` if the function `eqfun` is declared by the rule `eqfun(U,V)-> U | U=V`.
- The function `A.F` (project) selects feature F of ψ-term A. F may be passed as a string, an integer, or a sort. This generalizes record field selection of imperative languages. If the field does not exist, then it is created. If F is `@`, then the function residuates until the field name appears.

Example 8.10 This example illustrates project.

```
> A=s(a,b)?        % A has two fields named 1 and 2

*** Yes
A = s(a,b).
--1> B=A.2?        % Select the field 2

*** Yes
A = s(a,B), B = b.
----2> C=A.B?      % Create the field 'b'

*** Yes
A = s(a,B,b => C), B = b, C = @.
```

- The function `has_feature(F,X)` returns `true` if the term X has the feature F and `false` otherwise. This function does not residuate and hence must be used with caution. F may be passed as a string, an integer, or a sort.
- The function `features(X,M:string)` returns a list containing all the features attached to the root of term X which are visible in module M (see Section 10, page 66). The order of the features is unspecified. The module name M may be omitted; in that case the module in which the call to `features` textually occurs is taken. This function does not residuate and hence must be used with caution.

Example 8.11 This example illustrates `features`.

```
> A=features(thing(a,b,c,
|                    next => computer,
|                    age => 16,
|                    apple => worms))?

*** Yes
A = [1,2,3,age,apple,next].
```

- The function `root_sort(A)` returns the root sort of the term A. The result is a copy of A without arguments. This function does not residuate and hence must be used with caution.

Example 8.12 This example illustrates `root_sort`.

```
> A=root_sort(abc(1,2,3)),B=root_sort([a,b,c]),
|     C=root_sort(5.67)?

*** Yes
A = abc, B = cons, C = 5.67.
```

Lists are represented with the sorts `cons` and `nil`. The lists `[A|B]` and `[]` are parsed identically to `cons(A,B)` and `nil`.

- The function `strip(X)` yields a ψ-term whose sort is `@` and which has the attributes of X. The result has the identically same attributes, *i.e.*, the attributes are not copied. This function does not residuate and hence must be used with caution.

Example 8.13 This example illustrates `strip`.

```
> A=strip(siren(noise => loud,
|                 sound => unpleasant,db => 120))?

*** Yes
A = @(db => 120,noise => loud,sound => unpleasant).
```

Example 8.14 This is an example that shows `strip` does not copy the arguments of the stripped term.

```
> Y=strip(X:foo(2=>bar))?

*** Yes
Y = @(2 => _A: bar), X = foo(2 => _A).
```

- The function `copy_pointer(X)` yields a ψ-term whose sort is `root_sort(X)` and which has the attributes of X. The result has the identically same attributes, *i.e.*, the attributes are not copied. This function does not residuate and hence must be used with caution.
- The function `copy_term(X)` returns a copy of the ψ-term X. The copy has no subterms in common with X and has identical structure to X. So `copy_term(X:f(X))=A:f(A)`. The copy is not persistent (see Section 9). This function does not residuate and hence must be used with caution.

- The function `X===Y` (identity) yields `true` if the arguments X and Y are the same term, *i.e.*, if they have been explicitly unified together. Otherwise it returns `false`. This function does not residuate and hence must be used with caution.
- The function `X\===Y` yields `true` if the arguments X and Y are different terms, *i.e.*, if they have not been unified together. Otherwise it returns `false`. This function does not residuate and hence must be used with caution.

8.3 Arithmetic

8.3.1 Arithmetic calculation

These functions coerce the sort of their arguments and results to `real`. Many of the functions can be inverted. A local inversion is performed if the inversion is deterministic, *i.e.*, if only one solution is possible. For example, executing the goal `A=A/5` binds A to 0. The bit manipulation functions (`A/\B`, `A\/B`, `A>>B`, `A<<B`, `\(A)`) are not invertible. Executing the goal `25=A*A` residuates since there are two solutions 5 and –5. Applying a function followed by its inverse may not result in an identical value because of floating point roundoff error.

- The function `A+B` returns the sum of A and B.
- The function `A-B` returns the difference of A and B. The function may also be called with a single argument as `-(A)` in which case it returns the negative of A. The function does not curry in this case. The symbol `-` is defined as a unary operator which makes parentheses unnecessary when called with a single argument.
- The function `A*B` returns the product of A and B.
- The function `A/B` returns A divided by B. This is a floating point division.
- The function `A//B` returns the integer part of A divided by B, *i.e.*, the integer between 0 and A/B that is closest to A/B. The arguments A and B must be integers.
- The function `floor(A)` returns the largest integer less than or equal to A.
- The function `ceiling(A)` returns the smallest integer greater than or equal to A.
- The function `A/\B` returns the bitwise "and" of A and B. The arguments A and B must be integers. The low word (32 bits) of the result is valid. This function is not invertible.
- The function `A\/B` returns the bitwise "or" of A and B. The arguments A and B must be integers. The low word (32 bits) of the result is valid. This function is not invertible.
- The function `A>>B` returns the arithmetic (signed) right shift of A by B places. The arguments A and B must be integers. The low word (32 bits) of the result is valid. This function is not invertible.
- The function `A<<B` returns the arithmetic (signed) left shift of A by B places. The arguments A and B must be integers. The low word (32 bits) of the result is valid. This function is not invertible.

- The function `\A` returns the bitwise negation of A. The argument A must be an integer. The bitwise negation of the low word of A (32 bits) is taken and is sign-extended for the result. This function is not invertible.
- The function `sqrt(A)` returns the positive square root of A. It is invertible between the domain $]0, \infty[$ and the range $]0, \infty[$.
- The function `exp(A)` returns e^A where e is the base of the natural logarithms. It is invertible between the domain $]-\infty, \infty[$ and the range $]0, \infty[$.
- The function `log(A)` returns the natural logarithm of A. It is the inverse of `exp(A)`. It is invertible between the domain $]0, \infty[$ and the range $]-\infty, \infty[$.
- The function `sin(A)` returns the sine of the angle A, where A is expressed in radians. It is invertible between the domain $[-\pi/2, \pi/2]$ and the range $[-1, 1]$.
- The function `cos(A)` returns the cosine of the angle A, where A is expressed in radians. It is invertible between the domain $[0, \pi]$ and the range $[-1, 1]$.
- The function `tan(A)` returns the tangent of the angle A, where A is expressed in radians. It is invertible between the domain $]-\pi/2, \pi/2[$ and the range $]-\infty, \infty[$.
- The function `random(N:int)` returns a pseudo-random integer in the range $0, 1, ..., (N-1)$, where N is a positive integer. This function residuates if N is not a specific integer.
- The predicate `initrandom(S:real)` initializes the pseudo-random number generator with the seed S. This is useful when it is necessary to generate the same sequence of pseudo-random numbers repeatedly. Upon system startup, the pseudo-random number generator is initialized by calling `initrandom` with the current time.
- The function `genint` returns a new nonnegative integer each time it is called. It is guaranteed that all calls to `genint` during a run of Wild LIFE will return different integers.

8.3.2 *Arithmetic comparison*

These functions narrow the sort of their arguments to `real` and the sort of their result to `bool`. Their names are identical to Prolog's arithmetic comparisons. They will residuate if their arguments are not actual numbers in the sort hierarchy. They can be reversed in a few simple cases.

- The function `A>B` returns `true` if A is greater than B.
- The function `A>=B` returns `true` if A is greater than or equal to B.
- The function `A<B` returns `true` if A is less than B.
- The function `A=<B` returns `true` if A is less than or equal to B.

- The function `A=:=B` returns `true` if A is equal to B. This is different from = (the unification predicate) or `&` (the unification function). The comparison `=:=` is an arithmetic function which does not unify its arguments, always succeeds and returns `true` or `false`. Nevertheless, constraint lifting entails that if `(A=:=5)=true` then `A=5`.

- The function `A=\=B` returns `true` if A is not equal to B. This function unifies values in one case: `(A=\=5)=false`, where A will be unified with `5`.

8.3.3 Boolean arithmetic

These functions perform calculations on boolean terms, *i.e.*, terms whose values are the sorts `true` or `false`. The functions perform all possible local propagations. For example, calling `B=A and false` will cause B to be unified with `false` and calling `B=A and true` will cause B to be unified with A. The functions residuate if a unique result cannot be determined.

- The function `A and B` returns the logical "and" of A and B.

- The function `A or B` returns the logical "or" of A and B.

- The function `A xor B` returns the logical "exclusive or" of A and B.

- The function `not A` returns the logical "not" of A. It returns `true` if A is `false` and `false` is A is `true`.

8.4 Sorts

8.4.1 Sort calculation

These routines provide the means to calculate with sorts in the sort hierarchy and to navigate in the sort hierarchy. All functions that return sorts will return them in quoted form, so that properties attached to them are not executed.

- The function `glb(A,B)` returns the greatest lower bound of the root sorts of A and B in the current sort hierarchy. This function does not residuate and hence must be used with caution. See Section 4.1.3 for a full presentation of the glb operation.

- The function `lub(A,B)` returns the least upper bound of the root sorts of A and B in the current sort hierarchy. The lub is the dual operation to the glb. This function does not residuate and hence must be used with caution.

Example 8.15 This example illustrates the glb and lub built-in functions. It shows that both glb and lub create a disjunction if the result is a disjunctive sort.

```
> a <| c.
> a <| d.
> b <| c.
> b <| d.

> A=glb(c,d)?  % Result is disjunctive sort {a;b}
```

```
*** Yes
A = a.          % First solution
--1> ;

*** Yes
A = b.          % Second solution
--1> ;

*** No
> A=lub(a,b)?   % Result is disjunctive sort {c;d}

*** Yes
A = c.          % First solution
--1> ;

*** Yes
A = d.          % Second solution
--1> ;

*** No
```

- The predicate `subsort(A,B)` continuously enforces the relation `A:=<B`. It guarantees that the root sort of A is less than or equal to the root sort of B. `subsort` checks the relation when it is called, and then attaches itself as a residuation to B. Whenever the sort of B is refined, it is checked to be not lower than A.
- The function `children(S)` returns a list of the declared sorts that are immediately below S in the current sort hierarchy. The order of the sorts in the list is unspecified. Three built-in sorts (`int`, `real`, and `string`) have an infinite number of children. These are not enumerated, *i.e.*, `children(int)=[]`, `children(real)=[int]`, and `children(string)=[]`. This function does not residuate and hence must be used with caution.
- The function `parents(S)` returns a list of the declared sorts that are immediately above S in the current sort hierarchy. The order of the sorts in the list is unspecified. This function does not residuate and hence must be used with caution.

Example 8.16 This example illustrates the parents and children built-in functions.

```
> a <| c.
> a <| d.
> b <| c.
> b <| d.
```

```
> X=parents(a), Y=children(c)?

*** Yes
X = [c,d], Y = [a,b].
--1>

*** No
> X=parents(c), Y=children(a)?

*** Yes
X = [@], Y = [].
```

- The function `least_sorts` returns a list of the sorts that are immediately above {} in the current sort hierarchy. The order of the sorts in the list is unspecified.

8.4.2 Sort comparison

These functions perform sort comparisons and return `true` or `false`. The comparisons are done with the root sorts of the arguments. Because the sort hierarchy is a mathematical lattice, sort comparisons do not satisfy trichotomy (*i.e.*, it is possible for none of `A:<B`, `A:==B`, and `A:>B` to be true). As a result, there are more than six possible comparisons. Twelve of the 30 possible comparisons have been given a name.

The names of all sort comparisons start with a "`:`". The names are consistent with the names of the arithmetic comparisons, except for sort equality and nonequality which use == and \== instead of =:= and =\=.

These functions do not residuate and hence must be used with caution. They can be made order-independent by wrapping them in another function definition.

- The function `A:>B` returns `true` if A is greater than B in the sort hierarchy.
- The function `A:>=B` returns `true` if A is greater than or equal to B in the sort hierarchy.
- The function `A:<B` returns `true` if A is less than B in the sort hierarchy.
- The function `A:=<B` returns `true` if A is less than or equal to B in the sort hierarchy.
- The function `A:==B` returns `true` if A is equal to B in the sort hierarchy.
- The function `A:><B` returns `true` if A is comparable to B in the sort hierarchy, *i.e.*, their intersection is non-empty.
- The function `A:\>B` returns `true` if A is not greater than B in the sort hierarchy.
- The function `A:\>=B` returns `true` if A is not greater than or equal to B in the sort hierarchy.
- The function `A:\<B` returns `true` if A is not less than B in the sort hierarchy.

- The function `A:\=<B` returns `true` if A is not less than or equal to B in the sort hierarchy.

- The function `A:\==B` returns `true` if A is not equal to B in the sort hierarchy.

- The function `A:\><B` returns `true` if A is not comparable to B in the sort hierarchy, *i.e.*, their intersection is empty.

8.5 Strings

A string is a sequence of characters. Strings are provided as a separate sort in Wild LIFE. Compared to storing character sequences as root sorts of new ψ-terms, strings use less memory and they are not interned in the symbol table. The following built-ins are provided to manipulate strings.

8.5.1 String calculation

- The function `strlen(S:string)` returns the length of S. S must be a string. So `strlen("abcdef")=6`. This function residuates on S, *i.e.*, it waits until S is bound to a specific string.

- The function `substr(S:string,P:int,N:int)` (substring or string extraction) extracts N characters from string S starting from the character at position P. So `substr("abcdef",2,4)` yields `"bcde"`. It truncates the output if it would go beyond the end of S. So `substr("abcdef",5,4)="ed"`. This function residuates on S, P, and N.

- The function `strcon(S1:string,S2:string)` (string concatenation) returns the string obtained by appending S2 to the end of S1. So `strcon("abc","def")="abcdef"`. This function residuates on S1 and S2.

- The function `str2psi(S:string,M:string)` (string to ψ-term conversion) returns the ψ-term whose name consists of the same sequence of characters as S, and which is current to module M (see Section 10, page 66). The module name M may be omitted; in that case the module in which the call to `str2psi` textually occurs is taken. So `str2psi("foo")=foo` and `str2psi("34")='34'`. The latter is not the integer 34, but a symbol. This function residuates on S.

- The function `psi2str(X)` (ψ-term to string conversion) returns the string whose name consists of the same sequence of characters as the root sort of X. So `psi2str(int) = "int"`. This function never residuates. Hence, it is order-dependent and should be used with caution. It can be made order-independent by wrapping it in another function definition. The following definition waits until X:=<foo before firing: `psi2str_foo(X:foo) -> psi2str(X).`

- The function `asc(S:string)` returns the ASCII code of the first character of the string S. An error message is reported if S is not a string. This function residuates on S.

- The function `chr(I:int)` returns a string of length one containing the character whose ASCII code is I. An error message is reported if I is not an integer. This function residuates on I.

8.5.2 String comparison

The following functions perform string comparisons and return `true` or `false`. They will residuate if their arguments are not below `string` in the sort hierarchy. The names of all string comparisons start with "`$`". The names are consistent with the names of the arithmetic comparisons, except for string equality and nonequality which use == and \== instead of =:= and =\=.

- The function `A$>B` returns `true` if A is greater than B.
- The function `A$>=B` returns `true` if A is greater than or equal to B.
- The function `A$<B` returns `true` if A is less than B.
- The function `A$=<B` returns `true` if A is less than or equal to B.
- The function `A$==B` returns `true` if A is equal to B.
- The function `A$\==B` returns `true` if A is not equal to B.

8.6 Type checking

These boolean functions test whether an identifier is of a particular kind. These functions never residuate. Hence, they are order-dependent and should be used with caution. They can be made order-independent by wrapping them in another function definition.

- The function `is_function(X)` returns `true` if X is a function.
- The function `is_predicate(X)` returns `true` if X is a predicate.
- The function `is_sort(X)` returns `true` if X is a declared sort. A declared sort is one that has occurred in a `::` or <| declaration.
- The function `var(X)` returns `true` if X is `@` with no arguments.
- The function `nonvar(X)` returns `true` if X is not `@` or it has arguments, or both.
- The function `is_persistent(X)` returns `true` if X is a persistent term. See Section 9.

8.7 Input/output

In Wild LIFE, as in Prolog, input and output operations are extra-logical in that they have side-effects. Namely, Wild LIFE does not, upon backtracking, retrieve information that has been sent to an output port, or put back characters read from an input stream. Therefore good style dictates that input/output be avoided in the main body of a program.

Unless explicitly set otherwise by the user, all reading and writing is done from and to streams bound to the standard Unix I/O file descriptors `stdin` and `stdout`. We first describe the built-ins for reading and writing on the currently selected I/O stream. Then we describe how to modify the current stream.

8.7.1 Reading

The following built-ins are provided for reading from the current stream.

- The predicate `get(C)` reads the next character off the current input stream and unifies its ASCII code (an unsigned integer) with its argument C. If that next character is the end-of-file character `CTRL-D`, then `get` unifies its argument with the symbol `end_of_file`. Be mindful of the fact that if the argument passed has a sort other than `@` then when the end of file marker is reached, the predicate will simply fail (as `end_of_file` cannot unify with `string` for instance).
- The predicate `read(X)` reads a ψ-term off the input stream, quotes it, and unifies it with its argument X. The ψ-term being read must be properly terminated. The ψ-term being read must be consistent with the current set of operator declarations. Variable names appearing in the input stream are ignored.

Example 8.17 This example illustrates `read`.

```
> read(R)?
f(X,g(Y,X)).          % Typed in by the user

*** Yes
R = f(_A,g(@,_A)).
```

- The predicate `read_token(X)` reads one syntactic token from the input stream, quotes it, and unifies it with X. A token is either a symbol, a number, a string, a variable, any non-alphanumeric built-in or declared operator, or delimiters (parenthesis, bracket, brace, period, *etc.*). Variable names appearing in the input stream are ignored.

Example 8.18 This example illustrates `read_token`.

```
> read_token(S)?
foo                   % Typed in by the user

*** Yes
S = foo.
--1> read_token(T)?
S                     % Typed in by the user
```

```
*** Yes
S = foo, T = @.
```

- The predicate `load`(A_1, A_2, ...) loads all the definitions in the files A_1, A_2, *etc.*, in the order they appear. This predicate is non-strict, *i.e.*, it does not evaluate its arguments. Queries appearing in the files being loaded are proved as they are encountered and therefore take into account only definitions that textually occur before them. If one file is not found, then the remaining files are not loaded, an error message is reported, and the `load` fails. An error-free load always succeeds, even if queries in the loaded file fail. If a query appearing in a file fails then the remainder of the file following it is loaded anyway. A file being loaded may itself contain a `load` query. The outermost `load` is then momentarily "set aside" and the nested one is executed in the context reached thus far. If no error occurs, the outermost `load` is then resumed in the augmented context. Cyclic loadings are ignored, *i.e.*, a file is only loaded once during the scope of a `load`, even if it occurs in more than one `load` query.

- The function `load_path` may be defined by the user to return a directory name or a disjunctive term containing directory names. A directory name is given as a string, *e.g.*, `"/usr/lib/include"` is a valid directory name. During a `load` or `import` command, the current directory is searched first, followed by the directories returned by `load_path`, followed by the Lib, Tools, and Examples directories.

8.7.2 Writing

The following built-ins are provided for writing to the current stream.

- The predicate `put(C:int)` takes the integer ASCII code C of a character and outputs it to the current output stream. It is a dual to the `get` predicate.

- The predicates `write`(A_1, A_2, ...) and `pretty_write`(A_1, A_2, ...) print the ψ-terms A_1, A_2, *etc.*, according to the current operator declarations. The arguments are printed in lexicographical order of feature names, where all integer features are considered to be less than all non-integer features. The difference between the two predicates is that `pretty_write` will break up and indent a ψ-term if the output does not fit on a single line.[14] No line feed is issued after either of these two predicates.

- The predicates `writeq`(A_1, A_2, ...) and `pretty_writeq`(A_1, A_2, ...) print the ψ-terms A_1, A_2, *etc.*, according to the current operator declarations. The ψ-terms are printed with all necessary single and double quotes so that the ψ-terms may be read in again with `read`.

- The predicate `write_canonical(X)` prints its arguments in a canonical form. without operator declarations, and adding single and double quotes where necessary so that

[14] This makes `pretty_write` run marginally slower and use more memory.

the arguments may be read in again with `read`. This predicate is useful for information interchange between programs that do not necessarily have the same operator declarations.

- The predicate `nl` goes to the beginning of the next line, *i.e.*, it has the effect of printing a carriage return and line feed.
- The predicate `page_width(N)` changes the default page width to N. By default, the pretty-printer invoked by `pretty_write` uses a constant page width set to 80 characters. This value is changed dynamically to the number of characters $max(0, N)$. The pretty-printer is also used by the user interface to show the bindings of variables when a goal succeeds. N must be a nonnegative integer or an unbound variable. If N is an unbound variable, it is bound to the current page width.
- The predicate `print_depth(N)` changes the default maximum print depth to N. Only the first N levels of a ψ-term will be printed entirely. The rest is printed as "`...`" (three dots). N must be a nonnegative integer or an unbound variable. If N is an unbound variable, it is bound to the current print depth.

Since a ψ-term to be written may contain cycles, the Wild LIFE printer explores it entirely to detect any cycles encountered. It then prints the term, generating new tag names to reference identical terms which appear several times (and thus cycles, in particular). The tags generated are local to each call of the printer and are of the form _α where α is a capital letter, or a sequence of same, generated in alphabetical order; that is, `_A`, `_B`, `_C`, ..., `_Z`, `_AA`, `_AB`, *etc.*

Example 8.19 This example illustrates `page_width`.

```
> page_width(60)?

*** Yes
> big(one(abc,X:def,fgh(a,b,c,d),ijk,lmn(e,f,g(h,i,j),
|     h,i(12,23,34,45)),[a,b,c,d],[a,b](3=>10000),[X|X])).

*** Yes
> big(A),nl,write(A),nl,nl,pretty_write(A),fail?

one(abc,_A:def,fgh(a,b,c,d),ijk,lmn(e,f,g(h,i,j),h,i(12,23,3
4,45)),[a,b,c,d],cons(a,[b],10000),[_A|_A])

one(abc,
    _A:def,
    fgh(a,b,c,d),
    ijk,
    lmn(e,f,g(h,i,j),h,i(12,23,34,45)),
    [a,b,c,d],
    cons(a,[b],10000),
    [_A|_A])
*** No
```

Example 8.20 This example illustrates `print_depth`.

```
> print_depth(0)?

*** Yes
> X=f(f(f(f(a))))?

*** Yes
X = f(...).
--1> print_depth(1)?

*** Yes
X = f(f(...)).
--1> print_depth(2)?

*** Yes
X = f(f(f(...))).
--1> print_depth(3)?

*** Yes
X = f(f(f(f(...)))).
--1> print_depth(4)?

*** Yes
X = f(f(f(f(a)))).
```

Example 8.21 This example illustrates the difference between `write` and `writeq`. The former prints terms without quotes, the latter keeps all single and double quotes.

```
> write('f o o')?
f o o
*** Yes
> writeq('f o o')?
'f o o'
*** Yes
> write("f o o")?
f o o
*** Yes
> writeq("f o o")?
"f o o"
*** Yes
```

Example 8.22 This is an example of a self-reproducing query. It uses `writeq` and `write` to reproduce its input exactly.

```
> '=X,writeq(X),write(X)?'=X,writeq(X),write(X)?
'=X,writeq(X),write(X)?'=X,writeq(X),write(X)?
*** Yes
X = '=X,writeq(X),write(X)?'.
```

Example 8.23 This is a shorter self-reproducing query.

```
> X: (write(X), put(63))?
X: (write(X), put(63))?
*** Yes
X = (write(X), put(63)).
```

This works because the query is itself a ψ-term, and hence may be cyclic. A question for the reader: is an even shorter self-reproducing query possible? [15]

8.7.3 Parsing a string

The function `parse(S:string,X,V)` returns the ψ-term resulting from parsing the string `S` according to the current operator declarations. It sets the flag `X` to either `query`, `declaration`, or `error` depending on the status of the parse. It sets the flag V set to `true` if the parsed form contains variables, and `false` otherwise. Parse never fails. Variable names that appear in S are significant and interpreted in the current context. The call `parse("1+2.")` returns the quoted ψ-term `1+2`. The string S must be properly terminated, *i.e.*, it must end with "?" or ".".

Example 8.24 This example illustrates `parse`.

```
> X=1?

*** Yes
X = 1.
--1> F=parse("X+3.",S), G=eval(F)?
```

[15]The answer is yes; find it!

```
*** Yes
F = X + 3, G = 4, S = declaration, X = 1.
----2>

*** No
X = 1.
--1> F=parse("X+3",S,V)?

*** Yes
F = X + 3, S = error, V = true, X = 1.
```

The first call to `parse` correctly returns the ψ-term `1+3`. In the second call to `parse`, the binding `S=error` is due to the missing period at the end of the string `"X+3"`. That is, `parse` reports a *syntactic* error, not a semantic one.

8.7.4 Operator declarations

In the same manner as Prolog, Wild LIFE allows run-time modification of input/output syntax through operator declarations. The predicate `op(P,K,N)` declares that the operator named N exists with precedence P and kind K. This predicate is non-strict, *i.e.*, it does not evaluate its arguments. The precedence must be an integer from 1 to 1200. The kind must be one of `xf`, `yf`, `fx`, `fy`, `xfx`, `xfy`, and `yfx`. This predicate may be used to inspect or to create new operators. See appendix B (page 102) for a list of all predefined operators in the system. The predefined operators are as compatible with ISO Standard Prolog as the language allows.

Example 8.25 This example illustrates user-defined operators.

```
> op(1000,fx,if)?
> op(900,xfx,then)?
> op(500,fx,go)?
> op(600,fx,the)?
> op(700,xfx,is)?
> op(1200,xf,quickly)?

> write(if the weather is nice then go swimming quickly)?
if the weather is nice then go swimming quickly
*** Yes
```

Example 8.26 Return the list of all operators whose precedence is less than the precedence of `*`, *i.e.*, which bind tighter than `*`.

```
> A=bagof(Z,(op(X,3=>(*)),op(Y,3=>Z),Y<X))?

*** Yes
A = [^,&,mod,`,:,.], X = @, Y = @, Z = @.
```

☉☉ The `op` predicate recognizes keywords, and therefore the three calls `op(X,Y,Z)`, `op(precedence => X, kind => Y, functor => Z)`, and `op(X,Y,Z,precedence => X, kind => Y, functor => Z)` are equivalent. Because the head is a ψ-term, any mixed or incomplete list of arguments may be specified, even `op(precedence => X, 3 => Z)`. In addition, unlike Prolog's `op`, which is a *declaration*, Wild LIFE's `op` can be dually, and indifferently, used in *both* assertion *and* query mode. Thus, `op(precedence => X, kind => xfy, functor => Z)` will successively bind X and Y to all pairs of precedence weights and functor symbols of kind `xfy`. This is quite handy to define new operators' precedences relatively to known ones as functions (or constrained by) the defined values, all without being aware of their actual specific (and often arbitrary) values.

8.7.5 Files and streams

File I/O operations in Wild LIFE are reduced to a simple form: files may be opened in either read or write mode and closed. It is possible to open several files for input and/or output with `open_in` and `open_out`, and to switch between them with `set_input` and `set_output`.

For consistency, all trace messages are always output to `"stdout"`, all errors and warnings to `"stderr"`, and all program output to the stream selected by the user. The default initial selections are `"stdin"` for input and `"stdout"` for output.

The following built-in predicates provide file I/O operations.

- The predicate `exists_file(F)` succeeds if and only if the file `F` can be opened for input. This is quiet and so can be used for testing the presence of a file.

- The predicate `open_in(F,S:stream)` opens the file `F` for input and selects the stream `S` as current input. The argument `S` is set to a ψ-term whose root sort symbol is `stream` containing all information relative to the status of reading from this file. This predicate fails with an error message if file F cannot be opened.

- The predicate `open_out(F,S:stream)` opens the file `F` for output and selects the stream `S` as current output. The argument `S` is set to a ψ-term containing all information relative to the status of writing into this file. This predicate fails with an error message if file F cannot be opened.

- The predicate `set_input(S:stream)` sets the input stream to `S`. The stream `S` must have been initialized by `open_in`.

- The predicate `set_output(S:stream)` sets the output stream to `S`. The stream `S` must have been initialized by `open_out`.

- The predicate `close(S:stream)` closes the stream `S` which must have been initialized by `open_in` or `open_out`. It also flushes the corresponding I/O buffer. If `S` was an input stream, `close(S)` automatically selects `stdin` as current input stream. It behaves analogously for output streams and `stdout`. All streams are closed on normal termination of Wild LIFE.

All file handling built-ins (`load, import, open_in, open_out, exists_file`) require a string as filename argument.

Example 8.27 This example gives a short program for copying files.

```
copy_file(F1,F2) :-
    open_in(F1,S1),           % Open file F1
                              % Select it for input
    open_out(F2,S2),          % Open the target file
    open_out(stdout,S3),      % Set output stream to stdout
    write("Copying from """,F1,""" to """,F2,""" ... "),
    set_output(S2),           % Set the output to file F2
    repeat,
    get(X),                   % Read a character
    ( X=end_of_file,          % If end of file F1 is reached
      close(S1), close(S2),   % then close both files
      write("done."),         % (output is reset to stdout)
      !                       % and cut out the repeat loop
    ;                         % else
      put(X),                 % output the character
      fail
    ).                        % and fail to repeat loop
```

Here is an example of its use.

```
> copy_file(xxxooo,junk)?
Copying from 'xxxooo' to 'junk' ... done.
*** Yes
```

8.8 System-related built-ins

8.8.1 *The* Wild LIFE *system*

This section summarizes the built-ins that allow inspection and modification of Wild LIFE system-related properties.

- The predicate `listing`($A_1, A_2, ...$) lists the definitions of the identifiers A_1, A_2, *etc.*. This predicate is non-strict, *i.e.*, it does not evaluate its arguments. This built-in is quite useful to inspect a loaded program. It knows about functions, sorts (including the sort hierarchy), predicates, and global variables. For example, try the query `listing(23)?`.

- The predicate `verbose` toggles the Wild LIFE interpreter between two modes, quiet and verbose. The interpreter starts up in quiet mode. Verbose mode gives execution time statistics about each query and shows the garbage collections when they occur. The statistics given in verbose mode are the following.

Name	Meaning
cpu	CPU time of the query in seconds.
goals	Number of internal goals executed by the query.
stack	Size of backtrackable stack in bytes.
heap	Size of nonbacktrackable stack in bytes.
goals	Number of internal goals on goal stack.
choice points	Number of entries on choice point stack.
trail entries	Number of entries on trail stack.

Example 8.28 This example shows how to declare a function with variable arity, *i.e.*, the number of arguments is determined by each call.

```
> X:sum -> sum_all(features(X),X).
> sum_all([F|FL],X) -> X.F+sum_all(FL,X).
> sum_all([]) -> 0.

*** Yes
> verbose, A=sum(1,2,3,4,5)?
*** Verbose mode is turned on.

*** Yes  [0.000s cpu, 80 goals, 7916 stack, 196624 heap]
*** Stack depths [0 goals, 0 choice points, 1 trail entry]
A = 15.
```

- The predicate `statistics` prints information about the current memory usage of the system.
- The predicate `trace(X)` is used to enter and exit trace mode. In trace mode, all internal goals are printed to stdout. If X is `false` then trace mode is disabled. This is the default. If X is `true` then trace mode is enabled. If X does not exist, then trace mode is toggled.
- The predicate `step(X)` is used to enter and exit single-step mode. In single-step mode, an internal goal is printed and then the system waits for user input to continue. Pressing ⟨`CR`⟩ will execute one internal goal. Pressing an integer N followed by ⟨`CR`⟩ will execute N internal goals. If X is `false` then single-step mode is disabled. This is the default. If X is `true` then single-step mode is enabled. If X does not exist, then single-step mode is toggled. In single-step mode, type `h` to get online help.
- The predicate `abort` forces an immediate return to the Wild LIFE top level.
- The predicate `halt` forces an immediate termination of the Wild LIFE process.
- The predicate `gc` forces an immediate garbage collection. No messages are printed unless verbose mode is active.

8.8.2 The Unix system

This section summarizes the built-ins that allow Wild LIFE to interact with the Unix operating system.

- The function `system(S:string)` executes the command S under the shell `sh` and returns the resulting exit code. The function fails with an error message if a shell could not be created or if S is not a string.
- The function `getenv(S:string)` returns a string that contains the value of the environment variable S. The function fails if and only if the environment variable does not exist. The function fails with an error message if S is not a string.

Example 8.29 This example illustrates `system`.

```
> A=system("ls")?
DEMOS_README    dictionary.lf     flowers.lf    prime.lf
Flowers.doc     display_terms.lf  gauss.lf      queens.lf
Gauss.doc       flo_README        hamming.c     schedule.lf
Schedule.doc    flo_custom.lf     hamming.lf    simple.lf
SuperLint/      flo_flowerdef.lf  machine.lf    soap.lf
all_demos.lf    flo_gram.lf       magic.lf      solve.lf
boxes.lf        flo_utils.lf      nl.lf         xxxooo.lf
dictionary.c    flo_xtools.lf     palette.lf

*** Yes
A = 0.
--1>

*** No
> A=system("core")?
sh: core: cannot execute

*** Yes
A = 256.
```

- The function `argv` returns a list of the command line arguments given as strings.

Example 8.30 This example illustrates `argv`.

```
% wild_life apple pie
Wild_Life Interpreter Version 1.0
Copyright (C) 1991-93 DEC Paris Research Laboratory
No customizing file loaded.
```

```
> A=argv?

*** Yes
A = ["/udir/rmeyer/LIFE/MODULE/wild_life","apple","pie"].
```

8.8.3 Timekeeping

The following functions provide an interface to the flow of time.

- The function `cpu_time` returns the amount of user CPU time in seconds that the Wild LIFE process has used so far.
- The function `real_time` returns the amount of wall clock time (in seconds) that has elapsed since a particular moment of time in the past. The location of this moment of time is fixed for any given Unix system. The date of this moment depends on the particular Unix system.
- The function `local_time` returns a ψ-term that represents the local time. For example, the goal `write(local_time)` will write `time(day => 14,hour => 18,minute => 9,month => 10,second => 35,weekday => 3,year => 1992)` if the date is Wednesday, Oct. 14, 1992, 6:09:35 pm. The month field ranges from 0 to 11, where 0 represents January. The day field gives the day of the month and ranges from 1 to 31. The weekday field ranges from 0 to 6, with 0 representing Sunday. The hour field ranges from 0 to 23 (24-hour clock), where 0 represents midnight. The minute and second fields range from 0 to 59. This function is based on the underlying Unix date and time library.

8.9 Loading files with term expansion

The standard way in Wild LIFE to run a preprocessor on program rules is to define the preprocessor as a predicate. Program rules are written as queries to the preprocessor. For example, a DCG (Definite Clause Grammar) expander could be written by defining a predicate named `-->`, declaring it as an operator, and then writing clauses as calls to `-->`, namely as queries `Head --> Body?`.[16]

Because it is awkward to treat program rules as queries in this way, Wild LIFE provides a `term_expansion` facility. It generalizes the similarly named facility of Prolog. Term expansion is useful to expand terms into facts without having to write these terms as queries and without having to define their main functor as a predicate or a function. Term expansion is done only when loading files.

Example 8.31 This example illustrates how to use term expansion to provide a simple tracing facility.

[16] A preprocessor that does DCG expansion and much more is included in the system and described in appendix F.

```
> expand_load(true)?
> term_expansion((H:-B), (H:-write(root_sort(H)),nl,B)) :- !.
> load("/tmp/x")?
*** Loading File "/tmp/x.lf"

> listing(third)?

third(_A,_B) :-
        write(third),
        nl,
        _A = [@,@,_B|@].

*** Yes
```

This will trace the execution of `third`. The file `/tmp/x.lf` contains the single clause:

```
third(L, X) :- L=[_,_,X|_].
```

8.9.1 Defining term expansion clauses

The predicate `term_expansion(A,B)` expands the term A into the term or list of terms B. The term A may be anything at all, including a predicate declaration, a function declaration, or a sort declaration. The predicate `term_expansion` is dynamic and may be extended by the programmer.

Example 8.32 This example shows the initial definition of `term_expansion`.

```
Wild_Life Interpreter Version 1.0
Copyright (C) 1991-93 DEC Paris Research Laboratory
No customizing file loaded.
> listing(term_expansion)?

dynamic(term_expansion)?
% 'term_expansion' is a user-defined predicate
with an empty definition.

*** Yes
```

8.9.2 Using term expansion when loading files

The predicate `expand_load(A,W)` modifies the behavior of the `load` command with respect to term expansion. The default is not to do term expansion. `expand_load` has no effect on interactive input. The options A and W have the following effect:

- If the option A is `true`, then expand the terms and assert them. If A is `true` or `false`, the first option is set to the value of A. If A is a free variable then it is bound to the current value of the option.

- If the option W is `true`, then write the expanded rules and queries in a file with suffix "`.exp`". This file can later be loaded to avoid re-expanding all the rules. If W is `true` or `false`, the second option is set to the value of W. If W is a free variable then it is bound to the current value of the option.

These options are recursive: if `file1` is loaded with some options, and loads `file2`, then `file2` will be loaded with the same options. If the second option is set, the `load(file2)` query in `file1` will be rewritten as `load_exp("file2.exp")` in `file1.exp`. `load_exp` never does term expansion.

9 Global variables, persistent terms, and destructive assignment

This section covers three new concepts: *global variables*, *persistent terms*, and *destructive assignment*. These concepts are designed to provide clean and efficient replacements for most uses of `assert` and `retract`.

Global and persistent variable names are part of the name space that contains predicates, functions, and sorts. The same symbol cannot denote both a predicate and a global variable.

9.1 Global variables

A global variable is a logical variable whose name is visible throughout the program. To be precise, it is visible from all clauses within its defining module or any module to which it is exported. A global variable behaves exactly as if it were an extra parameter passed to all predicates, functions, and sorts. Global variables must be declared.

Example 9.1 This example illustrates global variables.

```
> global(warfare)?      % Declare a global variable

*** Yes
> warfare=34?           % Unify the variable with 34

*** Yes
--1> write(warfare)?    % Write the value
34
*** Yes
----2> .
> write(warfare)?       % Backtracking undoes the unification
@
*** Yes
```

Global variables are essentially syntactic sugar and a programming convenience. They should be used sparsely as program maintainability may suffer otherwise. Other than having a larger scope for its name, a global variable acts exactly like a local variable.

A good example of the use of a global variable (possibly combined with backtrackable destructive assignment) is to keep track of the reasoning used in some expert system, without having to explicitly pass an extra parameter around to all the predicates or functions used.

9.2 Persistent terms

A persistent term is a ψ-term that does not change its value on backtracking. It is "read-only." It may not be modified through unification and functions may not residuate on it. It can be modified only through explicit calls to nonbacktrackable assignment `<<-` (see Section 9.4.2). This can be viewed as having a global database (a set of graphs) with named entry points on certain nodes. All subterms of a persistent term are also persistent. Information may be shared between persistent terms.

Persistent terms are stored on the heap, just like clauses (see Section 8.8.1). Persistent terms cannot be unified together. They can be modified only through destructive assignment. All interaction between local terms and persistent terms is through *matching*. An error is reported if the match fails. Any attempt at unifying two persistent terms yields an error.

A persistent term may be stored in a standard variable. Modifications of the term are unaffected by backtracking. Access to the term through the standard variable is affected by backtracking: if one backtracks before the point in which the standard variable obtains access to the persistent term, then the standard variable gets its original value back and the persistent term becomes inaccessible. Its space is recovered by garbage collection.

9.3 Persistent variables

A persistent term may be stored in a global variable. The variable is then called a *persistent variable*. In this case, the value never becomes inaccessible. Parts of persistent terms may be shared between variables. Persistent variables must be declared. The `listing` predicate knows about global variables, but it does not (currently) differentiate between persistent and global variables.

Example 9.2 This example illustrates persistent variables.

```
> persistent(trouble)?      % Declare a persistent variable

*** Yes
> trouble <<- with_harry?   % Assign a value to the variable
                            % with destructive assignment
*** Yes
--1> write(trouble)?        % Write the value
with_harry
*** Yes
----2> .
> write(trouble)?           % Backtracking has no effect
with_harry
*** Yes
```

◕◕ The following commands:

```
> global(tactics)?
> tactics <<- retreat?
```

are not sufficient to make `tactics` a persistent variable because when backtracking beyond the point where `tactics` was bound to the persistent term the binding will be lost (in particular when returning to the top-level command line).

9.4 Destructive assignment

Wild LIFE provides a clean integration of destructive assignment in a single-assignment language. The integration is based on two kinds of terms: normal and persistent terms. The former are backtrackable, *i.e.*, they regain their former values on backtracking. The latter are nonbacktrackable, *i.e.*, changes to them are not undone on backtracking. Normal and persistent terms may be matched together. This results in a flow of information from the persistent to the normal term, never in the other direction. Any attempt to modify a persistent term except through its destructive assignment operation results in failure.

Normal and persistent terms each have their own destructive assignment operation. Therefore there are two kinds of destructive assignment: backtrackable and nonbacktrackable. Both of these are useful in real programs. See Section 12.7 (page 83) for a non-trivial example of the correct use of these two built-ins.

9.4.1 Backtrackable destructive assignment

The predicate `X<-Y` overwrites X with Y. X and Y are standard (backtrackable) ψ-terms. Backtracking past this statement will restore the original value of X. For example:

```
> X=5, (X<-6; X<-7; succeed), write(X), nl, fail?
6
7
5

*** No
```

This predicate is very useful for building “black boxes” that have clean logical behavior when viewed from the outside but that need destructive assignment to be implemented efficiently.

9.4.2 Nonbacktrackable destructive assignment

The predicate `X<<-Y` overwrites X with a *persistent* copy of Y. Modifications to X after it has been made persistent are not backtrackable. If you backtrack to a point before X is made persistent, then X is restored to its original (backtrackable) value. For example:

```
> X=5?

*** Yes
X = 5.
--1> X <<- 10?    % Make X persistent, with value 10

*** Yes
```

```
X = 10.
----2> X <<- 20?  % X gets value 20, nonbacktrackably

*** Yes
X = 20.
------3>          % Type <CR> to go back

*** No
X = 20.
----2>            % X is nonbacktrackably 20!

*** No
X = 5.            % X is restored to backtrackable 5
```

It is important not to confuse `<-` and `<<-`. The former can be used on the standard data structures in a program. The latter creates a special kind of data structure, the persistent term, which is useful for managing information that must not go away on backtracking. For example, the implementation of `bagof` uses persistent terms. Attempting to use `<-` on persistent terms results in an error.

The use of `A <<- B` where A is a local variable allows the creation of "temporary" persistent terms. They are temporary because the binding to them is lost on backtracking before the instant in which the variable became persistent.

Example 9.3 This example illustrates nonbacktrackable destructive assignment.

```
> persistent(this)?
> p :- write(this).    % 'this' is '@' when 'p' is defined
> this <<- q(a,b,c)?   % Assign a value to 'this'
> p?                   % Call 'p'
q(a,b,c)               % which prints the value of 'this'
*** Yes

> this=q(D,E,F)?       % Unify the persistent term 'q(a,b,c)'
*** Yes                % with the local term 'q(D:@,E:@,F:@)'
D = a, E = b, F = c, this = q(D,E,F).
                       % Succeeds since 'q(a,b,c) <| q(@,@,@)'
                       % D, E and F contain persistent terms
```

Example 9.4 This example illustrates that subterms of a persistent term are persistent.

```
> persistent(that)?
> that <<- thing(int)?
> that=thing(5)?
*** No                 % Fails since 'int |> 5'
```

```
> that=thing(X)?
*** Yes                  % Succeeds since 'int <| @'
X = int, that = thing(X).

--1> X=5?
*** No                   % Subterm 'int' of persistent term
                         % is persistent and 'int |> 5'
```

Example 9.5 This example defines an efficient version of `bagof` using <<-. It is efficient because <<- does incremental copying to the heap. That is, parts of the term that are already on the heap are not copied.

```
non_strict(local_bagof)?

local_bagof(X,G) -> M |
     L<<-[],
     ( evalin(G),                      % Prove G
       L<<-[evalin(X)|copy_pointer(L)], % Record X binding
       fail                            % Force backtracking
     ;
       M<-copy_term(L)                 % Copy persistent term
     ).                                % Back onto the stack
```

Both `bagof` and `local_bagof` execute in linear time.

Example 9.6 This example illustrates that on a persistent term, the function "." (project) will nonbacktrackably create a new feature if the required one was not present.

```
> persistent(this)?
> this<<-q(a,b,c), write(this)?
q(a,b,c)
*** Yes
--1> this.2=B?

*** Yes
B = b.              % B is bound to a persistent term
----2>

*** No
--1> this.new=B, write(this)?  % 'new' feature is added
q(a,b,c,new => @)
*** Yes
B = @, this = q(a,b,c,new => B).
```

```
----2>              % B is bound to a persistent term

*** No
--1> write(this)?
q(a,b,c,new => @)   % B no longer exists, but the
                    % 'new' feature still does
```

9.5 Quoting

Global and persistent variables can be quoted like functions (see Section 6.5, page 26). A quoted variable is not dereferenced. This allows global and persistent variables to be part of asserted predicates and functions.

Example 9.7 This example illustrates the use of quoting global variables when asserting a clause.

```
> global(it)?
> it=one_two_three, P=p(it), Q=q(`it),
|    assert(P), assert(Q)?

*** Yes
P = p(one_two_three), Q = q(it).
--1>

*** No              % Now 'it' is worth '@'
> p(X),q(Y),Z=it?

*** Yes
X = one_two_three, Y = @, Z = Y.
```

A listing shows the difference:

```
> listing(p,q)?

p(one_two_three) :- succeed.

q(it) :- succeed.
```

The same applies for persistent variables. Of course, for the difference to be obvious their value must be changed, and this can only be done with <<-.

9.6 Summary of built-ins

The following built-ins are provided for global variables and persistent terms.

- The predicate `global`(A_1, A_2, ...) declares A_1, A_2, *etc.*, as global variables. This predicate is non-strict, *i.e.*, it does not evaluate its arguments. Each argument A_i is either an uninterpreted identifier or a ψ-term of the form `V <- E` where V is an uninterpreted identifier and E an expression. The latter form initializes V with the evaluated result of E. If there is an error in any A_i then none of the A_i are declared. This predicate should be used only as a declaration, *i.e.*, in a query and not in a definition.

- The predicate `persistent`(A_1, A_2, ...) declares A_1, A_2, *etc.*, as persistent variables. This predicate is non-strict, *i.e.*, it does not evaluate its arguments. Each argument A_i must be an uninterpreted identifier. If there is an error in any A_i then none of the A_i are declared. This predicate should be used only as a declaration, *i.e.*, in a query and not in a definition.

- The predicates `A<-B` and `A<<-B` implement backtrackable and nonbacktrackable destructive assignment.

- The function `is_persistent(X)` returns `true` if X is a persistent term and `false` if X is a normal term, *i.e.*, X is backtrackable.

- The predicate `display_persistent(X)` is used to enter and exit a mode in which persistent terms are displayed differently from normal terms. This built-in is intended for debugging purposes. If X is `false` then persistent terms are displayed in the same manner as normal terms. This is the default. If X is `true` then persistent terms are preceded by a dollar sign "`$`". If X does not exist, then the display mode is toggled.

10 Modules

The module system creates an entirely separate set of symbols for each module. By *symbol* we mean any identifier (*i.e.*, a predicate, function, or sort) or feature name. The symbol name space is partitioned into three subspaces for predicate, function, and sort names. Feature names are in an independent space: a symbol may always be used as a feature name.

By *current module* we mean the module that determines the scope of the symbols at a particular time during execution. A current module exists at all times during program execution, both interactively and in a program.

A mechanism is provided which allows symbols to be accessed across modules. For a symbol to be visible outside of its defining module, it must be declared *public* in the module.

The syntax for an explicit reference to a given symbol defined in another module is `module#symbol`. Following standard terminology, we call this a *qualified* reference. For example, the syntax `built_ins#write` is legal if you are within module `"built_ins"`, or if `write` is declared public in module `"built_ins"` (which it is). If the symbol contains non-alphanumeric characters, then the reference becomes `'module#symbol'`.

It is not necessary to specify the module if the symbol you want to access is known in the current module. The standard way of doing this is to *open* the module you are interested in, with the built-in `open`. At that point, all public symbols appearing in the module are visible in your current module.

10.1 Standard modules

Four standard modules are defined:

- Module `"built_ins"`. This defines all built-in operations, including predicates, functions, and sorts.
- Module `"syntax"`. This defines the minimal symbols required for parsing LIFE files. This comprises operator declarations, the operator symbols themselves, and all non-alphabetic symbols related to parsing (such as `[`, `]`, `{`, `}`, `?`, and so forth).
- Module `"x"`. This contains all of the built-ins related to the X interface. A program using the X interface must first load and open it with the command `import("x")`. For an example of its use, see the X toolkit presented in appendix H.
- Module `"user"`. This is the current module at system startup. It is the default module for interactive input to the system.

Currently the modules `"syntax"` and `"built_ins"` are always open, so their symbols may be accessed without specifying a module name. There is no means to override this.

10.2 Using features

To make things easier for the LIFE programmer, features are public by default. If you want to have private features, then the predicate `private_feature` can be used in the same way as `private`. It is possible (but unwise) to define a feature as being private while the corresponding symbol is public.

The function `features` will only return those features which are visible from within the module the call appears in.

Example 10.1

```
> module("secret")?
> public(prison)?
> private_feature(entrance)?
> prison(entrance => tunnel).

> module("user")?
> open("secret")?

> P:prison(door => guarded)?

*** Yes
P = prison(door => guarded,entrance => tunnel).
--1> display_modules(true)?

*** Yes
P = secret#prison(door => user#guarded,
                  secret#entrance => secret#tunnel).
```

```
--1> F=features(P)?

*** Yes
F = [user#door],
P = secret#prison(door => user#guarded,
                  secret#entrance => secret#tunnel).
```

The feature `entrance` is private to `secret` and so when in module `user` the function `features` only sees the `door`. This is true for all built-ins that manipulate features.

10.3 Overloading

The module system allows symbols to be overloaded. This works because modules allow the distinction to be made between the new symbol and the old (typically built-in) definition.

Example 10.2

```
Wild_Life Interpreter Version 1.0
Copyright (C) 1991-93 DEC Paris Research Laboratory
No customizing file loaded.
> private(+)?
*** Warning: local definition of '+'
             overrides 'syntax#+'

> op(X,Y,'syntax#+'),op(X,Y,+)?
> A:list   + B:list   -> append(A,B).
> A:string + B:string -> strcon(A,B).
> A:real   + B:real   -> A 'syntax#+' B.

*** Yes
> write([a,b,c]+[d,e,f]),nl?
[a,b,c,d,e,f]

*** Yes
> write("abc"+"def"),nl?
abcdef

*** Yes
> write(3+4),nl?
7

*** Yes
```

This works but is only usable from within a single module. If you try to export the overloaded definition of +, a clash results between `syntax#+` and the exported +.

The following technique gets around the problem:

Example 10.3

```
> module("overload")?
> public(+)?
  (and so forth)

> module("charley")?  % Also opens 'syntax' and 'built_in'
> private(+)?         % Override 'syntax#+' locally
> open(overload)?
> alias(+,'overload#+')?
```

10.4 Summary of built-ins

- The predicate `module(M:string)` sets the current module to M. The argument M must be a string.[17] Typically, this predicate is put at the beginning of a file to create a new module. Setting the current module to the same module twice (or more) has no further effect. Whenever a file is loaded which switches to a different module, the current module reverts to what it used to be once the file is closed. This predicate should be used only as a declaration, *i.e.*, in a query and not in a definition.

- The function `current_module` returns the name of the current module. Upon startup, the current module is `"user"`.

  Example 10.4

  ```
  > X=current_module?

  *** Yes
  X = "user".
  --1> .
  > module("charley")?

  *** Yes
  charley> X=current_module?

  *** Yes
  X = "charley".
  charley--1> module("user")?

  *** Yes
  X = "charley".
  ```

[17] There is a good reason for this: if the argument were allowed to be any symbol, then the symbol is created in the module which was previously current. This is undesired behavior.

The prompt always shows the name of the current module, except if this module is `"user"`.

- The predicate `import`(A_1, A_2, ...) loads and opens the modules A_1, A_2, *etc.*, which are assumed to be in the files of the same name (modulo suffix and search path). This predicate is non-strict, *i.e.*, it does not evaluate its arguments. The arguments must be strings.[18] For example, `import` works correctly if the file is called `"/tmp/foo.lf"` and the module is called `"foo"`. If one of the files or modules does not exist, then the remaining files are not loaded, an error message is reported, and `import` fails. Cyclic loadings are ignored, *i.e.*, a file is only loaded once during the scope of an `import`, even if it occurs in more than one `import` query. This predicate should be used only as a declaration, *i.e.*, in a query and not in a definition.

- The predicate `public`(A_1, A_2, ...) declares the symbols A_1, A_2, *etc.*, as public. This predicate is non-strict, *i.e.*, it does not evaluate its arguments. They may then be accessed by other modules, either by a qualified reference (as `module#symbol`) or, if opened in the other module, simply as `symbol`. Typically, the `public` declarations are placed just after the `module` declaration, at the beginning of a module. This predicate should be used only as a declaration, *i.e.*, in a query and not in a definition.

  It is not possible to tamper with another module's private components. The call `public(built_ins#very_private_part)?` from within module `user` results in an error message.

- The predicate `private`(A_1, A_2, ...) declares the symbols A_1, A_2, *etc.*, as private. That is, from now on they are only accessible by qualified reference (as `module#symbol`). The `private` declaration is typically used to implement overloading (see Section 10.3). This predicate should be used only as a declaration, *i.e.*, in a query and not in a definition.

- The predicate `private_feature`(A_1, A_2, ...) declares the feature names A_1, A_2, *etc.*, as private. This is important because feature names are public by default. This predicate should be used only as a declaration, *i.e.*, in a query and not in a definition.

- The predicate `open`(A_1, A_2, ...) makes all the public symbols from the modules A_1, A_2, *etc.*, visible from within the current module without having to explicitly qualify them. The arguments must strings.

Example 10.5 This example illustrates opening a module.

```
> open("built_ins")?
> write("hello")?
hello
*** Yes
```

[18] This is necessary to prevent the filename symbol from being defined in more than one module.

Here the symbol `write` references `built_ins#write`.

- The predicate `display_modules(X)` toggles or switches the module display mode. If X is `false` then terms are displayed without module names. in the same manner as normal terms. This is the default. If X is `true` then terms are displayed with their module names. The system will then display `module#xxx` instead of `xxx`. This is very useful for debugging Wild LIFE programs, using the `listing` predicate. If X does not exist, then the module display mode is toggled.

Example 10.6 This example illustrates `display_modules`.

```
> display_modules(true), A=hello?

*** Yes
A = user#hello.
```

11 Rule-base management

The predicates in this section have been added for compatibility with Prolog. They may not be supported in the compiler. They should be used only when it is necessary to create or modify a *program* during execution. They should *not* be used for storing *data* (*i.e.*, ψ-terms). They should not be used if the program requires a global name to store a term, and/or if it is required that a term continue to exist on backtracking. Section 9 provides better solutions for both of these cases.

It is strongly discouraged to modify a routine during that routine's execution. The current release of Wild LIFE provides for immediate update semantics in certain cases (as given below). It does not implement the defensible semantics of [11].

11.1 Adding rules

The built-in predicates `assert(C)` and `asserta(C)` add the clause C to the program. The argument C may be of two forms, `(Head :- Body)` or simply `Head`, the latter being equivalent to `(Head :- succeed)`. The `Head` must be a dynamic predicate. With `assert`, the clause `C` will be added as the last rule to be tried for that predicate. With `asserta`, it is added as the first rule.

Example 11.1 This example illustrates `assert`.

```
> assert(q :- write(rule1)),assert(q :- write(rule2))?
```

```
*** Yes
> q?
rule1
*** Yes
--1> ;
rule2
*** Yes
```

When the last rule for a goal is being used, Wild LIFE does not create a choice-point for that goal, so if you add a new clause, it will not take effect, this is shown in the following example:

```
> p(1).

*** Yes
> p(A),write(A),assert(p(2)),fail?
1
*** No
```

The alternative, `p(2)`, was not considered because when the rule `p(1)` was used, there was no alternative clause at that time.

11.2 Deleting rules

The predicate `retract(C)` removes the first clause which unifies with C. If there are more than one, then backtracking will successively remove the others.

Example 11.2 It is possible to write the function `genint` which returns a new distinct symbol each time it is called in the following manner:

```
genint_counter(0).

:: genint(N) | retract(genint_counter(N)), M=N+1,
               assert(genint_counter(M)).
```

This is not the best way to write `genint`. A better way is to use a persistent term:

```
persistent(genint_counter)?
genint_counter<<-0?
genint -> copy_term(genint_counter) |
  genint_counter<<-genint_counter+1.
```

Example 11.3 Wild LIFE supports immediate update semantics. If you retract a clause which is currently being used then any currently active clause (one which is in the process of being executed or one which is reachable by backtracking) will not see the clause.

```
> p :- write(aha), retract(p:-B), retract(p:-C).
> p :- write(boo).
> p?
aha
*** Yes
--1> ;     % There are no further solutions to p

*** No
```

11.3 Inspecting rules

The predicate `clause(C)` unifies C with the first clause in the program that is unifiable with C. On backtracking, it will be unified with all successive clauses that are unifiable with C.

Example 11.4

```
> p(1).

*** Yes
> p(2) :- write(hello).

*** Yes
> clause(A:p :- B)?

*** Yes
A = p(1), B = succeed.
--1> ;

*** Yes
A = p(2), B = write(hello).
--1> ;

*** No
```

11.4 Function definitions

It is possible to use the above predicates (`assert`, `retract`, and `clause`) with function declarations. The argument is of the form `(Head -> Result)`. Again you have to be careful with choice-points, and new definitions will not modify previously existing ψ-terms which had already been evaluated.

Example 11.5

```
> A=f,assert(f->sdsd),B=f?

*** Yes
A = f, B = f.
```

The fact that predicates and functions are represented with ψ-terms makes it easy to write meta-interpreters. But the differences between various possible implementations of `assert` and `retract` are just a reminder of the inherent pitfalls of using self-modifying code. This code is very difficult to debug and is compiled much less efficiently. If you use these predicates often, you will find that Wild LIFE will spend quite a lot of its time collecting garbage.

11.5 Summary of built-ins

- The predicate `dynamic`($A_1, A_2, ...$) makes the routines A_1, A_2, *etc.*, dynamic, *i.e.*, they may be modified during execution. This predicate should be used only as a declaration, *i.e.*, in a query and not in a definition. This predicate is non-strict, *i.e.*, it does not evaluate its argument.

Example 11.6 Attempting to prove a goal of a predicate without clause definitions results in an error, unless the predicate has been declared `dynamic`, in which case it simply fails.

```
> foo?
*** Error: 'foo' is not a predicate or a function.

*** Abort
> dynamic(foo)?

*** Yes
> foo?

*** No
```

- The predicate `static`($A_1, A_2, ...$) makes the routines A_1, A_2, *etc.*, static, *i.e.*, they may not be modified during execution. Any attempt to do so results in an error. To modify them, they must be made dynamic with a dynamic declaration. This predicate should be used only as a declaration, *i.e.*, in a query and not in a definition. This predicate is non-strict, *i.e.*, it does not evaluate its argument.
- The predicate `assert(C)` asserts the clause C (of the form `(H:-B)` or `H`) or the function rule C (of the form `(H->B)`) at the end of the current definition. H should be instantiated to a legal function or predicate name.

- The predicate `asserta(C)` asserts C at the beginning of the current definition. Otherwise it behaves identically to `assert`.
- The predicate `clause(C)` unifies C with the first rule or clause that is unifiable with it. On backtracking, unify C with the successive rules or clauses that unify with it. C should have root sort `->` or `:-`.
- The predicate `retract(C)` unifies C with the first rule or clause that is unifiable with it. Remove this item from the program database. On backtracking, unify and remove with the successive rules or clauses that unify with C.
- The predicate `setq(H,E)` replaces the definition of function H by a function containing the single rule `H->V` where V is the result of evaluating the expression E. If H is a predicate name or a declared sort then an error message is given.

  ◑◑ `setq` is obsolete and will not be supported in the compiler. Persistent variables provide the same ability in a cleaner fashion and should be used instead (see Section 9, page 60).

12 Example programs and programming techniques

This section illustrates programming techniques in LIFE through interesting example programs. For many more examples, look at the Examples and Tools directories in the Wild LIFE 1.0 release package.

12.1 Generating prime numbers

Enumerating all positive integers can be done in an elegant manner by using the definition `natural -> {0;1+natural}`. Defining prime numbers can be done by declaring a sort `prime` and attaching a prime-testing routine to the sort. This allows the fact of being prime to be remembered by the number itself. This is a good illustration of the expressive power of the sort hierarchy.

```
prime := I:int | length(factors(I))=1.

factors(N:int) -> cond(N<0,
                       {},
                       factorize(N,2)).

factorize(N,P) -> cond(P*P>N,
                       [N],
                       cond(R:(N/P)=:=floor(R),
                            [P|factorize(R,P)],
                            factorize(N,P+1))).
```

The first rule reads as: “a prime is an integer, I, such as the number of dividers of I is 1”. The function “factors” yields a list containing the factorization of its argument. Let’s have a look at what this program does:

```
> write(factors(6450)),nl,write(factors(127))?
[2,3,_A:5,_A,43]
[127]
*** Yes
> 6=prime?

*** No
> 43=prime?

*** Yes
> P=prime?

*** Yes
P = prime~.
--1> P=29?

*** Yes
P = 29: prime.

--2> repeat,write(prime&natural," "),fail?
0:prime 1:prime 2:prime 3:prime 5:prime 7:prime 11:prime
13:prime 17:prime 19:prime 23:prime 29:prime 31:prime
37:prime 41:prime 43:prime 47:prime 53:prime ...
```

A number of sort `prime` will not be checked twice. As `P:prime` has no value, the function `factors` residuates, which causes `length` to residuate too, then the instant an integer with a value is unified with P, these two expressions are released, and succeed or fail. By coupling `prime` and `natural` it is possible to generate all prime numbers.

12.2 PERT scheduling

This section presents an algorithm for PERT scheduling that illustrates the advantages of out-of-order execution and of object-orientation. The program calls the functions that calculate the scheduling information before their arguments are known. The functions are attached to the declared sort `task`, providing for a clean data encapsulation.

PERT (Program Evaluation and Review Technique) is a methodology for planning big projects. In particular, it is used to schedule subtasks that comprise a bigger task. For example, the big task of building a house can be divided up into many smaller tasks: architectural design, buying a plot of land, contracting for the building, the plumbing, the electricity, interior decorating, and so forth. Each of these subtasks is dependent on a given set of other subtasks to start. Each of the subtasks has a duration. An important problem is to find the earliest and latest starting times of each task such that the big task is completed as soon as possible. If there is no limit on the number of tasks that may be done in parallel, then this problem has a linear-time solution. A Wild LIFE program to solve this problem can be written as follows. This program is one of the examples provided with the release. First, define a sort `task` that represents the information relevant to a task:

```
:: A:task( duration => D:real,
           earlyStart => earlyCalc(R),
```

```
             lateStart   => {1e500;real},
             prerequisites => R:{[];list} )
     | !, lateCalc(A,R).
```

This sort has attached to it the functions `earlyCalc` and `lateCalc` to do the calculations. Function `earlyCalc` determines the earliest time that task A can start:

```
earlyCalc([]) -> 0.
earlyCalc([B|ListOfActs]) ->
        max(B.earlyStart+B.duration,earlyCalc(ListOfActs)).
```

Function `lateCalc` determines the latest time that A's prerequisites can start and still finish before A starts:

```
lateCalc(A,[]) -> succeed.
lateCalc(A,[B:task|ListOfActs]) ->
        lateCalc(A,ListOfActs) |
        assign(LSB:(B.lateStart),
               min(LSB, A.earlyStart-B.duration)).

% Wait until B is an integer before doing the assignment:
assign(A,B:int) -> succeed | A<-B.
```

Taken together, the above definitions form a self-contained program. This program does all the calculation necessary to determine the earliest and latest start time of each task, given the dependencies and the durations. For example, a possible session is:

```
> import("schedule")?
*** File "schedule.lf" loaded

*** Yes
> A1=task(duration=>10),
|     A2=task(duration=>20),
|     A3=task(duration=>30, prerequisites=>[A1,A2])?

*** Yes
A1 = task(duration => 10,
          earlyStart => 0,
          lateStart => 10,
          prerequisites => []),
A2 = task(duration => 20,
          earlyStart => 0,
          lateStart => 0,
          prerequisites => []),
A3 = task(duration => 30,
          earlyStart => 20,
          lateStart => Infinity,
          prerequisites => [A1,A2]).
```

This says that task A3 can start the earliest at time 20. Activity A1 has a slack of 10: it can start as early as time 0 and as late as time 10 without slowing down the project. Activity A2 must start at time 0; it cannot start later without slowing down the project.

The problem may be described mathematically as follows. Given n tasks numbered 1, 2, ..., n. For each task i its duration is denoted d_i (`duration`) its earliest and latest start times are denoted e_i and l_i (`earlyStart` and `lateStart`), and the set of tasks it depends on is denoted P_i (`prerequisites`). Given are all values of d_i and P_i. The problem is to calculate the values of e_i and l_i. This reduces to the following set of equations:

$$\begin{aligned} e_i &= \max_{j \epsilon P_i}(e_j + d_j) \quad (1 \le i \le n) \\ l_i &= \min_{i \epsilon P_j}(e_j - d_i) \quad (1 \le i \le n) \end{aligned}$$

Both the maximum and the minimum operations run over all values of j that satisfy their condition. The first equation means that task i cannot start until all the tasks that it depends on have finished. The second equation means that task i must end before the earliest start time of all the tasks that depend on it.

A great advantage of writing the program in LIFE is the order-independence. The program can be written in a straightforward way by exactly following the equations. The given information (values of d_i and P_i) may be given in any order. If sufficient information is given to solve the equations, the result will always be correct. Since each function invocation is calculated only once, the result is calculated in linear time, regardless of the amount of sharing in the dependency graph. It is in general difficult to predict when the different calculations will be done, but this is irrelevant because it is not necessary.

12.3 Cryptarithmetic: SEND+MORE=MONEY

This example shows an efficient way of solving the standard benchmark test "SEND+MORE=MONEY" where each letter codes one digit and no two letters code the same digit. The algorithm is based on test-and-generate. That is, a series of function calls is done which all suspend. Then the variables are instantiated to all possible digits. The suspended functions act as passive constraints to prune the search. The computation terminates successfully only when an assignment of digits to variables is found that is consistent with all the passive constraints.

```
solve :-
    % Solutions with M=0 are uninteresting:
    M=1,

    % The arithmetic constraints:
    C3 + S + M = O + 10*M,
    C2 + E + O = N + 10*C3,
    C1 + N + R = E + 10*C2,
         D + E = Y + 10*C1,

    % The all-distinct constraints:
    diff_list([S,E,N,D,M,O,R,Y]),

    % Generating binary digits:
    C1=carry, C2=carry, C3=carry,

    % Generating decimal digits:
```

```
      S=decimal, E=decimal, N=decimal, D=decimal,
      O=decimal, R=decimal, Y=decimal,

      % Print the result:
      nl,
      write(' SEND     ',S,E,N,D),nl,
      write('+MORE    +',M,O,R,E),nl,
      write('-----    -----'),nl,
      write('MONEY    ',M,O,N,E,Y),nl,
      nl, fail.

  decimal -> {0;1;2;3;4;5;6;7;8;9}.
  carry -> {0;1}.

  diff_list([]).
  diff_list([H|T]) :-
      generate_diffs(H,T), diff_list(T), H=<9, H>=0.

  generate_diffs(H,[]).
  generate_diffs(H,[A|T]) :- generate_diffs(H,T), A=\=H.
```

This program solves the problem very quickly, despite the fact that Wild LIFE is only an interpreter. It is interesting that this solution is of the same order of efficiency as one based on finite domains.[19]

```
> import("solve")?
*** File "solve.lf" loaded

*** Yes
> solve?

 SEND     9567
+MORE    +1085
-----    -----
MONEY    10652

*** No
```

In less than a second on a DECstation 3100,[20] Wild LIFE prints a solution and proves it is unique. Just to get an idea of the power of constraints, we wrote a program in C which solves the same problem using a generate and test method. It has seven nested loops in which the program explicitly tests the difference constraints by marking those digits already used. On the same machine, its CPU time is 1.0 seconds.

Notice also that Wild LIFE does no special preprocessing of constraints or static analysis (other than sort encoding which isn't used here) except for local propagation.[21]

[19]There are presently no finite domains in Wild LIFE as such, although sorts allow a similar kind of filtering.
[20]A DECstation 3100 has speed similar to a SPARCstation 1.
[21]See Section 12.11.

12.4 Concurrent programming

Here is a little program that shows how a committed-choice programming style can be imitated in Wild LIFE. The program is transliterated from an FCP (Flat Concurrent Prolog) example in Shapiro's survey article on concurrent logic programming [18].

LIFE's function suspension mechanism (*i.e.*, residuation) is used to communicate between functions. In the terminology of concurrency: a recursive function acts like a process. Communication between processes is done through unification of shared variables. Synchronization is done through residuation. Task switching is completely data-driven and hence the scheduling policy is non-fair.

Here is a sample session:

```
> A=nsift([2,3,4,5|L])?

*** Yes
A = [2,3,5|@], L = @~.       % The system waits on L
--1> L=[6,7,8,9|L2]?         % Refining L resumes
                             % the computation
*** Yes
A = [2,3,5,_A:7|@], L = [6,_A,8,9|L2], L2 = @~.
                             % Now it waits on L2
----2> L2=[10,11,12,13|L3]? % Refining L2 resumes
                             % the computation
*** Yes
A = [2,3,5,_A:7,_B:11,_C:13|@],
L = [6,_A,8,9|L2],
L2 = [10,_B,12,_C|L3],
L3 = @~.                     % Now it is waiting on L3
```

The list A contains only prime numbers. Shared objects are given system-generated names; *e.g.*, the variable _A that marks the instance of 7 which occurs in both the lists A and L. The function nprimes passes a list of integers to nsift:

```
> A=nprimes(100)?

*** Yes
A = [2,3,5,7,11,13,17,19,23,29,31,37,41,43,47,53,59,61,67,71,
73,79,83,89,97].
```

Since the complete list is passed to `nprimes` (see its definition), a complete answer can be generated. Here is the source code of `nsift`.

```
nprimes(N) -> nsift(integers(2,N)).

% Create a stream of integers:
integers(From,To) ->  cond(From > To,
                           [],
                           [From | integers(From+1,To)]).
```

```
% Remove multiples of P:
filter([]) -> [].
filter([X|In],P) -> cond((X mod P =\= 0),
                         [X | filter(In,P)],
                         filter(In,P)).

% Filter out multiples of each element:
nsift([]) -> [].
nsift([P|Ns]) -> [P|nsift(filter(Ns,P))].
```

As an optimization, here is a modified version that passes a maximum sift value equal to the square root of the maximum input. This is much more efficient than the first version.

```
primes(N) -> sift(integers(2,N),sqrt(N)).

% Filter out multiples of each element:
sift([]) -> [].
sift([P|Ns],Max) -> [P | sift(cond(P =< Max,
                              filter(Ns,P),Ns),
                              Max)].
```

12.5 Encapsulated programming

This example shows how you can create a routine that behaves like a process with encapsulated data. The caller cannot access the routine's local data except through the access functions ("methods") provided by the routine.

```
% Encapsulated object-oriented process without streams

% Initialization:
new_counter(C) :- counter(C,0).

% Access predicate:
access(X,C) :- C=[X|C2], C<-C2.

% The counter:
counter([inc|S],    V) -> counter(S,V+1).
counter([set(X)|S],V) -> counter(S,X).
counter([see(X)|S],V) -> counter(S,V) | X=V.
counter([stop|S],   V) ->
    true | write("Counter closed with stop."), nl.
counter([_|S],      V) ->
    counter(S,V) | write("Message not understood."), nl.
counter([],         V) ->
    true | write("Counter closed with end-of-stream.").
```

This defines a `counter` process. Access to the process is by the variable C. The internal state of the process is the value of the counter, which is held in V, the second argument of `counter`. The use of backtrackable destructive assignment is essential. What happens is that `counter` creates a new object (the residuated recursive call) through its execution, and backtrackable destructive assignment gives this object the same name as the original object which has disappeared. This is very close to Actor semantics. Here is an example of its use:

```
> new_counter(C)?        % Initialize new counter object

*** Yes
C = @~.
--1> access(inc,C)?      % Increment the counter

*** Yes
C = @~.
----2> access(inc,C)?    % Increment the counter

*** Yes
C = @~.
------3> access(see(X),C)?  % Get the counter's value

*** Yes
C = @~, X = 2.
```

This creates a new counter object (with initial value 0) which is accessed through C. The counter is incremented twice and then its value is accessed.

12.6 Classes and instances

To teach LIFE to non-LIFE users or even non-Prolog users, it is very important to tie its concepts to concepts that are widely known. Two such concepts "class" and "instance", which are common in object-oriented languages such as C++ and Smalltalk. The declaration and use of sorts can be described precisely in terms of the latter two concepts by the following rules:

1. A class corresponds to a sort. Classes are declared by sort definitions, of which the two basic ones are:

   ```
   % Like a 'struct',
   % this adds fields to a class definition:
   :: class(field1=>value1, field2=>value2, ...).

   % Class1 inherits all properties of class2:
   class1 <| class2.
   ```

2. Instances are created by mentioning the class name during execution. For example, executing:

   ```
   > X=int?
   ```

   will create an instance of the class `int`. Each mention of `int` creates a fresh instance. Therefore, executing:

   ```
   > X=int, Y=int?
   ```

creates two *different instances* of the class `int` in X and Y. We can do:

```
> X=int, Y=int, X=56, Y=23?
```

We can refine X to `56` and Y to `23`. Obviously, this would not be possible if X and Y were the same instance.

3. The Wild LIFE system assumes that mentioning a class name during execution *always* creates a fresh instance that is different from all other instances of the class. For example:

```
> X=23, Y=23?
```

creates two *different instances* of the class `23`. With the function `f` defined as:

```
> f(A,A) -> 1.
```

the call `f(X,Y)` will *not* fire, since X and Y are different instances. To make it fire, X and Y must be the *same* instance. In Wild LIFE, the only way to do this is to unify them explicitly by doing `X=Y`. For example:

```
> X=23, Y=23, X=Y, write(f(X,Y))?
```

will write `1`, *i.e.*, the function `f` will fire.

4. Sometimes it would be useful to have classes that only have *one* instance. For example, it would be nice if all the instances of the class '23' were the same integer `23`, *i.e.*, if the class `23` had a unique instance `23`. This is not possible in Wild LIFE 1.0.

12.7 Using destructive assignment to calculate term size

This section defines the function `term_size(T)` which calculates the size of a ψ-term. The algorithm for calculating the size is a good example of the correct use of destructive assignment. It demonstrates the use of both backtrackable and nonbacktrackable destructive assignment (`<-` and `<<-`). See Section 9.4 for a discussion of these two built-ins.

The size of a ψ-term is defined as the number of nodes it contains. The algorithm counts the nodes by traversing the ψ-term and using backtrackable destructive assignment to mark nodes that have already been visited. The mark is the ψ-term `Seen`, which is created locally to `term_size`, and hence is guaranteed not to occur in the term being explored.

After the traversal is finished, the algorithm backtracks to unmark all the marked nodes and to recover all memory used during the traversal. The result is retained by storing it in an anonymous persistent term with nonbacktrackable destructive assignment. The persistent term is created *before* the choice point so that it is unaffected by the backtracking.

```
% Return a list containing the values of all features of X:
feature_values(X) -> map(project(2=>X),features(X)).

% Sum all the elements in a list:
```

```
sum([V|Vs]) -> V+sum(Vs).
sum([]) -> 0.

term_size(X) -> N |
    V<<-@,                  % Create an anonymous persistent
                            % term for the result
    ( V<<-term_explore(X,Seen), % Calculate the size
      fail                  % Remove effects of calculation
    ;
      N=copy_term(V)        % Copy the result back to
    ).                      % a normal logical term


term_explore(X,Seen) -> V |
    ( X===Seen,             % Skip already-counted nodes
      !,
      V=0
    ;
      FV=feature_values(X), % Get the features of X
      X<-Seen,              % Mark X as having been counted
      V=1+sum(map(term_explore(2=>Seen),FV))
    ).
```

Here is a sample execution:

```
> A=term_size(p(a,b,c,d))?

*** Yes
A = 5.
--1>

*** No
> A=term_size(B:[a,b|B])?

*** Yes
A = 4, B = [a,b|B].
```

12.8 Using a ψ-term as an array

This section presents a Sieve of Eratosthenes program that calculates the sequence of primes up to a given limit. It uses a ψ-term as an array to store the primality information. The features of the ψ-term are the integers whose primality are being checked.

```
global(sieve)?
global(limit)?

main :-
    write("N=?"),
    read_token(limit & int),
    next_prime(2),
    nl.
```

```
remove_multiples(P,M) :-
    cond(M<limit,
         (sieve.M<-multiple_of(P),remove_multiples(P,M+P))).

next_prime(P) :-
    P<limit,
    !,
    SP=sieve.P,
    ( SP=prime(P),
      !,
      write(P,' '),
      remove_multiples(P,2*P)
    ;
      succeed
    ),
    next_prime(P+1).

next_prime(P).
```

Here is a sample execution:

```
> main,nl,nl,pretty_writeq(sieve)?
N=?20
2 3 5 7 11 13 17 19

@(2 => prime(_A: 2),
  3 => prime(_B: 3),
  4 => multiple_of(_A),
  5 => prime(_C: 5),
  6 => multiple_of(_B),
  7 => prime(_D: 7),
  8 => multiple_of(_A),
  9 => multiple_of(_B),
  10 => multiple_of(_C),
  11 => prime(11),
  12 => multiple_of(_B),
  13 => prime(13),
  14 => multiple_of(_D),
  15 => multiple_of(_C),
  16 => multiple_of(_A),
  17 => prime(17),
  18 => multiple_of(_B),
  19 => prime(19))
*** Yes
```

12.9 Memoization

This example shows how `asserta` may be used to implement an efficient way of finding the list of moves needed to solve the Hanoi towers problem. This example is adapted from “The Art of Prolog” [19].

```
hanoi(1,A,B,C,[[A,B]]).

hanoi(N:int,A,B,C,Moves:append(Moves1,[[A,B]|Moves2])) :-
     N>1,
     hanoi(N-1,A,C,B,Moves1),
     hanoi(N-1,C,B,A,Moves2),
     asserta(hanoi(N,A,B,C,Moves) :- !),
     write("Solved problem for N = ",N),
     nl.
```

The effect of using `asserta` is that solutions of intermediate problems are remembered. This technique is often called "memoization".

```
> hanoi(3,_,_,_,M)?
Solved problem for N = 2
Solved problem for N = 3

*** Yes
M = [[_A,_B],[_A,_C],[_B,_C],[_A,_B],[_C,_A],
      [_C,_B],[_A,_B]].
--1> ;

*** No
> listing(hanoi)?

hanoi(3,_A,_B,_C,
       [[_A,_B],[_A,_C],[_B,_C],[_A,_B],[_C,_A],
        [_C,_B],[_A,_B]]) :-
     !.
hanoi(2,_A,_B,_C,[[_A,_C],[_A,_B],[_C,_B]]) :-
     !.
hanoi(1,_A,_B,@,[[_A,_B]]) :-
     succeed.
hanoi(_A: int,_B,_C,_D,_E: append(_F,[[_B,_C]|_G])) :-
     _A > 1,
     hanoi(_A - 1,_B,_D,_C,_F),
     hanoi(_A - 1,_D,_C,_B,_G),
     asserta((hanoi(_A,_B,_C,_D,_E) :- !)),
     write("Solved problem for N = ",_A),
     nl.

*** Yes
```

Because `hanoi` is first called with "_" (the most general sort) for the tower names, it stores the most general solution for each value of N.

12.10 Method inheritance in the graphical interface toolkit

In the graphical interface toolkit supplied with Wild LIFE (see appendix H), `implies` is used to perform "method inheritance". Consider the following sort hierarchy:

```
f <| d.
f <| e.

d <| c.
e <| c.

c <| a.
b <| a.
```

and a predicate p which has one definition per sort:

```
p(X:a) :- pa(X). % pa is a method of sort a
p(X:b) :- pb(X). % pb is a method of sort b
p(X:c) :- pc(X). % pc is a method of sort c
 ...
```

With `implies` we can call all methods of p that are supersorts of the argument X. This is done by means of the following predicate:

```
all_p(X) :-
     ( p(X),
       fail
     ;
       succeed
     ).
```

For any X, `all_p(X)` will execute all the clauses of p defined for supersorts of X. For instance, `all_p(X:e)` will execute `pa(X), pc(X), pe(X)`. In the terminology of the graphical interface toolkit, imagine now that the sorts are *look* sorts, and p is a drawing routine, and you see how looks are inherited. In the toolkit, the inheritance from boxes, looks, and constructors (*i.e.*, multiple inheritance) is handled using this technique.

12.11 Structural constraints and arithmetic constraints

Wild LIFE does not do any global constraint solving. It does purely local constraint solving. In that sense, it is more like Prolog than like $CLP(R)$. Calling a function adds a matching (implication) constraint, and calling a predicate adds a unification (equality) constraint.

Arithmetic functions in Wild LIFE do local propagation. That is, they handle all cases where the value of one or more arguments can be determined *uniquely* from the others. For example:

```
> 23=10+X?

*** Yes
X = 13.
```

Another example:

```
> A=A+C?

*** Yes
A = real, C = 0.
```

This example residuates (*i.e.*, suspends):

```
> A=A*B?

*** Yes
A = real~, B = real~.
```

(note the tildes) since there are two solutions: A=0 or B=1.

For real numbers *CLP*(R) does more: in addition to local propagation, it does global constraint solving with linear equalities and linear inequalities on real numbers (using incremental Gaussian elimination and incremental Simplex, based on Fourier elimination). On the other hand, Wild LIFE does local constraint solving on ψ-terms, which are rooted graphs that live in a hierarchy. So for example, Wild LIFE can check structural constraints like graph subsumption, which *CLP*(R) cannot.

Example 12.1 This example illustrates the use of graph subsumption as a constraint.

```
> f(X:s(a=>t(b=>X))) -> true.  % Define a function

*** Yes
> A=f(G)?          % Check if graph G implies
                   % the cyclic graph X:s(a=>t(b=>X))
*** Yes
A = @, G = @~.
--1> G=s(a=>t(b=>X))?          % Make G an acyclic graph
                               % It doesn't fire yet!
*** Yes
A = @, G = s(a => t(b => X))~, X= @~.

----2> G=X?        % Adding the cycle fires the function
                   % (The function result is A = true)
*** Yes
A = true, G = s(a => t(b => G)), X = G.
```

13 Fine points for would-be wizards

13.1 Functional variables and apply

When a functional expression F(A) is evaluated whose function symbol is a variable F, it looks as if the expression's root sort is a variable and applied to the arguments. In fact, this is not what happens in Wild LIFE. The syntax `F(a,b,c)` is simply syntactic sugar for `apply(functor => F,1 => a,2 => b,3 => c)`. It is converted into this internal form by the parser. If the feature `functor` occurs in F it is ignored. The function `apply` is transparent to the user. But you should virtually never need to resort to using it explicitly. Only in rare cases is it unavoidable to use it, as shown in the following example.

Example 13.1 Let us define a function `mapkey` that is similar to `map` except that it does not apply the mapped function on its first argument but on a specified position label. As Wild LIFE stands now, there is no way to do this without explicitly using `apply` as follows.

```
mapkey(A,F,[]) -> [].
mapkey(A,F,[H|T]) ->
     [FAH:apply(functor => F)|mapkey(A,F,T)]
     | H = FAH.A.
```

We can use this to compute the values of an integer (say, 10) in cyclic groups $\mathbb{Z}/p\mathbb{Z}$ where the values of p are given in a list:

```
> L=mapkey(2,mod(10),[1,2,3,4,5,6,7,8,9])?

*** Yes
L = [0,0,1,2,0,4,3,2,1].
```

13.2 Query levels

The observant user will notice that the query level (shown by the prompt) is not always systematically incremented after every query extension. In fact, the specific rule for increasing the query level is the following: if the query extension (1) contains at least one variable *or* creates a choice-point *or* creates an X window, *and* (2) succeeds, *then* the level is increased. Otherwise, the level stays the same (*viz.*, if the query extension fails, or contains no variable and succeeds with no choice-point). Each time interaction goes back to a previous query level, the variable bindings corresponding to that level are displayed.

Example 13.2 This example illustrates the fine points of query level incrementing.

```
> X=a?

*** Yes
X = a.
```

The query level is incremented because X appears in the query.

```
--1> a=a?

*** Yes
X = a.
```

Despite success, the query level is unchanged because no variable appears in the query and no choice point is created.

```
--1> a=a ; b=b?

*** Yes
X = a.
```

The query level is incremented because a choice-point was created proving this query extension.

```
----2> ;

*** Yes
X = a.
```

The level goes back to 1 because the second disjunct `b=b` succeeds without containing a variable nor creating a choice-point.

```
--1> a=a ; b=b ; c=c?

*** Yes
X = a.
```

The query level is incremented because a choice-point was created proving this query extension.

```
----2> ;

*** Yes
X = a.
```

Level stays at 2 because the second disjunct `b=b;c=c` succeeds but creates a choice-point. Finally, we type `.` to pop to top level.

```
----2> .
>
```

13.3 Predicate and function positions

Wild LIFE makes a syntactic distinction between a *predicate position* and a *function position*. A predicate position is any place in a program where the interpreter expects to find a predicate. This includes at the query prompt, in a clause body, and certain arguments of the built-ins `call_once`, | (such-that), and `cond`. Similarly, a function position is where the interpreter expects to find a function. This includes all arguments of predicates and functions and all function bodies.

In the simplest case, predicates are used only in predicate positions and functions are used only in function positions.

In general, a predicate position may contain any term that will eventually be bound to a predicate or to one of the sorts `true` or `false`. When the term is bound to a predicate, then that predicate is executed. When the term is bound to `true` then execution succeeds. When the term is bound to `false` then execution fails. If the term is incompatible with these three possibilities, then an error is reported. If the term is compatible with these three possibilities, but it is not yet known which holds, then the predicate position residuates. It follows that predicate execution is *order-independent*: the same result is obtained for a pure program that terminates no matter in what order the predicates in its predicate positions become known.

Example 13.3 This example illustrates that the same result is obtained no matter when the predicate is known in a predicate position.

```
> p(a).         % Define a single fact

> Q=p(X),Q?     % The predicate is known immediately

*** Yes
Q = p(X), X = a.
--1>

*** No
> Q,Q=p(X)?     % The predicate is known with a delay

*** Yes
Q = p(X), X = a.
--1>
```

In general, a function position may contain any term. If the term is a function, then the term is evaluated. If the term is a predicate, then the term is considered as a sort with only parent @ and only child {}. If the term is a ψ-term, then it is left unchanged. The arguments of a term are considered to be function positions, and hence are evaluated if they themselves are functions.

A predicate position may contain a predicate itself, a function that eventually returns a predicate, or a term that is eventually bound to a predicate. The predicate position does not have to be bound immediately.

Example 13.4 This examples illustrates a function in a predicate position.

```
> f(A:int) -> write(A).   % Wait until A is an integer

> f(23)?                  % Write 23
23
*** Yes
> X,X=f(Y),Y=23?          % Write 23
23
*** Yes
X = write(Y), Y = 23.
```

The order of predicate evaluation, function evaluation, and unification makes no difference; in a pure program that terminates the result is always the same.

A function in a function position is always evaluated unless the function position is non-strict, in which case the function is returned unchanged. A predicate or function can be declared to have non-strict arguments through the `non_strict` declaration.

13.4 Compact sort definitions

A complete facility for sort declarations must allow any combination of:

- Attaching attributes to a sort. This is declared with the operator `::`.
- Declaring single or multiple inheritance relationships from supersorts. This is declared with the operator `<|`. Multiple inheritance from several supersorts is specified by using more than one declaration or by writing the supersorts as a disjunction.
- Attaching a constraint to a sort. This is declared with the such-that operator `|`.

The above comprise the base primitives that allow all sort declarations possible in Wild LIFE. A sort that is declared with one of these primitives is called a *declared sort*. In this section we introduce the built-in `:=` that often permits declarations with a suggestive relationship to mathematical notation. This built-in provides no additional expressive power. It is pure syntactic sugar.

Table 2 summarizes all the possible combinations of sort declarations allowed in Wild LIFE. Notation: `t, u, v, w` denote sort symbols; `(attr)` following a sort denotes either a non empty attribute list or nothing (*i.e.*, not `()` but simply absence of an attribute list altogether); `const` represents a constraint of the same form as is allowed in the body of a definite clause.

The declaration marked (8) is syntactic sugar for:

```
:: t(attr)  | const.
t <| u.
t <| v.
t <| w.
```

The declaration marked (10) is syntactic sugar for:

```
:: u(attr1)  | const.
:: v(attr2)  | const.
:: w(attr3)  | const.
u <| t.
v <| t.
w <| t.
```

◑◑ Please note the asymmetry of behavior of `:=`. On one hand, `t := u.` declares `t` as a *sub*sort of `u`; but on the other hand, `t := {u;v;w}.` declares `t` as a *super*sort of `u`, `v`, and `w`. An annoying consequence of this is that `t := {u}.` is equivalent to `u <| t.`, whereas `t := u.` is equivalent to `t <| u.`. Similarly, while `t := u.` means the same as `t <| u.`, it is *not* the case that `t := {u;v;w}.` means `t <| {u;v;w}.`, but instead means `u <| t. v <| t. w <| t.`. We are aware that these anomalies may not be palatable to all (not even to some of us!). However, one can systematically use the consistently behaving `<|` and avoid ever using `:=`. Nevertheless, `:=` still offers a conveniently simple shorthand to define such sorts such as:

```
tree := { leaf ; node(left => tree, right => tree) }.
```

which declares that a tree is a leaf or a node whose left is a tree, and whose right is a tree. Some may indeed argue that this looks like a more direct and natural translation than writing:

(1)	`:: t(attr).`
(2)	`:: t(attr) | const.`
(3)	`t(attr) <| u.`
(4)	`t(attr) <| u | const.`
(5)	`t := u(attr).`
(6)	`t := u(attr) | const.`
(7)	`t(attr) <| {u;v;w}.`
(8)	`t(attr) <| {u;v;w} | const.`
(9)	`t := {u(attr1);v(attr2);w(attr3)}.`
(10)	`t := {u(attr1);v(attr2);w(attr3)} | const.`

(1)	[attributes]
(2)	[attributes,] constraint
(3)	[attributes,] inheritance
(4)	[attributes,] inheritance, constraint
(5)	same meaning as (3)
(6)	same meaning as (4)
(7)	[attributes,] multiple inheritance
(8)	[attributes,] multiple inheritance , constraint
(9)	[attributes,] multiple inheritance
(10)	[attributes,] multiple inheritance , constraint

Table 2: Possible sort declarations

```
leaf <| tree.
node <| tree.
:: node(left => tree, right => tree).
```

or even the equivalent shorter form:

```
leaf <| tree.
node(left => tree, right => tree) <| tree.
```

Be that as it may, we leave it to your taste to choose what fits you best.

13.5 Sort encoding

Wild LIFE uses a special binary encoding of the declared sorts so that a glb can be quickly and efficiently calculated even if the hierarchy contains a very large number of sort definitions [1]. Each declared sort is assigned one bit vector with a value that reflects its place in the hierarchy. The value of the bit vectors is irrelevant to the programmer, but for the curious, the query `print_codes?` will show them.[22] The upshot of this is that there is an encoding phase which can be likened to compilation. This is invisible to the user.

A constant without any definitions (*i.e.*, an uninterpreted identifier) is considered to be alone in its class, *i.e.*, it can only be unified with `@` or itself. It is not encoded.

13.6 Printing convention

Wild LIFE uses a systematic convention for printing the arguments in the body of a term. First the arguments of numeric positions are printed using the natural number ordering, then the arguments corresponding to word attributes using the lexicographic ordering on the attribute labels. All symbolic attribute labels are printed explicitly. A numeric position is printed only if necessary. Namely, position 1 is never printed explicitly; and any other position is printed explicitly only if a lower position is missing.

Example 13.5

```
> X=f(2=>t2,a=>ta,t1,@,4=>t4,b=>tb,ab=>tab)?

*** Yes
X = f(t1,t2,4 => t4,a => ta,ab => tab,b => tb).
```

Other important points if you are interested in parsing input are that:

- parentheses are removed from expressions wherever they are unnecessary,
- `a op b` where `op` is an infix operator is syntactic sugar for `op(1 => a, 2 => b)`,
- `op a` where `op` is a prefix operator is syntactic sugar for `op(1 => a)`,

[22] For example, execute the query `import("x"),print_codes?`.

- likewise `a op` where `op` is a postfix operator is syntactic sugar for `op(1 => a)`.

Infix notation is respected by the printer if the term has exactly two subterms. Otherwise, prefix notation is used. Prefix notation can always be used, even for infix or postfix sort symbols.

14 Hints to write more efficient programs

14.1 Garbage collection

Garbage collection is expensive. The best way to write a program is certainly to avoid using it, which means that memory should be recovered automatically when possible.

Using backtracking enables the program to recover memory space without garbage collection, simply because nearly all the space used between the creation of the choice-point and the failure that causes the backtrack is recovered.

Consider the following example:

```
foo(0) :- !.
foo(X) :-
     X > 0,
     foo(R:(X-1)), foo(R), foo(R), foo(R).

foo(N)?  % Computes foo(0) 4^N times
```

During the query `foo(7)?` Wild LIFE will garbage collect several times before succeeding. A way to avoid garbage collection is to force backtracking:

```
foo2(0) :- !.
foo2(X) :-
     X > 0,
     ( foo2(R:(X-1)), foo2(R), foo2(R), foo2(R), fail
     ; succeed
     ).
```

The modified predicate `foo2(7)` does not need garbage collection and runs 30% faster. This technique may be used if there is no need to keep track of what has been done between the creation of a choice point and the failure. Nonbacktrackable assignment to persistent variables can be used to keep information while backtracking. For example, we could compute 4^N with the following program:

```
persistent(result)?

foo3(0) :- result<<-result+1.
foo3(X) :-
     X > 0,
     ( foo3(R:(X-1)), foo3(R), foo3(R), foo3(R), fail
     ; succeed
     ).

power3(N) :- result<<-0, foo3(N), X=result, write(X).
```

Calling `power3(8)` writes 65536. The remaining garbage collections can be avoided if <<− is used less often. The following program uses <<− four times less often than the previous one, and runs about 45% faster:

```
persistent(result)?

foo4(0,Counter,X,X+1) :- !.
foo4(N,0,_,Y) :- !,
     N > 0,
     ( foo4(R:(N-1),3,result,X1),
       foo4(R,3,X1,X2),
       foo4(R,3,X2,X3),
       foo4(R,3,X3,X4),
       result<<-X4,
       fail
     ; Y=result
     ).
foo4(N,P,X,Y) :-
     N > 0,
     foo4(R:(N-1),C:(P-1),X,X1),
     foo4(R,C,X1,X2),
     foo4(R,C,X2,X3),
     foo4(R,C,X3,Y).

power4(N) :- result<<-0, foo4(N,_,0,X), write(X).
```

This technique can always be used when the program's result is given through nonbacktrackable operations (such as writing, drawing, or asserting).

14.2 Residuation

Residuation is a very useful tool to write efficient programs when a kind of coroutining behavior is expected. In many "generate and test" programs (such as eight queens), residuation enables the programmer to put the test predicates before the generation predicates: this means that all constraints are set before any generation is done, and thus the satisfaction of the constraints is checked at every generation step. This greatly decreases the complexity of the program.

Residuation can be used to mimic the behavior of delaying primitives such as when or wait declarations and freeze. Here again, residuation is used to increase efficiency or to avoid the non-termination of programs.

Nevertheless, residuation should be avoided when possible, since it is a complex and expensive operation. In many cases, the programmer knows quite accurately how the variables in a rule are instantiated, and can rewrite these so as to avoid as many useless residuations as possible. For example:

```
z(V1+V2*2^(-L)) :- r(V1),p(V2,L)?
```

should be rewritten:

```
z(V) :- r(V1), p(V2,L), V = V1+V2*2^(-L).
```

if you know that the values of V1, V2 and L are computed by r and p.

14.3 Partial evaluation

In programs where functions are used to denote constants, it is sometimes possible to do a bit of partial evaluation, to avoid computing these constants at runtime. For example:

```
charHeightLogo -> 60.
y0 -> 0.
cellHeight -> 20.
yLogo  -> y0+charHeightLogo.
yTitle -> yLogo+2*cellHeight
unit -> 10.
scale(X) -> X*unit.
side -> scale(50).
```

should be rewritten:

```
setConst(X,Y) :- assert(X->Y).

charHeightLogo -> 60.
y0 -> 0.
cellHeight -> 20.
setConst(yLogo, y0+charHeightLogo)?
setConst(yTitle, yLogo+2*cellHeight)?
unit -> 10.
scale(X) -> X*unit.
setConst(side, scale(50))?
```

In this way, the values of `yLogo`, `yTitle`, and `side` are computed at load time, and will not have to be recomputed at runtime. This technique can of course not be used with dynamic "constants," but may be really useful if the evaluated constants are used often.

15 Compatibility with Prolog

It is but a small step for a Prolog programmer to start programming in LIFE. If he stays in the Prolog-like subset of LIFE, then he can start immediately. The following is a complete list of the differences that can cause problems porting Prolog programs to LIFE.

- Ψ-terms have no arity. They can have an arbitrary number of arguments and arguments may be added at will at run-time. Therefore arity can not be used to distinguish predicates. In Prolog, a functor is a pair F/N, where F is the functor name and N is the arity. In Prolog, two functors are unifiable if and only if they have both the same name and the same arity. Many Prolog programmers take advantage of this and use the same symbol with different arities to name distinct predicates. Clearly, this practice is no longer valid in Wild LIFE.[23] Indeed, two ψ-terms with the same principal sort symbol but different numbers of arguments, or with different subterm attributes altogether, can very well

[23] This is not a very serious limitation of compatibility as this practice is generally considered a bad one by serious Prolog programmers, and Prolog programs can usually be rewritten in a straightforward way to avoid it.

unify. In fact, they may unify even with distinct root sorts, as long as these have a non-bottom glb, and a ψ-term may acquire new attributes as a problem is solved (*i.e.*, as more information is learned about the object).

Example 15.1 Consider the following predicate definition:

```
pred(A,B,C) :- write(A), write(B), write(C).
```

This is how Prolog and Wild LIFE behave when confronted with the same queries: In SICStus Prolog:

```
?- pred(1,2,3).
123
?- pred(A,B,C).
_26_60_94
?- pred(A,B,C,D).
WARNING: predicate 'pred/4' undefined.
?- pred(A,B).
WARNING: predicate 'pred/2' undefined.
```

In Wild LIFE:

```
> pred(1,2,3)?
123
*** Yes
> pred(A,B,C)?
@@@
*** Yes
A = @, B = @, C = @.
--1> .
> pred(A,B,C,D)?
@@@
*** Yes
A = @, B = @, C = @, D = @.
--1> .
> pred(A,B)?
@@@
*** Yes
A = @, B = @.
--1> .
> pred?
@@@
*** Yes
```

- Terms may be cyclic. LIFE's ψ-terms may be cyclic (they are rational trees). They are unified correctly, matched correctly, read correctly, written correctly, and asserted correctly. Terms are read and written as linear text using the operator "`:`" to represent sharing and cycles.

- The interactive user interface is different. Additions to the rule-base are terminated with "`.`" and queries are terminated with "`?`". There is no dummy `user` file. Queries may be extended incrementally. See Section 3.2 (page 3) for an example.

- Symbols that represent functions behave differently from Prolog. For example, `A=(+)` in Prolog will bind A to the atom `'+'`. In Wild LIFE it will create the curried function `'+'`. To use `'+'` as an uninterpreted symbol in Wild LIFE it must be quoted, *e.g.*, as ``A=`(+)`` (with backquote). When manipulating arbitrary sequences of characters, for example, textual output, one should use a string instead of a single-quoted symbol. For example, call `write("+")` to write the character `+`.

- Strings are represented differently. In Prolog, a string is a short-hand for a list of integer ASCII codes. In Wild LIFE, a string is sort in its own right. All strings are subsorts of the built-in sort `string`. This allows a representation for strings in Wild LIFE that uses much less space.

- Some built-ins are different. See the chapters on built-in predicates and functions for a full list of Wild LIFE's built-ins. Here is a short list of the important differences.
  - `functor/3` and `arg/3` do not exist. The latter is replaced by "`.`", which is called "project". For more information see Section 8.2 (page 38).
  - `is/2` does not exist because it has become superfluous. The LIFE query `A=B+4` is more flexible than the Prolog version `A is B+4`. The LIFE version works with any instantiation pattern of its arguments.
  - Prolog uses `true/0` and `fail/0` to represent success and failure. Wild LIFE makes a more consistent choice by using `succeed` and `fail` for success and failure, and reserving `true` and `false` for the boolean sorts returned as values of boolean functions.
  - The standard order comparisons `==/2`, `\==/2`, `@</2`, `@=</2`, `@>/2` and `@>=/2` do not exist. These are made obsolete by ψ-terms.
  - The "univ" built-in `=../2` does not exist. It is made obsolete by ψ-terms.
  - `bagof` in Wild LIFE does no existential quantification and `setof` is not implemented. See Section 8.1 (page 36).
  - `write/1` and `writeq/1` may have any number of arguments in Wild LIFE.

- Lists are represented with the sort `cons` instead of the dot functor `./2`. The dot is a built-in function used for field selection. For example, `A.foo` accesses the field `foo` of the term A.

- There is no if-then-else operator `->/2`. It is replaced by the `cond` function. See Section 8.1 (page 34). The symbol `->` is used to define function rules.

- Operator declarations are kept as compatible as possible with the ISO standard for Prolog (see appendix B, page 102). The following differences exist. The operator precedence for `->` is 1200 (instead of 1050) since it is used to declare function rules. The operator precedence for arithmetic comparisons is 600 (instead of 700) since comparisons occur in expressions. No operator declarations are given for `is`, `\=`, `=..`, `**`, `?-`, `rem` and the standard order comparisons (`==`, `\==`, `@<`, `@=<`, `@>` and `@>=`).

- Disjunctions are the same as in Prolog. In addition to these, there is also a kind of disjunction, the *type disjunction*, which returns a value. Type disjunctions may be used inside terms. They allow compacter code. They are written with braces { and } instead of parentheses. For example, the following two queries are equivalent:

```
> (A=[1|_]; A=[2|_]; A=[3|_]; A=[4|_]; A=[5|_])?
```

```
> A=[{1;2;3;4;5}|_]?
```

- Negation-as-failure does not work exactly as you might expect if functions are involved. See Section 5.3.3 (page 16).

- DCG expansion in Wild LIFE adds arguments with labels `in_dcg` and `out_dcg`. See appendix F (page 109).

16 Conclusion: the experience of Wild LIFE

We hope that this short handbook will incite readers to program in LIFE. The interpreter we describe, Wild LIFE version 1.0, has been stable for almost a year in its present state. It has been used and tested extensively, and we are confident of its robustness and usefulness. It has most of the functionality of the complete language.

Getting to grips with LIFE is not hard if you are familiar with Prolog. For Prolog programmers, the extra power of ψ-terms, sorts and functions is a welcome addition that often makes programs more readable, more concise and more efficient.

We are using the Wild LIFE interpreter as a foundation to build a compiled system. The emphasis in the compiler is twofold: efficiency and scalability. We are building a streamlined and powerful system that will make LIFE into a language that is every bit as fast and usable as the best existing implementations of Prolog [12]. To help us in this endeavor, please send us your comments and your Wild LIFE programs, so we can use them as fuel for the compiler design.

A LIFE versus Prolog

It is our experience that once you have used LIFE you will not feel like ever using Prolog again. The reason is simple: LIFE provides clean and elegant solutions to a number of Prolog's most glaring deficiencies. Here is a list:

1. Functions, including correct arithmetic
2. Object-orientation
3. C-like records
4. Expandable data-structures: arrays and hash-tables
5. Types and multiple inheritance
6. Correct manipulation of cyclic structures
7. Coroutining and constraints
8. Global variables
9. Clean destructive assignment
10. Persistent data structures

Semantically, most of the above features are consequences of the two ways in which LIFE extends Prolog:

- Herbrand terms are replaced by ψ-terms.
- Call-by-matching is added (Prolog only has call-by-unification).

Most Prolog programs can be easily converted to run under LIFE. Section 15 (page 97) lists the differences between Prolog and LIFE.

B Predefined operators

This section lists all the predefined operator declarations in Wild LIFE 1.0. As far as possible, the declarations have been kept compatible with the ISO Prolog standard, which is substantially embodied in most current Edinburgh-style Prolog systems.

<table>
<tr><th>Precedence</th><th>Kind</th><th>Operators</th></tr>
<tr><td>1200</td><td>xfx</td><td><| :- -> :=</td></tr>
<tr><td>1200</td><td>fx</td><td>::</td></tr>
<tr><td>1150</td><td>xfx</td><td>|</td></tr>
<tr><td>1100</td><td>xfy</td><td>;</td></tr>
<tr><td>1000</td><td>xfy</td><td>,</td></tr>
<tr><td>900</td><td>fy</td><td>\+</td></tr>
<tr><td>700</td><td>xfx</td><td>= <- <<-</td></tr>
<tr><td>675</td><td>yfx</td><td>xor or</td></tr>
<tr><td>650</td><td>yfx</td><td>and</td></tr>
<tr><td>625</td><td>fy</td><td>not</td></tr>
<tr><td>600</td><td>xfx</td><td>< =< > >= =:= =\= === \=== $< $=< $> $>= $== $\== :< :=< :> :>= :== :>< :\< :\=< :\> :\>= :\== :\><</td></tr>
<tr><td>500</td><td>yfx</td><td>+ - \/ /\</td></tr>
<tr><td>400</td><td>yfx</td><td>mod * / // << >></td></tr>
<tr><td>200</td><td>xfy</td><td>^</td></tr>
<tr><td>200</td><td>fy</td><td>- \</td></tr>
<tr><td>150</td><td>yfx</td><td>.</td></tr>
<tr><td>100</td><td>xfy</td><td>&</td></tr>
<tr><td>75</td><td>fy</td><td>` (backquote)</td></tr>
<tr><td>50</td><td>xfy</td><td>:</td></tr>
</table>

C Glossary

- **Attribute** An attribute is a pair consisting of a label (or field name or feature) and an associated ψ-term. See Section 4.2 (page 11).
- **Bottom** The **sort** that denotes the empty set. The occurrence of bottom in a calculation causes an immediate failure and backtracking to the most recent choice point. It is written as {}.
- **Class** See **declared sort**.
- **Constrained sort** A **declared sort** that has a routine attached to it. This routine behaves as a daemon or dynamic constraint. See Section 7.1.2 (page 30).
- **Constraint** A constraint is a relation between variables. For example, `A=B+4` is a numeric constraint between A and B, and `A=person(age=>B)` is a structural constraint between A and B. The semantics of LIFE can be explained simply in terms of primitive constraints [6].
- **Declaration** See **definition**.
- **Declared sort** A **sort** that has been defined in a `::` or <| declaration. This declaration corresponds to a class definition in an object-oriented language. Declared sorts have data and/or routines attached to them and they are part of a hierarchy.
- **Definition** An assertion that is added to the program. It is terminated with a period "`.`". Also known as a **declaration**. Assertions are predicate clauses, function rules, or sort declarations.
- **Directive** A **query** that occurs in a file.
- **Dynamic routine** A **routine** that may be modified during program execution. The use of dynamic routines should be extremely rare. In most cases, persistent terms should be used instead. For more information on persistent terms see Section 9 (page 60). For more information on dynamic routines see Section 11 (page 71).
- **Feature** A feature is the field name of an **attribute**. For example, in the ψ-term `person(age=>25)`, `age` is a feature. See Section 4.2 (page 11).
- **Function** A **routine** that is called by **matching** and that returns a value. Functions do not guess their answer; they wait until their arguments are sufficiently instantiated to execute. See also **matching** and Section 6 (page 17).
- **Hierarchy** See **inheritance hierarchy**.
- **Identifier** An identifier is any integer or floating point number or character sequence. The character sequence must be surrounded by single quotes if it does not start with a lower-case letter or if it contains non-alphanumeric characters other than underscore. The character sequence may be surrounded by double quotes. A quote character inside a quoted sequence is represented by two quotes. An identifier is either a **declared sort**, a **function**, a **predicate**, or an **uninterpreted identifier**. The first definition of an uninterpreted identifier as function, declared sort, or predicate, means that identifier will always be in that category.
- **Inheritance hierarchy** The partially ordered set of all sorts. It corresponds to the inheritance hierarchy in an object-oriented language. It has a **top** element (which represents the set of all possible objects) and a **bottom** element (which represents the

empty set).

- **Label** See **feature**.
- **LIFE** (Logic, Inheritance, Functions, Equations) A programming language that uses ψ**-terms** as its basic data structure and **unification** and **matching** as its basic operations.
- **Matching** Checking whether one object implies the other. For example, `woman(age=>25)` implies `woman`, so the match succeeds. Matching corresponds to logical implication. See Section 6.2 (page 19).
- **Predicate** A **routine** that is called by **unification** and does not return a value. Predicates may guess an answer; if this answer is incorrect later on, control flow will return to the predicate (by backtracking) and it may produce other answers. See Section 5 (page 14).
- Ψ**-term** The basic data structure of LIFE. Ψ-terms are generalized Prolog terms. They are extensible records that are part of a hierarchy. They have a **root sort** (which corresponds to the type of the record) and **attributes** (fields with values which themselves are ψ-terms). Fields may be added at will and the record's root sort may be refined at will. See Section 4 (page 7).
- **Query** A question that is asked of the system. It ends with a question mark "?". Unless a routine with side effects is called, this does not modify the program.
- **Residuation** What happens with a function call when there is not enough information in the arguments to fire the function nor to fail. The function suspends, or residuates, until there is enough information to decide one way or the other.
- **Root sort** The principal sort of a ψ-term. For example, the root sort of `woman(age=>25)` is `woman`. It corresponds to the main functor in Prolog.
- **Routine** A **function** or a **predicate**.
- **Sort** An identifier that represents a set of objects. A sort corresponds intuitively to a type or class. Sorts may be refined, for example `real` may be refined to `int`. See also **declared sort**, **undeclared sort** and Sections 4.1 (page 8) and 7 (page 29).
- **Static routine** A **routine** that cannot be modified during program execution. Attempting to do so results in an error. See Section 11 (page 71).
- **Symbol** See **identifier**.
- **Top** The **sort** that denotes the set of all records. It corresponds to an unbound variable in Prolog. It is written as `@`.
- **Type** See **sort**.
- **Undeclared sort** An identifier that is an **uninterpreted identifier**, a **predicate**, or a quoted **function**. It is treated as a sort whose only parent is **top** and whose only child is **bottom** (see also **inheritance hierarchy**).
- **Unification** Making two objects equal by restricting the values of each. For example, the unification of `person(age=>25)` and `woman` is `woman(age=>25).` Unification corresponds to logical equality. See Section 4.4 (page 12).
- **Uninterpreted identifier** An identifier that is not a **function**, **predicate**, or **declared sort**. It is treated as a **sort** whose only parent is **top** and whose only child is **bottom** (see also **inheritance hierarchy**).

D Practical information about Wild LIFE 1.0

The system is available by anonymous ftp from gatekeeper.dec.com. After logging in, enter the command `cd pub/plan`, and then the command `bin` to enable binary transfer mode. Then enter the command `get Life1.0.tar.Z` to get the system. Uncompress and untar this file to obtain the Life1.0 directory. See the README file for further instructions.

The Wild LIFE 1.0 system release contains the following.

- The license agreement.
- The C and LIFE source code of Wild LIFE 1.0.
- Documentation. This includes this handbook, a manpage, and documentation files for the tools and example programs. It also includes a list of known bugs and information about porting Wild LIFE 1.0 to various platforms.
- A set of tools written in LIFE. This includes the X interface, the graphical interface toolkit, the accumulator preprocessor, a debugger, a profiler, an extended user interface shell, and a LIFE tokenizer and parser.
- A set of libraries written in LIFE. This provides collections of useful routines that are organized in modules to be imported when needed.
- A set of example programs written in LIFE. This includes SuperLint, a lint-like checker for C with user-customizable checking rules, a flower drawing program with an extensive X interface, an incremental Gaussian equation solver, a program to graphically display ψ-terms, and a simulator of the PRL snack machine. It also includes various smaller programs (queens, boxes, PERT scheduler, Hamming problem, magic squares, natural language parsing, etc.), some of which use the X interface.
- A test suite. This is a set of more than three hundred programs containing more than 30000 lines of LIFE code. These programs contain exhaustive tests of the capabilities of Wild LIFE along with code fragments from real programs. The programs are accompanied with their inputs and correct outputs and two scripts, `check` and `check_all`, which can be used to test the correctness of the implementation.

The following email addresses are relevant to the LIFE language and the Wild LIFE system:

- `life-users@prl.dec.com`. This is a mailing list of people using LIFE or interested in specific aspects of LIFE, whether theory, implementation, or applications. It is meant as a public forum to answer questions and share programs and ideas. It is not meant to report bugs, although it may be used to ask public opinions about surprising behavior of Wild LIFE that may turn out to be a bug and to warn others against confirmed bugs.
- `life-request@prl.dec.com` or `life-users-request@prl.dec.com`. These addresses are used to request to be put on, or removed from, the life-users mailing list.
- `life-bugs@prl.dec.com`. When you strongly suspect a bug (*i.e.*, after reading the handbook and polling life-users's opinion about the symptoms), try to find the *smallest* self-contained program that illustrates the bug and mail it to this address together with a script that shows the bug.

E Manpage

```
                                               wild_life(1)
NAME
    wild_life - interpreter for the LIFE language

SYNTAX
    wild_life [ options ] [ arguments ]

DESCRIPTION
    LIFE (Logic, Inheritance, Functions, Equations) is an experimental
    programming language with a powerful facility for structured type
    inheritance.  LIFE reconciles styles from Functional Programming
    and Logic Programming by implicitly delegating control to an
    automatic suspension mechanism.  This allows interleaving
    interpretation of relational and functional expressions which
    specify structural dependencies on objects.

    The Wild_Life interpreter is a fully functional implementation of
    the LIFE language.  It has a comfortable user interface with
    incremental query extension ability.  It contains an extensive set
    of built-in operations as well as an X Windows interface.

    The Wild_Life interpreter is especially suited for rapid prototyping
    of applications dealing with complex data.  It contains a tool for
    rapid building of interactive window-based interfaces and a powerful
    preprocessor.  The Wild_Life interpreter was originally developed as
    part of the Paradise project at the DEC Paris Research Laboratory.
    Its development is continuing in the Proteus project.

OPTIONS

    -q   Quiet mode.  Forces completely silent operation, i.e., no user
         interface information (prompts, variable values, Yes/No
         messages, startup banner, exit banner) will be printed.  This
         allows Wild_Life to be used as an element of a Unix pipe with
         minimal hassle.  Errors, warnings, trace messages, program
         output (with the write statement etc.), and file I/O are still
         output.  As always, errors and warnings are output to stderr,
         trace information to stdout.  In 'verbose' mode the quiet mode
         is disabled, which allows the user to inspect a misbehaving
         Wild_Life when it is being used as a pipe element.

    -memoryN
    -memory=N
         Start up the system with N words of available memory.  This
         memory is shared between data and programs.  Virtual memory
         usage is close to 2*N because of the half-space garbage
```

```
        collection algorithm used.  The default value of N is 2000000.

    arguments
        All command line arguments are available to the LIFE program.
        The function argv returns a list of strings, where each string
        is one command line argument.  For example, if the system is
        started with "wild_life -q foo" then argv returns the list
        ["wild_life", "-q", "foo"].

FILES
    Life1.0/Doc       (documentation)
    Life1.0/Examples  (example programs)
    Life1.0/Lib       (libraries)
    Life1.0/Tools     (programming tools)
    Life1.0/CLife     (wild_life as C library)
    Life1.0/Tests     (test suite)
    Life1.0/Source    (source code)

EXAMPLES
    The directory Life1.0/Examples contains a set of example programs.
    Each of these programs is in its own module and can be loaded
    directly into the interpreter with the 'import' command.

    The following example shows how to run a program that solves the
    SEND+MORE=MONEY puzzle:

      % wild_life
      Wild_Life Interpreter Version 1.0
      Copyright (C) 1991-93 DEC Paris Research Laboratory
      No customizing file loaded.
      > import("solve")?
      *** File "/udir/rmeyer/LIFE/PUBLIC/Examples/solve.lf" loaded

      *** Yes
      > solve?

       SEND     9567
      +MORE    +1085
      -----    -----
      MONEY    10652

      *** No
      > listing(solve#solve)? % In module "solve", list predicate 'solve'.

      solve :-
              _A = 1,
              _B + _C + _A = _D + 10 * _A,
              _E + _F + _D = _G + 10 * _B,
              _H + _G + _I = _F + 10 * _E,
              _J + _F = _K + 10 * _H,
```

```
            diff_list([_C,_F,_G,_J,_A,_D,_I,_K]),
            _H = carry,
            _E = carry,
            _B = carry,
            _C = decimal,
            _F = decimal,
            _G = decimal,
            _J = decimal,
            _D = decimal,
            _I = decimal,
            _K = decimal,
            nl,
            write(" SEND      ",_C,_F,_G,_J),
            nl,
            write("+MORE     +",_A,_D,_I,_F),
            nl,
            write("-----     -----"),
            nl,
            write("MONEY     ",_A,_D,_G,_F,_K),
            nl,
            nl,
            fail.

      *** Yes
      > halt?

      *** Exiting Wild_Life  [1.850s cpu, 0.000s gc (0.0%)]
      %
```

BUGS

See the installation's README file for a list of known bugs.

CURRENT OWNERS

rmeyer@prl.dec.com (Richard Meyer)
vanroy@prl.dec.com (Peter Van Roy)

AUTHORS OF OBJECT

Richard Meyer
Peter Van Roy
Bruno Dumant (grammar preprocessor, graphical interface toolkit)
Jean-Claude Herve (X Windows interface)
Hassan Ait-Kaci, Seth Copen Goldstein, Abder Aggoun (contributions)

AUTHORS OF DOCUMENTATION

Hassan Ait-Kaci
Bruno Dumant
Richard Meyer
Andreas Podelski
Peter Van Roy

F The accumulator preprocessor

The accumulator preprocessor is a powerful tool to simplify the development of large programs. The preprocessor does a source-to-source transformation that adds accumulators to predicates. An *accumulator* provides a means of calculating a value incrementally. For example, the incremental calculation could be the building of a list. Implementing an accumulator consists in adding two arguments to each predicate and chaining the arguments between goals inside each clause. This passes the intermediate values around during the calculation. The preprocessor can also expand single arguments (called "passed arguments" below) which is useful to pass global information to procedures.

To speed up Wild LIFE's start up time, the preprocessor is not loaded by default. Any program using it must first import (*i.e.*, load and open) the preprocessor module with the command `import("accumulators")`.

Example F.1 This example shows how to use the accumulator preprocessor to write a program `main(N,L)` that takes an input N and generates a list L of integers from 1 to N. The accumulator `myacc` is used to accumulate the elements of the list. This example is shown here in its entirety to show how to use the preprocessor and to give a flavor of its the abilities.

```
> import("accumulators")?
*** Loading File "Tools/accumulators.lf"
*** Loading File "Tools/std_expander.lf"
*** Loading File "Tools/acc_declarations.lf"

> acc_info(myacc,X,In,Out,acc_pred=>(Out=[X|In]))?
> pred_info(loop,myacc)?

> loop(0) :-- !?
> loop(N) :-- N+myacc, loop(N-1)?
> main(N,L) :-- loop(N) with myacc([],L)?

> listing(loop,main)?

loop(0, in_myacc=>A, out_myacc=>A) :- !, succeed.
loop(A, in_myacc=>B, out_myacc=>C) :- D=[A|B],
    loop(A-1, in_myacc=>D, out_myacc=>C).

main(A,B) :- loop(A, in_myacc=>[], out_myacc=>B).

> main(10,L)?

*** Yes
L = [1,2,3,4,5,6,7,8,9,10].
```

The declaration `acc_info` declares the accumulator `myacc`. The declaration `pred_info` declares that predicate `loop` uses `myacc`. The predicates `loop` and `main` are defined using queries with root `:--` (*i.e.*, terminated with `?`) instead of definitions with root `:-`. The listing of `loop` and `main` shows how the accumulator `myacc` is added. Complete explanations of all these items are given below.

The accumulator preprocessor does a generalization of the DCG (Definite Clause Grammar) expansion of Prolog. Definite Clause Grammars (DCGs) are the standard example of a preprocessor for a single accumulator, which is in this case a difference list. This is a standard technique, described in Prolog textbooks, for example in Sterling and Shapiro's Art of Prolog [19]. The Wild LIFE preprocessor replaces Prolog terms by ψ-terms.

The generalized technique, called EDCG (Extended Definite Clause Grammar), was developed and implemented for the Aquarius compiler [16, 17]. It has proven extremely useful in the development of large Prolog programs. It has been used by the authors and others to develop various compilers, simulators, analyzers and test generators.

The Wild LIFE preprocessor jointly developed by Dumant and Van Roy provides much extra functionality over the EDCG preprocessor. It is being used in the development of the LIFE compiler.

F.1 Accumulators

F.1.1 Basic examples and syntax

The rules to be expanded are written:

```
Head :-- Body?
```

`Head` is like any clause head; `Body` is like any predicate definition body except that some special symbols may appear in predicate places (see below: accumulation and other features).

The predicates occurring in the rule are expanded according to the `pred_info` declarations attached to them, that tell the preprocessor which arguments have to be added.

```
pred_info(p,castor)?
```

This declaration states that the accumulator `castor` should be expanded when `p` is encountered. The arguments of `pred_info` may also be lists, to relate a set of predicates and a set of accumulators.

◔◔ If you use named features, be careful that there are no conflicts with the features added by the preprocessor. These features always start with `in_` or `out_`.

Example F.2 (adapted from Van Roy)

```
pred_info([p,q],[castor,pollux])?
pred_info(r,castor)?
```

The clause `p :-- p, q, r, s?` is translated into:

```
p(in_castor => A,in_pollux => B,
  out_castor => C,out_pollux => D) :-
```

```
p(in_castor => A,in_pollux => B,
  out_castor => E,out_pollux => F),
q(in_castor => E,in_pollux => F,
  out_castor => G,out_pollux => D),
r(in_castor => G,out_castor => C),
s.
```

F.1.2 Accumulation

The principal operation performed by an accumulator is to accumulate! The way accumulation is performed for a given accumulator is specified through `acc_info` declarations. These declarations usually contain a predicate (or a sequence of predicates) used for accumulation, that may take three external arguments:

- the two terms implementing the accumulator (`In,Out`);
- the data to be accumulated (`X`).

An `acc_info` declaration contains the name of the accumulator, references to these three terms, and the accumulation predicate:

```
acc_info(AccName,X,In,Out,acc_pred => AccPred)?
```

There are also other optional arguments to `acc_info` declarations which will be introduced later. The syntax for accumulation in a rule is the following:

`X + Acc`: accumulate `X` in accumulator `Acc`.

Example F.3

```
acc_info(fwd,X,In,Out,acc_pred => (Out = [X|In]))?
pred_info(foo,fwd)?

foo :-- 4+fwd, foo?
```

is translated into:

```
foo(in_fwd => A,out_fwd => B) :-
     C = [4|A],
     foo(in_fwd => C,out_fwd => B).
```

The expression in `acc_pred` just replaces the `X + Acc` goal, with the proper arguments instantiated.

F.1.3 Other features

- Initialization.

  An accumulator that has to be expanded and doesn't appear in the head of the clause, is initialized (this happens when `with` is used, see below). Initialization information may be given in the `acc_info` declaration (`in_start` and `out_start` features).

Example F.4

```
acc_info(einstein, in_start=>mc2, out_start=>energy)?
pred_info(t,einstein)?

s :-- t?
```

is translated into:

```
s :- t(in_einstein => mc2,out_einstein => energy).
```

as `einstein` doesn't appear in the head.

If `acc_info` is used several times to define the same accumulator, only the last declaration is taken into account. The system gives a warning.

- `X is Acc` unifies X with the current value of the accumulator Acc.

Example F.5 (with the previous declarations)

```
p :-- p, X is castor, write(X), Y is pollux, write(Y), p?
```

The obtained clause is:

```
p(in_castor => A,in_pollux => B,
  out_castor => C,out_pollux => D) :-
    p(in_castor => A,in_pollux => B,
      out_castor => E,out_pollux => F),
    G = E,
    write(G),
    H = F,
    write(H),
    p(in_castor => E,in_pollux => F,
      out_castor => C,out_pollux => D).
```

It is also possible to write:

- `Acc is X`: same meaning as `X is Acc`
- `Acc1 is Acc2`: unify current value of `Acc1` with current value of `Acc2`

Warning: the expansion fails if none of the arguments of `is` is an accumulator supposed to be expanded.

- `insert(X,Y,Acc)` inserts X and Y in the chain implementing Acc. This primitive is provided as an escape to standard LIFE code. It should not be needed in most cases.

Example F.6 (with the previous declarations)

```
r :-- r, insert(a,b,castor), r?
```

The obtained clause is:

```
r(in_castor => A,out_castor => B) :-
    r(in_castor => A,out_castor => C),
    a = C,
    b = D,
    r(in_castor => D,out_castor => B).
```

- `cond` and disjunctions.

  Expansion is performed inside `cond` and disjunctions.

- Code insertion.

  The preprocessor may be told not to expand a piece of code by inserting it between brackets.

Example F.7 (with the previous declarations)

```
p :-- q , {p}?
```

The obtained clause is:

```
p(in_castor => A,in_pollux => B,
  out_castor => C,out_pollux => D) :-
    q(in_castor => A,in_pollux => B,
      out_castor => C,out_pollux => D),
    p.
```

- Use of cut.

  Cuts may be used anywhere and are never expanded. "`!`" is in fact syntactic sugar for "`{!}`".

- Meta-programming.

  Variables may be used as symbols in the body of the rules:

Example F.8 (with the declarations above)

```
p(X) :-- X?
```

is translated into:

```
p(X,
  in_castor => B,in_pollux => C,
  out_castor => D,out_pollux => E) :-
    interpret_symbols(X,
                      @(castor => B,pollux => C),
                      @(castor => D,pollux => E),
                      @,
                      cname => default_C,
                      gram => false).
```

The features of `interpret_symbols` contain the necessary information to interpret `X` when it gets bound; `interpret_symbols` will residuate until `X` is no longer @. This way, if `X` is instantiated to some symbol `foo`, the first rule will behave exactly like:

```
p :-- foo?
```

A warning is given at expansion time to indicate that variables are expanded using `interpret_symbols`.

☹☹ `interpret_symbols` is a non-strict predicate.

- Changing the argument names.

  If you want special names to be given to the arguments implementing an accumulator, you may specify them in the `acc_info` declaration:

  ```
  acc_info(box,in_name => "input",out_name => "output")
  ```

  With this declaration, `box` will be expanded using `input` and `output` instead of `in_box` and `out_box`.

F.2 Operations on accumulators

F.2.1 Context of an expansion

The tools we have described above don't give good answers to some practical questions such as:

- How can we specify rules like:

  ```
  head :- body1(in => In,out => Inter),
          body2(in => Inter,out => Out).
  ```

  The problem here is that the accumulator is not expanded in the head of the clause.

- How can we link different accumulators together? There is no reason why different accumulators always have to be isolated from each other.

In fact, the accumulators expanded in the head of the clause are very important in the expansion of the clause, because only these accumulators are linked (the others are just initialized). The `with` operator has been designed to let other sets of accumulators play the same role.

An expansion is characterized by two concepts:

- Its *scope*: the set of terms affected by the expansion.
- Its *context*: the set of accumulators that will be linked together during the expansion.

Until now, the scope of an expansion was always the whole clause, and its context the set of accumulators expanded in the head. `With` enables the programmer to define the scope and the context of an expansion. `With` is defined as an infix operator of precedence 800 and kind xfy, *i.e.*, it binds tighter than the control operators used in clauses ("`,`", "`;`", "`:-`", and so on), but looser than functional expressions.

Example F.9 Let us consider the following program:

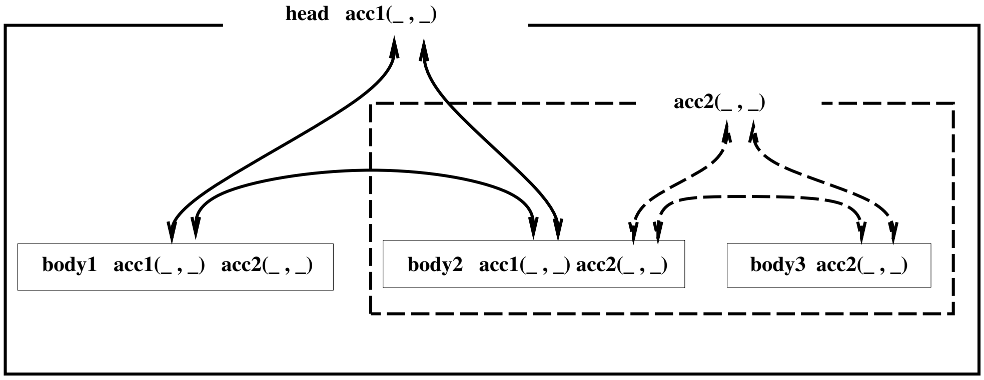
```
pred_info([head,body1,body2],acc1)?
pred_info([body1,body2,body3],acc2)?

head :-- body1, (body2,body3) with acc2.
```

The initial scope of the expansion is the whole clause, with context {`acc1`}: This means that `acc1` will be expanded and linked in `head`,`body1`,`body2`.

`With` defines a new context, {`acc2`}, with scope `(body2,body3)`: this means that `acc2` will be expanded and linked in `body2` and `body3`, but not linked with the arguments of `body1` since `body1` doesn't belong to the scope of this expansion.

This may be represented this way: The scopes are represented by the big rectangles, with the corresponding contexts on their upper edge. Each pair of expanded arguments is represented by a term `acc(_,_)` where the first argument is the input, and the second the output. The arrows represent the unified arguments.



The clause is thus expanded to:

```
head(in_acc1=>A,out_acc1=>B) :-
    body1(in_acc1=>A,out_acc1=>C,in_acc2=>@,out_acc2=>@),
    body2(in_acc1=>C,out_acc1=>B,in_acc2=>@,out_acc2=>D),
    body3(                       in_acc2=>D,out_acc2=>@).
```

When an accumulator appears in the right hand side of a `with` *and* in the parent context, the expansion of the accumulator inside and outside the scope of the `with` are totally independent: arguments appearing inside the scope of the `with` won't be linked with arguments appearing outside.

Example F.10

```
p :-- r, q with castor?
```

is translated into:

```
p(in_castor => A,in_pollux => B,
  out_castor => C,out_pollux => D) :-
    r(in_castor => A,out_castor => C),
    q(in_castor => E,in_pollux => B,
      out_castor => F,out_pollux => D).
```

The arguments implementing the accumulator `castor` in `q` are not linked with the others. `pollux` is regularly expanded.

F.2.2 Operations

The notion of context gives the possibility to perform operations on accumulators. In a context, an accumulator is represented by a pair of arguments `(In,Out)`. In the example above, the castor accumulator is represented by the pair `(A,C)` in the head context, and the pair `(E,F)` in the context defined by the `with` operator.

We define three operations on these pairs of arguments: composition, inversion, equality.

- Composition: =>

  `(A,B) => (C,D)` returns `(A,D)`; `B` and `C` are unified.

- Inversion: `inv`

  `inv((A,B))` returns `(B,A)`

- Equality: =

  `(A,B) = (C,D)` unifies `A` and `C`, and `B` and `D`.

Moreover, direct access to the pairs of arguments is given, which allows the accumulators to be initialized in a convenient way.

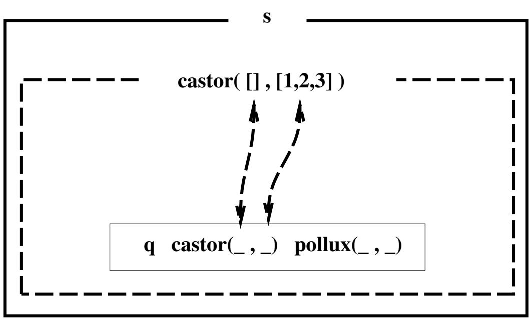
The right hand side of the `with` operator contains a conjunction of constraints to be applied to the new context: either simple constraints meaning: "expand this accumulator locally" or more complex like: "compose these two accumulators", or "make these accumulators equal". What follows is a set of example clauses and their translations.

- `s :-- q with castor([],[1,2,3])?`

  When arguments are given to accumulators in a scope of a `with`, the first one is unified with the input of the accumulator, the second with the output. The above clause is translated into:

  ```
  s :- q(in_castor => [],in_pollux => @,
         out_castor => [1,2,3],out_pollux => @).
  ```

  This may be represented graphically by:

- `s :-- q with castor => pollux?`

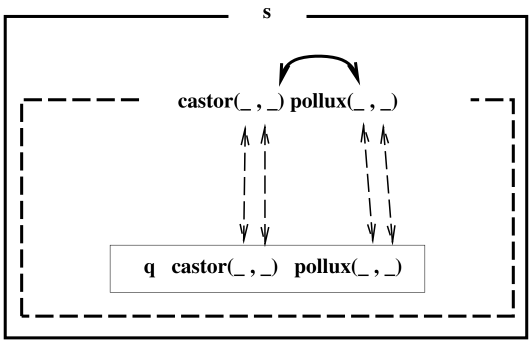
  "=>" is the composition operator. It links the output of its left hand side with the input of its right hand side:

  ```
  s :- q(in_castor => @,out_castor => A,
         in_pollux => A,out_pollux => @).
  ```

  The same clause could have been specified in the following way:

  ```
  s :-- q with (castor(\_,X),pollux(X,\_)).
  ```

Graphically:



The constraints imposed by the right argument are represented by solid arrows, the other links by dashed arrows.

- `s :-- q with inv(castor) => pollux?`

  "`inv`" inverts input and output arguments. In this example, you may notice that both inputs are linked together.

```
s :- q(in_castor => A,in_pollux => A,
       out_castor => @,out_pollux => @).
```

- `s :-- q with castor = pollux?`

  Equality of accumulators means unification of both input and output.

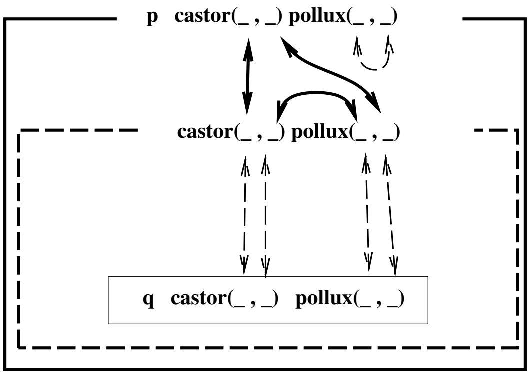
```
s :- q(in_castor => A,in_pollux => A,
       out_castor => B,out_pollux => B).
```

- `p :-- q with glob(castor) = castor => pollux?`

  `glob(castor)` is a reference to the `castor` accumulator appearing in the parent context.

```
p(in_castor => A,in_pollux => B,
  out_castor => C,out_pollux => B) :-
    q(in_castor => A,in_pollux => D,
      out_castor => D,out_pollux => C).
```

Graphically:



- All these things may be used together.

```
p :-- (q, r, s)
    with inv(glob(castor)) =
         castor(begin) => pollux(2 => end)?
```

is translated into:

```
p(in_castor => A: end,in_pollux => B,
  out_castor => C: begin,out_pollux => B) :-
      q(in_castor => C,in_pollux => D,
        out_castor => E,out_pollux => A),
      r(in_castor => E,out_castor => D),
      s.
```

- Initialization may also be performed at the context level.

  The initialization rule given above should be rewritten: An accumulator is initialized if it doesn't appear in the context of the expansion. But it is possible to force initialization, using `init`.

```
acc_info(einstein,in_start=>mc2,out_start=>energy)?
pred_info(t,einstein)?

s :-- t with init(einstein)?
```

is translated into:

```
s :- t(in_einstein => mc2,out_einstein => energy).
```

You may specify which argument has to be initialized by giving a second argument to `init`: `in` or `out`. For example, this clause:

```
s :-- t with init(einstein,in)?
```

is translated into:

```
s :- t(in_einstein => mc2,out_einstein => @).
```

This clause:

```
s :-- t with init(einstein,out)?
```

is translated into:

```
s :- t(in_einstein => @,out_einstein => energy).
```

If the second argument of `init` isn't `in` or `out`, then both are initialized.

F.3 The DCG accumulator

The accumulator preprocessor provides built-in definitions that mimic Prolog's DCG translation. There are a few differences with Prolog DCGs, mainly due to the use of ψ-terms instead of Herbrand terms. The main differences are that the arguments added have features `in_dcg` and `out_dcg` and that declarations given interactively must be terminated with `?`. If `expand_load(true)` is called, then declarations in a file may be terminated with "`.`".

Example F.11 This example shows how to use DCGs in Wild LIFE.

```
> import("accumulators")?
*** Loading File "Tools/accumulators.lf"
*** Loading File "Tools/std_expander.lf"
*** Loading File "Tools/acc_declarations.lf"

> a --> b, c?

> listing(a)?

a(in_dcg => _A, out_dcg => _B) :-
     b(in_dcg => _A, out_dcg => _C),
     c(in_dcg => _C, out_dcg => _B).

*** Yes
```

F.3.1 Definition

The DCG accumulator is predefined as follows:

```
acc_info( dcg, Term, Xs, Ys, acc_pred => 'C'(Term,false,Xs,Ys))?
```

The `'C'(Terms,FoldOk,Xs,Ys)` function is used for accumulation. Its arguments have the following meaning:

- `Terms`: the list representing the non-terminals to be recognized;
- `FoldOk` : a boolean telling whether `Terms` may be folded into the program or not;
- `Xs`: the input stream of non-terminals;
- `Ys`: the output stream of non-terminals.

You may replace the `'C'` function by your own version:

```
set_C(my_C)?
```

`'C'` is only used during the expansion of your grammar in LIFE clauses, so you can (and should) reset its value to the default after translation (with the query `reset_C?`) The tokenizer and parser written in LIFE (see Tools directory) redefine `'C'` for their own needs.

F.3.2 DCG syntax

The standard DCG syntax is supported by this preprocessor:

```
Head --> Body?
```

will expand the DCG accumulator in all predicates of `Head` and `Body` even if there is no `pred_info` declaration. The other accumulators will be expanded according to the declarations.

Accumulation in the DCG may be specified either with the above notation `(X + dcg)`, or using the standard list notation of DCGs:

```
foo --> [3], bar?
```

is translated into:

```
foo(in_dcg => [3|A],out_dcg => B) :-
    bar(in_dcg => A,out_dcg => B).
```

F.3.3 Implementation notes

- Folding Terminals.

  The DCG accumulator has been optimized to allow the folding of terminals (*i.e.*, terms to be accumulated in the DCG accumulator).

  A translation of: `foo --> bar, [1]?` could be:

  ```
  foo(in_dcg => A,out_dcg => B) :-
      bar(in_dcg => A,out_dcg => C),
      C = [1|B].
  ```

  The last statement may be folded into the bar predicate, yielding:

  ```
  foo(in_dcg => A,out_dcg => B) :-
      bar(in_dcg => A,out_dcg => [1|B]).
  ```

This translation is more efficient than the previous one, and the meaning of the two clauses are identical, as long as the bar predicate does not contain things like destructive assignments on its `out_dcg` feature.

When code is inserted — for instance a cut — we have to make sure that this folding does not bind variables occurring before the insertion. Consider the following clause:

```
foo --> !, [1]?
```

This first translation of it (with folding):

```
foo(in_dcg => [1|A],out_dcg => A) :-
     !.
```

does not have the same behavior as this one (without folding):

```
foo(in_dcg => A,out_dcg => B) :-
     !,
     A = [1|B].
```

The first translation is not correct w.r.t. the usual meaning of cut. The translator we propose here deals with this in a very simple way: no folding is performed after a cut or a code insertion. `FoldOk` is the variable used in the translator telling whether folding is authorized or not .

- Code Insertion.

  There are two ways of translating a rule like:

```
foo --> {pred}?
```

  Either as:

```
foo(in_dcg => A,out_dcg => B) :- pred, A = B.
```

  or as:

```
foo(in_dcg => A,out_dcg => A) :- pred.
```

  In most cases, those two translations will have exactly the same behavior, but if some side-effect is expected from `pred`, they may differ. This is the case if `pred` is a cut for instance. The translation used here is the second one, namely:

```
foo --> {pred}?
```

  is equivalent to:

```
foo --> [], {pred}?
```

  The other alternative may be obtained by writing:

```
foo --> {pred}, []?
```

F.4 Passed arguments

A passed argument is just an extra feature added to a predicate, according to some pred_info declaration. The use of passed arguments is less interesting since the introduction of global variables in LIFE.

```
pass_info(ball)?
```

declares ball as a passed argument. There are other optional arguments to this declaration that will be explained later.

To declare that a predicate uses this argument, just add ball to the list in pred_info.

pred_info(basket,[ball])?

The expansion will add a ball feature to all occurrences of basket. If it is not present in the head of the definition, it may be initialized: if the pass_info has a start feature, its value will be used to initialize the passed argument.

Example F.12

```
pass_info(kick,start => quick)?
pass_info(ball)?

pred_info(foot,[ball,kick])?
pred_info(basket,ball)?

basket :-- foot, basket?
```

is translated into:

```
basket(ball => A) :-
     foot(ball => A,kick => quick),
     basket(ball => A).
```

All rules related to expansion of disjunctions, cond, cuts, and insertion of code still hold. The is primitive may be also used with passed arguments. Passed arguments may be used in the left hand side of with operators, but no operations may be performed on them. Their value may be initialized by giving them an argument, but not (yet) using init.

pass_info declaration may also specify an "accumulation" predicate. This predicate can of course only take two arguments, the value to accumulate, and the value of the passed argument.

Example F.13

```
pass_info(hash,X,Pass,acc_pred => (Pass.X = true))?
pred_info(store,hash)?

store(X) :-- X+hash?
```

is translated into

```
store(A,hash => B) :-
    B.A = true.
```

F.5 Common problems and debugging

Debugging a program written using the accumulator expander is not an easy task. One of the most common mistakes is to forget the expansion of facts or to type `:-` instead of `:--`. To help the programmer, a directive may be added to the programs so that warnings are given each time a predicate with an associated `pred_info` declaration occurs in a non expanded rule. This directive is

```
check_expansion?
```

It only works with `expand_load(true)`, and for rules ending with a period.

Example F.14 Consider a file with the following declarations:

```
acc_info(acc)?
pred_info(pred,acc)?

pred :-- pred.
pred.
pred :- pred.
other_pred :- pred.
```

A warning will be generated for each of the last three clauses.

Example F.15 This example illustrates a common problem when mixing DCG predicates with non-DCG predicates. The problem occurs when the predicates contain other accumulators in addition to the DCG accumulator. The problem is solved through judicious use of the `with` operator. Consider the following definition of predicates `a` and `c`:

```
> import("accumulators")?
> acc_info(acc)?
> pred_info([a,b,c,d],acc)?

> c :-- r,s,t? % Non-DCG predicate

> a --> b,c,d? % DCG predicate
```

The expansion of `a` adds the two DCG arguments to `b`, `c`, and `d`:

```
> listing(a)?

a( in_acc => _A, in_dcg => _B,
  out_acc => _C,out_dcg => _D) :-
        b( in_acc => _A, in_dcg => _B,
          out_acc => _E,out_dcg => _F),
        c( in_acc => _E, in_dcg => _F,
          out_acc => _G,out_dcg => _H),
        d( in_acc => _G, in_dcg => _H,
          out_acc => _C,out_dcg => _D).
```

The problem is that `c` is not a DCG predicate, hence it does not know what to do with the two DCG arguments! The fix is to insulate `c` from the DCG accumulator by using `with`:

```
> a --> b, c with dcg, d?  % Correct definition
> listing(a)?

a( in_acc => _A, in_dcg => _B,
  out_acc => _C,out_dcg => _D) :-
        b( in_acc => _A, in_dcg => _B,
          out_acc => _E,out_dcg => _F),
        c( in_acc => _E, in_dcg => @,    % c is bypassed
          out_acc => _G,out_dcg => @),
        d( in_acc => _G, in_dcg => _F,
          out_acc => _C,out_dcg => _D).
```

The DCG arguments are now correctly chained between `b` and `d`, bypassing `c`.

F.6 Term expansion

The preprocessor includes term expansion clauses for `:--`, `-->`, `pred_info`, `acc_info` and `pass_info`. This means that if you execute `expand_load(true)`, then all rules loaded from files may be written as definitions instead of as queries, *i.e.*, ending with a period instead of a question mark.

G The X interface

Wild LIFE provides an X interface that allows programming X applications at the LIFE level. To speed up Wild LIFE's start up time, the X library is not loaded by default. Any program using X must first import (load and open) the X interface module with the command `import("x")`. For examples of how these routines are used, look at the sample programs provided with the Wild LIFE release.

A "+" before a variable name means the field is an input, and a "−" means it is an output. Arguments mentioned as default may be left out; in that case the default values are used. Certain arguments may not be changed by the LIFE programmer; they may only be created by X routines and passed into X routines. This includes the Display, Window, and Font arguments. A few arguments must be strings; these are indicated by the notation "`:string`".

For additional functionality that makes it very easy to create interactive window-based applications (with buttons, menus, and so on) the graphical interface toolkit should be imported. See appendix H for more information.

G.1 Event mask values

The named event mask types are:

```
xNoEventMask, xKeyPressMask, xKeyReleaseMask,
xButtonPressMask, xButtonReleaseMask, xEnterWindowMask,
xLeaveWindowMask, xPointerMotionMask, xPointerMotionHintMask,
xButton1MotionMask, xButton2MotionMask, xButton3MotionMask,
xButton4MotionMask, xButton5MotionMask, xButtonMotionMask,
xKeymapStateMask, xExposureMask, xVisibilityChangeMask,
xStructureNotifyMask, xResizeRedirectMask,
xSubstructureNotifyMask, xSubstructureRedirectMask,
xFocusChangeMask, xPropertyChangeMask, xColormapChangeMask,
xOwnerGrabButtonMask.
```

G.2 Primitive control operations

- `xOpenConnection (-Display, +Screen:string)`

  Open an X connection on the specified screen. The default value of `Screen` is the contents of the environment variable `DISPLAY`. The field containing this value may be omitted.

- `xCloseConnection (+Display)`

  Close the connection opened by a `xOpenConnection`.

- `xCreateWindow (+Display, +X, +Y, +Width, +Height, -Window, color => +Color, windowtitle => +WindowTitle, icontitle => +IconTitle, eventmask => +EventMask)`
  Open a window on the specified display at `(X,Y)` with the given width and height, and with the given background color. `EventMask` contains

a bitwise `or` of the accepted events. The default values are `Color: xWhite`, `WindowTitle: "Life"`, `IconTitle: "Life"`, `eventmask: xKeyPressMask\/xButtonPressMask\/xExposureMask` (where `\/` is the bitwise or function). The fields containing these values may be omitted.

- `xShowWindow (+Window)`

  Show the window. There are no default values.

- `xHideWindow (+Window)`

  Hide the window. There are no default values.

- `xRefreshWindow (+Window)`

  Refresh the window. There are no default values.

- `xPostScriptWindow (+Window, +Filename:string)`

  Output the window in a PostScript file. The default value of `Filename` is `"X.ps"`.

- `xGetWindowGeometry (+Window, -X0, -Y0, -Width, -Height)`

  Return the geometry of the window. There are no default values.

- `xSetWindowGeometry (+Window, +X0, +Y0, +Width, +Height)`

  Modify the geometry of the window. There are no default values.

- `xSetWindowColor (+Window, +Color)`

  Modify the background color of the window. There are no default values.

- `xDestroyWindow (+Window)`

  Destroy the window. There are no default values.

- `xRequestColor (+Window, +Red, +Green, +Blue, -Color)`

  Return a color entry in the color map of the window with the closest RGB. The arguments Red, Green, and Blue must be integers in the range 0 through 255. There are no default values.

- `xRequestNamedColor (+Window, +Name, -Color)`

  Return the color entry in the color map of the window of the named color. The argument Name must be a string recognized by the X system; a list of these is given through the interactive X command `xco`.

- `xFreeColor (+Window, +Color)`

  Free a color allocated by `xRequestColor` or `xRequestNamedColor`. There are no default values.

G.3 Primitive drawing operations

- `xDrawLine (+Window, +X0, +Y0, +X1, +Y1, function => +Function, color => +Color, linewidth => +LineWidth)`

  Draw the line from `(X0,Y0)` to `(X1,Y1)` on the window with the given function, color, and linewidth. The default values are `Function: xCopy`, `Color: xBlack`, and `LineWidth: xThinLine`. The possible values of `Function` are: `xClear`, `xAnd`, `xAndReverse`, `xCopy`, `xAndInverted`, `xNoop`, `xXor`, `xOr`,`xNor`, `xEquiv`, `xInvert`, `xOrReverse`, `xCopyInverted`, `xOrInverted`, `xNand`, `xSet`.

- `xDrawRectangle (+Window, +X0, +Y0, +Width, +Height, function => +Function, color => +Color, linewidth => +LineWidth)`

  Draw the rectangle starting at upper-left corner `(X0,Y0)` of specified width and height on the window. The default values and possible values are the same as for `xDrawLine`.

- `xDrawArc (+Window, +X0, +Y0, +Width, +Height, +StartAngle, +ArcAngle, function => +Function, color => +Color, linewidth => +LineWidth)`

  Draw an arc in the rectangle `(X0,Y0,Width,Height)`, starting at angle `StartAngle` relative to the 3-o'clock position and extent `ArcAngle`, angles given in degrees. The default values and possible values are the same as for `xDrawLine`.

- `xDrawOval (+Window, +X0, +Y0, +Width, +Height, function => +Function, color => +Color, linewidth => +LineWidth)`

  Draw an oval in the rectangle `(X0,Y0,Width,Height)` on the window. The default values and possible values are the same as for `xDrawLine`.

- `xFillRectangle (+Window, +X0, +Y0, +Width, +Height, function => +Function, color => +Color)`

  Same as `xDrawRectangle` but filled with a given color.

- `xFillArc (+Window, +X0, +Y0, +Width, +Height, +StartAngle, +ArcAngle, function => +Function, color => +Color)`

  Same as `xDrawArc` but filled with a given color.

- `xFillOval (+Window, +X0, +Y0, +Width, +Height, function => +Function, color => +Color)`

  Same as `xDrawOval` but filled with a given color.

- `xFillPolygon (+Window, +PointsList, +Function, +Color)`

  Fill a polygon described by a list of points (*e.g.*, `[(100,100), (200,300), (300,100)]`). The polygon is closed automatically if the last point of the list does not coincide with the first point.

- `xLoadFont (+Display, -Font, +FontName:string)`

  Load the specified font name. Valid font names are system-dependent. They may be found in `/usr/lib/X11/fonts`. The default font name is `"9x15"`.

- `xDrawString (+Window, +X0, +Y0, +String, font => +Font, function => +Function, color => +Color)`
  Draw a string at `(X0,Y0)` with the specified font on the window. The font has to be loaded with `xLoadFont`. This function does not affect the background pixels of the bounding box of the string. Note: `(X0,Y0)` is the lower left coordinates of the string. The default values and possible values are the same as for `xDrawLine`.

- `xDrawImageString (+Window, +X0, +Y0, +String, font => +Font, function => +Function, color => +Color)`
  Draw a string at `(X0,Y0)` with the specified font on the window. The font has to be loaded with `xLoadFont`. The background pixels of the bounding box of the string are filled with the background color of the window. Again, `(X0,Y0)` is the lower left coordinates of the string. The default values and possible values are the same as for `xDrawLine`.

- `Event = xGetEvent (+Window, eventmask => +EventMask)`

  Return an event in the window which matches the given mask. The currently implemented events are `mouse_event`, `keyboard_event`, and `expose_event` such that:

  ```
  E: mouse_button button => B:bool, x=>X:int, y=>Y:int)
  E: keyboard_event (keycode=>K, char=>C:int)
  E: expose_event
  ```

  For a list of the default values and possible values of EventMask see `xCreateWindow`. If there is no event available, the function residuates (waits) until one is. Multiple calls to `xGetEvent`, on the same window and/or on multiple windows, may be pending at the same time.

H The graphical interface toolkit

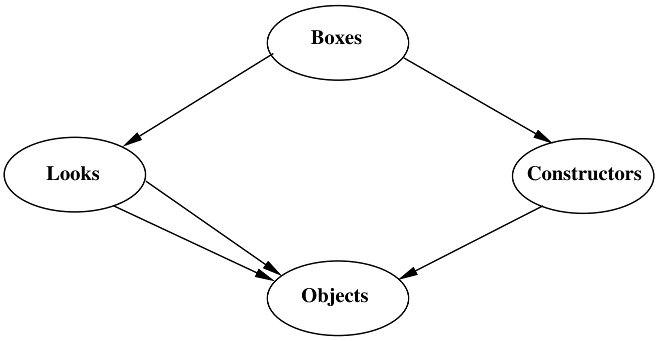
H.1 Introduction

`xtools` is a simple toolkit to build interactive window-based X applications in Wild LIFE. It provides the user with the basic functionality of bigger toolkits, in short the ability to use buttons, text fields, menus, and sliders. Composite objects containing these primitives can be created arbitrarily at run-time. The toolkit is built on top of the basic X interface described in the previous section. The toolkit module is loaded and opened with the command `import("xtools").`

The toolkit is organized around three concepts, namely boxes, looks, and constructors.

- *boxes* are used to compute the sizes and positions of objects on the screen. All screen objects manipulated by the toolkit are subsorts of `box`.
- *constructors* are used to build and initialize screen objects. All objects that have a behavior (i.e. not simple graphical objects, but real widgets) inherit from one constructor type. Ten of them are predefined.
- *looks* are used to describe the appearance of screen objects. An object may be a subsort of several look types (four such subsorts are predefined), and will inherit the appearance of these "looks".

These three concepts are defined as sorts and are organized in the following inheritance hierarchy (multiple inheritance is possible from looks):



The next sections give details about boxes, constructors, looks, and the predefined objects inheriting from these. An example program is provided with the system (in file `ex_tools.lf`) and should help the user to get started.

H.2 Boxes and their placement constraints

All the objects manipulated by the toolkit are boxes. A box is defined by the following type declaration:

```
:: box(X,Y,width => DX,height => DY,
```

```
        border => B,
        mother => M).
```

X and Y are integers giving the coordinates of the top left corner of the box, DX and DY are integers giving the dimensions of the box. Boxes may contain other boxes: the mother feature of a box points to the box that contains it, if any. The border feature is the width of reserved space on each side of a box. It has a default value d_border.

The following sections list the placement constraints on boxes that are implemented in the toolkit. These constraints may be accumulated and imposed in any order. The local constraint propagation of Wild LIFE guarantees that if the constraints are consistent and enough information exists to determine a placing, it will be determined. If the constraints are inconsistent, then they will fail and cause backtracking.

H.2.1 Boxes used as padding

Some boxes are only used to reserve space between objects:

- h_box(W) is a function that returns a box of width W.
- v_box(H) is a function that returns a box of height H.
- null_box is a box of zero size. It is the sort:

```
null_box <| box.
:: null_box(width => 0,height => 0).
```

The values in h_box(W) and v_box(H) may be negative.

H.2.2 Positioning

- Relative positioning

  The toolkit offers a number of primitives to place boxes:

l_above	c_above	r_above
l_below	c_below	r_below
t_left_of	c_left_of	b_left_of
t_right_of	c_right_of	b_right_of

  The letter prefixes have the following meaning: l stands for left, r for right, t for top, b for bottom, and c for center.

  Each of these primitives is a function returning the smallest box containing its two arguments; for instance, Box1 l_above Box2 returns the smallest box containing Box1 and Box2, such that:

  - Box1 is above Box2, and
  - their left sides are aligned.

  These primitives will set and try to resolve the placement constraints.

- Containment

  The toolkit offers two primitives to express that one box contains another:

  `contains` `containing`

  Syntax:

  ```
  Box1 contains Box2
  Box = Box1 containing Box2
  ```

  Both of these primitives express the same containment constraint. The difference is that `contains` is a predicate and `containing` a function. If no size is specified for Box1, it will be given the same size as that of Box2. The function call `containing` returns the containing box, in this case `Box1`.

  If `Box1` has a `border` feature worth `Border` (in pixels), it will be used to reserve a space of that width around the box. In this case, `Box1` will be larger than `Box2`.

- Refined positioning

  There are also some primitives that set finer constraints:

`ll_aligned`	`lr_aligned`	`lc_aligned`	`rr_aligned`	`rc_aligned`
`tt_aligned`	`tb_aligned`	`tc_aligned`	`bb_aligned`	`bc_aligned`
`cc_v_aligned`	`cc_h_aligned`			

  These are predicates. The first letter of the predicate name applies to the first argument, the second to the second argument. As before, `l` stands for left, `r` for right, `t` for top, `b` for bottom, and `c` for center.

  For instance:

  ```
  Box1 lr_aligned Box2
  ```

  will force the left side of `Box1` to be aligned with the right side of `Box2`.

  ```
  Box1 cc_v_aligned Box2
  ```

  will force the centers of `Box1` and `Box2` to be vertically (`v`) aligned.

H.2.3 Lists

Lists are just syntactic sugar to express the vertical or horizontal alignment of boxes. The following list-handling primitives are provided:

`vl_list`	`vc_list`	`vr_list`
`ht_list`	`hc_list`	`hb_list`
`menu_list`		

All these functions are defined as prefix operators. The call `vl_list List_of_Boxes` returns the box containing all the boxes of the list, such that each of them is `l_above` the next one in the list. As before, `l` stands for left, `r` for right, `t` for top, `b` for bottom, and `c` for center. The call `menu_list List_of_Boxes` first constrains all the boxes of the list to be of the same size, then returns `vl_list List_of_Boxes`. It is very easy to make your own kind of list, using the implementation of these as an inspiration.

H.2.4 Sizes of boxes

A very useful constraint predicate is `same_size`. `same_size(List_of_Boxes)` will force all the boxes of the list to have the same height and width. In the same way, `same_height(List_of_Boxes)` will force all the boxes of the list to have the same height, and `same_width(List_of_Boxes)` will force all the boxes of the list to have the same width.

Sizes of boxes are computed on the fly, using a subsort of box: `t_box`. It has the following features:

```
t_box(h_space => HSpace,
      v_space => VSpace,
      text => Text,
      font_id => Fid)
```

`Text`	The text appearing in the box.
`VSpace`	The *total* amount of vertical space reserved around the text.
`HSpace`	The *total* amount of horizontal space reserved around the text.
`Fid`	The font ID used. Default is bold (see below for an explanation of font IDs).

If no size is already given for a box, and if it is a subtype of `t_box`, then its size is computed according to `Text`, `Fid`, `VSpace` and `HSpace`.

H.2.5 Creating a box

In order to be displayed and to work, a box has to be *created*.

`create_box(Box)` calls the constructor of `Box` (if it is a subsort of a constructor) and the drawing routine (if the box is a subsort of a look sort). If a box contains other boxes, you only need to call `create_box` for the parent box: the call is propagated to the boxes' children.

`create_box` must be called only after all positioning constraints have been declared. It is in fact possible to separate completely the positioning and the creation. `create_box` may be called several times with the same argument. Later calls than the first will have no effect.

H.3 Main constructors

H.3.1 Panels

- `panel_c`

  A `panel_c` consists of a top-level window containing widgets.

  Features: (optional)

  ```
  panel_c(title => Title)
  ```

  `Title`: title of the window and icon

  Beware: the positions of top-level windows are usually modified by the window manager.

- `sub_panel_c`

  A simple sub-window that deals with refresh events. It is used by slide bars.

H.3.2 Buttons

The following button types are provided, listed with the relevant features:

```
push_c(action => Action)
on_off_c(on => On, action => Action)
text_field_c(action => Action)
menu_button_c(menu => Menu)
```

`Action` Buttons of sort `on_off_c`, `push_c`, `text_field_c` have a feature `action` that describes the action activated by the button. The default action is succeed. If the button is an `on_off_c` or a `push_c`, the action is activated each time the mouse is pressed and released inside the button. If the button is a `text_field_c`, the action is activated each time the return key is pressed and the button active.

`On` Buttons of type `on_off_c` have a boolean feature `on` that describes their state. `On` is a persistent term.

`Menu` A `menu_button` has a feature `menu` that must contain a term of sort `menu_panel_c`.

To distinguish between the mouse buttons, a persistent variable `button_pressed` is modified each time a mouse button is pressed. Its value is 1 for the first button, 2 for the second button, etc.

H.3.3 Menus

- `menu_panel_c`

  A `menu_panel_c` is essentially a `panel_c` with a different kind of window. A menu panel is always positioned under its associated menu button. In fact, a menu panel may contain any object, exactly like a panel.

- `item_c`

  Features:

  ```
  item_c(action => Action)
  ```

  `Action` The action associated with the item.

H.3.4 Sliders

A slider is just a moving button. It may move either vertically (`v_slider_c`) or horizontally (`h_slider_c`), inside the box that contains it.

Features:

```
*_slider_c(min => Min,max => Max,value => Value,action => Action)
```

`Min,Max,Value` The position of the slider is associated with a real value that is constrained to stay between `Min` and `Max`. `Min` and `Max` must be given by the user. `Value` is a persistent term.

`Action` Each time the value of the slider is updated by moving the slider, `Action` is executed.

H.4 Looks

H.4.1 Look types

- `text_box`

  A `text_box` appears as text. `text_box` is of course a subsort of `t_box`.

  Features:

```
text_box(text => Text,
         text_state => State,
         text_color_id => Tid,
         true_text_color_id => TTid,
         font_id => Fid,
         offset  => Offset).
```

  (a `text_box` also has the features of `t_box`)

`Text`	The text appearing in the box.
`Offset`	Default value is `d_offset`.
`Offset = 0`	The text is centered in the box.
`Offset > 0`	The text is flushed left, and `Offset` is the distance between the left border of the box and the beginning of the text.
`Offset < 0`	The text is flushed right, and `Offset` is the distance between the right border of the box and the end of the text.
`State`	A boolean describing the state of the button. `State` is a persistent term.
`Tid`	The color ID used when State is false. The color value is found in `main_colors` (see Colors below). Default value is `d_text`.
`TTid`	The color ID used when `State` is true. Default value is `d_text`.
`Fid`	The font ID used. Default is `bold`.

- `frame`

  A `frame` corresponds to the 3D border of a button.

  Features:

```
frame(frame_state => State,
      flat => Flat,
      color_id => Cid)
```

`State`	If `State` is true, the frame is sunken, otherwise it is raised. State is a persistent term.
`Flat`	If `Flat` is true, then there is no raised position: When `State` is false, the frame appears flat.
`Cid`	The color ID used. The actual color values are found in the persistent variables `highlight_colors` and `shade_colors`.

- `field`

  A field is a colored rectangle.

  Features:

```
field(field_state => State,
      color_id => Cid,
      true_field_color_id => TFid)
```

`State`	A boolean describing the state of the button. `State` is a persistent term.
`Cid`	The color ID used when `State` is false. The color value is found in `main_colors` (see Colors below). Default is `d_field`.
`TFid`	The color ID used when `State` is true. Default is `d_selected_field`.

- `led`

  An led is just like an LED, *i.e.*, it is a small light that can be on or off.

  Features:

```
led(led_state => State,
    led_on_color_id => LedOn,
    led_off_color_id => LedOff)
```

`State`	A boolean describing the state of the led. `State` is a persistent term.
`LedOn`	The color ID used when `State` is true. The color values are found in `main_colors`, `highlight_colors` and `shade_colors`, depending on the part of the led (see Colors below). Default is `d_led_on`.
`LedOff`	The color ID used when `State` is false. Default is `d_led_off` .

H.4.2 Inheritance of looks

Looks are inherited through subtyping. For instance, `on_off_button` is a subsort of `text_box`, `led` and `frame`. Note that the `color_id` feature appears in `frame` and `field`. Therefore, they should be compatible.

H.4.3 Colors and fonts

Colors and fonts are stored in tables. There are three tables for colors (`main_colors`, `highlight_colors`, `shade_colors`) and one table for fonts. Colors and fonts are accessed through identifiers that may be any atom. All objects have default colors (stored in `xtools_constants`). To change the color of an object, you have to:

- Store a color in the appropriate table, with the ID you have chosen, using the predicate `def_color(Table,Id,Color)` (for a font: `def_font(Id,Font)`).
- Set the appropriate color ID of the object to Id.

Example H.1 To have a class of text boxes with red text:

```
def_color(main_colors,my_id,red)?

my_txt <| text_box.
:: my_txt(text_color_id => my_id)
```

As the same `color_id` feature appears in `field` and `frame`, if a color is defined for the ID `I` in `main_colors`, then the corresponding colors (for the same ID `I`) in `shade_colors` and `highlight_colors` should be respectively a dark and light version of the same color.

To load a new color, use `new_color`:

Example H.2 To add to the color table, the color with RGB values 180,190,190, type:

```
X = new_color(180,190,190)?
```

`new_color` returns the color corresponding to the RGB values.

Example H.3 To load a new font, use `new_font`:

```
X = new_font("helvetica_bold18")?
```

`new_font` returns the font corresponding to the string. The string must be one of the names obtained by typing xlsfonts (Unix command).

All widgets (objects with their own window, in short all subsorts of constructors) have a `color_id` feature to set the background color of the window.

Two font IDs are predefined:

```
bold  : "-*-helvetica-bold-r-*-*-14-*-*-*-*-*-*-*"
medium: "-*-helvetica-medium-r-*-*-14-*-*-*-*-*-*-*"
```

The following colors are loaded by default:

```
aquamarine, black, blue, 'blue violet', brown, 'cadet blue',
coral, 'cornflower blue', cyan, 'dark green', 'dark olive green'
'dark orchid', 'dark slate blue', 'dark slate grey',
'dark turquoise', 'dim grey', firebrick, 'forest green',gold,
goldenrod, green, 'green yellow', grey, 'indian red', khaki,
'light blue', 'light grey', 'light steel blue', 'lime green',
magenta, maroon, 'medium aquamarine', 'medium blue', 'medium orchid',
'medium sea green', 'medium slate blue', 'medium spring green',
'medium turquoise', 'medium violet red', 'midnight blue',
'navy blue', orange, 'orange red', 'orchid', 'pale green', pink,
plum, red, salmon, 'sea green', sienna, 'sky blue', 'slate blue',
'spring green', 'steel blue', 'light brown', thistle, turquoise,
violet, 'violet red', wheat, white, yellow, 'yellow green'.
```

These are loaded when the X toolkit is loaded, and may be used wherever a `Color` parameter is indicated.

H.5 The hierarchy of graphical interface objects

Here is the object hierarchy of the graphical interface toolkit.

- Panels:

```
panel <| panel_c.
panel <| frame.

menu_panel <| menu_panel_c.
menu_panel <| frame.

slide_bar <| sub_panel_c.
slide_bar <| frame.

v_slide_bar <| sub_panel_c.
v_slide_bar <| v_slide_l.

h_slide_bar <| sub_panel_c.
h_slide_bar <| h_slide_l.
```

- Buttons:

```
push_button <| push_c.
push_button <| text_box.
push_button <| frame.

on_off_button <| on_off_c.
on_off_button <| led.
on_off_button <| text_box.
on_off_button <| frame.

text_field_button <| text_field_c.
text_field_button <| field.
text_field_button <| frame.
text_field_button <| text_box.

menu_button <| menu_button_c.
menu_button <| frame.
menu_button <| text_box.

menu_item <| item_c.
menu_item <| frame.
menu_item <| text_box.
```

The complete definition is in Tools/xtools.lf.

H.6 Screen objects

The screen objects manipulated by the X toolkit are subsorts of looks and/or constructors. They usually have an additional feature that stipulates how the states of the look and that of the constructor are linked (`change_state`).

I The C-LIFE interface

I.1 Description

This interface provides a simple but powerful means of calling the Wild LIFE interpreter from within C programs. Routines are provided to:

- state facts,
- issue queries,
- recover results,
- extract data from ψ-terms,
- manipulate the current query status of Wild LIFE:
  - generate more solutions,
  - reset the system.

The interface behaves in pretty much the same way as the top-level of the Wild LIFE interpreter, so being familiar with the interpreter (and needless to say, the LIFE programming language) is necessary. This also makes the interface very easy to use. No means are provided to build ψ-terms directly other than through successive queries.

I.2 A simple example

The following is a simple C program that calls Wild LIFE to print `"Hello World!!"`.

```
#include "c_life.h"

 main(int argc, char *argv[])
 {
      WFInit(argc,argv);
      WFProve("write(\"Hello World!!\")?");
 }
```

One can compile it with:

```
> cc -o hello hello.c c_life.a -lm -lX11
```

and execute it:

```
> hello
Hello World!!
```

I.3 Summary of functions and prototypes

- Initialize Wild LIFE
  `void WFInit(int argc,char *argv[])`
- Submit a query:
  `int WFInput(char *query);`
- Get a variable's value:
  `PsiTerm WFGetVar(char *name);`
- Get the type of a ψ-term:
  `char *WFType(PsiTerm psi);`

- Get the value of a ψ-term if it's a double:
  `double WFGetDouble(PsiTerm psi, int *ok);`
- Get the value of a ψ-term if it's a string:
  `char *WFGetString(PsiTerm psi, int *ok);`
- Count the features of a ψ-term:
  `int WFFeatureCount(PsiTerm psi);`
- Get the value of a feature:
  `PsiTerm WFGetFeature(PsiTerm psi, char *featureName);`
- Get all the feature names as a NULL-terminated array of strings:
  `char **WFFeatures(PsiTerm psi);`
- Prove a goal and report an error (to stderr) on failure:
  `WFProve(char *goal);`

I.4 Memory management

As LIFE is a non-deterministic programming language, it is best to view it as a coroutine running in tandem with the C program, and which is queried by the C program. Its execution state may be very different from the calling C program.

Wild LIFE uses its own memory management scheme (with garbage collection). At each call to `WFInput` the interpreter changes state, and may completely re-map its memory layout, thus rendering obsolete any C variables pointing into the LIFE memory space. Here is an example of dangerous programming:

```
PsiTerm a;
double n;
WFProve("A=123?");
a=WFGetVar("A");
WFProve("B=fact(A)?"); /* Might cause a call to the GC */
n=WFGetDouble(a,NULL); /* Random results-might even crash */
```

To avoid this problem, do not keep values of type PsiTerm in C variables across calls to Wild LIFE.

The other side of the coin is that thanks to the garbage collector, it is not necessary for the C program to worry about freeing memory in LIFE's memory space (in fact, doing so would corrupt the integrity of the system). Currently the only function requiring the C programmer to free memory is `WFFeatures` which allocates a string array with `malloc`. The array has to be freed with `free`. The strings within it should be left alone since they are in LIFE's space.

I.5 An exhaustive example

The following example displays the correct (and short-cut) use of all the current features of the interface. Read it carefully as some of these are not documented elsewhere.

The program generates the following output:

```
> cc -g -o demo demo.c /udir/rmeyer/LIFE/MODULE/c_life.a -lm -lX11
> demo
Welcome to Wild-LIFE!!
WFInput succeeded and there may be more answers
true=false? failed (demo.c, line 51)
C=4
```

```
1
2
3
4
; failed (demo.c, line 69)
6
7
message(three+four,equals => 7)
The type of A is user#message
A has 2 features
sum=7
the first feature is: 'three+four'
feature 1 => built_in#string
feature equals => built_in#int
Linking X library...
ok
% Here it runs the 'queens' program...
```

The program is in the file "`demo.c`", here is a listing:

```
/* Example C program calling Wild-LIFE */
#include "../Source/c_life.h"

main(argc,argv)
int argc;
char *argv[];
{
  int ans;

  PsiTerm a;
  PsiTerm sum;
  char **features;
  int i;
  double value;
  int ok;

  /*** Initialize Wild-LIFE ***/
  WFInit(argc,argv); /* Currently doesn't use the arguments */

  /*** Submit a query ***/
  /* \042 is a quote sign (") */
  ans=WFInput("write(\042Welcome to Wild-LIFE!!\042),nl?");

  /* Deal with Wild-LIFE response */
  switch(ans) {

  case WFno:
    printf("WFInput failed\n");
    break;

  case WFyes :
```

```
    printf("WFInput succeeded\n");
    break;

  case WFmore:
    printf("WFInput succeeded and there may be more answers\n");
    break;
  }

  /*** This query fails and so prints an error message ***/
  WFProve("true=false?");

  /*** Solve a simple constraint ***/
  WFProve("A=B+C?");
  WFProve("B=1?");
  WFProve("A=5?");
  WFProve("write('C=',C),nl?");
  WFProve(".");

  /*** Backtrack over 4 solutions ***/
  WFProve("A={1;2;3;4},write(A),nl?");
  WFProve(";");
  WFProve(";");
  WFProve(";");
  WFProve(";"); /* No more at this point */

  /*** Backtrack over only the first 2 solutions ***/
  WFProve("A={6;7;8},write(A),nl?");
  WFProve(";");
  WFProve("."); /* Return to top level */

  /*** Build a psi-term and query it ***/
  WFProve(
   "A=message(\042three+four\042,equals=>3+4),write(A),nl?");

  /* Read the variable 'A' */
  a=WFGetVar("A");
  if(!a) { /* Error checking, here for demonstration only */
    fprintf(stderr,"Couldn't read variable 'A'\n");
    exit(1);
  }

  /* Print the type of 'A' */
  printf("The type of A is %s\n",WFType(a));

  /* Get the number of features of 'A' */
  printf("A has %d features\n",WFFeatureCount(a));

  /* Get the feature 'equals' */
  sum=WFGetFeature(a,"equals");
  if(!sum) { /* Error checking, here for demonstration only */
```

```
    fprintf(stderr,"Couldn't read feature 'equals'\n");
    exit(1);
  }

  /* Get the value of 'sum' */
  value=WFGetDouble(sum,&ok);
  if(!ok) { /* Error checking, here for demonstration only */
    fprintf(stderr,"'sum' is not a real number\n");
    exit(1);
  }
  printf("sum=%lg\n",value);

  /* Get the first feature */
  /* You can use NULL in WFGetDouble and WFGetString if you are */
  /* sure the psi-term contains a value of the correct type. */
  printf("the first feature is: '%s'\n",
       WFGetString(WFGetFeature(a,"1"),NULL));

  /* Get the features as a NULL terminated string array */
  features=WFFeatures(a);
  if(features) {
    for(i=0;features[i];i++) {
      printf("feature %s => %s\n",
           features[i],
           WFType(WFGetFeature(a,features[i])));
    }
    free(features); /* Recommended */
  }
  else { /* Error checking, here for demonstration only */
    fprintf(stderr,"'A' has no features\n");
    exit(1);
  }

  /* Run the queens program */
  WFProve("import(\042queens\042)?");
  WFProve("queens?");

  /* Loop over each solution */
  do {
    sleep(1);
    printf("retrying\n");
    ans=WFInput(";");
    printf("ans=%d\n",ans);
  } while(ans);
}
```

References

1. Hassan Aït-Kaci, Robert Boyer, Patrick Lincoln, and Roger Nasr. Efficient implementation of lattice operations. *ACM Transactions on Programming Languages and Systems*, 11(1):115–146 (January 1989).

2. Hassan Aït-Kaci and Jacques Garrigue. Label-selective λ-calculus. PRL Research Report 31, Digital Equipment Corporation, Paris Research Laboratory, Rueil-Malmaison, France (1993).

3. Hassan Aït-Kaci and Patrick Lincoln. LIFE—A natural language for natural language. *T.A. Informations*, 30(1–2):37–67 (1989). Association pour le Traitement Automatique des Langues, Paris, France.

4. Hassan Aït-Kaci and Roger Nasr. LOGIN: A logic programming language with built-in inheritance. *Journal of Logic Programming*, 3:185–215 (1986).

5. Hassan Aït-Kaci and Roger Nasr. Integrating logic and functional programming. *Lisp and Symbolic Computation*, 2:51–89 (1989).

6. Hassan Aït-Kaci and Andreas Podelski. Functions as passive constraints in LIFE. PRL Research Report 13, Digital Equipment Corporation, Paris Research Laboratory, Rueil-Malmaison, France (1991).

7. Hassan Aït-Kaci and Andreas Podelski. Towards a meaning of LIFE. PRL Research Report 11, Digital Equipment Corporation, Paris Research Laboratory, Rueil-Malmaison, France (1991).

8. Hassan Aït-Kaci and Andreas Podelski. Towards a meaning of LIFE. In Jan Maluszyński and Martin Wirsing, editors, *Proceedings of the 3rd International Symposium on Programming Language Implementation and Logic Programming (Passau, Germany)*, pages 255–274. Springer-Verlag, LNCS 528 (August 1991).

9. Hassan Aït-Kaci and Andreas Podelski. Order-sorted feature theory unification. PRL Research Report 32, Digital Equipment Corporation, Paris Research Laboratory, Rueil-Malmaison, France (May 1993).

10. William F. Clocksin and Christopher S. Mellish. *Programming in Prolog*. Springer-Verlag, Berlin, Germany, 2nd edition (1984).

11. Tim Lindholm and Richard A. O'Keefe. Efficient implementation of a defensible semantics for dynamic Prolog code. In *Proceedings of the Fourth International Conference on Logic Programming*, pages 21–39. MIT Press (May 1987).

12. Richard Meyer. Compiling LIFE. Technical Report 8, Digital Equipment Corporation, Paris Research Laboratory (September 1993).

13. Lee Naish. *Negation and Control in Prolog*. Springer-Verlag, LNCS 238 (1986).

14. Lee Naish. Negation and quantifiers in NU-Prolog. In *Proceedings of the 3rd International Symposium on Logic Programming*, pages 624–634. Springer-Verlag, LNCS 225 (July 1986).

15. Richard O'Keefe. *The Craft of Prolog*. MIT Press, Cambridge, MA (1990).

16. Peter Van Roy. A useful extension to Prolog's Definite Clause Grammar notation. *ACM SIGPLAN Notices*, pages 132–134 (November 1989).

17. Peter Van Roy. *Can Logic Programming Execute as Fast as Imperative Programming?* PhD thesis, Department of Computer Science, University of California at Berkeley (December 1990).

18. Ehud Shapiro. The family of concurrent logic programming languages. *ACM Computing Surveys*, 21(3):412–510 (1989).

19. Leon Sterling and Ehud Shapiro. *The Art of Prolog*. Series in Logic Programming. MIT Press, Cambridge, MA (1986).

Index

0 **The Wild LIFE Handbook (prepublication edition)**

Hassan Aït-Kaci, Bruno Dumant, Richard Meyer, Andreas Podelski, and Peter Van Roy

PARIS RESEARCH LABORATORY
85, Avenue Victor Hugo
92563 RUEIL MALMAISON CEDEX
FRANCE