APERO Language Reference

Version 2.0

Charles Pecheur, Université de Liège (B), Systèmes et Automatique[1].
<pecheur@montefiore.ulg.ac.be>

This document document specifies APERO, a language based on pattern matching for describing text transformation rules. An implementation of this language is used as a basis for providing data type extensions for the LOTOS language.

1. INTRODUCTION

APERO is a language for describing text translation operations. It is based on pattern matching: each elementary translation rule is a pair ‹source pattern, target pattern›, so that text that matches the source pattern is translated according to the target pattern. We shall use the term translation in a generic sense, whereas other related words (transformation, rule, expansion) will be given a more precise meaning.

This formalism is derived from "Macro By Example" (MBE), a macro definition formalism used for the Scheme language [Kohlbecker87]. The main difference between MBE and APERO is that MBE applies directly to syntactic structures (s-expressions), whereas APERO has to operate on "raw" text.

Numerous variations have been considered for this formalism. This document (version 2.0) covers the language as implemented in APERO release 2.00. It is foreseen that several versions, including substantial enhancements and/or modifications, will follow.

"APERO" is initially the name of the tool for which the language was developed. It is a loose acronym for "ACT ONE pre-processor". The gastronomic underlying metaphor is not senseless either.

1.1. Conventions

Data structures and functions are described in a conventional notation inspired from the syntax of the ML language.

Syntax specifications use BNF conventions with the following extensions:

- non-terminals are italicized, terminals are double quoted.
- extended BNF meta-symbols are:

  pat1 **|** pat2 (alternatives)

  pat … (repetition)

  (pat) (grouping)

  [pat] (optional part).

---

[1]This work is supported by the CEC under EC-CA Project 001:76099 "EUCALYPTUS".

2. THE APERO MODEL

2.1. Lexical model

APERO transformations are applied to *raw text*, i.e. sequences of characters. However, APERO translation rules see this text as a sequence of *words*. A *phrase* is any such sequence, and may be seen as a concatenation of *sub-phrases*. This lexical phase brings some complexity to the overall model but discharges the main translation phase from coping with such aspects.

A *tokenized text* is a raw text containing words, so that each word is a contiguous sequence of characters of the raw text, but successive words need not be contiguous. Characters between two words are referred to as *white space*.

The rules for *scanning* raw text for tokens and *pretty printing* tokens into raw text are not part of the APERO language itself, instead, they are a parameter of the formalism that is instanciated in each application (for example, APERO tools use rules compatible with LOTOS syntax — see tool documentation for details). Their purpose is to skip non-significant characters in source text, such as spaces or comments, and re-introduce spacing in target text. Pretty printing a sequence of tokens and then scanning the resulting text need not return the same (or an equivalent) sequence of tokens, though this is a safe solution in most practical cases.

2.2. Translation levels

Two levels of translation activity are considered:

1. *single transformation*: given a collection of pairs ‹*source pattern*, *target pattern*› and a phrase, *match* a source pattern against this phrase and *expand* the corresponding target pattern into another phrase.
2. *full transformation*: given a collection of transformations ‹*trigger*, *rules*› and a tokenized text, *catch* those parts of the text that correspond to a trigger, apply the single translation corresponding to rules to the caught phrase and replace it by the obtained result.

2.3. Word equivalence

The word comparison criterion used in catching and matching is referred to as *word equivalence* and is another parameter of the model. The essential option for this parameter is whether lower and upper case letters are equal.

Note:

This equivalence can be defined as equality between canonical forms of the words, obtained through a provided canonical reduction mapping (e.g. conversion of lower case to upper case).

2.4. Variables and values

APERO makes use of *environments* that bind *values* to *variables*. Variable *identifiers* occur as binding instances in source patterns and as bound instances in target patterns (A local binding construct allows binding of local variables in target patterns as well).

Variable values are in general not atomic: when a variable identifier occurs in a source pattern that is matched repeatedly, the several words matched by the identifier are grouped to form a structured value. A value can be either:

- an *atomic value* consisting of one word,
- an *iterated value* consiting of a sequence of values,
- an *optional value* consisting of either a value or a distinguished null value.

The structure of values of a given variable is a constant that can be determined by syntactical inspection. We call this the *type* of the variable. Thus iterated variables will always have iterated values, etc.

More formally:

type *varval* = **AV**(*word*) **|** **IV**(*varval* …) **|** **OV**([*varval*])

type *env* = { *varid*=*varval*, … }

type *vartype* = **AT** **|** **IT**(*vartype*) **|** **OT**(*vartype*)

type *sig* = { *varid* : *vartype*, … }

and we have:

AV(*word*) : **AT**

$v_i : t$ => **IV**($v_1 \ldots v_n$) : **IT**(t)

OV() : **OT**(t)

$v : t$ => **OV**(v) : **OT**(t)

where *varval* : *vartype* is a typing relation as in common practice.

3. SYNTAX

The following grammar defines APERO syntax. Two non-terminals can be taken as distinguished (root) symbol: *fullspec* for full transformations, *singlespec* for single transformations.

fullspec ::=
    transfo ...

singlespec ::=
    rule ("|" *rule*) ...

transfo ::=
    selector ":" *rule* ("|" *rule*) ... ["|"] "."

selector ::=
    str ".." *str*

rule ::=
    spat ... "->" *tpat* ...

spat ::=
    str
| *idpat*
| *spat* [*sep*] "..."
| "[" *spat* ... "]"
| "«" *spat* ... "»"
| "(" *spat* ... ")"

tpat ::=
    apat

| *tpat* [*sep*] "..." [*range*]
| *opat* [":" *tpat*]
| "!" *bind* ("," *bind*) ... ":" *tpat*
| "(" *tpat* ... ")"

apat ::=
str
| *id*
| *npat*
| """ *tpat* ... """

opat ::=
"[" *tpat* ... "]" [*range*]
| "?" *bpat* ("," *bpat*) ... ":" *tpat*

sep ::=
".." *str*

npat ::=
number
| "#" *id*
| "@" *apat*
| *npat nopn npat*
| "(" *npat* ")"

bind ::=
idpat "=" *apat*
| *splitpat* "=" *lexpr*

lexpr ::=
id "..."
| "..." *id*
| "(" *npat* ":" *npat* ")"

range ::=
"{" *id* ("," *id*) ... "}"

bpat ::=
apat "==" *apat*
| *apat* "~=" *apat*
| *npat comp npat*
| "~" *bpat*
| *bpat* "&&" *bpat*
| *bpat* "||" *bpat*
| "(" *bpat* ")"

splitpat ::=
idpat "..."
| "(" *idpat* ... *idpat* "..." *idpat* ... ")"

idpat ::=
| "_"
| *id*

nopn ::=
"+" | "-" | "*" | "/" | "%" | "**" | "//"

comp ::=
"=" | "<>" | "<" | "<=" | ">" | ">="

3.1. Lexical Syntax

The lexical analysis recognizes the keywords and other terminals of the above syntax according to the following rules:

number ::=
["~"] digit …

id ::=
letter (letter | digit | "_") …

str ::=
"`" (*strchar* | *neststr*) … "´"

strchar ::=
{any printable char except """, "\", "`" and "´"}
| "\n" | "\t" | "\v" | "\b" | "\r" | "\f"
| "\" *digit digit digit*
| "\"" | "\\" | "\´" | "\`"
| "\"{newline}

neststr::=
str

digit ::=
"0" | … | "9"

letter ::=
"A" | … | "Z" | "a" | … | "z"

Moreover:

- any white character (space, tab, line break, …) or comment between lexical units is ignored. Comments are any text between "(*" and "*)". Properly balanced comments nested inside comments are allowed.
- *number* are converted to their decimal integer value.
- The following translations are performed on *strchar* inside *str*:
  - "\n" —> {newline}, "\t" —> {tab}, "\v" —> {vertical tab}, "\b" —> {backspace}, "\r" —> {carriage return}, "\f" —> {form feed}.
  - "\"*ddd* —> character with octal ASCII code *ddd*.
  - "\"" —> """, "\\" —> "\", "\´" —> "´", "\`" —> "`".
  - "\"{newline} is removed.

  However, these translations are not performed in *neststr*.

Notes:

- "~" is used for negative numbers.
- "\" followed by any other character than those mentioned is an error.
- The octal notation requires three digits (not one or two).
- a string may span several lines. Line breaks that are not escaped by "\", are included in the string.

- The *neststr* syntax allows an APERO program to process APERO text (or another language with similar string syntax). For example,

  `"`('hello \n')...\n'"`

  denotes the string

  `"('hello \n')..."`{newline}

  where only the second `"\n"` is translated because the first one occurs in a *neststr*.

4. USAGE

Eventually a formal semantics should be provided for the language. However it was felt that a good explanation of the role of each operator in natural language was more urgent than a rigorous mathematical definition.

4.1. Source patterns

A source pattern *spat* is matched against a source phrase *phrase* resulting in an environment that associates values built from *phrase* to variables in *spat*. Since the match may fail this operation is a partial function.

If several matches are possible for a given pattern and phrase, the one with the longest match for the leftmost sub-pattern is chosen.

Atomic patterns

- A constant pattern *str* matches one word if this word is equivalent to *str*. No binding is generated.
- A variable pattern *id* matches any single word and binds the variable to the atomic value consisting of that word.
- A wildcard pattern "_" matches any single word and but does not generate a binding.
- A fused pattern "`"`" *spat* "`"`" matches one word if *spat* matches the decomposition of this word into single characters and returns the resulting bindings.

Sequential patterns

A sequential pattern

$spat_1 \ldots spat_n$

matches any sequence of sub-phrases $phrase_1 \ldots phrase_n$ where each $spat_i$ matches the sub-phrase $phrase_i$. The resulting environment is the union of the bindings of each sub-match, with the requirement that no two distinct *spat* contain the same identifier.

Iterated patterns

There are two variants of the iterated pattern syntax, either with or without separator.

- *spat* `"..."` matches any contiguous sequence of sub-phrases that match *spat*.
- *spat* `".."` *str* `"..."` matches any sequence of sub-phrases that match *spat*, separated by *str*.

In either case, each variable *id* of *spat* is bound to an iterated value grouping its values for each match of *spat*.

Optional patterns

An optional pattern

"`[`" *spat* "`]`"

matches either a phrase matched by *spat* or an empty phrase.

Each variable *id* of *spat* is bound to an optional value. This value is the value of *val* from the match of *spat* in the first case, or a null value in the second case.

Parentheses

Parentheses in source patterns are only for syntactical grouping. Thus the pattern "`(`" *spat* "`)`" is equivalent to *spat*.

Notes:

- APERO source pattern matching corresponds to regular expression search. This can be done in several ways, given here by increasing complexity and efficiency:
  - By a simple, syntax-directed search function that tries the longest match for each sub-pattern. This is straightforward to implement but will not work correctly for any pattern. For example, the pattern

    ```
    `<' v ... `>'
    ```

    will never match (any closing ">" is captured by the iterated pattern).
  - By a backtracking version of the former, that will handle the previous example correctly. This solution will still fail for patterns for which infinite matches exists, such as nullable iterated patterns:

    ```
    () ... `b'
    ```
  - By reducing the pattern to a non-deterministic finite state automaton and simulating the latter with a backtracking mechanism. This is probably the best compromise but requires to associate complex operations to transitions in order to build the values for the variables.
  - By reducing the pattern to a deterministic finite state automaton. This gives the fastest solution, but this makes the extraction of variable values very difficult.
- Words are atomic w.r.t. source pattern matching: atomic patterns will not match parts of words. For example, the word sequence (`abc def`) is matched by "`` `abc' `def' ``" but NOT by "`` `ab' `cd' `ef' ``". However, fused patterns can be used for matching individual characters of a word on a more elaborate basis. For example, the pattern "`"x y _ ..."`" matches any word of at least two letters and binds `x` and `y` to the two initial letters.

4.2. Target patterns

A target pattern *tpat* is expanded in an environment *env*, issued from the successful match of the corresponding source pattern and/or enclosing local bindings. The result is a phrase. Expansion may lead to two kinds of error conditions:

- static error conditions, resulting from incompatibility between the usage of variables in *tpat* and their types in *env*. These errors can be determined by syntactic inspection of the patterns.

- dynamic error conditions, resulting from particular values associated to the variables. These can only be determined during a particular expansion, when those values are known.

Atomic patterns

- A constant pattern *str* expands into the single word *str*.
- A variable pattern *id* expand into the value of *id* in *env*, provided that this value is atomic. It is a static error if *id* is not atomic or is not defined in *env*.
- A fused pattern `"‹"` *tpat* `"›"` expands into a single word, formed by the concatenation of all words of the expansion of *tpat* in *env*.
- A numeric expression *npat* expands into a single word that is the decimal representation of its numeric evaluation in *env*. `"~"` is used for negative numbers.

Sequential patterns

A sequential pattern

$tpat_1 \ldots tpat_n$

expands into the sequence of the expansions of each $tpat_i$ in *env*.

Iterated patterns

There are four variants of the iterated target pattern syntax, either with or without separator and range. The most general form is:

tpat `".."` *str* `"..."` `"{"` id_1 `","` … `","` id_n `"}"`

This form expands into a sequence of expansions of *tpat*, separated by *str*, in environments where each id_i ranges over its iterated value in *env*. It is a static error if some id_i is not iterated or is not defined in *env*; and a dynamic error if all iterated values of id_i do not have the same number of values.

If the separator is omitted then the pattern expands into a contiguous sequence.

If the range is omitted then all free variables of *tpat* that are iterated in *env* are used. Moreover it is a static error if no such variable exists in *tpat*.

Optional patterns

The general form for an optional pattern is

`"["` *tpat* `"]"` `"{"` id_1 `","` … `","` id_n `"}"`

If the optional value of all id_i in *env* is null, the pattern expands to an empty phrase. If the optional value of all id_i in *env* is non-null, *tpat* is expanded in an environment where each id_i is bound to its non-null optional value in *env*. It is a static error if some id_i is not optional or is not defined in *env*; and a dynamic error if optional values of id_i are not either all null or not null.

If the range is omitted then all free variables of *tpat* that are optional in *env* are used. Moreover it is a static error if no such variable exists in *tpat*.

Conditional patterns

The syntax for a conditional pattern is

`"?"` $bpat_1$ `","` … `","` $bpat_n$ `":"` *tpat*

Each $bpat_i$ is evaluated as a boolean in *env* until either a false value is obtained or all $bpat_i$ have been expanded to true. In the first case, the pattern expands into an empty phrase, in the second case, *tpat* is expanded into *env*.

Patterns with default cases

Default cases are extensions of the two preceding forms. The pattern

opat "`:`" *tpat*

where *opat* is either an optional or a conditional pattern, expands into the expansion of *opat* if it leads to the non-empty case, or else to the expansion of *tpat*.

Local binding patterns

The syntax for a local binding pattern is

"`!`" $bind_1$ "`,`" … "`,`" $bind_n$ "`:`" *tpat*

The pattern expands into the expansion of *tpat* in the union of *env* and the bindings resulting from the evaluation of each $bind_i$ in *env*.

Parentheses

Parentheses in target patterns are only for syntactical grouping. Thus the pattern "`(`" *tpat* "`)`" is equivalent to *tpat*.

4.3. Numeric expressions

A numeric expression *npat* is evaluated in an environment *env* resulting in an integer numeric value. It is subject to the same kinds of static and dynamic errors as a target pattern.

- A constant expression *number* evaluates to *number*,
- A counting expression "`#`" *id* evaluates to the number of values in the iterated value of *id* in *env*. It is a static error if *id* does not exist or is not iterated in *env*.
- A word expression "`@`"*apat*, where *apat* is an atomic target pattern, evaluates to the numeric value of the single word expansion of *apat* in *env*, considered as the decimal representation of an integer value. It is a dynamic error if it is not. "`~`" is used for negative numbers.
- An operation expression *npat nop npat'* evaluates *npat* and *npat'* in *env* and combines the results via the numeric binary operation denoted by *nop*, which is any of the following:

Operation	Definition
x `+` y	sum.
x `-` y	difference.
x `*` y	product.
x `/` y	integer quotient, floored. Error if $y = 0$.
x `**` y	x to the y'th power. 0 if $y < 0$.
x `%` y	remainder of quotient of x by y (modulo). Error if y = 0.
x `//` y	logarithm of x in base y, floored. Error if $x < 1$ or $y \leq 1$.

- Parentheses in numerical expressions are only for syntactical grouping. Thus "`(`" *npat* "`)`" is equivalent to *npat*.

4.4. Boolean expressions

A boolean expression *bpat* is evaluated in an environment *env* resulting in an boolean value. It is subject to the same kinds of static and dynamic errors as a target pattern.

- An word comparison expression *apat wcomp apat'* expands the atomic target patterns *apat* and *apat'* in *env* and compares the resulting words via the word comparison operation denoted by *wcomp*, which is any of the following:

Operation	Definition
x `==` *y*	equivalence.
x `~=` *y*	non-equivalence.

- An numeric comparison expression *npat comp npat'* evaluates the numeric expressions *npat* and *npat'* in *env* and compares the resulting numbers via the comparison operation denoted by *comp*, which is any of the following:

Operation	Definition
x `=` *y*	equality.
x `<>` *y*	non-equality.
x `<` *y*	less.
x `<=` *y*	less or equal.
x `>` *y*	more.
x `>=` *y*	more or equal.

- A negation expression "`~`"*bpat* evaluates to the negation of the evaluation of *bpat* in *env*.
- An operation expression *bpat bop bpat'* evaluates *bpat* and *bpat'* in *env* and combines the results via the boolean binary operation denoted by *bop*, which is any of the following:

Operation	Definition
x `&&` *y*	and.
x `||` *y*	or.

- Parentheses in boolean expressions are only for syntactical grouping. Thus "`(`" *bpat* "`)`" is equivalent to *bpat*.

4.5. Bindings

A binding has the general form

left "`=`" *right*

The evaluation of this form in *env* returns a collection of bindings *id*=*val*, where *id* is a variable from *left* and *val* results from some evaluation of *right* in *env*. It is subject to the same kinds of static and dynamic errors as a target pattern. In particular, the syntactic structure of *left* and *right* must match and it is a static error if they don't.

In all kinds of bindings, the wildcard "_" may be used in place of an identifier *id*. In this case the binding that would have been generated is not generated.

Atomic bindings

The evaluation of an atomic binding

id "=" *apat*

is a binding *id*=*word*, where *word* is the single word resulting from the expansion of the atomic pattern *apat* in *env*.

List bindings

The evaluation of a list binding

"(" idx_1 ... idx_m *idy* "..." idz_n ... idz_1 ")" "=" *lpat*

is a collection of bindings of variables to parts of an iterated value *ival* that is the expansion of *lpat* in *env*. *lpat* may have three forms:

- The expansion of *id* "..." is the value of *id* in *env*. It is a static error if *id* is not iterated or not defined in *env*.
- The expansion of "..." *id* is the value of *id* in *env* in reverse order. It is a static error if *id* is not iterated or not defined in *env*.
- The expansion of "(" $npat_1$ ":" $npat_2$ ")" is the range of numbers between the numeric evaluation of $npat_1$ and $npat_2$ in *env*, inclusively. It is empty if $npat_1$ > $npat_2$.

idx_i is bound to the i^{th} first value in *ival*, idz_i to the i^{th} last value and *idy* to the remaining values (grouped as an iterated value). It is a dynamic error if *ival* contains less than (m+n) values.

if m=n=0, the parentheses may be omitted in *left*. Hence *idy* "..." is the same as "(" *idy* "..." ")"

Notes:

- The syntax may sometimes seem heavy but is coherent: iterated variables always appear in the scope of an ellipsis "...".
- In list bindings, the values in *ival* is not atomic.
- Bindings are the strongest candidates for language improvements. More general patterns should be allowed for *left* and *right*.

4.6. Transformations

Single transformations

A single transformation has the general form

$rule_1$ "|" ... "|" $rule_n$

where each $rule_i$ is a pair *spat* "->" *tpat*.

A single transformation is applied to a phrase and returns another phrase. The source pattern of each rule is matched in turn against the source phrase. When a match is found, the corresponding target pattern is expanded in the obtained environment and the obtained phrase is returned. An error condition is signalled if no source pattern matches.

Full transformations

A full transformation has the general form

transfo$_1$ ... *transfo*$_n$

where each *transfo*$_i$ has the form *str*$_1$ ".." *str*$_2$ ":" *rules* ".". *str*$_1$ and *str*$_2$ are refered to as the *start* and *end keywords* of the transformation, respectively, and *rules* define an associated single transformation.

A full transformation is applied to a raw text and returns another raw text. The source text is scanned and searched for any word equivalent to a start keyword of one of the *transfo*$_i$. When such a keyword is found, it is captured with the following words up to a word equivalent to the corresponding end keyword. Any text outside such keyword pairs is copied to the target text without change, including white space. If both keywords are identical, then the captured text is any single occurrence of (a word equivalent to) that keyword.

When a sequence of words bracketed by some keyword pair has been captured, the single transformation associated to the keywords is applied to the phrase consisting of those words. In the default case, the resulting phrase is pretty printed and inserted in place of the originally captured text. Optionally, the same full transformation may be recursively applied to the resulting text, until no more keywords can be found.

It is a static error if two transformations have equivalent starting keywords.

Notes:

- When a single transformation has been applied, the target phrase is first pretty printed (e.g. inserting white space between some words), then scanned again before the full transformation is applied recursively. Therefore the words are formed according to the tokenizing rules in effect, NOT from the expansion of the target pattern.
- The part of raw text that is replaced when a transformation is applied spans from the first character of the start keyword to the last character of the end keyword; i.e. white space just before and after the delimiting keywords is copied unchanged.

5. DIRECTIONS FOR ENHANCEMENTS

This section discusses some considered directions for improving the APERO formalism.

Generalized local binding:

The current version of local bindings only allow some simple variants. The syntax for left and right could be generalized to allow for more general and powerful binding expressions. There are at least three different ways to go:

- match similarly structured source and target patterns component-wise,
- expand a target pattern into a (flat) phrase, then match it against a source pattern,
- expand and pretty print a target pattern into raw text, then scan and match it against a source pattern.

We will call them *structural*, *flat* and *raw bindings*, respectively. The order given corresponds to increasing versatility, but also growing computation cost and complexity of use: as there is less inherent structure in the intermediate form, more hints have to be introduced in the contents to ensure the desired bindings afterwards. For example, let's compare flat and structural binding. Interleaving two iterated variables `x`, `y` of equal length into one iterated variable `z` could be obtained with the following flat binding:

```
z ... = (x y) ...
```

There is no equivalent solution with structural binding alone. On the other hand, suppose we want to duplicate a doubly iterated variable `x` into `y`. This is straightforward by structural binding between `x ... ...` and `y ... ...`. With flat binding however, we have to "mark" in some way where the nested sequences end; as for example in:

```
(y ...) .. `$´ ... = (x ...) .. `$´ ...
```

with the added difficulty that `$` may not occur in `x`.

Flat and structured bindings can be combined in a natural way:

- elementary bindings have the form *spat* = *tpat* and offer flat binding,
- the pattern composition operators (sequence, iteration, option) are introduced on bindings and induce structural binding.

For example, given an iterated variable `x` and an iterated optional variable `y`, the following binding creates a doubly iterated variable `z`, whose n-th sub-list contains the constant words `a`, `b`, the n-th element of `x` and the n-th element of `y`, if present.

```
(z ... = `a´ `b´ x [y]) ...
```

<u>New operations:</u>

The addition of some operations would make the language richer and easier to use:

- Iterated variables as boolean values: define a boolean expression "`##`"*id* that would be true iff the optional value of *id* is non-null. This form would be germane to the numeric expression "`#`"*id* over iterated variables.
- Iteration over prefix/suffix: define a right pattern for bindings that expands to the iterated list of all prefixes or suffixes of some iterated value. For example:

  ```
  (prefixes ...) ... = <<(1 2 3)
  ```

  would bind prefixes to the two-fold iterated value (() (`1`) (`1 2`) (`1 2 3`)).
- Set-oriented operations: it is often needed to consider a collection of values as a set instead of a list. One will need, for example, to know if some word is present in a list of options or to avoid declaring the same symbol twice.

<u>Multi-word atomic values:</u>

In the present version, atomic variable values are single words. It is not possible to capture and pass to the target text, a portion of source text that spans over several words, including the separating white space.

Such an addition would require several changes:

- The source pattern syntax should be modified so that a variable identifier may be associated to any source pattern (in which case the variable is bound to the part of text captured by the pattern).
- The semantics of matching would have to be defined over tokenized text instead of phrases.

More drastically, the whole preliminary tokenizing phase could be suppressed and means be provided for specifying patterns at the character level within APERO. This would surely be much more general, but:

- It would require a more involved description effort from the user, who would have to describe all the lexical aspects,
- The performance cost would probably be consequent, because character-level regular expression searches would be required over the entire text.

Source pattern extensions

- Bracketed patterns: this is a new source pattern form that would allow in some extent the matching of non-regular expressions. The form contains a pair of words ‹$word_1$,$word_2$› and matches any sequence of words bracketed with correctly balanced pair of $word_1$ and $word_2$. This is a classic feature of conventional macro processors (cpp, m4) w.r.t. parentheses.

  To go further, such an operator could be used as a replacement for the keyword catching mechanism of full transformations.
- Other triggering patterns: the double keyword triggering mechanism for full transformations is somewhat restrictive. It could also be useful to trigger transformations on distinct but identical start and end keywords or some particular points within a text: start/end of line, start/end of text, each line, the whole text. Note that the source pattern is trivially null on punctual or single-word triggers.
- Flag variables: in some cases, no value needs to be captured inside an iterated (resp. optional) source pattern. However knowledge of the number of matches is required: typically, to know about the presence or absence of a constant optional keyword without capturing the keyword.

  An elegant way to achieve this is to associate range variables to source patterns, with the same syntax as for target patterns. These variables are iterations (resp. options) over a new distinguished "flag" type containing no accessible value. When the pattern is matched, the variables are bound to iterated (resp. optional) values that can be used in target ranges, but it is a static error to access the corresponding sub-values.
- Bounded iterations: the idea is to refine the iterated source pattern by an optional specification of the minimum and/or maximum number of matches that are allowed. With such an extension, the optional pattern could become a particular case of the iterated pattern (0 to 1 match).

Global environment

We could allow bindings to be provided at the full transformation level so that they are available in all environments. Moreover, this global environment could also contain pervasive bindings such as commonly used values (e.g. {newline}, {tab}, {space}, etc.).

Top level considerations

Some more imperative style features, classically found in usual text processing programs, could be added in APERO:

- Imperative-style variables: this would allow state information to persist between successive single transformations. Any translation algorithm parameter could also be modified dynamically via a pre-defined pseudo-variable.
- In-line definitions: in the present setting the definition of the transformations and the input text are clearly distinct entities, each with its own syntax. In common macro-processors such as cpp or m4, the macro definitions occur in the same text stream as the input text and are parsed incrementally. This offers

some new possibilities, most importantly the ability to use APERO definitions within APERO definitions. On the other hand, it requires more syntax conventions to avoid confusions between APERO syntax and elements of the input text.

- Side-effect operations: for generating an error message, aborting the translation, reading or writing a separate file, …

6. REFERENCES

[Kohlbecker87] Kohlbecker E., Wand M.
"Macro By Example: Deriving Syntactic Transformations from their Specifications"
Proceedings of the 14th Annual ACM Symposium on Principles of Programming Languages, Munich, West Germany, January 1987.

7. FRENCH TERMINOLOGY

This annex gives the mapping between French and English words that refer to APERO terminology.

English	French
catch	intercepter
end keyword	mot-clé final
expand	expanser
full transformation	transformation complète
match	apparier
pattern	forme
phrase	phrase
raw text	texte brut
single transformation	transformation simple
source	source
start keyword	mot-clé initial
target	cible
tokenized text	texte découpé
translation	traduction
white space	espacement
word	mot