Pattern Matching In Python

This paper describes a model of pattern matching implemented using the Python programming language. It's designed with the following objectives:

- To describe the style of pattern matching found in the SNBOL4, Icon and OmniMark programming languages to those who don't have an opportunity to use those languages.
- To provide examples of the advantage of a streaming approach to pattern matching as found in the OmniMark language, its predecessor the Hugo language, and in Unix's "lex" utility.
- As an attempt to fit pattern matching into the Python language in a more expressive and powerful manner than the existing "re" pattern matching library.
- To explain pattern matching in an accessible fashion, and to show that it's not such a difficult subject.

This description and the accompanying implementation should be considered a work in progress. It's not intended as a criticism of the alternative grep/Perl/re style. Rather it's intended to present an alternative style, based on that used in the SNOBOL4, Icon and OmniMark programmming languages, that has sufficient history and merit to warant serious consideration.

In addition, the implementation is best viewed as a prototype. It's no great shakes performance-wise. There's no compile-time optimizations, which would greatly help. But it runs quite nicely for quite a variety of uses.

---

1. Different Kinds of Pattern Matching

When I started looking into Python, having heard that it had text pattern matching support, and read in the recent (version 2.3) Python docs that it had adopted the Icon programming language's model for iterator and generators, I had high hopes that I'd find pattern matching of a similar sort to that in the Icon language. What I found was a module ("re") implementing a string-based "regex" grep-like pattern matching, similar to that in Perl, that can be applied to Python strings.

1.1 Icon-Like Pattern Matching

Pattern matching in the Icon language is different from that in Perl. It's an integral part of the language, like pattern rules in Perl, but its syntax is much closer to that of other expressions, similarly to arithmetic or string-valued expressions, and all the pattern operations are provided as operators or functions in the language. The advantage of this approach is two-fold:

- Pattern matching can be used in a wide variety of contexts, and that it can be readily extended in a variety of ways, allowing users to build application-specific pattern types and operations.
- It fits better into the syntax of Icon, and would fit better into the syntax of Python than does the Perl model.

Icon's style of pattern is a development of that in the earlier programming language SNOBOL4, and was the inspiration for the pattern matching facilities of the OmniMark programming language.

So I've been putting some time into seeing whether I could use Python to implement something closer to what Icon provides. And it turns out that a quite close, and very usable implementation is possible. The rest of this document describes a few modules that implement a new pattern matching model for use in

Python.

1.2 A Short History

The style of pattern matching described in this document started out in the 1960's in the SNOBOL language, a programming language for processing text whose design and implementation was lead by Ralph E. Griswold. In the 1970's he lead the design and implementation of a more general language, Icon, that had similar pattern matching features with a syntax closer to the general expressions in the language.

In the late 1970's I lead the implemenation of a programming language called Hugo at the Canadian Printing Office (the Queen's Printer) in Ottawa and used a variant of the Icon pattern language, the major difference being that it used a non-backtracking model (see later). In the late 1980's I adapted this model, still non-backtracking, for use in the OmniMark language. The current OmniMark pattern matching langauge is a development of this original design and its implementation.

The following Python pattern matching sublanguage is a mixture of those in SNOBOL4, Icon, Hugo and OmniMark. Syntactically its closest to Icon and Hugo. It has both a backtracking implementation, like SNOBOL4 and Icon, and non-backtracking implementation, like Hugo and OmniMark. It avoids the low-level explication of the model as appears in Icon, and looks to the user more like SNOBOL4, Hugo or OmniMark. Compromises have had to be made to fit into what can be currently impelemented in Python. On the other hand, there are things here that don't appear in any of those languages (like the "P1 - P2" pattern), so enjoy!

---

2. Some Examples

Here's a simple example:

```
if "abcdefgh23;ijklmn" ^ AnyOfP (string.letters) [4]:
   print "found four letters"
else:
   print "didn't find four letters"
```

What this "if" statement does is check if the string "abcdefgh23;ijklmn" starts with four letters (of course it does):

- "^" applies a pattern to a string or streaming input. It returns True if it successfully matches, or False if the pattern fails to match.
- "AnyOfP (string.letters)" creates a pattern that matches a letter. (The Python module "string" is a useful source of sets of characters that can be used with AnyOfP and NoneOfP.)
- "[4]" takes a pattern and returns a pattern that matches it four times.

Here's a slightly more complex example:

```
subject = MatchingInput ("abcdefgh23;ijklmn")
if subject ^ AnyOfP (string.letters) [4] >> "four letters":
   print "found four letters: \"" + subject ["four letters"] + "\""
else:
   print "didn't find four letters"
```

The "if" statement checks for four letters again, but additionally saves them away for later use:

- "MatchingInput" creates an input stream that can be used for matching. It's what the matching operator ("^") wants as its first argument. (If "^" is given just a string, it coerces to a MatchingInput.)

- ">>" assigns what's matched by the pattern to a dictionary entry in the "^" operator's first argument, in this case named "four letters".
- "["four letters"]" retrieves a named entry from a MatchingInput's dictionary.

All the common (and some uncommon) pattern matching operators are available:

```
if "abcdefgh23;ijklmn" ^ (AnyOfP (string.letters) [4] | \
                          AnyOfP (string.digits) [2:] & IsP (";")):
   print "found four letters or two or more digits and a semicolon"
else:
   print "didn't find four letters or two or more digits and a semicolon"
```

The "if" statement has been extended to matching either four letters, or matching two or more digits plus a tailing semicolon (only in the case of finding digits):

- "|" means "or" -- the first alternative found is the one used.
- "&" means "and" -- both what precedes it and what follows it. As is usual, "&" binds more tightly than does "|".

Multiple separate matchings can be applied to an input:

```
subject = MatchingInput ("abcdefgh23;ijklmn")
if subject ^ AnyOfP (string.letters) [4] >> "four letters":
   print "found four letters: \"" + subject ["four letters"] + "\""
elif subject ^ AnyOfP (string.digits) [2:] >> "the digits" & IsP (";"):
   print "found two or more digits followed by a semicolon: \"" + \
         subject ["the digits"] + "\""
else:
   print "didn't find four letters or two or more digits and a semicolon"
```

In this example, if the four letter match fails, the digit match is attempted on the same input.

As a final example, here is scanning through an input and matching as much as possible:

```
subject = MatchingInput ("abcdefgh23;ijklmn")
while True:
   if subject ^ AnyOfP (string.letters) [4] >> "four letters":
      print "found four letters: \"" + subject ["four letters"] + "\""
   elif subject ^ AnyOfP (string.digits) [2:] >> "the digits" & IsP (";"):
      print "found two or more digits followed by a semicolon: \"" + \
            subject ["the digits"] + "\""
   elif subject ^ AnyOfP (string.letters) [1:3] >> "1 2 3 letters":
      print "found less than four letters: \"" + \
            subject ["1 2 3 letters"] + "\""
   else:
      break
```

In this example:

- "while True" is used to repeatedly match patterns against the input.
- "break" is used when no pattern will match -- it's time to give up.
- When a successful match is made against an input, the data it matches is consumed from the input, so that any following pattern matching is applied against the next thing input.

---

3. Matching Inputs

The string "abcdefgh23;ijklmn" or the "subject" in the previous examples is a what a pattern is matched against. It's a "matching input", or in Icon's terms, a "subject".

A matching input is just like a stream input (an input file, for example), except that it manages the input for the benefit of pattern matching:

- It holds input that's already read in so that if a pattern fails to match, the held input is available for another match.
- It flushes out what's already matched so, for example, one doesn't need to keep a whole file in memory to examing and/or transform it.
- It holds a dictionary of matched and saved parts of the input for later use.
- It allows multiple nested matches against the same input.

---

4. Backtracking vs. Non-backtracking Pattern Matching

(This section can be skipped if you want to quickly start working with the implementation.)

There are two underlying models of how patterns work: backtracking and non-backtracking. A simple example of the difference between the two is the following match:

```
"aab" ^ (IsP ("a") | IsP ("aa")) & IsP ("b")
```

- Backtracking pattern matching "backs up" when it fails and attempts any available alternatives. Under the backtracking regime, the first "a" will be matched, then an attempt will be made to match the "b", but the second "a" will be found and the "b" match attempt will fail. At that point the pattern matching will back up to the second alternative of the "or" part of the pattern, and the "aa" will be matched, leading to a successful match for the "b". So the pattern match will succeed.
- Non-backtracking pattern matching doesn't back up to find an alternative of an already successful match, so when the "b" match fails, the pattern as a whole fails.

The apparent weakness of non-backtracking pattern matching isn't as serious as might at first seem the case. For one thing, there's often a simple rewrite of the pattern that performes the desired match. For the above pattern, either of the following will do:

```
"aab" ^ (IsP ("aa") | IsP ("a")) & IsP ("b")
"aab" ^ IsP ("a") [1:2] & IsP ("b")
```

Another pattern that illustrates the difference between the two approaches is the following:

```
"abcd.efg" ^ MoveP (1) [0:] & IsP (".")
```

([0:] in this notation matches zero or more of the pattern it qualifies.)

Again, this match succeeds under backtracking pattern matching and fails under non-backtracking pattern matching:

- There are two backtracking pattern matching models of what goes on here.

  In the first, all the input is first matched by "MoveP (1) [0:]" and the attempted match to "." fails. Then the shorter alternatives of "MoveP (1) [0:]" are tried until it matches "abcd" and the following match of "." succeeds.

In the second model, "MoveP (1) [0:]" matches the shortest possibility and when "." isn't found, attempts the progressively longer alternatives until it does.

- Under non-backtracking pattern matching "MoveP (1) [0:]" matches all that it can, the "." fails to match, and the pattern immediately fails.

Again, non-backtracking pattern matching can be made to work with a simple rewrite:

```
"abcd.efg" ^ NoneOfP (".") [0:] & IsP (".")
```

Advantages of non-backtracking pattern matching include:

- Non-backtracking pattern matching is generally more efficient, as the path through them is more straight-forward and less back-and-forth. Backtracking pattern matching can often be optimized by a good compiler into the non-backtracking case, but doing so becomes difficult or impossible where the pattern is no logically equivalent to a simple regular expression.

  Bactracking pattern matching languages often provide additional operators to direct pattern matching, largely to help avoid the use of inefficient and worthless paths.

- Non-backtracking patterns are generally easier to understand, again because of the straight-through route successful matches take.

Both backtracking and non-backtracking pattern matching have their place. Non-backtracking is an entirely adequate choice most of the time. Backtracking is the choice when pattern matching requirements start to become complex.

The Python pattern matching implementation that accompanies this paper includes both bactracking and non-backtracking versions of the patterns.

---

5. A Short Reference Manual

5.1 Matching Inputs

Both matching inputs and patterns are implemented as Python objects. Patterns don't have any properties or methods of use to the user -- they are used when matching. Matching inputs, on the other hand, have a variety of useful properties, and they are described here.

Matching inputs are created in one of three ways: explicit creation from a string, explicit creation from a file or file-like value, and implicit creation from a string when it's passed as the first argument of the "^" operator:

```
MatchingInput (string)
MatchingInput (file)
string ^ pattern
```

Where "file" is referred to as the argument of MatchingInput, what's really required is only any object that has a .read method that returns input in the same manner as does the .read method of an opened file object.

5.1.1 MatchingInput [...]

Indexing a matching input with a string value accesses the matching input's dictionary.

```
keyvalue = subject ["key"]
```

(In this example and the following, "subject" is a value of type MatchingInput. The term "subject" is taken

from the Icon programming language.)

A matching input's dictionary contains the values saved within a pattern using the SetP function or >> operator.

5.1.2 MatchingInput.Sets

The .Sets property of a matching input is an explicit reference to its dictionary. All common dictionary operations can be performed on it, including the useful ".has_key" method.

```
keyvalue = subject.Sets ["key"]
```

5.1.3 MatchingInput.ClearSets

Without any arguments, the .ClearSets method clears the matching input's dictionary, so that it has no entries.

```
subject.ClearSets ()
```

With one or more arguments, .ClearSets clears only the specified entries (if they exist -- they don't need to) from the matching input's dictionary.

```
subject.ClearSets ("key", "value")
```

5.1.4 MatchingInput.AllMatched

The .AllMatched property contains all of whatever was matched by the last use of the "^". If the match attempt failed (or prior to any match), the value of .AllMatched is None. .AllMatched means you don't need to save the whole of a pattern's match in the matching input's dictionary.

```
subject.AllMatched
```

5.1.5 MatchingInput.SetBufferSize

The .SetBufferSize method sets the size of the buffer used by a matching input. Normally a matching input's buffer is just as large as it needs to be for any current pattern matching. However, this can be inefficient. So .SetBufferSize is provided to up the buffer size used. It can be any positive value, but too too big can be inefficient, so care needs to be taken using this method. If you don't know, or don't particulary care, ignoring this method is just fine

```
subject.SetBufferSize (1024)
```

5.1.6 lastMatch

The lastMatch function (it's not a method), returns the matching input object most recently passed to a "^" operator or Match function.

```
keyvalue = lastMatch () ["key"]
if lastMatch () ^ AtTheEndP (): ...
```

lastMatch has no arguments. lastMatch is useful when you dont't have a name for what was being scanned. For example, where "usersName" is a string value, you can go:

```
if usersName ^ NoneOfP (" ") [1:] >> "first" & AnyOfP (" ") [0:]:
   print "first name =", lastMatch () ["first"]
   if lastMatch () ^ MoveP () [1:]:
      print "other names =", lastMatch () # because of str
```

5.1.7 str

Python's "str" function can be applied to a matching input object. If it is, it consumes all the remaining input from that matching input and returns it, leaving nothing for any future match.

```
alltherest = str (subject)
print subject
```

5.2 The Match Operator

```
Match (matchinginput, pattern)
matchinginput ^ pattern
string ^ pattern
```

The pattern is applied to first argument, which can either be a string or the result of invoking MatchingInput. If the match succeeds, the matched input is "consumed" and the following input available for a following match.

If the first argument is a matching input object, then that object can again be used to continue matching on the same input. Alternatively, the user can invoke the lastMatch function to retrieve the matching input that was used by a previous match. If the first argument of a Match operator is a string, then the lastMatch function is the only way to access input captured by the pattern, and the only way to continue further matching on it.

The "^" operator returns True or False, depending on whether it succeeds. It's typically used as the argument of an "if" or "while".

The "^" operator updates the matching input value returned by the lastMatch function immediately prior to returning its result.

5.3 Infix Pattern Operators

Some of the basic pattern operations are implemented as infix operators. Each of these operators has an equivalent function form.

5.3.1 Or Pattern

Match either the first or second pattern, the first pattern in preference.

```
pattern | pattern
OrP (pattern, pattern)
```

The function form of OrP can have zero or more arguments. With zero arguments it always fails. Otherwise if any of the argument patterns match, it matches.

5.3.2 And Pattern

Match both the first and second patterns, in that order.

```
pattern & pattern
AndP (pattern, pattern)
```

The function form of AndP can have zero or more arguments. With zero arguments it always succeeds. Otherwise all of the argument patterns much match, in the order given.

5.3.3 Arbitrary Match Pattern

Match zero or more instances of the first pattern, followed by one instance of the second pattern. The minimum number of matches of the first pattern is done to satisfy the second pattern.

```
pattern ^ pattern
ArbP (pattern, pattern)
```

The ArpP pattern is useful for testing something anywhere in input data.

Note that the ArbP operator is "^", just as is the matching operator. There's no conflict, because you get one when the first argument is a matching input and the other when both arguments are patterns. Care needs to be taken with parenthesization when using both in an expression: "input ^ pattern1 ^ pattern2" is in error because Python reads it as if entered "(input ^ pattern1) ^ pattern2" -- "input ^ (pattern1 ^ pattern2)" does the right thing.

5.3.4 Set Pattern

Each matching input has an associated dictionary (its ".Sets") property, in which parts of a matched pattern can be saved. SetP matches the given pattern and then sets the matched text to the dictionary entry with the key "string". If the match fails, the dictionary entry is unchanged.

```
pattern >> string
SetP (string, pattern)
```

5.3.5 SetOne Pattern

Like SetP, SetOneP matches the given pattern and then sets the matched input to the dictionary entry with the key "string". Unlike SetP, SetOneP requires that only one input token be matched, and it sets the dictionary entry to the one token matched -- not to a sequence, like SetP. If there is not exactly one token matched SetOneP fails. If the match fails, the dictionary entry is unchanged.

```
pattern << string
SetOneP (string, pattern)
```

SetOneP is useful for matching non-text input, where the individual tokens are of importance, rather than sequences of them.

5.3.6 Within Pattern

```
pattern + pattern
WithinP (pattern, pattern)
```

Match first pattern, and then match the second pattern within what was matched by the first pattern. The "+" pattern matches what is matched by the second pattern.

An example of WithinP's utility is the following:

```
line_width = 80
while True:
   if subject ^ NoneOfP ("\n") [line_width + 1] + \
                (NoneOfP (" ") [1:] & IsP (" ")) [1:]:
      print subject.AllMatched
   elif subject ^ NoneOfP ("\n") [0:line_width] >> "line" & IsP ("\n") [0:1]:
      print subject ["line"]
   else:
      break
```

which does simple formatting on text input. The "if" picks up part of a line (81 characters), matches the part of it that consists of whole words (followed by spaces), and prints it on a line. The "elif" picks up tail ends of lines and pieces with more than 80 characters that have no spaces in them. The "else" exits when

there's no more input.

5.3.7 Except Pattern

Match first pattern, but succeed only if the second pattern does not match the whole of what's matched by the first pattern. ExcpeptP is useful where a complex pattern matches a little too much and it's easier to say what shouldn't be matched than revise the first pattern.

```
pattern - pattern
ExceptP (pattern, pattern)
```

5.4 Prefix Pattern Operators

A couple of useful pattern operations are implemented as prefix operators. Each of these operators has an equivalent function form.

5.4.1 Test Pattern

Match the pattern, but consume no input data. TestP is useful for "looking ahead" in the input to see what follows.

```
+ pattern
TestP (pattern)
```

5.4.2 Not Pattern

Match the pattern and then fail to match if the pattern matches, and succeed if the pattern failed to match. On success no input data is consumed.

```
- pattern
NotP (pattern)
```

5.4.3 Once Pattern

Match pattern but if it yields more than one alternative match, use only the first one yielded. For example ~ P [0:] (or OnceP (P [0:])), where "P" is a pattern, will match the largest repetition of "P", but will not attempt to match any smaller repetitions, even if doing so might produce a successful match.

```
~ pattern
OnceP (pattern)
```

OnceP can help speed patterns up, when it's known that the it's either the first match attempt or nothing -- useless matches aren't attempted. OnceP only makes sense in for the backtracking implementation. For the non-backtracking implementation there is only one alternative ever matched (or none), so OnceP just invokes the pattern passed to it.

5.5 Repetition and Optional Patterns

Repetition and optional patterns are the things usually denoted by "+", "*" and "?" in "regex" pattern syntax. In this implementation, those operators aren't available, so an alternative is used:

```
pattern [count]
```

Matches the given pattern "count" times. "count" can be zero or more. The pattern fails if the pattern matches less than "count" times.

```
pattern [count1:count2]
```

Matches the given pattern at least "count1" times, and no more than "count2". The pattern fails if the pattern matches less than "count1" times. It does not attempt to match more than "count2" times.

Either "count1" or "count2" can be omitted. If "count1" is omitted it defaults to zero. If "count2" is omitted, it defaults to unlimited -- it'll match as many as their are. Some common useful combinations are the following:

```
pattern [0:]        # zero or more
pattern [:]         # also zero or more
pattern [1:]        # one or more
pattern [0:1]       # zero or one -- i.e. optional
pattern [:1]        # also zero or one
```

There are function forms, here shown with their equivalent operator forms:

```
RepP (pattern)                  # pattern [:]
RepP (pattern, count)           # pattern [count:]
RepP (pattern, count1, count2) # pattern [count1:count2]
PlusP (pattern)                 # pattern [1:]
OptP (pattern)                  # pattern [0:1]
NOfP (pattern, count)           # pattern [count]
```

In the backtracking implementation each of these patterns by default attempt matching with the largest number of repetitions first, and then back down if that attempt doesn't produce a successful match -- it backs down until it results in a successful match or until the minimum count is reached, after which it fails.

This behaviour can be reversed. Where a range of itterations is specified (i.e. where a colon is used) there's an additional option: as a third argument you can specify a single character string containing "s" (or upper-case "S") which indicates that the smallest number of repetitions should be tried first, and then incremented until it results in a successful match or until the maximum count is reached. For example, the following means start with zero repetitions and if that doesn't work, try again with one, etc.:

```
pattern [::"s"]
```

For the non-backtracking implementation the "s" option is interpreted as follows:

- If "s" is not specified, find the largest number of repetitions that matches and use that. If doing so causes a failure later on, don't attempt any alternatives.
- If "s" is specfied, only attempt the smallest number of allowed repetitions. It's effectively equivalent to saying: use the lower bound as the upper bound value as well.

5.6 Pattern Functions

5.6.1 Is Pattern

Match if "string" is the next thing in the input.

```
IsP (string)
```

5.6.2 AnyOf Pattern

Match if the next input token (character) is one of those in "charset". "charset" must be a string. To match any character, use MoveP (1).

```
AnyOfP (charset)
```

5.6.3 NoneOf Pattern

Match only if the next input token (character) is none of those in "charset".

```
NoneOfP (charset)
```

5.6.4 Pos Pattern

Match only if the current input position is that specified. "position" must be an integer value. At the beginning of matching the input position is zero, and is incremented for each input token matched.

```
PosP (position)
```

If the value of "position" is a negative number, then match if the current position is the length of the input plus the position. In general, PosP (position) with a negative position is equivalent to:

```
TestP (Move (- (position + 1)) & AtTheEndP ())
```

It might help to note that PosP (-1) is equivalent to AtTheEndP ().

5.6.5 Unordered Pattern

Match if the given patterns match in any order. UnorderedP is similar to AndP, but AndP requires that the patterns be matched in the order given. UnorderedP can have zero or more arguments. With no arguments, like AndP, it always succeeds.

```
UnorderedP (pattern, pattern, ...)
```

5.6.6 AtTheEnd Pattern

Match only if there is no more input.

```
AtTheEndP ()
```

5.6.7 Move Pattern

Match the given number of input tokens. "number" must be an integer. Match fails if there are not the given number of input tokens left in the input. Without an argument, MoveP matches one input token.

```
MoveP (number)
```

5.6.8 Bal Pattern

Match as much of the input as is properly nested with respect to the given delimiters. "delimiters" must be a string. Each pair of characters is taken to be an opening/closing pair. So, for example, BalP ("()[]{}") matches input balanced with respect to "("/")", "["/"]" and "{"/"}" pairs.

```
BalP (delimiters)
```

5.6.9 Longest Pattern

Match pattern but if it yields more than one alternative match, use only the longest one yielded: the one that consumes the most input.

LongestP (pattern)

LongestP only makes sense in for the backtracking implementation. For the non-backtracking implementation there is only one alternative ever matched (or none), so LongestP just invokes the pattern passed to it.

5.6.10 Another Pattern

Match another instance of what was previously matched and set to the dictionary entry "string". In other words, match "IsP (subject [string])", but delay the access of the dictionary entry until the actual AnotherP match is attempted. If there is no dictionary entry with the key "string", then the match fails.

```
AnotherP (string)
```

5.6.11 ClearSet Pattern

Clear the matching input's dictionary:

- If no argument is given, remove all entries.
- If one or more arguments are given, clear only those entries whose keys are specified. If there's no dictionary entry with a given key, just ignore it.

```
ClearSetP (string, ...)
```

ClearSetP always succeeds and consumes no input.

The ClearSetP pattern is equivalent to using the .ClearSets method of a matchingt input. It's provided to allow clearing to be done during pattern matching.

5.6.12 StartOf Pattern

Match if and only if the next input token is one of those in the "charset", and either it's the first token in the input, or the previous token is not in the "charset". For example: "StartOfP (string.uppercase)" matches at the start of a string of uppercase letters.

```
StartOfP (charset)
```

5.6.13 EndOf Pattern

Match if and only if the previous input token is one of those in the "charset", and either it's the last token in the input, or the next token is not in the "charset". For example: "EndOfP (string.uppercase)" matches at the end of a string of uppercase letters.

```
EndOfP (charset)
```

5.6.14 Succeed Pattern

Cause the pattern as a whole to succeed without matching any more input. No matter where SucceedP appears in a pattern, it ends the matching by that pattern.

```
SucceedP ()
```

5.6.15 Fail Pattern

Cause the pattern as a whole to fail without attempting to match any more input. No matter where FailP appears in a pattern, it ends the matching by that pattern.

```
FailP ()
```

5.6.16 Fence Pattern

Successfully match without consuming any input, but if an alternative is asked for, cause the pattern as a whole to fail. In the non-backtracking implementation, alternative attempts are not tried, so all this function does is succeed, not consuming any input.

```
FenceP ()
```

5.6.17 Yes Pattern

Match without consuming any input. Equivalent to AndP ().

```
YesP ()
```

5.6.18 No Pattern

Fail to match. Equivalent to OrP ().

```
NoP ()
```

YesP and NoP are handy as default values of pattern variables.

5.7 Operator Precedence

The Python operators used in this implementation have different precedences. For example, "&" binds more tightly than does "|" but less tightly than "+" or "-". Precedence affects how expressions are written, especially with respect to when parenthesization is needed, so here's a quick outline of the precedence levels of these operators (taken from the Python 2.3.3 Language Reference), with the least tightly binding first and the most tightly last:

```
|
^
&
>>
+ - (infix)
+ - (prefix)
[...] (repetition)
```

An important consideration is that "|" is the only thing that binds more loosely than the match operator ("^"). Parenthesization is generally not required within a pattern on the right-hand of a "^" if it doesn't contain "|" and doesn't need parenthesization for other reasons.

---

6. Pattern Matching Text

The patterns and facilities defined earlier are not text-specific -- they can be applied to any kind of input stream. They work just fine for text-specific pattern matching, but it helps to have a few extra facilities when working with text. To support this additional functionality, there's an extra module (textpatterns_nb.py or textpatterns_b.py) that implements the following patterns:

6.1 Letter Pattern

Match a single letter.

```
LetterP ()
```

6.2 Digit Pattern

Match a single digit.

```
DigitP ()
```

6.3 LineText Pattern

Match a single character so long as it's not a new-line character (LF or CR).

```
LineTextP ()
```

6.4 Whitespace Pattern

Match a single whitespace character.

```
WhitespaceP ()
```

6.5 Range Pattern

Match a single character with a value no less than that of "lowbound" and no more than "highbound". "lowbound" and "highbound" should be single characters (or other comparable values if the matching input is other than characters).

```
RangeP (lowbound, highbound, excluding)
```

The third argument, "excluding", is optional. If specified it should be a string of characters. If the single character otherwise matched by RangeP is one of the "excluding" characters, then RangeP fails to match. Use of the third argument is effectively shorthand for:

```
RangeP (lowbound, highbound) - AnyOfP (excluding)
```

6.6 Nocase Pattern

Match the string "text" (like the pattern IsP) but ignore the case of the text. In other words, match if it's the same in uppercase or lowercase. The matched text doesn't have to be all uppercase or lowercase, each character is compared separately.

```
NocaseP (text)
```

---

7. Defining Your Own Patterns

It's easy enough to assign a pattern to a name. For example:

```
HexDigit = AnyOfP (string.hexdigits)
```

To define your own pattern creation function only requires writing a function that returns a pattern:

```
def HexDigitP ():
  return AnyOfP (string.hexdigits)
```

You may need to add your own logic to a pattern returning function. In fact the best way to do it is to define a pattern object. A bit of work is required to define your own pattern object. However, all the

definitions are quite small, so it's not as difficult as it might look to start with.

7.1 More on Matching Inputs

To help in defining matching inputs have methods and attributes in addition to those described earlier:

```
.Buffer
```

A buffer in which the matching input holds the input that has been read already. It is guarranteed that there will be at least one character prior to the current input position -- allowing one-look-back matches to be done.

7.1.1 MatchingInput.Offset

The count from the beginning of the input of the first token in .Buffer.

```
.Offset
```

7.1.2 MatchingInput.Pos

The current position in the input buffer, counting from the start of the input.

```
.Pos
```

The next input token (of a matching input called "subject") is therefore at location: "subject.Buffer [subject.Pos - subject.Offset]".

7.1.3 MatchingInput.GetText

Ensure that there are at least "count" more input tokens (characters) available. .GetText returns the "count" characters made available. If the end of the input is reached, and there are less than "count" more characters in the input, then only those available are returned.

```
.GetText (count)
```

7.1.4 MatchingInput.AtTheEnd

Return True if there are no more input characters -- False if there are. .AtTheEnd may read more input to be sure whether or not there are more input characters.

```
.AtTheEnd ()
```

7.2 Implementing for Non-Backtracking Patterns

So equiped with these facilities, here's the non-backtracking definition of a slightly complex pattern -- it's the implementation of the StartOfP pattern defined as part of the accompanying implementation:

```
class StartOfP (Pattern):
   def Match (self, subject):
      return not subject.AtTheEnd and \
        subject.Buffer [subject.Pos - subject.Offset] in self.charset and \
        (subject.Pos == 0 or \
         subject.Buffer [subject.Pos - subject.Offset - 1] not in self.charset)
   def __init__ (self, charset):
      self.charset = charset
```

The parts of the definition are:

- A class definition that declares the class to be a subclass of the class Pattern.
- An .__init__ method that accepts the arguments required to create the pattern. In this case it's just the set of characters that signify the start of what's to be started.
- A .Match method that takes a matching input as its argument and returns True or False depending on whether it matches.

In this instance, the .Match method examines the characters both before and at the current position in the input and ensures:

- that either the current character is the first in the input or that the previous character is not in the given character set, and
- that there is at least one more input characters, and that it is in the given character set.

That pretty well covers it.

7.3 Implementing Backtracking Patterns

The above implementation of StartOfP works with the non-backtracking implementation but needs modification to work with the backtracking implementation. The big difference between the two is that the backtracking implementation uses Python's generator functionality. Here's the backtracking StartOfP:

```
class StartOfP (Pattern):
   def Match (self, subject):
      if not subject.AtTheEnd and \
         subject.Buffer [subject.Pos - subject.Offset] in self.charset and \
         (subject.Pos == 0 or \
          subject.Buffer [subject.Pos - subject.Offset - 1] \
                not in self.charset):
         yield None
   def __init__ (self, charset):
      self.charset = charset
```

The class definition and the object creation (.__init__) is the same, as is the condition in the "if" statement. The difference is how the .Match method returns its value.

Instead of returning a True/False value, .Match must "yield" for each successful result. This makes it a Python "generator". When it returns, it signals the end of the sequence of what it's yielding. In the case of this StartOfP, .Match yields once, on success, and not at all, on failure.

The yield statement doesn't actually yield a value. Just the fact that it yields is sufficient to indicate a successful match. (None is yielded to satisfy Python's syntactic requirements.) The change in value of the .Pos property of the matching input delimits what input data has been matched.

Patterns such as OrP can yield a success more than once if, for example, both alternatives of the OrP succeed.

Again that's it.

---

8. Pattern Matching On Other Than Text

The dynamic nature of Python's type system means that there's no reason for pattern matching to be limited to input streams of characters. All of the above patterns can be applied to a stream of any type. In particular:

- The argument of MatchingInput can be a string, a Unicode string, or any object with a .read method that returns a list of input tokens.
- Where "charset" or "delimiters" appears as an argument of a pattern, it can be a string, a Unicode string, or a list of tokens of any type.
- Where "character" is used in a pattern's description, it can be a string character, a Unicode character, or any token of any type.

Two interesting non-text possibilities for pattern matching are:

- Processing streams of parsed XML tokens.
- Processing streams of lexical token of a programming or other specification language, in the manner of the venerable "yacc" Unix utility.

But that's another story.

---

9. Implementation Considerations

The implementation that accompanies this paper can readily be ported to other programming languages. Python is a convenient first target because of the flexibility of its dynamic type system and a few other useful implementation tools.

In any language, the particularities of that language will determine in large part just how the implementation and its user-level interface will look, so what operators are used and how they fit together will change from language to language.

The requirements on any language for implementing an Icon-like pattern matching facility are the following:

- There must be some form of persistent data value, either using objects as in this Python implementation, or using "closures". Here's a closure-based implementation of the "StartOfP" pattern:

```
def StartOfP (charset):
   def Match (subject):
      return not subject.AtTheEnd and \
        subject.Buffer [subject.Pos - subject.Offset] in self.charset and \
        (subject.Pos == 0 or \
         subject.Buffer [subject.Pos - subject.Offset - 1] not in self.charset)
   return Match
```

  The difference from the object-based implementation is that now "StartOfP" is a function that returns another function, that the persistence of the "charset" argument to "StartOfP" is used to make it available when the Match function that is returned is called at a later time, and that there's no more of that "self" object-oriented stuff going on.

- For infix and prefix operators (and in some languages suffix operators) to be defined, there needs to be some form of user-defined operator overloading. In Python there's a somewhat limited set of operators that can be user-defined, and they can only be defined on user-defined object types (classes). So in this implementation, objects were used.
- Persistence is also required for implementing matching input, but objects do that job nicely, holding a buffer and having properties that manipulate the buffer.

What would make things even nicer are the following:

- A static type system, so that appropriate pattern matching operators could be defined on other than pattern values. For example, the "IsP" pattern is irritating. It would be nice to have a prefix operator that would do the job. The Icon language uses "=" as a prefix operator for text matching.

- A more general operator definition facility would be nice, so that operators more appropriate to pattern matching could be defined. For example, instead of "^" for feeding a matching input to a pattern, a named infix operator "matches" would be more expressive, as in:

  ```
  if subject matches ="<" & nameP & =">":
  ```

- The existing overloadings of the basic arithmetic operators "+" and "*" make it more difficult to use in other additional ways. Their definitions weren't bad in the original Python language, but as the language has grown in use and in the range of its applications, early decisions such as this become less and less appropriate. This is a phenomenon not unique to Python, but is an indication that the time is coming when it would be best to conceive of a successor to the Python language.

- A static type-based operator overloading would help. This is needed so that user-defined operators could be defined directly on types other than objects.

---

10. Where To Get It

The package is available as a ZIP file: patternmatching.zip. It includes:

- patternmatching.html: this document.

- PatternMatchingInPython.txt: this document as a text file.

- patterns_nb.py: the non-backtracking implementation.

- patterns_b.py: the backtracking implementation.

  The backtracking implementation seems to run about 50% slower than the non-backtracking implementation. So given that, and that the non-backtracking implementation is easier to understand, it's best to use the non-backtracking implementation when it does the job, and this backtracking one when its additional functionality is required.

- textpatterns_nb.py: a non-backtracking implementation of text-specific patterns.

- textpatterns_b.py: a backtracking implementation of text-specific patterns.

- MatchingInput.py: the implementation of the MatchingInput class.

- patternusers_nb.py: the non-backtracking version of a collection of useful functions that use patterns in their implementation. The file describes the use of its functions, and their definitions should be short enough to explain where the documentation falls short.

- patternusers_b.py: the backtracking version of the same.

- text2html.py: an example of using the implementation. This program was used to convert the text version of this document to HTML. It uses the patterns_nb.py pattern module, but can equally use the patterns_b.py module.

In the "example1" folder there's another example of using the pattern matching facilities:

- makeslides.py: This program converts a text file into a set of slides using a template HTML file.
- makeslides.txt and template.html: Input to the example.
- slides.css and bullet.gif: Material used by the resulting "slides".

---

11. Updates

The following changes have been made since the original posting of this material:

18 August 2004:

- The lastMatch function has been added.
- The str function has been added for the MatchingInput type.
- A bit of reorganization has been done -- subsections have been added for each operator and function -- to make things easier to find.
- In the implementation, all classes have been made Python "new style".
- Fixed an error in the "within" function in patternusers_nb.py and patternusers_nb.py.
- Fixed an error in the RangeP in textpatterns_nb.py and textpatterns_b.py and added a third optional argument to that pattern.
- format.py and renamed text2html.py has been updated to support a table of contents, an author declaration, and intradocument links.
- The makeslides.py example has been added.

25 July 2004:

- Added "A Short History".
- Replaced patternusers.py with patternusers_b.py and patternusers_nb.py so that the importing works properly.

2 July 2004:

- The patternusers.py module has been added.
- format.py has been updated.
- An error in the ArbP function in patterns_nb.py has been corrected.

Thu Sep 09 21:05:02 2004