TR-I-0055

# Typed Feature Structures II
# The Language and its Implementation

Rémi ZAJAC

1988.12

Abstract

A previous report has given the motivations and has described the formal framework for the integration of attributes structures and feature structures. This report describes the functional interpreter which has been developped in order to demonstrate the feasability of the approach.

ATR Interpreting Telephony Research Laboratories
ATR 自動翻訳電話研究所

# Typed Feature Structures : the Language and its Implementation

Rémi Zajac*

*ATR Interpreting Telephony Research Laboratories*
*Twin 21, MID Tower*
*2-1-61 Shiromi, Higashi-ku, Osaka 540, Japan*

*[zajac%atr-ln.atr.junet@uunet.uu.net]*

**Abstract:** A Machine Interpretation system will presumably use various specialized linguistic programming languages. At some points, there will probably be transitions from an attributed tree to a feature structure, and vice versa. A common point between attribute structures and feature structures is that attribute structures could be considered degenerated feature structures.

On the other hand, most grammar formalisms using unification of feature structures introduce a variety of augmentations that hamper the declarativeness of the formalism and the monotonicity of the computation of these equational formalisms.

In the spirit of functional programming (à la LISP), we have proposed a framework for such augmentations based on the interpretation of feature structures as objects (not as denotations). We have introduced the associated functions derived from set theory. For example, union is defined as a weak form of unification.

We have developed an experimental functional interpreter to test these ideas. The basic data structure of the language is the typed feature structure. Operations available include conditional instructions and sequence of instructions, boolean expressions, assignment, unification, and various functional operations on typed feature structures. Definitions include type definitions, template definition and (recursive) function definitions. This interpreter has been integrated in an Earley parser. The grammar formalism offer the grammar writer enhanced expressive power and allows using a more modular approach to grammar developement.

A previous report has given the motivations for this approach, and described the formal framework used for the language. This report describes the language and its implementation.

**Key Words:** Equational formalisms, unification, functionnal formalisms, feature structures, attribute structures, functional programming, equational programming.

---

* Visiting researcher from GETA, UJF-CNRS, 38041 Grenoble, cedex 53X, France.

## TABLE

# 1. INTRODUCTION

## 1.1 Purposes of the implementation

The language we introduce is based on typed feature structures. The approach has been motivated and described in [Zajac 88]. We view feature structures not as denotation of sets, but as objects standing for themselves. We argue that this interpretation is the most commonly used by many computationnal linguists – not theoreticians, but people who actually develop grammars for specific applications. One consequence is to allow the use of a functional programming style, such as the LISP style, as opposed to an equational programming style, such as the PROLOG style. The main characteristic of our language is that it allows freely merging the functional style of programming and the declarative style used in most of the proposed unification-based grammar formalisms, using the same data structure. This feature is new in the field of computational linguistics, and should be explored further.

The present implementation of the language is intended to be an experimental testbed for a specialized language for linguistic programming that could include a feature structure-like data structure. Consequently, this implementation is by no means definitive, and other implementations could be developed depending on the insight gained while developing programs and grammars.

## 1.2 Scope of the report

This report describes a language for manipulating typed feature structures. In its present form, it cannot stand as a grammar formalism in itself, even if it can be developed in this direction. Nevetheless, this language has already been integrated in an Earley parser developed by [Nicolas 88], and is used as an evaluator for expressions describing the structures associated to the symbols of a grammar rule. This report is a description of the rules's expression evaluator.

# 2. THE LANGUAGE

## 2.1 Introduction

The language is quite simple and offers considerably less power than full general programming languages, such as C or LISP. The first reason is that it is designed to be used as part of a more general specialised language for linguistic programming. The second reason is that this language is in an experimental stage. Furthermore, developing a full programming language without preliminary tests and experiments with a proptotype is not the most secure way to get good and usable tools. Nevertheless, it has been designed to be easily augmented, and even in its present state, integrated in a parser, it has all the basic capabilities of unification-based grammar formalisms such as D-PATR [Shieber 86].

## 2.2 Objects: syntax, definition and evaluation

### *A. True, false and undefined values*

The false value in a logical framework for a feature structure is generally represented by $\perp$, the bottom of the type lattice. However, as we are also working in a functional framework, we need to relate a logical boolean value to a functional value. There is a special symbol `*undef*` which represents the *absence of value*, and has a special interpretation for each operation (see the corresponding definitions). This could be used to determine if a feature has a bound value or not.

Each value returned by the interpreter has an associated type, which is related to the type of the expression evaluated. The combination of a value and a type gives a truth value. There are only two possibilities:

1. The value has the boolean type: {} represents the truth value FALSE, and any other value represesnt the value TRUE (see the behavior of `*undef*` in the definitions of boolean operators).

2. The value has either the number type, the string type, or a structure type: `*undef*` represents the FALSE value, and any other value represent the TRUE value.

One should note that if the empty set {} does not have the boolean type, it is interpreted as TRUE: the empty set is a value, and not the absence of a value.

## B. Numbers

Three kinds of numbers are defined: integers, ratios and reals. Predicates and arithmetic operations are defined on each kind of number, and there is no need for different operators's symbols for integers or ratios. The kind of number is recognised by its syntax, and the arithemetic interpreter makes the necessary conversions.

An integer is written as a sequence of decimal digits, optionally preceded by a sign (+ or -) :

| | |
|---|---|
| -0 | zero (always equal to +0 !) |
| 6 | positive integer |
| -33. | negative integer |

A ratio is either an integer or the ratio of two integers, the numerator and the denominator, separated by a '/'. The denominator is not preceded by a sign. If the result of some operation is a ratio, it will always be printed in its canonical form: the greatest common divisor of the denominator and the numerator is one: -4/6 is a non-canonical form of -3/2.

A real number is a sequence of digits optionally preceded by a sign, followed by a decimal point, and optionally, a sequence of digits: -3.1415 the negation of $\pi$.

Operations on numbers are boolean tests and arithmetic operations.

## C. Strings

Lists of characters are represented as strings, enclosed by double quotes ("). A character can be any printing character of the ASCII set. Some characters may have a special role (", for example): to include such characters in a string, they must be preceded by the *escape* character '\'. This rule applies to '\' itself: one must double this character to include it in a string.

The ordering for strings is the lexicographic ordering: upper case letters have the same rank as lower case letters, digits are ranked before letters, non alphanumeric characters are not taken into account: this is the «dictionary ordering» (see details in section 2.3.B).

Operations on strings are boolean tests and functions (e.g., concatenation).

## D. Atomic symbols

An atom is either a number, a string or an alphanumeric symbol beginning with a letter. An atom is a «ground» value denoted by itself. The only operation that can be performed on it is the equality test.

## *E. Structures*

Structures are the basic data structure of the language, as the list is the basic data structure of LISP.

A structure is a record-like data structure. It is a generalisation of the feature structures used in unification-based grammar formalism: each feature has a type [Shieber 86, Aït-Kaci 84], and atomic features are either symbols, numbers or strings (atomic structures).

In the present version of the language, the type system has a degenerated lattice structure and type definitions are intended to be able to define legal domains of values. Types need not to be explicitly stated in a feature structure, as they must be statically declared: the interpreter associates each symbol of a feature with its declared type. The user can create, modify or destroy structures using assignment, unification, union, intersection of structures.

There are four kinds of elementary operations which are defined on structures:

- predicates without side effects are tests, for example inclusion, membership, etc. These predicates return a boolean value. See section 2.5.B Boolean functions on structures.
- predicates with side effects are unification and generalisation. They return a boolean value and modify their arguments. See section 2.5.C Boolean predicates on structures.
- functions create new objects from their arguments: union, conjunction, intersection, etc. See section 2.7 Functions on structures.
- assignment which takes the object returned by the right-hand-side of the assignment, and puts it at the location specified by the left-hand-side (erase the previous content). See section 2.6.B Assignment.

A structure is written in approximatly the same manner as traditionnal feature-structures, surrounded by braces (usual notation for sets). Syntacticaly, there is no distinction between atomic structures and complex structures: as every feature has to be declared, the system knows the type of each feature. For example, the structure [Pollard and Sag 87]

```
[ PHON  Kim
  SYN   [LOC   [HEAD [MAJ    N
                      NFORM  NORM]
               SUBCAT []]]
 SEM    [CONT KIM]]
```

will be written :

```
{ PHON:"Kim",
  SYN:{LOC:{HEAD:{MAJ:N,
                  NFORM:NORM},
            SUBCAT }},
 SEM:{CONT:KIM}
```

The differences from traditional notations are the following:

- an atomic structure is written between braces, as for complex feature-structures;
- feature SUBCAT is written without values: this means that the values might be anything;
- if feature SUBCAT were written SUBCAT:{}, this would mean that the value of feature SUBCAT is the *empty set*, and not that the feature is a complex feature.

## *F. Paths, variables and expressions*

Variables are used to store and access structures. A variable is an alphanumeric symbol prefixed by a star: `*cat31` is a variable symbol.

Sub-structures are accessed using paths. A path is a variable symbol optionally followed by feature symbols concatenated with a period: `*cat33.cat.subcat` is a path. It can be used to get the sub-structure dominated by the features cat and subcat of a structure stored in variable `*cat33`. It can also be used in the left-hand-part of an assignment to replace one sub-structure with another:

```
*cat33.cat.subcat <- {cnoun}.
```

An expression is a recursive combination of operators and operands: operators can take either constants, variables, paths, or, recursively, complex expressions as operands.

## *G. Templates*

A template is used to store a frequently used constant structure. The definition of a template is static and cannot be modified by assignment (but it can be redefined). A template symbol is an alphanumeric symbol prefixed with a %: `%agr` is a template symbol.

A template symbol can occur anyplace a structure is expected: the template symbol is simply replaced with its definition.

### *H. Functions*

Functions are the operations which are used to manipulate structures. There are two kinds of functions: pre-defined functions and user defined functions. Pre-defined functions may be simple operations (addition of numbers) or more complex operations (conjunction of structures, print function,...). The user may define his own functions by composition of other functions. In a function definition, recursive definitions are allowed. A user defined function symbol is an alphanumeric symbol prefixed with a & sign. It takes the same place as any pre-defined operator sign or pre-defined funcion identifier.

A function always returns a structure as the result of its evaluation. It could be used to build a new structure or to modify existing structures. Some functions act on the environment of the intepreter itself (for printing, writing of modifying some interpreter's parameters).

### *I. Programs*

A program consists of a set of definitions: type definitions, template definitions and functions definitions. Type definitions are mandatory. Running a program consists of loading the definitions in the interpreter memory and evaluating some expression (combination of functions). The input data can be fed interactively or read in a file. The output can be printed at the terminal or in a file.

## 2.3 Types

There are two kinds of types: atomic types and complex types. There are 3 pre-defined atomic types: string, number, and symbol. The three following sub-sections describe the operations on strings and numbers. The last sub-section describes the type system for complex types. Operations on complex types are described in the following sections (2.4, ...).

### *A. Numbers*

*Predicates*

The result of a predicate *P*, n1 *P* n2, is undefined if n1 or n2 is undefined:

```
n1 P ? → n1;   ? P n2 → n2.
```

EQUALITY: **n1 = n2** is true if n1 equals n2, false if n1 is different from n2.

INEQUALITY: **n1 /= n2** is true if n1 is different from n2, false if n1 equals n2.

STRICT INFERIORITY: **n1 < n2** is true if n1 is less than n2, false if n1 is greater than or equal to n2.

INFERIORITY: **n1 <= n2** is true if n1 is lesser than or equal to n2, false if n1 is greater than n2.

*Functions*

ADDITION: **n1 + n2** returns the sum of n1 and n2. If n1 or n2 is undefined, n2 or n1 is returned: `n1 + ? → n1; ? + n2 → n2.`

SUBTRACTION: **n1 - n2** returns the difference of n1 and n2. If n2 is undefined, n1 is returned. If n1 is undefined, the result is undefined: `n1 - ? → n1; ? - n2 → ?.`

MULTIPLICATION: **n1 * n2** returns the product of n1 and n2. If n1 or n2 is undefined, n2 or n1 is returned: `n1 * ? → n1; ? * n2 → n2.`

DIVISION: **n1 / n2** returns the ratio of n1 and n2. If n2 is undefined, n1 is returned; if n1 is undefined, the result is undefined: `n1 / ? → n1; ? / n2 → ?.`

## *B . Strings*

*Predicates*

The result of a predicate *P*, s1 *P* s2, is undefined if s1 or s2 is undefined:

```
s1 P ? → s1;   ? P s2 → s2.
```

The ordering is the lexicographic «ordering of the dictionnary»:

- when a string contains letters or digits, all others characters are stripped for comparison;
- upper case and lower case letters are of the same rank;
- digits precede letters;
- non alpha-numeric characters precede digits;
- non alpha-numeric characters are ordered using the ASCII ordering.

EQUALITY: **s1 = s2** is true if s1 equals s2, false if s1 is different from s2.

INEQUALITY: **s1 /= s2** is true if s1 is different from s2, false if s1 equals s2.

STRICT INFERIORITY: **s1 < s2** is true if s1 is less than s2, false if s1 is greater than or equal to s2.

INFERIORITY: **s1 <= s2** is true if s1 is less than or equal to s2, false if s1 is greater than s2.

*Functions*

CONCATENATION: **s1 + s2** returns the concatenation of s1 and s2: `x + y` → `x.y`. If s1 or s2 is undefined, s2 or s1 is returned: `s1 + ?` → `s1; ? + s2` → `s2.`

Example: `"Ab" + "cD"` → `"AbcD"`

SUFFIX SUBTRACTION: **s1 /- s2** returns s1 minus the greatest common suffix of s1 and s2: `x.u /- y.v` → `x.` If s2 is undefined, s1 is returned. If s1 is undefined, the result is undefined: `s1 /- ?` → `s1; ? /- s2` → `?.`

Examples: `"abcd" /- "xcd"` → `"ab"`

`"abcd" /- "x"` → `"abcd"`

`"abc" /- "abc"` → `""`

SUFFIX INTERSECTION: **s1 /* s2** returns the geatest common suffix of s1 and s2: `x.u /* y.u` → `u.` If s1 or s2 is undefined, s2 or s1 is returned: `s1 /* ?` → `s1;` `? /* s2` → `s2.`

Examples: `"abcd" /- "xcd"` → `"cd"`

`"abcd" /- "x"` → `""`

`"abc" /- "abc"` → `"abc"`

PREFIX SUBTRACTION: **s1 -/ s2** returns s1 minus the greatest common prefix of s1 and s2: `u.x .- u.y` → `x.` If s2 is undefined, s1 is returned. If s1 is undefined, the result is undefined: `s1 -/ ?` → `s1; ? -/ s2` → `?.`

Examples: `"abcd" /- "abx"` → `"cd"`

`"abcd" /- "x"` → `"abcd"`

`"abc" /- "abc"` → `""`

PREFIX INTERSECTION: **s1 */ s2** returns the geatest common prefix of s1 and s2: `u.x */ u.y → u.` If s1 or s2 is undefined, s2 or s1 is returned: `s1 */ ? → s1;` `? */ s2 → s2.`

Examples: `"abcd" */ "abx" → "ab"`

`"abcd" */ "x" → ""`

`"abc" */ "abc" → "abc"`

## *C. Structures*

All kinds of complex typed feature-structures (structures for short) must be declared. In the current implementation, there is one general type, called 'DECORATION', which must always be defined. All top structures belong to this type (this limitation will be removed in future versions of the language). A declaration of structures might be as follows:

```
:TYPES
DECORATION = logic : logic-t,
         sf    : functions,
         k     : classes,
         lex   : string,
         cat   : cat-t;
logic-t = :cardinality 1,
          pred, arg1, arg2, arg3;
functions = :cardinality 1,
            gov, subj, obj1, obj2, atgov, atsubj, atobj;
```

All possible features for a type must be listed. A complex feature has a type: the feature SF has type FUNCTIONS, the feature LEX has the pre-defined type STRING. It is possible to define some constraints on the legal set of values of a type: maximum number of values, and minimum number of values. The key-word `:CARDINALITY` introduces a couple of integers: `:CARDINALITY 0 5` means that the minimum number of value of a feature having this type is zero, and the maximum number is five.

The default cardinality is `:CARDINALITY 0 <number of declared features>`. When only one number is specified, it is the *maximum* cardinality.

When a feature must have only one atomic value, it must be declared as either a number, a string, or a user's defined type with maximum cardinality 1 (such as the feature FS of type FUNCTIONS in the previous example).

The rationale for type declaration is to have the possibility of type-checking during loading of expressions, and during evaluation of expressions: an evaluation will fail as soon as an illegal structure is built. Before an expression is effectively loaded in the memory of the interpreter, the reader performs checks to ensure that all objects have been declared: a program that can be loaded is not only syntacticaly correct, but must also be semantically correct, with respect to the declarations of objects, thus removing from the programmer the burden of debugging an error which occurs because of the existence of an illegal object.

In this version, all features must have a unique explicitly defined type. In a future version, we envisage the introduction of anonymous types, and also of a type structure with inheritance, allowing for more flexibility.

## 2.4 Templates and functions

### A. Templates

A template is simply a name for a constant structure. Writing an expression, one can use the name in place of the whole structure thus allowing for more concise expressions. A structure can be either a complex one or an atomic structure (string, number).

Templates are declared as follows:

```
:TEMPLATES
V = {syn:{loc:{head:{maj:V}}} ;
mainV = {syn:{loc:{head:{aux:-, inv:-}}}} ;
finiteV = {syn:{loc:{head:{vform:fin}}}};
```

In an expression, a template name must be prefixed with the % sign: `%finiteV`, for example. The name is simply replaced with its definition (in fact, this replacement occurs at loading for type checking, and for speeding-up computation).

### B. Functions

One original feature of our language is that it offers to the grammar writer not only templates (which can be defined as constant functions), but also true function definitions. As templates are written in place of constant structures, functions are written in place of expressions. Writing an expression, one can replace some expression with a call to a function. This also allows for more concise expressions.

One limitation of the present implementation is that expressions are not allowed inside structures: structures are manipulated through path accesses, as in most unification based grammar formalisms.

Recursive definitions are allowed. For example, the factorial function can be defined as follows:

```
:FUNCTIONS
fact(*x)  = COND
                 *x=0   : 1;
                 *x/=0  : *x * &fact(*x - 1)
               ECOND;
```

A function is defined with any kind of expression. During evaluation of a function call, the function call is replaced by the function definition (the function «body»), with all variables in this expression bound to the values found in the function call. Functions can be used to define sets of equations between paths (set of unifications). An agreement wich is written as a set of equations between paths can be defined once, and called in several distinct places. If the writer wants to modify the agreement condition, he need only modify the definition.

For example, in an expression, a function call might be `&fact(5)` and the evaluation will return `120`. It can also be called with a path `&fact(*A.num)`, assuming that the feature `num` of structure `*A` is an integer. It can also be called with any expression that will return an integer. The effect of the recursive definition of factorial is, in fact, to replace the expression `&fact(3)` with the expression `3 * &fact(3-1)` which reduce to `3 * (2 * &fact(2-1))`, etc.

## 2.5 Boolean expressions

There are two kinds of boolean expressions:

- Boolean functions that return a boolean value and do not modify the operands;
- Boolean predicates that return a boolean value and modify the arguments.

This distinction is introduced to differentiate boolean operations, such as the test of equality of two objects, and operations that fall in the unification framework (in the present implementation, unification and generalisation).

## *A. Boolean operators*

Boolean operators take boolean values and return boolean values. These operations are functions which are evaluated immediatly. They should be distinguished from conjunction-disjunction-complement operations which are defined to describe sets of feature structures. The latter are not defined in this implementation. However, a set of equations between paths can be written as a boolean AND expression with the same expected behavior (this is not true for OR and NOT operations). To be able to implement such behavior when a boolean expression is evaluated as FALSE, all modifications on structures that were done during the evaluation of this expression are undone. The practical interpretation is simple: these operations are primarily intended to write tests (in conditional expressions, see section 2.6.E Conditional). However, if the evaluation suceeds, the modifications become permanent. Alternative sets of path equations can be written as

```
    eq1 & eq2 & eq3
  | eq4 &eq5
```

The first AND expression `eq1 & eq2 & eq3` is evaluated. If it succeds, the second is not evaluated. If it fails on `eq3` for example, the unifications of `eq1` and `eq2` are undone, and the second AND expression is evaluated. The result of the whole expression is then the result of this evaluation.

As the evaluation of an AND expression is performed in order, the grammar writer can use this ordering to control the evaluation (for example, to evaluate simple constraints first). Of course, the result of the evaluation of a set of equation is independant or any particular order of equations.

AND: **x & y** → `if x=TRUE and y=TRUE then TRUE else FALSE.` The rules for the undefined value are: `x & ? → x; ? & y → y.`

OR: **x | y** If the first operand evaluates to TRUE, the second is not evaluated.

```
x | y → if x=TRUE
            then TRUE
             else if y=TRUE then TRUE else FALSE;
```

The rules for the undefined value are: `x | ? → x; ? | y → y.`

NOT: **^ x** → `if x=TRUE then FALSE else TRUE.` The rule for the undefined value is: `^ ?  → ?.`

## *B. Boolean functions on structures*

Boolean functions on structures take structures as operands and return a boolean value. They do not modify the operands.

In definitions of operations, e represents an atomic structure, f represents a feature structure (atomic or complex), and l represents a list of feature-structures.

UNIFIABLE: **x ?>< y** tests if x and y can be unified: if x >< y succeeds, returns TRUE otherwise returns FALSE.

UNIFIED: **x ><? y** tests if x and y have been unified: returns TRUE if they have been unified (they share their values) and FALSE if not.

EQUALITY: **x = y** checks recursively the feature/value pairs of x and y: all paths of x must be in y and vice versa. Note that this does not take sharing into account. Equality could be defined as: `{l1}={l2}` $\rightarrow$ `∀ x ∈ {l1}, x ∈ {l2}` <u>and</u> `∀ x ∈ {l2}, x ∈ {l1}`

INEQUALITY: **x /= y** $\rightarrow$ <u>if</u> `x=y` <u>then</u> `FALSE` <u>else</u> `TRUE.`

INCLUSION: **x < y** or **y > x** checks recursively that the feature/value pairs of x are in y: all paths of x must be in y. Note that sharing is not taken into account. Inclusion can be defined as follows:

```
1. {} < {l2} → TRUE

2. {l1} < {} → FALSE

3. {e1{l1}} < {e1{l2}, l3} → {l1} < {l2} or {e1{l1}} < {l3}

4. {f1} < {f2, l2} → {f1} < {l2}

5. {f1, l1} < {l2} → {f1} < {l2} and {l1} < {l2}
```

MEMBERSHIP WITH EQUALITY TEST: **x @ y** returns TRUE if the feature-structure x belongs to the structure y: it is true if there is at least one feature-structure of y which is *equals* to x. Membership with equality test can be defined as follows:

```
1. f ∈ {} → FALSE

2. e{l} ∈ {e{l1}, l2} → l = l1 or e{l} ∈ {l2}

3. f ∈ {f1, l1} → f ∈ {l1}
```

MEMBERSHIP WITH INCLUSION TEST: **x @< y** returns TRUE if the feature-structure x belongs to the structure y: it is true if there is at least one feature-structure of y which contains x (x is *included* in it). The definition is the same as for @, with equality test = in line 3 replaced by inclusion test <.

MEMBERSHIP WITH UNIFIABLE TEST: **x @>< y** returns TRUE if the feature-structure x belongs to the structure y: it is true if there is at least one feature structure of y which can be *unified* with x. The definition is the same as for @, with equality test = in line 3 replaced by unifiable test ?><.

## *C. Boolean predicates on structures*

There are two operations implemented: unification and generalisation. Both can be used in writing a set of equations: unification and generalisation are distributive relative to each other.

UNIFICATION: **x >< y** If x and y are paths in the expression, they have as the new value the conjunction (see section 2.7.A Conjunction and union) of the sub-structures dominated by x and y, and are co-indexed.

If x is a path and y is a constant structure (and not a sub-structure accessed through a path), the new value of x is the conjunction of y and the sub-structure dominated by x (and vice versa if x is a constant structure and y a path). There is, of course, no co-indexing of x itself.

The types of x and y are checked: if the new value for x or y is not of the type of x or y, the unification fails (returns FALSE), otherwise it succeeds and returns TRUE.

GENERALISATION: **x <> y** If x and y are paths in the expression, they have as the new value the intersection (see section2.7.B Intersection and strong intersection) of the sub-structures dominated by x and y, and are co-indexed.

If x is a path and y is a constant structure (and not a sub-structure accessed through a path), the new value of x is the intersection of y and the sub-structure dominated by x (and vice versa if x is a constant structure and y a path). There is, of course, no co-indexing of x itself.

The types of x and y are checked: if the new value for x or y is not of the type of x or y, the unification fails (returns FALSE), otherwise it succeeds and returns TRUE. In fact, the only thing which is checked during evaluation is the maximum cardinality constraint. The minimum cardinality constraint is checked after evaluation.

## 2.6 Control structures

### *A. Constants, variables and paths*

A constant is a structure (either atomic or complex) written as itself: the evaluation of a constant returns the constant itself: the number 5 returns 5, the structure `{cat:noun}` returns `{cat:noun}`.

A variable is a means of storing and accessing a structure. Lexically, it is written as an identifier prefixed with a star. The evaluation of a variable returns its value, the structure which has been associated with it. If the structure `{cat:noun}` is associated with the variable `*X`, the evaluation of `*X` returns `{cat:noun}`.

A path is either a variable or a variable with a concatenation of feature identifiers. A path is a means of accessing a sub-structure of the variable. Using the the previous exemple, the path `*X.cat` accesses to the value of `cat`: the evaluation of `*X.cat` returns `{noun}`.

### *B. Assignment*

The assignment function is one of the basic operations for modifying structures (the others are unification and generalisation). Assignment takes two arguments: a path and a constant structure (result of some evaluation), and replaces the structure dominated by the path with the constant structure.

For example, suppose that the structure `{cat:noun}` is associated with the variable `*X`. If we want to change the value of `cat` from `noun` to `verb`, we can write `*X.cat <- {verb}`, and `*X` will have the new value `{cat:verb}`.

Type checking is performed: if the value is of the type of the path, the assigment function succeds and returns the value assigned (which is equivalent to the boolean value TRUE), otherwise the assignment fails and returns the boolean value FALSE. This peculiarity is introduced in order to use assignment in the same way as unification, which can succeed or fail. Of course, when the order of a set of unifications and generalisations is not important (the result is the same for any order of equation), it is important for assignment. As AND expressions are evaluated in order, the grammar writer can use this ordering to control the order of evaluation of a set of equations written as an AND expression.

## C. Function and template calls

A template call is simply a template identifier prefixed with the % sign: it is replaced with its definition. For example, the template mainV defined as `mainV = {syn:{loc:{head:{aux:no, inv:no}}}}` is called `%mainV`, and returns `{syn:{loc:{head:{aux:no, inv:no}}}}`.

A function call has the form `&<function identifier>(<list of arguments>)`. The variables in the head of the definition (formal parameters) are replaced with the actual parameters (arguments of the function call). The actual parameters replace the formal parameters of the body of the function definition, and the body is evaluated. The function call returns the result of this evaluation. A function can, of course, modify structures in the environment (passing a path as a parameter of a function call).

For example, suppose that the variable `*X` contains the feature `num{3}`, and take the function `fact` defined in section 2.4.B:

```
fact(*x) = :COND *x=0  : 1;
                 *x/=0 : *x * &fact(*x - 1)   :ECOND;
```

The call of `&fact(*X.num)` will be evaluated as follows:

1. The call form is replaced with the body in which the formal parameter `*x` is replaced with the actual parameter 3, evaluation of the form `*X.num`:

```
:COND 3  = 0  : 1;
      3 /= 0 : 3 * &fact(3 - 1)   :ECOND;
```

2. This expression is evaluated: the first clause of the conditional 3=0 is evaluated as FALSE, and does not apply; the second clause applies and the expression is further rewritten:
`3 * &fact(3-1)`, then:

→ `3 *  &fact(2)`

→ `3 * (2 *  &fact(1))`

→ `3 * (2 * (1 * &fact(0)))`

→ `3 * (2 * (1 * 1))`, which is eventually rewritten as 6.

## *D. Sequence*

The SEQUENCE control structure allows evaluating a list of instructions in order. An instruction can be any expression. Each instruction of the sequence is separated form the following instruction with a semi-colon. For example, the following operations permutate the values of `*X.tamp` (say A) and `Y*.tamp` (say B):

```
:BEGIN
 *Z.tamp <- *X.tamp ;
 *X.tamp <- *Y.tamp ;
 *Y.tamp <- *Z.tamp ;
:END
```

If the values of `*X.tamp` and `*Y.tamp` were respectively A and B before the evaluation of the sequence, after the execution `*X.tamp` contains B and `*Y.tamp` contains A.

The result of the evaluation of a sequence is the result of the evaluation of the *last* instruction of the sequence. For example, if during the execution of a sequence, an instruction which is a unification fails, the evaluation continues regardless of this particular result, and returns the result of the evaluation of the last instruction.

## *E. Conditional*

The conditional instruction, borrowed from LISP, allows writing expressions which are evaluated depending on certain conditions. See as example, the factorial function section 2.4.B.

The conditional instruction is a sequence of clauses, separated with a semi-colon. Each clause has a condition part and an action part (separated with a colon). The action part is in turn a sequence of instructions separated with a comma. The general form is

```
:COND
  <cond1> :: <inst11>, <inst12>, ...;
  <cond2> :: <inst21>, <inst22>, ...;

  <condN> :: <instN1>, <instN2>, ... <instNM>
:ECOND
```

The evaluation of the conditional is as follows: the clauses are evaluated in sequence. For each clause, the condition part is first evaluated: if the result is FALSE, the evaluation of the clause stops and returns FALSE, and the following clause is evaluated. If the condition returns TRUE, the action part is then evaluated: each instruction of the action is evaluated in sequence.

The result of the conditional is the result of the last instruction of the clause. If no condition applies, the conditional returns FALSE.

Note that if some condition has side-effects (modify some structure), and if the whole condition returns FALSE, these modifications are undone.

## 2.7 Functions on structures

### *A. Conjunction, union and unification*

UNION: **x + y** returns the union of the sets of feature-structures x and y. This union is recursively defined as follow:

```
1     {} + {l2} → {l2}
2     {l1} + {} → {l1}
3     {e1{l1}, l2} + {e1{l3}} → { e1{[{l1}+{l3}]} } + {l2}
4     {f1, l1} + {f2} → {f1, [{l1}+{f2}] }
5     {l1} + {f2, l2} → ( {l1} + {f2} ) + {l2}
```

As described in [Zajac 88], when some feature-structure is co-indexed, there are two cases: if it is a feature which is present in both operands (line 3 of the definition above), this co-indexing is removed in the result; if the feature-structure is present in only one of the operands, the co-indexing remains in the result.

CONJUNCTION: **x ++ y** returns the union of the sets of feature-structures x and y, as defined above, *but* when a feature is present in both operands (as in line 3 in the definition of union), they are co-indexed and they their values. This is the so-called «non-destructive unification»: the structure which is returned is a new structure.

EXAMPLES

Assume that:

```
*x <- {subject:{agr:{per:3, gen:fem}}
*y <- {predicate:{agr{num:plu}}
```

For union, there is no co-indexing, and futher modification of `subject.agr` will not modify `predicate.agr`.

```
                              {agr:{per:3,
*x.agr + *y.agr →                   gen:fem,
                                    num:plu}}
```

and neither *x nor *y are modified. The same holds for conjunction ++, but the value returned may have new co-indexing for some features inside the structure.

For unification, `subject.agr` and `predicate.agr` are co-indexed, and each futher operation on one will also affect the other.

```
                               {agr:{per:3,
*x.agr >< *y.agr →                   gen:fem,
                                    num:plu}}
```

but *x and *y have now new values:

```
*x <- {subject:{agr.#1:{per:3, gen:fem, num:plu}}
*y <- {predicate:{agr.#1:{per:3, gen:fem, num:plu}}
```

## *B. Intersection and strong intersection*

INTERSECTION: **x * y** returns the intersection of the sets of feature-structures x and y. The intersection is recursively defined as follow:

```
1    {} * {l2} → {}

2    {l1} * {} → {}

3    { e1{l1}, l2 } * { e1{l3} } → { e1{[{l1}*{l3}]} }

4    {f1, l1} * {f2} → {l1} * {f2}

5    {l1} * {f2, l2} → ( {l1}*{f2} ) + ( {l1}*{l2} )
```

When a feature which appears in both operands (as in line 3 of the definition), and this feature is co-indexed, there are two cases: if the two features are co-indexed together, nothing is changed; when they are not co-indexed together, the co-indexing is removed in the result.

STRONG INTERSECTION: **x ** y** returns the intersection of the structures x and y, as defined above, *but* when two features are co-indexed in one of the operands and appear also in the result, this co-indexing remains. That means that the values of co-indexed features in the result are not strictly the intersection of values of the operands, as in conjunction. See [Zajac 88] for a complete definition.

EXAMPLES

```
{subject:{agr.#1:{per:3,                {subject:  {agr:{per:3,
                  gen:fem,      *                        gen:fem},
                  num:plu},

 predicate:agr.#1}                       predicate:{agr:{num:sin}}}


            {subject:  {agr:{per:3,
→                                   gen:fem,
             predicate:{agr:num}}
```

For intersection, it is as if there were no co-indexing: only the values of each features are taken into account. But for strong intersection, co-indexing remains.

```
{subject:{agr.#1:{per:3,                {subject:  {agr:{per:3,
                  gen:fem,    **                         gen:fem},
                  num:plu},

 predicate:agr.#1}                       predicate:{agr:{num:sin}}}


            {subject:{agr.#1:{per:3,
→                                    gen:fem,
                                     num},
             predicate:agr.#1}
```

## *C. Difference*

DIFFERENCE: **x - y** returns x minus the set of feature-structures y. The difference is recursively defined as follows:

```
1     {} - {l2} → {}

2     {l1} - {} → {l1}

3     {e1{l1}, l2} - { e1{l3} } → {l2}

4     {f1, l1} - {f2} → {f1} + ( {l1} - {f2} )

5     {l1} - {f2, l2} → ( {l1} - {f2} ) - {l2}
```

Features in the result which are co-indexed in the first operand have the same co-indexing in the result.

EXAMPLE

```
{subject:{agr.#1:{per:3,              {subject:  {agr:{per:3,
                 gen:fem,        -                     gen:fem}}
                 num:plu},

 predicate:agr.#1}


      {predicate:{agr:{per:3,
→                      gen:fem,
                       num:plu}}
```

## 2.8 Inputs/outputs

Input and ouput functions are provided to the grammar writer. The grammar writer can use output functions to display at the terminal some partial result during evaluation, to print results in a file, etc. He can use input functions to read some previously stored information or to query the user during interactive execution (for interactive parsing, for example).

### *A. The read function*

There are two read functions: `reread` and `read`. `reread` opens the file, and reads in the file at the beginning whether `read` continues to read in the file after the previous reading of that file. If it is the first time the `read` function is called, it must be the `reread` function. The `read` function always reads a structure, a definition or an expression, and each is evaluated: the evaluation of a structure is the structure itself, a definition is loaded in memory, an expression is evaluated. The `read` function returns the result of the evaluation.

The `read` function is called without parameters to read an object at the user's terminal. If there is a parameter, it should be a string containing the name of a file (`reread` must always read in a file):

| | |
|---|---|
| `&read` | reads an object at the terminal |
| `&reread("def.tfs")` | reads an object at the beginning of file `def.tfs` |

A `read` operation performed after the end of the file is an error. A `write` operation performed on a file which is being read, and vice versa, is an error. A file must be closed using the `close` function before reading it again:

`&close("def.tfs")` closes the file `def.tfs`

### *B. The write function*

There are two writing functions: `rewrite` and `write`. `rewrite` writes an object at the begining of a file, whether `write` continues to write after the previous write in the same file. If the `rewrite` function is called for a file, it opens the file, the previous content is erased, and it writes at the beginning of the file. `rewrite` always erases the previous contents of the file, and writes from the beginning.

The `write` and `rewrite` functions take one or two parameters: the first is the object to be written in the file (a structure), and the second is the file name (as a string of characters).

`&write(<exp>)` writes the object returned by the evaluation of <exp> on the terminal;
`&rewrite(<exp>, "def.tfs")` writes the object returned by the evaluation of `<exp>` at the beginning of file `def.tfs`.

A read operation may be performed on a file which has been written and closed. Limitation: in the present version, the `write` function writes indexed structures; the `read` function cannot yet read indexed structures.

# 3. RUNNING THE INTERPRETER

## 3.1 Calling the interpreter

To call the interpreter is very simple: type the command `tfs`. The interpreter will prompt 'TFS>' when ready. To exit from the interpreter, type CONTROL-X: you are in the LISP environment. You can re-call the interpreter typing `(tfs)` or exit LISP with `(exit)`. To run the interpreter from the LISP environment, see section 3.6.

The interpreter has several modes set by the function `&param`. When this function is called without arguments, it simply displays the current parameters. When called with two arguments, the function sets the parameter (first element) to the value (second element): `&param [ (p, v) ]`. The parameters are the following.

UNIFICATION: T or NIL. This parameter allows switching from true unification to «pseudo-unification» [Tomita 88]. The effect of setting this parameter to NIL is that conjunction is simply replaced with union in every occurence of conjunction and unification.

CHECK: T or NIL. The NIL value disables type checking: unification and assignment always suceed.

SHARE: T or NIL. When this parameter is set to NIL, all co-indexing is removed, and the unification parameter is set to NIL. Recall that when sharing is removed, feature structures are completely equivalent to attribute structures.

## 3.2 Loading definitions

Before running the interpreter, you must define all the features you will use. They are more conveniently defined in a separate file which is loaded at the beginning of a session, rather than defining the features interactively. A file may contain any kind of definition or expression: types, templates and functions definitions, and also expressions which are evaluated (a convenient means of loading complex feature structures).

The function is `&load("file1", "file2",...)`

## 3.3 The read-eval-print top-level loop

Once you have loaded your definitions, you may evaluate expressions interactively, and also augment interactively your set of definitions (types, templates and functions). See appendix for an example of a session.

## 3.4 Errors

Some errors may occur: undefined features, an attempt to read a file which is not opened, etc. In such cases, the interpreter displays an error message and asks if you want to use the LISP debugger. This is not recommended. It is only to be used for debugging the interpreter itself.

## 3.5 Environment inquiries

At the interpreter level, you can have some information on the state of your environment. There are two functions, `&objlist` which print at the terminal the list of the current objects, and `&objects` which print the definitions of the objects. The objects could be `variables`, `functions`, `templates`, and `types`.

Without arguments, the functions print all objects on the terminal: `&objects` or `&objlist`.

With one argument, the functions print objects of the kind defined by the argument (can be a set of arguments), on the terminal:
`&objects(variables)`      `&objects({variables,functions})`.

With two arguments, the second is string which contains a file where the objects are printed:
`&objects(functions, "myfunc")`.

Limitation: in the present version, types, templates and function definitions are printed in the LISP format. The TFS printer is used for variables (feature structures).

## 3.6 LISP interface

The TFS interpreter is implemented on the top of LISP. Once the interpreter is loaded, one can go back in the LISP environment and call the TFS interpreter using the function `(tfs "<expression>")`, where the expression to be evaluated is passed as a LISP string (without argument, this function calls the interactive evaluation loop of the interpreter).

The corresponding function of calling the LISP interpreter `&lisp("<S-expression>")`, has not yet been implemented.

## 4. IMPLEMENTATION NOTES

The interpreter is written in COMMON-LISP, and does not use any implementation dependant feature. Thus, it is in principle completetly portable.

Several general remarks on the implementation are in order.

This implementation has essentially been devised to test the idea of integrating a functionnal framwork and a feature based unification framework. Thus, it has not been designed for performance, and for a new version, other implementation choices should be made for a number of parts.

Some parts are yet incomplete (such as the TFS printer for types, templates and function definitions), other parts need to be be revised (e.g. to allow function calls inside a feature-structure). This does not seriously alter the validity of the approach, but for a really usable version, they are, of course, critical.

### 4.1 The TFS reader and printer

The TFS reader is implemented as an (almost) LL(1) parser written in LISP. It uses two standard parsing modules that could easily be used for writing other readers. There are three modules: the parser module, the lexical reader, and the syntactic reader. The parser and the lexical reader modules are very generals, and the syntactic reader is the only module which takes into account the TFS syntax proper.

The parser module has several interface functions and variables :

(parse-file <string>) : calls the parser on the file <string>
(parse-string <string>): calls the parser on the string of characters <string>
(parse-open-file <string>): calls the parser on the file <string>. The file is not closed, and the parser can be called again on this file.
(parse-open-string <string>): calls the parser on the file <string>. The file is not closed, and the parser can be called again on this file.
(parse-next-stream) : continues to parse on the file opened with parse-open-file or parse-open-string. The calls may be recursive: parse-open-x may open another file, which becomes the input for parse-next-stream. The previous file is not lost: when this file is closed, the previous file again becomes the input for parse-next-stream.
(close-parser-streams) : closes the current streams used by the parser. The previous ones are reinstated.

The result of the parse is put in the variable *parsing-result*.

Two functions help to cope with syntactic errors:
(syntactic-error <unit>) : should be called when a syntactic error is detected. Build an error message using <unit> (a string or a non-terminal identifier), and the current state of the parser as described by the function (set-parsing-state <NT>). NT is a non terminal of the grammar. It could be a CONS (<NT> . <idf>) used, for example, as (NT_RULE . S) and is used to generate the beginning of the message "Error in rule S ...".

The parser uses a modified version of the LISP read-table, where the only macro-characters are \ (escape character) and " (string macro character). The definition of space characters are not changed. The following reading functions are provided.
(get-character) : reads a character in the current input stream.
(peek <skip> <char>) : tests if the following character is <char>. Space characters are skipped if <skip> is T.
(peek-alpha <skip>) : tests if the following character is a letter.
(peek-digit <skip>) tests if the following character is a digit.
(peek-number <skip>) tests if the following character is the beginning of a number (+, - or a digit).
(peek-alpha-num <skip>) tests if the following character is a digit or a letter.
(unread <char>) : unreads the character read by (get-character).

The lexical reader provides several pre-defined lexical units (syntax of identifiers, variables, numbres, strings, operators,...). The user can redefine all these except the key-word reader. This is because key-words are used to select the appropriate entry point in the grammar when the parser is called. To write a parser, the programmer must provide not only the syntactic part, but also a table of key-words to which are associated the functions defining the syntactic constructs identified by key-words. The default entry point must be specified in the variable *expression*. This default entry point is used when the unit to be parsed does not begin with a key-word.

The syntax of TFS expressions is defined in the appendix. This syntax (and the corresponding reader) is used in the interactive reader of TFS expressions, in the `&read` and `&reread` function and in the `&load` function.

## 4.2 Types, templates and functions

Definitions of types, templates and functions are put in association lists. As type information is not coded in the feature structures, there are two tables for types: one contains the type definition, and the other contains the set of features with their associated types.

## 4.3 Evaluation of expressions

The set of variables is represented as an association list. The co-indexing information is maintained in an array. An expression is represented as an S-expression, where the feature structures are represented as an imbedding of lists. This structure is produced by the TFS reader and evaluated by the TFS expression evaluator.

## 4.4 Unification, conjunction, union,...

Operations on feature structures are implemented formthe specifications given in [Zajac 88]. No optimisation has been made.

Operations on numbers use the LISP equivalent functions. Operations on string have a straightforward implementation.

## 5. CONCLUSION

LISP has been proven useful for building large systems. An equivalent language for building large Machine Interpretation systems could be very useful. Such a specialized language would allow a computational scientist to integrate various sub-components such as a speech recognition engine, a syntactic parsing engine, and a dialog management engine, within an unified syntax and a unique user interface. It would allow development of several sub-parts of a Machine Interpretation system using a common syntax an a common environment for all sub-parts, and to integrate these sub-parts to build a single large system. A functionnal language is a good candidate for such a task.

We have previously shown that feature structures can be related to attribute structures which are used in a functionnal framework [Zajac 88], describing a common theoretical framework for both approaches. An interpreter has been successfuly implemented to validate this approach, and proves the feasability of introducing a functionnal calculus for feature structures. Feature structures could be manipulated with such an language. Therefore a unification based engine can be imbedded in a clean way in a functionnal framework.

A preliminary attempt has been made to integrate this interpreter in the parser engine developed by Y.Nicolas [Nicolas 88]. For this occasion, the grammar formalism has been extended to include tree type (with a limited syntax and partial unification): the motivation is that a language with the capabilities for writing all parts of a Machine Interpretation system should be able to manipulate not only feature structures, but also tree structures, lattice structures, etc.

## Appendix 1: examples of definitions

```
:TYPE DECORATION = subject: SUBJECT, predicate:PREDICATE, category: CATEGORY;

:TYPE SUBJECT = agr:AGR, case: CASE, maj: CLASS;

:TYPE PREDICATE = agr:AGR, verb: VERB, object: OBJECT, category: CATEGORY;

:TYPE VERB = category: CATEGORY;

:TYPE OBJECT = case: CASE, maj: CLASS;

:TYPE CATEGORY = maj: CLASS, case: CASE, vform: VFORM, nform: NFORM, aux: YN;

:TYPE  CLASS = n, v, a, p, d, adv;

:TYPE CASE = nom, acc, gen;

:TYPE NFORM = norm, it, there;

:TYPE VFORM = fin, inf, ger;

:TYPE AGR = per: NUMBER, gen: GENDER, num : LNUMBER;

:TYPE GENDER = mas, fem, neu;

:TYPE LNUMBER = sin, plu, neu;

:TYPE YN = y, n;


:FUNCTION
fact(*x)=
 :COND
   *x=0 :: 1 ;
   *x>0 :: *x * &fact(*x-1)
 :ECOND;
```

## Appendix 2: a session with the interpreter

```
atr-ln> tfs

TFS>
                    +-------- Summary of line commands --------+
                    !                                          !
                    !  ^A     beginning of the line            !
                    !  ^E     end of the line                  !
                    !  ^B     backward                         !
                    !  ^F     forward                          !
                    !  ^D     delete 1 character               !
                    !  ^K     delete the end of the line       !
                    !  ^I TAB previous command                 !
                    !  ^N     next command                     !
                    !  ^L     previous ^K                      !
                    !  ^X     exit TFS, return to LISP         !
                    +------------------------------------------+

TFS> &load("hs.tfs")
> TYPE DECORATION.
> TYPE SUBJECT.
> TYPE PREDICATE.
> TYPE VERB.
> TYPE OBJECT.
> TYPE CATEGORY.
> TYPE CLASS.
> TYPE CASE.
> TYPE NFORM.
> TYPE VFORM.
> TYPE AGR.
> TYPE GENDER.
> TYPE LNUMBER.
> TYPE YN.
File hp.tfs loaded.
> NIL

TFS> *a <- {subject: {agr: {per: 2, gen: fem}}}
> {subject: {agr: {per: 2,
                   gen: fem}}}

TFS> *b <- {predicate: {agr: {num: plu}}}
> {predicate: {agr: {num: plu}}}

TFS> *c <- *a + *b
> {subject: {agr: {per: 2,
                   gen: fem}},
   predicate: {agr: {num: plu}}}

TFS> *c
> {subject: {agr: {per: 2,
                   gen: fem}},
   predicate: {agr: {num: plu}}}

TFS> *c.subject.agr >< *c.predicate.agr
> {per: 2,
   gen: fem,
   num: plu}
```

```
TFS> *c
> {subject: {agr.#0: {per: 2,
                      gen: fem,
                      num: plu}},
   predicate: {agr.#0}}

TFS> *a1 <- {subject: {agr: {per: 1}, case: nom}}
> {subject: {agr: {per: 1},
             case: nom}}

TFS> *b1 <- {subject: {agr: {num: sin, gen: mas}, maj: n}}
> {subject: {agr: {num: sin,
                   gen: mas},
             maj: n}}

TFS> *a1 >< *b1
> {subject: {agr: {per: 1,
                   num: sin,
                   gen: mas},
             case: nom,
             maj: n}}

TFS> *x <- {category: {vform: fin}, predicate: category, subject}
> {category: {vform: fin},
   predicate: category,
   subject}

TFS> *y <- {predicate: {object, category, verb: {category: {aux: y}}}}
> {predicate: {object,
               category,
               verb: {category: {aux: y}}}}

TFS> *x.category >< *x.predicate.category
> {vform: fin}

TFS *x
> {category.#4: {vform: fin},
   predicate: {category.#4},
   subject}

TFS> *y.predicate.category >< *y.predicate.verb.category
> {aux: y}

TFS> *y
> {predicate: {object,
               category.#5: {aux: y},
               verb: {category.#5}}}

TFS> *x >< *y
> {category.#6: {vform: fin,
                 aux: y},
   predicate: {category.#6,
               object,
               verb: {category.#6}},
   subject}

TFS> &load("fact.tfs")
> FUNCTION fact.
File fact.tfs loaded.
> NIL
```

```
TFS> &objects(functions)

((|&fact| . #S(FUNC-DECL :ARGS (|*x|)
                         :BODY (COND ((= (|&pathvalue| |*x|)
                                          #S(H-ATOM :NAME 0 :INDEX -1))
                                       (SEQ #S(H-ATOM :NAME 1 :INDEX -1)))
                                      ((> (|&pathvalue| |*x|)
                                          #S(H-ATOM :NAME 0 :INDEX -1))
                                       (SEQ
                                        (* (|&pathvalue| |*x|)
                                           (|&fact|
                                            (-
                                             (|&pathvalue| |*x|)
                                             #S(H-ATOM :NAME 1 :INDEX -1)))))))))
> NIL

TFS> &fact(5)
> 120

TFS> 2/3 + 5/3
> 7/3

TFS> "abc" + "DEF"
> "abcDEF"


> "abcd"


> "abcd"


> ""

TFS> "abcd" */ "abde"
> "ab"

TFS>
Call (tfs) to return to TFS...
NIL
Lisp> (exit)
```

## Appendix 3: syntax of TFS

The syntax is described using BNF notation extended with the following notations:

- square brackets [] are used to note optional parts
- braces { } are used to note iteration parts (Kleene star).

```
<types> ::=      :TYPES      <list-type-decl>
<type> ::=       :TYPE     <type-declaration>

<templates> ::=     :TEMPLATES <list-template-decl>
<template> ::=      :TEMPLATE <template-declaration>

<functions> ::=     :FUNCTIONS <list-func-decl>
<function>  ::=     :FUNCTION <function-declaration>

<list-type-decl>  ::=  { <type-declaration> }
<list-template-decl>  ::=  { <template-declaration> }
<list-function-decl>  ::=  { <function-declaration> }

<type-decl> ::=     <type-idf> = [ :CARD <integer> ]
                    <attr-idf> [ : <type-idf> ] { ,   <attr-idf> [ : <type-idf> ] } ;

<template-decl> ::=   <template-idf>  =  <constant-structure>  ;

<function-decl> ::=   <function-idf>   [( <structure-variable>  {  ,  <structure-variable>  }  )]
                      = <expression> ;

<expression>  ::= :COND <cond-exp> :ECOND
                |  :BEGIN <seq-exp> :END
                |  <boolean-exp>

<cond-exp> ::= <clause> { ; <clause> }

<clause> ::= <boolean-exp> :: <expression> { , <expression> }

<seq> ::=    <expression> { ; <expression> }

<boolean-exp> ::= <boolean-term>  [ '|' <boolean-exp> ]
```

```
<boolean-term> ::= [ ^ ] <boolean-fact> [ & <boolean-term> ]

<boolean-fact> ::= ( <boolean-exp> ) | <relational-exp>

<relational-exp> ::= <functional-exp> [ <rel-op> <functional-exp> ]

<functional-exp> ::= <functional-term> [ <op2> <functional-exp> ]

<functional-term> ::= <functional-fact> [ <op3> <functional-term> ]

<functional-fact> ::= ( <functional-exp> ) |
                      <number> | <string> | <fs> | <path> |
                      <template-idf> | <function-call>

<constant-structure> ::= <number> | <string> | <fs>

<fs> ::=    '{' [ <1fs> { , <1fs> } ] '}'
<1fs> ::=   <idf> [ <fs> ]

<func-call> ::= <func-idf> [ ( <functional-exp> { , <functional-exp> } ) ]

<path> ::= <v-idf>[.<idf>]*

<v-idf> ::=     *<idf>
<func-idf> ::=      &<idf>
<template-idf> ::= % <idf>
<idf> ::= <alpha-num-string>

<rel-op> ::=   = | /= | < | <= | > |  >= | <> | >< | ?>< | ><?
          |    @ | @< | @>< | <-
<op2> ::= + | ++ | – | /- | -/
<op3> ::= * | ** | / | /* | */
```

## REFERENCES

**Hassan Aït-Kaci**, 1984, *A Lattice Theoretic Approach to Computation Based on a Calculus of Partially Ordered Type Structures*, PhD. Dissertation, University of Pennsylvania.

**Hassan Aït-Kaci**, 1986, *An Algebraic Semantics Approach to the effective Resolution of Type Equations*, Theoretical Computer Science 45, pp 293-351.

**Hassan Aït-Kaci and Roger Nasr**, 1986, *LOGIN: a Logic Programming Language with Built-in Inheritance*, J. of Logic Programming, 3, pp 185-215.

**W. Bennett and J. Slocum**, 1985, *The LRC Machine Translation System*, Computational Linguistics 11/2-3, April-September.

**Ch. Boitet, P. Guillaume and M. Quezel-Ambrunaz**, 1980, *Manipulation d'arborescences et parallélisme: le système ROBRA* , COLING-80.

**Ch. Boitet, D. Bachut, N. Verastegui and R. Gerber**, 1988, *ARIANE portable, Dossier des Spécifications Externes, Le langage TETHYS*, GETA-ADI.

**R.J. Brachman and J.G. Schmolze**, 1985, *An Overview of the KL-ONE Knowledge Representation System*, Cognitive Science 9/2, pp 171-216.

**Alain Colmerauer**, 1971, *Les SYSTEMES-Q, un formalisme pour analyser et synthétiser des phrases sur ordinateur*, Groupe TAUM, Université de Montréal.

**Marc Dymetman**, 1987, *RATP: un nouveau formalisme de la classe des grammaires d'unification*, Unpublished paper, April 1987.

**G.E. Heidorn, K. Jensen, L.A. Miller, R.J. Byrd and M.S. Chodorow**,1982 , *The EPISTLE text-critiquing system*, IBM Syst. Journal, 21/3.

**K. Jensen and G.E. Heidorn**, 1983, *The fitted parse : 100% parsing capability in a syntactic grammar of English*, Proc. of the Conf. on Applied Natural Language Processing, pp 93-98, Santa-Monica, California, February.

**Ron Kaplan and J. Bresnan**, 1982, Lexical Functional Grammar, a Formal System for Grammatical Representation. In J. Bresnan (ed.), *The Mental Representation of Grammatical Relations,* The MIT Press, 1982, pp 173-381.

**Robert T. Kasper and William C. Rounds**, 1986, A Logical Semantics for Feature Structures, *Proc. of the 24th Annual Meeting of the ACL*, 10-13 June, Columbia University, New-York, pp 257-266.

**Martin Kay**, 1984, *Functional Unification Grammar: a Formalism for Machine Translation*, COLING-84.

**J. Nakamura, J. Tsujii and M. Nagao**, 1984, *Grammar Writing System (GRADE) of Mu-Machine Translation Project and its Characteristics*, COLING-84.

**Yves Nicolas**, 1988, *Pragmatic Extensions to Unification Based Formalisms*, ATR Interpreting Telephony Research Laboratories.

**Carl Pollard and Ivan A. Sag**, 1987, *Information-based Syntax and Semantics*, CSLI, Lectures Notes Number 13.

**J. A. Robinson and E. E. Sibert**, 1982, LOGLISP: an alternative to PROLOG, in *Machine Intelligence*, volume 10, J.E. Hayes, D. Michie and Y-H. Pao eds., Ellis Horwood Limited, pp 399-419.

**Stuart M. Shieber**, 1986, *An Introduction to Unification-based Approaches to Grammar*, CSLI, Lecture Notes Number 4.

**Jonathan Slocum**, 1984, *METAL: the LRC machine translation system*, ISSCO Tutorial on Machine Translation, Lugano, Switzerland, April 2-6.

**Gert Smolka**, *A Feature Logic with Subsorts*, LILOG-REPORT 33, IBM Deutschland GmbH, Stuttgart, May 1988.

**Masaru Tomita** (ed.), 1988, *The Generalized LR Parser/Compiler Version 8.1: User's Guide*, CMU-CMT-88-MEMO, 26 January 1988.

**Rémi Zajac** , 1988, *Operations on Typed Feature Structures: Motivations and Definitions,* ATR Interpreting Telephony Research Laboratories, Technical Report TR-I-0045, October 1988.

-o-o-o-o-o-o-o-o-o-o-o-o-o-