PATTERN MATCHING

version 1.1

by Dmitry A. Kazakov
(mailbox@dmitry-kazakov.de)

Pattern matching is a powerful tool for syntax analysis. The main idea of pattern matching comes from the **SNOBOL4** language (see the wonderful book THE SNOBOL4 PROGRAMMING LANGUAGE by R. E. Griswold, J. F. Poage and I. P. Polonsky). Some of the pattern expression atoms and statements were taken from there. One can find that patterns are very similar to the Backus-Naur forms. Comparing with the regular expressions (used by **grep** and **egrep** in UNIX) patterns are more powerful, but slower in matching.

The current version works under Windows and UNIX.

Download (match_1_1.tgz **tar** + **gzip**, Windows users may use WinZip)

---

1. Pattern expression

A pattern is a character string. All keywords can be written in both upper and lower cases. A pattern expression consists of *atoms* bound by *unary* and *binary* operators. Spaces and tabs can be used to separate keywords.

Atoms (terms of the expression):

- LETTER
- LOWER_CASE_LETTER
- NL
- Substring
- SUCCESS
- UPPER_CASE_LETTER
- User defined patterns

Unary operators:

- Big repeater
- Finite repeater
- Labels
- Little repeater
- NOEMPTY
- NOT

Binary (infix) operators:

- Catenation
- Alternation
- Assignment

1.1. Atoms

Atoms are terms of the pattern expression. Each atom is a very simple pattern that matches something.

1.1.1. Substring atom

This atom is a substring that matches itself. Such a substring must be put in quotes `''` or `""`. Very often it is necessary to match case independent substring. In this case the substring must be put in <> quotes. The case of letters in <> does not matter. Empty substring is allowed too. It matches empty chain of characters. This atom is assumed to be in any place where a pattern must be on a syntactical reason. Examples of substring atom:

Pattern	**Matches**
"ABC"	ABC
'ABC'	ABC
<ABC>	ABC, ABc, AbC, Abc, aBC, aBc, abC, abc

The character circumflex (^) in the substring has a special meaning. It converts the following character by inverting bit 6 of its ASCII code. For example, `^@` means NUL, `^M` means CR and so on. This allows you to include control characters and quotes in the substring body. The following table shows how circumflex works. Characters from pair columns give each other being specified with the circumflex.

How circumflex works

$^{100}/_{000}$	@	NUL	$^{120}/_{020}$	P	DLE	$^{140}/_{040}$	`	SP	$^{160}/_{060}$	p	0
$^{101}/_{001}$	A	SOH	$^{121}/_{021}$	Q	DC1	$^{141}/_{041}$	a	!	$^{161}/_{061}$	q	1
$^{102}/_{002}$	B	STX	$^{122}/_{022}$	R	DC2	$^{142}/_{042}$	b	"	$^{162}/_{062}$	r	2
$^{103}/_{003}$	C	ETX	$^{123}/_{023}$	S	DC3	$^{143}/_{043}$	c	#	$^{163}/_{063}$	s	3

104/004	D	EOT	124/024	T	DC4	144/044	d	$	164/064	t	4
105/005	E	ENQ	125/025	U	NAK	145/045	e	%	165/065	u	5
106/006	F	ACK	126/026	V	SYN	146/046	f	&	166/066	v	6
107/007	G	BEL	127/027	W	ETB	147/047	g	'	167/067	w	7
110/010	H	BS	130/030	X	CAN	150/050	h	(	170/070	x	8
111/011	I	HT	131/031	Y	EM	151/051	i	)	171/071	y	9
112/012	J	NL	132/032	Z	SUB	152/052	j	*	172/072	z	:
113/013	K	VT	133/033	[	ESC	153/053	k	+	173/073	{	;
114/014	L	NP	134/034	\	FS	154/054	l	,	174/074	|	<
115/015	M	CR	135/035	]	GS	155/055	m	-	175/075	}	=
116/016	N	SO	136/036	^	RS	156/056	n	.	176/076	~	>
117/017	O	SI	137/037	_	US	157/057	o	/	177/077	DEL	?

Circumflex at the end of a literal denotes itself. For instance, `'^'` specifies character literal containing the single character - circumflex.

1.1.2. Atom AnyOf

The atom matches any character from the given character set. Notation for this atom is a character string put in `{}` brackets. Order of characters in the string does not affect on the result of matching. Circumflex within the string body works the same way as for substring atom. For example, the pattern *`{.,;?!}`* matches one of punctuation marks.

1.1.3. Atom END

The atom END (*`END`* or `.`) matches the line end. It fails when the rest of the current line is not empty. On success it matches empty chain of bytes.

1.1.4. Atom BREAK

The atom BREAK (*`BREAK`* or _) matches non empty chain of spaces and tabs. It can match empty chain at a keyword boundary. Keyword is assumed to be a chain of letters and digits. Thus BREAK will fail if previous and current characters are letters or digits. This pattern is very useful to separate multiword keywords. For example: the pattern *`'key'`*_ will match *key* in `key(12)` and `key is`, but never in `keyword`.

1.1.5. Other atoms

Some other atoms are described in the following table:

Atom (notation)	Abbr.	Matches
`ANY`	*`%`*	Any character
`BLANK`	*`+`*	Any non empty chain of spaces and tabs
`DIGIT`	*`#`*	One of characters `0`, `1`, ..., `9` (see also UserBase)
`UPPER_CASE_LETTER`	*`U`*	One of `A`, `B`, ... , `Z`
`LOWER_CASE_LETTER`	*`W`*	One of `a`, `b`, ... , `z`
`LETTER`	*`L`*	One of `A`, `B`, ... , `Z`, `a`, `b`, ... , `z`
`CHARACTER`	*`C`*	One digit or letter

NL	/	Empty chain and skips the line (see)
FAILURE	F	Nothing and ends matching with failure
FENCE	:	Empty chain and prevents search for an alternative (see)
SUCCESS	S	Empty chain and ends matching with success

1.2. Binary (infix) operations

A pattern expression is constructed with the aid of atoms and operators. There are three binary operators *catenation*, *alternation* and *assignment*.

1.2.1. Catenation

There is no special notation for catenation of two patterns. You should just put them in a chain. Concatenated patterns are matched sequentially. The whole matching is successful if and only if each of them is matched. For example:

Pattern	Matches
`ANY LETTER`	Any character followed by a letter

1.2.2. Alternation

Alternatives are separated by the alternation operator, denoted as `OR`, `!` or `|`. For example:

Pattern	Matches
`'A'|'B'|'C'`	Either `A` or `B` or `C`

Note, that the order of alternatives is very important. For instance, the pattern `'A'|'AA'|'AAA'` will never match `AAA` because `A` is a prefix of `AAA`. The rule is that the pattern which is a prefix of another one must follow it in the alternation list. In our example `'AAA'|'AA'|'A'` would be correct. Note also, that in all cases, when the AnyOf atom can be used instead of the alternation, it is preferable, because it is matched faster and the time of matching depends on neither the number of alternated characters nor their order.

1.2.3. Assignment

Assignment (=) allows one to set the value of a variable to the part of a line (lines) matched by the pattern specified as the right argument of the assignment. This feature is based on interface between the caller and the match processor (see GetVaraibleId, AssignVariable and DeAssignVariable).

1.3. Grouping

Round and square brackets can be used for grouping patterns. Brackets can be nested. They allow us to make more complex expressions.

Pattern	Matches
`('A'|'B')('C'|'D')`	Either `AC` or `AD` or `BC` or `BD`
`('A'|'')`	Either `A` or empty string
`('A'|)`	Same as above
`['A']`	Same as above

1.4. Unary operators

There are several unary operators. All of them have prefix form, i.e. they must be specified before a pattern. If the pattern is a list of concatenated or alternated patterns it must be put in round braces because priority of any unary operator is greater than one of catenation or alternation, but less than one of assignment.

1.4.1. Repeaters

Sometimes it is necessary to repeat matching of a pattern. There are three kinds of repeaters.

The simplest repeater is *finite repeater*. To repeat a pattern desired number of times, one should place the number of repetitions before the pattern. For example, `5 DIGIT` matches five digits. Note that `5('A'|'B')` `indeed` matches five `A` or `B` letters, because finite repeater tries to match all available alternatives.

A more complex case is when it is unknown, how many times a pattern must be repeated. There are two indefinite repeaters `*` (*little repeater*) and `$` (*big repeater*).

The *little repeater* tries to match the repeated pattern as little times as possible. So first time it does not try to match the pattern at all. Then, if a failure occurs while matching of the following pattern expression, the repeater tries to match the pattern. On success the matching process will be continued from the pattern following the repeated pattern. So each time when matching fails, the *little repeater* makes an attempt to match the repeated pattern one more time. If it is impossible to match the repeated pattern, the *little repeater* tries to diminish the number of repetitions, so that if the pattern has alternatives the repeater will try to match all of them. And only if the repetition count is zero and it is impossible to match the repeated pattern the *little repeater* fails. Therefore the expression `*`***P***, where ***P*** is a pattern may be outlined as `(|`***P***`)(|`***P***`)(|`***P***`)...` (One may say that matching of an indefinite repeater is led by its right context).

Pattern	Matches
`*LETTER '.'`	A chain of letters followed by dot

In contrary to the *little repeater* the *big repeater* tries to match the repeated pattern as many times as possible. It starts with matching the repeated pattern so many times as possible. Then matching process continues to the rest of pattern expression. If it fails, the *big repeater* tries to match other alternative of the repeated pattern and then the pattern itself (as many times as possible). Only if it is impossible to find an alternative the repeater diminishes the number of repetitions. Therefore the expression `$`***P***, where ***P*** is a pattern is an equivalent to `(`***P***`|)(`***P***`|)(`***P***`|)....` For example, an arbitrary length string of letters can be specified as `$LETTER`.

1.4.2. Using atom FENCE with repeaters

All repeaters try to match all possible alternatives before failure. In many cases it is not necessary. Sometimes it is possible to optimize matching process with the aid of the atom FENCE (`FENCE` or `:`). In a given list of concatenated patterns FENCE disables any return to unmatched alternatives.

Suppose we want to build a pattern which will match a string containing a substring beginning with `A` and ending by `B`. We can write this pattern using little repeater and the atom ANY by the following way `*%'A'*%'B'`. With the string `AAACA` this pattern will work as follows:

Step	Unmatched	Actions
1	AAACA	1st `*%` matches empty substring

2	AAACA	`'A'` matches A1
3	AACA	2nd `*%` matches empty substring
4	AACA	`'B'` fails to match A2
5	AACA	2nd `*%` matches one character (A2)
6	ACA	`'B'` fails to match A3
7	ACA	2nd `*%` matches one more character (A3)
8	CA	`'B'` fails to match C4
9	CA	2nd `*%` matches one more character (C4)
10	A	`'B'` fails to match A5
11	A	2nd `*%` matches one more character (A5)
12		`'B'` fails to match nothing (end of line)
13		2nd `*%` fails and returns AACA back. `'A'` returns A
14	AAACA	1st `*%` matches one character (A1)
15	AACA	`'A'` matches A2
16	ACA	2nd `*%` matches empty substring
17	ACA	`'B'` fails to match A3

It is obvious that once `*%'B'` failed (step 13), there is no necessity to match other samples and the whole matching must fail. So we can optimize the pattern by placing FENCE after the first indefinite repeater `*%:'A'*%'B'`. This pattern will fail after step 13.

Let us consider another example. One way of matching a chain of hexadecimal letters is `$(DIGIT|'A'|'B'|'C'|'D'|'E'):`. Here FENCE allows to avoid useless attempts to match A instead of a digit, B instead of A and so on. In this case the alternatives of the repeated pattern do not overlap. Hence, once an alternative fails there is no need to try others.

1.4.3. Ellipsis

The pattern `*%:` is used so often that it was given a special notation - ellipsis (`...` or `..`). Ellipsis are used to match anything till something. For example, pattern `'A'...'B'` will match any chain which starts with A and ends by B.

1.4.4. NOT

The unary operator NOT (`NOT` or `^`) inverts the result of matching the pattern it precedes. Where the pattern ***P*** is successfully matched, `^`***P*** pattern fails. Where ***P*** fails, `^`***P*** successfully matches empty substring. So `^`***P*** means *empty if not* ***P***. Very often doubled NOT is used to test if a pattern matches something doing that without side effects. For example, `$LETTER:^^'*'` matches a chain of letters if and only if it is followed by `*`. But `*` itself remains unmatched.

1.4.5. NOEMPTY

The statement NOEMPTY (`NOEMPTY` or `?`) does not allow an empty substring to be matched. For example, `?($LETTER:)` matches only non-empty words.

1.4.6. Label

Each pattern may have one or more labels. The syntax is ***label*>*pattern***. A reference to the label works as if all the patterns following the label till an unbalanced bracket (`)` or `]`) or the end of the pattern expression, would be placed instead of the reference. Note that `]` is just an abbreviation for `|)` . Let the label Bool was somewhere defined as `Bool>'0'|'1']`. Then a reference to Bool will work exactly as `['0'|'1']`.

Using labels one can build recursive patterns. For instance, we can build a pattern to match a balanced chain of {} braces: `balanced>('{'[balanced]'}')`.

1.5. Special patterns

1.5.1. NL

The atom NL (`NL` or `/`) is used to leave the current line for the next one. For example, to match an item that occupies several lines. Note the difference between END atom and NL. The atom END just matches the end of a line but does not leave it. While the atom NL leaves the current line even if it is not completely matched. This feature is based on the interface between a user written caller and the match processor (see GetNextLine and GetPreviousLine for more information).

1.5.2. User defined patterns

The match processor interface allows one to use user defined patterns as well as embedded ones. The user defined patterns are pre translated patterns which addresses are returned by the GetExternalPattern function, which binds unknown identifiers with user defined patterns.

1.6. Example

Let us build a pattern that matches an expression of C language. A C expression is something limited by semicolon, an unbalanced bracket or comma. We should take into account C comments and literals. We will build the pattern by pieces. Let us start from the pattern for string literals:

```
'"' *('\\' | '\"' | '\'END/ | %): '"'
```

Note that simply `'"'...'"'` would be wrong, because in C a string literal may occupy several lines. We should also recognize digraphs \\ and \" as well.

The next pattern is for character literals (we recognize \' literal as a separate case):

```
"'\''" | "'"..."'"
```

The pattern for C and C++ comments looks like:

```
'/*' *(END/|%): '*/' | '//'...END/
```

The following pattern is not really necessary. It quickly skips spaces, tabs, numbers and identifiers:

```
BLANK | CHARACTER $CHARACTER:
```

Now let us put all these patterns together and add ANY and NL patterns as alternatives to skip special characters and line ends:

```
  BLANK                                  - spaces and tabs
| CHARACTER $CHARACTER                   - an identifier or number
| "'\''" | "'"..."'"                     - a character literal
| '"' *('\\' | '\"' | '\'END/ | %): '"'  - a string literal
| '/*'*(END/|%): '*/' | '//'...END/      - a comment
| %                                      - something else
```

```
| /                                                    - go to the next line
```

Let us repeat this pattern using little repeater, mark this as *Item* and add brackets recognition:

```
( Item>                                                - label for self call
*                                                      - repeater
  ( BLANK                                              - spaces and tabs
  | CHARACTER $CHARACTER                                - an identifier or number
  | '('item')'                                         - recursive call to itself
  | '['item']'                                         - same for square brackets
  | '{'item'}'                                         - same for braces
  | "'\''" | "'"..."'"                                 - a character literal
  | '"' *('\\' | '\"' | '\'END/ | %): '"'              - a string literal
  | '/*' *(END/|%): '*/' | '//'...END/                 - a comment
  | %                                                  - something else
  | /                                                  - go to the next line
  ) :                                                  - FENCE for optimization
  ^^{,;)]^=}                                           - terminator
)
```

The condition of repetition termination is: stop if one of `,;)]}` characters appears. Here `}` is specified using circumflex. Doubled NOT is added to disable matching of the terminating character, because it does not belong to the expression.

---

2. Programming with patterns

The file *`match.c`* contains all the mentioned below functions. It is possible to use them from any C/C++ program as well as from any other language that supports C interface. From the C run-time library only memory allocation functions (**malloc** and etc.) are used in *`match.c`*.

2.1. Basic functions

The following functions can be used in C/C++ or Ada 9X. They are declared in the *`match.h`* header file. In Ada 9X one should use bindings defined in the package Matching (file *`matching.ads`*).

2.1.1. match

C/C++

```
int match
(
   int          Length,
   const char * String,
   int        * Pointer,
   const char * Pattern
);
```

Ada 9X

```
function match
(
   Length  : int;
   String  : char_array;
   Pointer : access int;
   Pattern : char_array
)  return StatusCode;
```

The function matches the string pointed by the parameter `String` with the pattern in the internal format pointed by the parameter `Pattern` . The string may contain any characters including NUL ASCII. The

string length is defined by the parameter `Length` (0..). The string is matched from the character position specified by the parameter `Pointer`. The parameter is passed by address. It is modified to point to the first unmatched character of the string. The first character has the position 0.

Return (status code defined in `match.h`)

MATCH_SUCCESS	Success
MATCH_FAILURE	Failure of matching
MATCH_WRONG_PATTERN_FORMAT	Wrong pattern format
MATCH_TOO_LONG_STRING	String length exceeds 1 Gbyte
MATCH_CANNOT_RETURN	Cannot return to a previously left string
MATCH_NO_DYNAMIC_MEMORY	Dynamic memory overflow
MATCH_INTERNAL_ERROR	Unrecognized internal error

2.1.2. patran

C/C++

```
int patran
(
   int          Length,
   const char * String,
   int        * Pointer,
   char       * Pattern,
   int          Size
);
```

Ada 9X

```
function patran
(
   Length  : int;
   String  : char_array;
   Pointer : access int;
   Pattern : char_array;
   Size    : access int
)  return StatusCode;
```

The function translates the string pointed by the parameter `String` into the pattern internal format. The string can contain any characters including NUL ASCII. The parameter `Length` gives the string length (0..). The parameter `Pointer` specifies the string position where the pattern begins (0..). The parameter is passed by the address. After completion its value will point to the first character of the string following the pattern or the error position if the translation fails. The parameter `Pattern` points to the memory block where the translated pattern should be stored. The parameter `Size` specifies the block size in bytes. It is passed by address, and after successful completion will contain the number of actually used bytes. See also patinit and patmaker.

Return (status code defined in `match.h`)

MATCH_SUCCESS	Success
MATCH_BRACE_ERROR	Right brace does not match the left one
MATCH_DUPLICATE_LABEL	Re-definition of a label
MATCH_FAILURE	Failure. No pattern found
MATCH_MISSING_QUOTATION	Missing closing quotation marks
MATCH_MISSING_RIGHT_BRACE	One or more missing right braces
MATCH_NO_DYNAMIC_MEMORY	Dynamic memory overflow
MATCH_POSSIBLE_INDEFINITE_LOOP	Repetition of a pattern that matches null
MATCH_TOO_BIG_REPEATER	Repetition count is too big
MATCH_TOO_LARGE_PATTERN	Pattern is too large for provided buffer
MATCH_TOO_LARGE_STRING	Pattern length exceeds 1 Gbyte
MATCH_UNDEFINED_VARIABLE	Immediate assignment to unknown variable
MATCH_UNRECOGNIZED_CHARACTER	Unrecognized character
MATCH_UNRECOGNIZED_KEYWORD	Unrecognized keyword or undefined label
MATCH_RESERVED_KEYWORD	Empty or conflicting name of a variable

2.1.3. patinit

C/C++

```
const char * patinit
(
   int          Length,
   const char * String,
   int        * Pointer
);
```

Ada 9X

```
function patinit
(
   Length  : int;
   String  : char_array;
   Pointer : int
)  return chars_ptr;
```

The function translates the string pointed by the parameter `String` into the pattern internal format. The string can contain any characters including NUL ASCII. The parameter `Length` gives the string length (0..). The parameter `Pointer` specifies the string position where the pattern begins (0..). The parameter is passed by the address. After completion its value will point to the first character of the string following the pattern or the error position if translation fails. To store the translated pattern the function calls the user defined memory allocator AllocatePattern. The pointer to the translated pattern is returned as the result. On any error no pattern is allocated and zero is returned. One can consult MatchError global variable for the error code (same as one returned by patran).

2.1.4. patmaker

C/C++

```
const char * patmaker
(
   const char * String
);
```

Ada 9X

```
function patmaker
(
   String : char_array
)  return chars_ptr;
```

The function works like patinit, but translates a null-terminated string and uses **malloc** to allocate the translated pattern. Note that the whole string must be translated, otherwise MATCH_UNRECOGNIZED_KEYWORD error code is reported in MatchError.

2.1.5. freedm

C/C++

```
void freedm (void);
```

Ada 9X

```
procedure freedm;
```

The function releases all the memory allocated by the match function, which does not release the memory but keeps it allocated between subsequent calls.

2.2. C++ functions and data types

In C++ one may use the type *pattern* that wraps the described above functions (match, patran, patinit, patmaker and freedm). The *pattern* type is defined in *pattern.h* file. The *match.h* is automatically included by the *pattern.h*.

2.2.1. Constructor

```
pattern (const char * String);
```

The constructor translates its string parameter into the internal pattern format. When an error occurs the created pattern will match nothing. The error code can be obtained from the external variable MatchError.

2.2.2. Pattern internal representation

```
const char * Pattern () const;
```

This method returns the pointer to the pattern internal representation. The returned value can be used by the function match.

2.2.3. Matching methods

```
int operator |=
(
   const char  *  String,
   const pattern& Pattern
);
```

The **operator** |= matches the string specified by the left parameter with the pattern provided by the right parameter. It returns 1 if the pattern matches the whole string or 0, otherwise. The string must be null-terminated.

```
int operator >>
(
   const char  *  String,
   const pattern& Pattern
);
```

The **operator** >> matches a part of the string specified by the left parameter. Unlike the **operator** |=, the pattern given by the right parameter, can match a part but the whole string. The actual number of matched bytes is returned as the result. It allows to parse a string using consequential calls to the **operator** >>. For example:

```
const pattern Identifier = "LETTER $(CHARACTER|'_'):";
const pattern Blank = "_";
char          Line [80];              // Source line
int           Index;                  // Source line position
             . . .
Index  = 0;                           // From beginning
Index += &Line [Index] >> Blank;      // Skip an identifier
Index += &Line [Index] >> Identifier; // Skip blank field
```

Since there is a constructor from string, it is possible to use the text constants instead of patterns, like in:

```
Line [Index] >> "LETTER $(CHARACTER|'_'):"
```

Though it most cases, it would be a bad idea, because it requires a string to pattern translation each time the expression is calculated.

Both **operator** |= and **operator** >> set the variable MatchError (see match completion code).

2.3. Advanced pattern programming issues

There are several of global variables defined in *match.h*. They control the pattern translation and matching processes. All of them require C external linking (C name convention). If the variable is a pointer to a function it is always possible to set it to 0, when the function is not required, and this is the initial setting.

2.3.1. UserAlphas

```
extern char * UserAlphas;   UserAlphas : chars_ptr;
```

This is a NUL terminated string that contains the list of symbols interpreted as valid characters for identifiers of labels and variables. It has no effect on how atoms work. Normally UserAlphas contains 0. It means that only letters and digits are valid. If, for example, UserAlphas would point to `_#` then both underline and dash characters were valid too. Notice the difference between letters and digits: an identifier name always starts with a letter. The UserAlphas affects pattern translation process only.

2.3.2. UserBase

C/C++ Ada 9X

```
extern int UserBase;   UserBase : int;
```

This is an integer number that contains the current base for the atom DIGIT . Normally it is set to 10, so DIGIT means one of the following characters `0123456789`. Setting it to, say, 2 causes DIGIT be `0` or `1`. The value of UserBase could be greater than 10. For example, setting it to 16 causes DIGIT be one of `123456789abcdefABCDEF`. Note, that atoms CHARACTER and BREAK do not depend on the value of UserBase.

2.3.3. MatchError

C/C++ Ada 9X

```
extern int MatchError;   MatchError : constant StatusCode;
```

This variable contains the last error code set by match, patran, patinit or patmaker. In Ada 9X bindings StatusCode is defined as a separate type derived from *`int`*.

2.3.4. GetNextLine

C/C++ Ada 9X

```
extern int (* GetNextLine)
(
   const char ** Line,
   int         * Length
);
```

```
type GetNextLineCallBack is access
   function
   (
      Line   : access chars_ptr;
      Length : access int
   )  return int;
pragma Convention (C, GetNextLineCallBack);
GetNextLine : GetNextLineCallBack;
```

The function pointed by the variable GetNextLine is called by the pattern matching processor to pick up the next line to match (see NL). The parameter `Line` receives the address of the line. The parameter `Length` is the pointer to the length of the line (must be >= 0). The function should return 0 if there is no line (end of file equivalent). Other return values mean that `Line` and `Length` were correctly set.

2.3.5. GetPreviousLine

C/C++ Ada 9X

```
extern int
   (* GetPreviousLine)
(
   const char ** Line,
   int         * Length
);
```

```
type GetPreviousLineCallBack is access
   function
   (
      Line   : access chars_ptr;
      Length : access int
   )  return int;
pragma Convention (C, GetPreviousLineCallBack);
```

```
                                    GetPreviousLine : GetPreviousLineCallBack;
```

The function pointed by the variable GetPreviousLine is called to return back to the previous line. The parameter `Line` receives the address of the line. The parameter `Length` is the pointer to the length of the line. The function should return 0 if the previous line cannot be returned. In this case the matching process will be terminated with MATCH_CANNOT_RETURN error. Other return values mean that `Line` and `Length` were correctly set.

2.3.6. GetExternalPattern

C/C++

```
extern char *
   (* GetExternalPattern)
(
   char * Name,
   int    Length
);
```

Ada 9X

```
type GetExternalPatternCallBack is access
   function
   (
      Name   : char_array;
      Length : int
   )  return chars_ptr;
pragma Convention (C, GetExternalPatternCallBack);
GetExternalPattern : GetExternalPatternCallBack;
```

The function GetExternalPattern is called when the pattern translator discovers an undefined identifier. This identifier is passed to GetExternalPattern as a string. If it returns 0 the translator assumes that there is no such pattern and translation will fail. The parameter `Name` points to the identifier name. The parameter `Length` is the length of the name. The function should return the address of the pattern associated with the name or 0.

2.3.7. GetVariableId

C/C++

```
extern unsigned long
   (* GetVariableId)
(
   char * Name,
   int    Length
);
```

Ada 9X

```
type GetVariableIdCallBack is access
   function
   (
      Name   : char_array;
      Length : C.int
   )  return VariableID;
pragma Convention (C, GetVariableIdCallBack);
GetVariableId : GetVariableIdCallBack;
```

The function pointed by GetVariableId is called when the pattern translator discovers an assignment. The left argument of the assignment is passed to the function. It must return the identifier associated with the variable. The identifier is any non-zero number that will be passed to the AssignVariable and DeAssignVariable while pattern matching. Each variable must have an unique identifier. If 0 is returned, the pattern translation fails with MATCH_UNDEFINED_VARIABLE error code. The parameter `Name` of the function points to the variable name. The parameter `Length` is the name length. In Ada 9X bindings the VariableID is defined as a separate type derived from *`unsigned_long`*.

2.3.8. AssignVariable

C/C++

```
extern void
   (* AssignVariable)
(
   unsigned long VariableId,
   unsigned long Offset,
   unsigned long Length
);
```

Ada 9X

```
type AssignVariableCallBack is access
   procedure
   (
      Variable : VariableID;
      Offset   : unsigned_long;
      Length   : unsigned_long
   );
```

```
pragma Convention (C, AssignVariableCallBack);
AssignVariable : AssignVariableCallBack;
```

The function pointed by AssignVariable is called each time when a pattern assigned to a variable has been successfully matched. The value of the parameter `VariableId` is one returned by GetVariableId for the name the pattern is assigned to. The parameters `Offset` and `Length` define which part of the string was matched. Note that the string may occupy several lines (see GetNextLine) so `Offset` indicating the beginning of the matched substring, is counted from the beginning of the first line including all line ends (a line end is counted as one character). The parameter `Length` is the length of the matched substring counting all line ends (as one character) included in the substring.

2.3.9. DeAssignVariable

C/C++

```
extern void
   (* DeAssignVariable)
(
   unsigned long VariableId
);
```

Ada 9X

```
type DeAssignVariableCallBack is access
   procedure (Variable : VariableID);
pragma Convention (C, DeAssignVariableCallBack);
DeAssignVariable : DeAssignVariableCallBack;
```

A call to the AssignVariable does not mean that the pattern assigned to the variable has been finally matched. The matching may still fail in the future. In this case the previously assigned value should be discarded. The function pointed by DeAssignVariable is called to discard a value previously assigned by AssignVariable. The value of the parameter `VariableId` is one returned by GetVariableId and passed to AssignVariable. Note that DeAssignVariable is not called when the matching process is terminated by the FAILURE pattern even if some variables have been assigned.

2.3.10. AllocatePattern

C/C++

```
extern char *
   (* AllocatePattern)
(
   unsigned int Size
);
```

Ada 9X

```
type AllocatePatternCallBack is access
   function
   (
      Size : unsigned_long
   )  return chars_ptr;
pragma Convention (C, AllocatePatternCallBack);
AllocatePattern : AllocatePatternCallBack;
```

The function pointed by the variable AllocatePattern is called by patinit to allocate memory for the translated pattern. The parameter `Size` is the number of required bytes to store the pattern. The function must return a pointer to the allocated memory block or 0 (on error).

2.3.11. Internal representation

The internal pattern format was chosen to make patterns relocatable (one can copy and move patterns). A pattern with no user-defined patterns is portable (has a machine-independent representation). The following table describes the pattern format.

Code	Description
000	SUCCESS
001..127	A character from substring atom. The atom is specified as a sequence of characters belonging to the substring. If some of characters have the code outside the range 1..127, the pattern translator chooses another substring atom representation is used (see code 205

	below).
128..191	Finite repeater. The number of repetitions is the code minus 128. So the maximal repetition count available in this form is 63. For finite repeaters greater than 63, the alternative form is used (see code 246 below).
192	END
193	ANY
194	BLANK
195	DIGIT
196	UPPER_CASE_LETTER
197	LOWER_CASE_LETTER
198	LETTER
199	CHARACTER
200	FAILURE
201	BREAK
202	Null atom. This atom successfully matches an empty substring. The pattern translator uses this atom instead of the empty substring pattern in rare cases, when its omitting would change matching semantic. Usually empty substring atoms are just ignored.
203	NL
204	Case independent form of the substring atom. The following byte is treated as the substring length (0..255). The substring body starts from the next byte. Letters are specified in lower case. The pattern translator automatically splits substrings longer than 255 bytes into chains of shorter substring atoms put in round brackets.
205	Alternative form of the substring atom. It is used when the substring contains characters with codes out of the range 1..127. The following byte is treated as the substring length (0..255). The substring body starts from the next byte. Letters are specified in lower case. The pattern translator automatically splits substrings longer than 255 bytes into chains of shorter substring atoms put in round brackets.
206	AnyOf atom. The following two bytes are interpreted as follows. The first byte gives 5 high order bits of the lowest code of characters specified by the atom. The second byte gives the size (bits) of the bitmap table that follows it. The table contains one bit set for each character specified by the atom. The lowest bit of the first table byte corresponds to the character with the lowest code.
207..242	Reserved for future extensions
243	Ellipsis
244	Assignment. The following four bytes contain the number interpreted as the variable identifier.
245	Reference to a label. The following four bytes contain the signed relative offset to the target pattern. The offset is counted in the assumption that the next byte has the offset 0. It has portable format. The less significant bit of the first byte is the offset sign. Other 7 bits are less significant bits of the offset. The next byte contains the next 8 bits of the offset and so on.

246	*Big* finite repeater. The following four bytes contain the repetition count. The format is the same as described above.
247	User defined pattern. The following bytes contain the address (`char *`) of the target pattern. The *number* of bytes and the *address* format are *machine dependent*.
248	NOEMPTY
249	FENCE
250	NOT
251	*Little* infinite repeater
252	*Big* infinite repeater
253	Right bracket. The right square bracket is represented as two byte sequence: 254 253.
254	Alternation
255	Left bracket. No matter round or square.

The pattern translator always adds one unbalanced right bracket (code 253) at the end of the pattern. So that `('A'|'B')` is translated to 255, 65, 254, 66, 253, 253.

---

3. Pattern matching utility

The **match** utility is no more than just an example of pattern usage. It searches the specified files for a text matched by the given pattern.

> **match** *pattern filename*
> **match** ^ *filename*
> **match** *pattern* ^

Circumflex instructs **match** to get the pattern from the standard input or to send the pattern internal representation to the standard output.

3.1. Pattern expression extension

Only additional features are described here (see pattern expression syntax for more detailed information).

3.1.1. Assignment

Assignment allows you to set a variable value according to matching result. For example, `...line='abc'` will print onto standard output the number of the line where `abc` substring was found. There are several variables that may be assigned. Each of them has four forms of its name. For example,

- `put=P` prints the assigned text matched by the pattern ***P*** followed by one line skip. The text may contain several lines.
- `~put=P` immediately prints the text matched by the pattern ***P*** followed by one line skip. For the explanation of the difference between a *normal* and *immediate* assignment see below.

- *put&=P* prints the matched text without line skips.
- *~put&=P* immediately prints the matched text without line skips.

The difference between *immediate* and *normal* assignments is that for the normal one the variable is assigned only after successful matching of the whole pattern. The immediate assignment mark (~) forces the variable to be assigned after each matching of the assigned pattern, even if the matching of the whole pattern fails. Therefore by matching `12345` against *`$(put&=#):`*, **match** will print `5`, i.e. the last assigned value. In contrary to this, *`$(~put&=#):`* will cause **match** printing `12345`.

3.1.2. Variables

The following variables are defined:

Defined variables and what they print

line	The number of the line where the text matched by the assigned pattern starts (1..)
column	The line position where the text starts (1..)
length	The length of the text
file	The name of the file the matched text belongs to
where	The file name + the line number + the column number
put	The text matched by the pattern
frame	The whole lines containing the text matched by the pattern
light	Same as above, but but highlights matched substring, so you can see which part of each line was matched. The VT100 escape sequences are used for highlighting.

3.1.3. Pre-defined patterns

The following additional patterns are defined by **match**:

Pre-defined patterns

c_id	Identifier of C programming language
c_com	C comment (/* ... */). Multi-line comments are allowed. For example, **match** *`'...light=c_com'`* *.c shows all lines containing C comments and highlights their bodies.
cpp_com	C++ comment. It works like c_com, but understands // as well
c_str	String literal in C (like `"the text"`). Multiline literals are allowed
c_chr	Character literal in C (like `'\015'`)
c_op	Any sequence of C statements limited by an unbalanced bracket, comma, or semicolon. For example, argument of a function call. This pattern *knows* about comments, quoted strings and character literals. It is a very sophisticated pattern. You may take a look on the example of its construction.
cpp_op	Same as above but recognizes C++ comments.
c_blank	Spaces, tabs, line feeds and C comments
cpp_blank	Same as above, but for C++

3.1.4. User defined patterns

It is possible to make a set of your own patterns. When the **match** meets an unknown pattern name it tries to find an environmental variable with the same name. In case of success it translates the value of the variable. For instance, you can set ADA_COM variable to *`'--'...END`* in your environment and then use

ADA_COM pattern in any pattern expression.

3.2. Pattern matching process

The pattern matching process starts with the first line of the file.

- If it is impossible to match the current line the process re-starts from the beginning of the next line.
- On successful matching the process re-starts from the first unmatched byte of the file. Since line boundaries are not assumed to be characters, **match** automatically goes to the next line if the current line was completely matched.

3.3. Internal pattern representation

To see how a pattern looks in internal format, you can call match as follows:

match *pattern* ^.

For example,

match *"('a'|'b')"* ^

will produce the following output:

```
/*>
  Internal representation for pattern :
  ('a'|'b')
<*/
{
  0xff, 0x61, 0xfe, 0x62, 0xfd, 0xfd
}
```

3.4. Example

Let we want to see all calls to and declarations of a C function including the parameters of that calls and declarations. Let the function have name `foo`. We could do it as follows:

match *'...frame=("foo" c_blank "(" *(","|c_op): ")")'* *.c

3.5. Bugs

Using a *shell* you should be careful about non-alpha characters which could be interpreted by the *shell* as meta symbols.

The internal buffer has a limited size. Although **match** can match a substring that is longer than the buffer size, **match** cannot assign it. For example, the pattern *...light=c_com* that normally highlights C comments in a file, cannot do that for comments larger than the buffer size. In this case you will be provided with an error message.

The number of simultaneously referred external patters is limited by 16 items.

---

4. Changes log

Changes to the version 1.0:

- Licensing wording was corrected to comply with GMGPL;

---

5. Table of Contents