When *Maybe* is not good enough

Mike Spivey, May 2011

A glimpse into the prehistory of parser combinators.

1 Introduction

Many variations upon the theme of parser combinators have been proposed (too many to list here), but the central idea is simple: a parser for type α is a function that takes an input string and returns a number of results, each consisting of a value x of type α and a string that is the portion of input remaining after the phrase with value x has been consumed. The results are often organized into a list, because this allows a parser to signal failure with the empty list of results, an unambiguous success with one result, or multiple possibilities with a longer list.

$$\textbf{type}\ \mathit{Parser}_1\ \alpha = \mathit{String} \rightarrow [(\alpha, \mathit{String})].$$

Producing a list of results naturally leads to parsers that allow backtracking. If a phrase should consist of two parts A and B, and there are multiple ways of recognizing an A at the start of the input, then the parser for B can discard the first of these if necessary, and ask for a second or a subsequent way of parsing an instance of A, choosing the one for which the remaining input matches B. The lazy evaluation of Haskell is beneficial here, because results from A after the first need not be computed until they are demanded by the parser for B.

Nevertheless, it remains true that backtrack parsing is in itself inefficient, because the search tree in a parsing problem can become very large. A substantial fraction of the tree may be created and explored before finding a successful parse, consuming a lot of time. Also, as the parsing process advances, each choice between alternatives must be recorded in case it must be revisited later, and this can consume a lot of space, even if the alternatives are never explored. For the sake of efficiency, it is preferable where possible to substitute a different parser type,

$$\textbf{type}\ \mathit{Parser}_2\ \alpha = \mathit{String} \rightarrow \mathit{Maybe}\,(\alpha, \mathit{String}).$$

As we shall see, parsers with this type allow alternation but not backtracking, in the sense that two parsers can be combined into a single parser that succeeds if either of them would succeed on its own, but once it has produced a first result, there is no way to get it to produce another, even if both the component parsers would succeed on the original input. Using this parser type reduces the amount of fruitless searching, and allows the record of choices made in recognising a phrase to be discarded as soon as one of the choices succeeds.

If parsers based on *Maybe* are preferable to those based on lists, it is natural to ask what grammars allow them to work properly. If a grammar is unambiguous, then an input string will either fail to be in the language, or it will have exactly one derivation tree; we will say that the *Maybe*-based parser works correctly if in these two circumstances it returns *Nothing* or *Just x* respectively, for some value *x*.

2 Parser combinators

We shall want to experiment with both parsers that return a list of results and parsers that use *Maybe* instead. Luckily, the type class system of Haskell allows us to describe parser combinators in a way that is independent of the monad *m* that is used to deliver the results. As usual, we need to wrap the parser type in a **newtype** construction so we can make it into an instance of type classes.

$$\textbf{newtype}\ \mathit{Parser}\ m\ \alpha = \mathit{Parser}\ \{\ \mathit{runParser} :: \mathit{String} \rightarrow m\,(\alpha, \mathit{String})\ \}.$$

The type *Parser m α* contains parsers that accept a string and deliver results of type *α* using the monad *m*. If *m* is indeed a monad then so is *Parser m α*.

```
instance Monad m ⇒ Monad (Parser m) where
  return x = Parser (λs → return (x, s))
  xp ≫= f =
    Parser (λs → runParser xp s ≫= (λ(x, s′) → runParser (f x) s′)).
```

Haskell's type class *MonadPlus* describes monads that provide two additional operations *mzero* and *mplus*, for which we use here the notations $\varnothing$ and $\oplus$:

$$\begin{aligned}
&\textbf{class}\ \mathit{Monad}\ m \Rightarrow \mathit{MonadPlus}\ m\ \textbf{where}\\
&\quad \varnothing :: m\ \alpha\\
&\quad (\oplus) :: m\ \alpha \rightarrow m\ \alpha \rightarrow m\ \alpha.
\end{aligned}$$

Both the list type constructor [ ] and the *Maybe* constructor are declared in the standard library as instances of *MonadPlus*:

$$\begin{aligned}
&\textbf{instance}\ \mathit{MonadPlus}\ [\,]\ \textbf{where}\\
&\quad \varnothing = [\,]\\
&\quad (\oplus) = (+\!\!+).
\end{aligned}$$

$$\begin{aligned}
&\textbf{instance}\ \mathit{MonadPlus}\ \mathit{Maybe}\ \textbf{where}\\
&\quad \varnothing = \mathit{Nothing}\\[1ex]
&\quad (\mathit{Just}\ x) \oplus \mathit{ym} = \mathit{Just}\ x\\
&\quad \mathit{Nothing} \oplus \mathit{ym} = \mathit{ym}.
\end{aligned}$$

There are several equational laws that it is natural to consider in relation to these operations, motivated by the idea that $m\ \alpha$ in some sense represents a set of values of type α, with $\varnothing$ as the empty set and $\oplus$ as set union; and that $(\gg\!= f)$ applies f to each member of this set and collects the results. Like

the operations on sets, we expect $\oplus$ to be associative with $\varnothing$ as an identity element.

$$\begin{aligned}
&(\mathit{xm} \oplus \mathit{ym}) \oplus \mathit{zm} = \mathit{xm} \oplus (\mathit{ym} \oplus \mathit{zm}),\\
&\varnothing \oplus \mathit{ym} = \varnothing = \mathit{xm} \oplus \varnothing.
\end{aligned}$$

Also, we expect $(\gg\!= f)$ to act as a homomorphism.

$$\begin{aligned}
&\varnothing \gg\!= f = \varnothing,\\
&(\mathit{xm} \oplus \mathit{ym}) \gg\!= f = (\mathit{xm} \gg\!= f) \oplus (\mathit{ym} \gg\!= f).
\end{aligned} \qquad (*)$$

All these laws are satisfied by both lists and *Maybe*, with the exception of equation $(*)$, which is not satisfied by *Maybe*. A simple example shows this: if we define

$$\begin{aligned}&\mathit{even} :: \mathit{Int} \rightarrow \mathit{Maybe}\ \mathit{Int}\\ &\mathit{even}\ x = \textbf{if}\ x\ \text{`mod`}\ 2 \equiv 0\ \textbf{then}\ \mathit{Just}\ x\ \textbf{else}\ \mathit{Nothing}\end{aligned}$$

then the expression

$$(\mathit{result}\ 1 \gg= \mathit{even}) \oplus (\mathit{result}\ 2 \gg= \mathit{even})$$

evaluates to $\mathit{Nothing} \oplus \mathit{Just}\ 2 = \mathit{Just}\ 2$, but

$$(\mathit{result}\ 1 \oplus \mathit{result}\ 2) \gg= \mathit{even}$$

evaluates to $\mathit{Just}\ 1 \gg= \mathit{even} = \mathit{Nothing}$. The alternative value 2 is discarded as soon as the expression in parentheses succeeds with the value 1; this is beneficial in terms of efficient use of time and space, but in this example, it leads to failure, because the result 1 does not satisfy the subsequent test *even*. The law $(*)$ also fails for the parser monad *Parser Maybe* based upon *Maybe*, something that will turn out to be crucial later.

If m is an instance of *MonadPlus*, then so is *Parser m*. The additional operations are obtained by lifting the operations on m:

$$\begin{aligned}&\textbf{instance}\ \mathit{MonadPlus}\ m \Rightarrow \mathit{MonadPlus}\ (\mathit{Parser}\ m)\ \textbf{where}\\ &\quad \emptyset = \mathit{Parser}\ (\lambda s \rightarrow \emptyset)\\ &\quad \mathit{xp} \oplus \mathit{yp} = \mathit{Parser}\ (\lambda s \rightarrow \mathit{runParser}\ \mathit{xp}\ s \oplus \mathit{runParser}\ \mathit{yp}\ s).\end{aligned}$$

Given a list of results in $m\ \alpha$, where m is an instance on *MonadPlus*, we can reduce then to a single result by folding with $\oplus$.[1]

$$\begin{aligned}&\mathit{alt} :: \mathit{MonadPlus}\ m \Rightarrow [m\ \alpha] \rightarrow m\ \alpha\\ &\mathit{alt} = \mathit{foldr}\ (\oplus)\ \emptyset.\end{aligned}$$

We shall use the operations $\gg=$ and $\oplus$ to build parsers that handle concatenation and alternation in context free grammars. All that is missing now are the basic parsers that deal with individual characters. The parser *pChar c* compares the next character of the input with c and succeeds if they match, consuming the character c.

```
$pChar :: MonadPlus\ m \Rightarrow Char \rightarrow Parser\ m\ ()$
$pChar\ c =$
  $Parser\ (\lambda s \rightarrow$
    $\textbf{case}\ s\ \textbf{of}\ c' : s'\ |\ c \equiv c' \rightarrow return\ ((), s');\ \_ \rightarrow \varnothing).$
```

[1] Our function *alt* is called *msum* in the Haskell library.

For convenience, we also define *pString s* so that it recognizes the characters in the string *s* one after another.

$$\begin{aligned}&\mathit{pString} :: \mathit{MonadPlus}\ m \Rightarrow \mathit{String} \rightarrow \mathit{Parser}\ m\ ()\\ &\mathit{pString} = \mathit{foldr}\ (\gg)\ (\mathit{return}\ ())\cdot \mathit{map}\ \mathit{pChar}.\end{aligned}$$

3 Alternation without backtracking

A grammar is called $LL(1)$ if it can be recognized by a *deterministic top-down parsing machine*: that is, an automaton whose state is a stack of symbols yet to be found in the input. The possible moves of this automaton are to match a token from the top of the stack with the next token from the input, or to take a non-terminal symbol A from the top of the stack and replace it by the right-hand side s of a production $A \rightarrow s$. For the machine to be deterministic, it must be able to choose the correct production with only one token of lookahead, that is, knowing only the next token of the input.

Broadly speaking, a grammar is $LL(1)$ if, whenever alternatives $A \rightarrow B \mid C$ occur, the set of tokens that can start an instance of B is disjoint from the set that can start an instance of C. (The story is complicated a bit if either B or C can produce the empty string, but we can ignore that complication here.) If we build a parser for A from parsers for B and C by writing,

$$pA = pB \oplus pC,$$

then we can be sure that no input string would cause both pB and pC to succeed. So if pB succeeds, it is safe to rule out a subsequent attempt to apply pC, and that is what happens in a parser based on *Maybe*.

For the *Maybe*-based parser to work, it is certainly sufficient that the grammar is $LL(1)$. On the other hand, the $LL(1)$ condition is not necessary in all cases, as is easily shown by the grammar $S \rightarrow \mathsf{x}\,\mathsf{x} \mid \mathsf{x}\,\mathsf{y}$. This grammar is not $LL(1)$, because both alternatives start with x, so that with S on the stack and x as the lookahead, the automaton would not be able to choose between the two productions. Let's consider in some detail what happens when the parser,

$$pS = (pChar\ \text{'x'} \gg pChar\ \text{'x'}) \oplus (pChar\ \text{'x'} \gg pChar\ \text{'y'}),$$

is applied to input "xy" using the *Maybe* monad. First, the alternative

$$pChar\ \text{'x'} \gg pChar\ \text{'x'}$$

is tried; the first 'x' succeeds, but the second one fails, and this causes the whole alternative to fail. At this point, the alternation operator $\oplus$, seeing that its left operand has failed, is able to try its right operand, the parser $pChar\ \text{'x'} \gg pChar\ \text{'y'}$, on the original input. This parser succeeds, and the outcome of the whole parser is success.

The grammar $S \rightarrow \mathsf{x}\ \mathsf{x} \mid \mathsf{x}\ \mathsf{y}$ could, of course, be 'left-factored' to make it $LL(1)$:

$$S \rightarrow \mathsf{x}\ A,$$
$$A \rightarrow \mathsf{x} \mid \mathsf{y}.$$

But that is not the point! The unfactored grammar is *not* $LL(1)$, yet a *Maybe*-based parser still works correctly.

In a more complicated setting, *Maybe*-based parsers continue to work correctly even where the grammar fails the $LL(1)$ condition. For example, in a programming language grammar we could write,

$$stmt \rightarrow expr := expr \mid expr,$$

perhaps permitting a solitary expression as a statement in order to allow for procedures called for their side effect. A *Maybe*-based parser could work correctly with this grammar, recognising an expression at the beginning of a statement, then finding that it is not followed by :=, switching to the other alternative, and parsing the expression again as a statement in itself. It is surely true in this case that left-factoring the grammar would remove the need to parse the expression twice, making the parser more efficient. But again, that is not the point, because the parser already works correctly, even if a bit slowly, if it is directly based on the original grammar.

The ability of parser combinators to deal with grammars that are not $LL(1)$ is also useful when, perhaps favouring simplicity over robustness, we choose to build a parser that is not preceded by a scanner that divides the input into tokens. Many programming language grammars are $LL(1)$ over an alphabet of tokens, but not when the terminal symbols are taken to be individual characters; it is easy to recognize an **if** statement from its first token, but not so easy given only the information that the first character is 'i'. However, if we additionally exploit the bias of $\oplus$ towards its left-hand operand, parser combinators deal with this situation acceptably well.

Whereas left-factoring can improve the efficiency of a *Maybe*-based parser without spoiling its correctness, the same cannot always be said of right-factoring. For example, the grammar $S \rightarrow \mathsf{x}\,\mathsf{y} \mid \mathsf{x}\,\mathsf{x}\,\mathsf{y}$ gives the parser

$$\mathit{pString}\ \texttt{"xy"} \oplus \mathit{pString}\ \texttt{"xxy"}$$

that works perfectly, but factoring it as

$$S \rightarrow A\,\mathsf{y}$$
$$A \rightarrow \mathsf{x} \mid \mathsf{x}\,\mathsf{x}$$

gives a parser that no longer works properly:

$$(\mathit{pString}\ \texttt{"x"} \oplus \mathit{pString}\ \texttt{"xx"}) \gg \mathit{pString}\ \texttt{"y"}.$$

When given the input "xxy", this parser first recognizes the first 'x' with *pString* "x", then feeds the remaining input "xy" to the parser *pString* "y", which fails. With the *Maybe* monad, there is no possibility of backtracking to try the parser *pString* "xx" instead, so the whole parse fails. Notice that the equation $(*)$ would imply that the two parsers just considered are equivalent, so it is the failure of $(*)$ for the *Maybe* monad that is the root of the problem uncovered in this example.

So far, we have seen that all grammars that are *LL*(1) can be parsed with *Maybe*-based combinators, but also some grammars that are not *LL*(1). The next step is to introduce an undecidable problem, which we shall use later to show that there is, in general, no algorithm to decide whether a grammar can be parsed in this way or not.

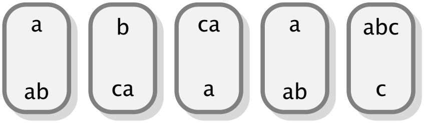

Figure 1: *A correct layout*

4 Post's correspondence problem

In Post's correspondence problem, we are given a list of tiles, each labelled with a pair of strings, and asked to find a sequence of copies of the tiles such that the strings obtained by concatenating the upper labels from each tile matches the one obtained by concatenating the lower labels (see Figure 1). Each given tile may be used once, several times, or not at all in the layout.

We can represent a tile by a pair of strings, and we say that a layout is *correct* if the upper and lower labels concatenate to give the same string:

```
type Tile = (String, String)

correct :: [Tile] → Bool
correct layout = (concat (map fst layout) ≡ concat (map snd layout)).
```

To find solutions for a given set of tiles, we can generate layouts in increasing order of length, representing each layout by a list of indices in the list of tiles:

```
choices :: Int → [[Int]]
choices n = concat (iterate step [[ ]])
  where step css = [ c : cs | c ← [0 . . n − 1], cs ← css ].
```

For example:

```
∗Main> choices 4
```

```
[[], [0], [1], [2], [3], [0, 0], [0, 1], [0, 2], [0, 3], [1, 0], ...
```

The solutions are those non-empty layouts where the upper and lower labels match.

```
solutions :: [Tile] → [[Tile]]
solutions tiles =
  [ cs | cs ← tail (choices (length tiles)), correct (map (tiles!!) cs) ].
```

Some sets of tiles have solutions, and others have none. Here is a set that does have solutions:

```
*Main> let tiles1 = [("b", "ca"), ("a", "ab"), ("ca", "a"), ("abc", "c")]
*Main> solutions tiles1
[[1, 0, 2, 1, 3], [1, 0, 2, 1, 3, 1, 0, 2, 1, 3], ...
```

The solution [1, 0, 2, 1, 3] describes the layout shown in Figure 1, with the string "abcaaabc" on both top and bottom.

If a problem has one or more solutions, then the function *solutions* will find them eventually by a blind search. Much better algorithms exist that do not search blindly: for instance, in the example only tile 1 can begin a layout, because the other tiles have top and bottom labels that start with different characters. Also, putting tile 1 down first leaves an extra 'b' on the bottom, and that means that tile 0 must come next, since other tiles do not start with a 'b' on top. So it is pointless to try any candidate solution that does not begin [1, 0, . . .], and yet the brute-force *solutions* function does just this, as well as trying many other sequences that make no sense. It would be an interesting exercise to write a better search algorithm.

It's obvious that if a problem has at least one solution, then it has an infinite number, because concatenating a solution with itself or another one will always give a further solution. Other problems have no solutions at all; for them, the function *solutions* will not return the empty list, but instead will run forever without producing any information. For some sets of tiles, we might be able to determine by inspection that there are in fact no solutions. For example, if there are no tiles where the top and bottom labels begin with the same character, then there is no tile that could start a correct layout, and it is certain that there are no solutions.

In general, however, *it is undecidable whether a given set of tiles has a solution*. I assume this result here without proving it in detail. The book by Sipser (2005) gives a proof by reduction from the halting problem for Turing machines.[2] A Turing machine and its initial tape can be represented by a set of tiles, in such a way that we can begin a correct layout by first putting down the initial state, then we can continue by adding tiles in a way that corresponds to a sequence of steps of the Turing machine. We will be able to complete the layout correctly and find a solution exactly if the Turing machine halts. This means that deciding whether the set of tiles has a solution is equivalent to deciding whether the Turing machine halts with the given input, and we know that problem to be undecidable.

Before returning to the problem of *Maybe*-based parser combinators, we will first mention a classic undecidable problem connected with parsing: the problem of deciding whether a grammar is ambiguous.

5 Ambiguity

Given a set of tiles, we can construct a context-free grammar that is ambiguous exactly if the set of tiles has a solution. The layout described in Section 4 is [1, 0, 2, 1, 3], leading to the string "abcaaabc", and we will encode this as

"3,1,2,0,1=abcaaabc".

To the left of the equals sign appears the list of tiles chosen; it appears in reverse order for ‘technical reasons’ that will emerge soon. To the right appears the concatenated sequence of labels from the tiles, in this case the same string whether we take the upper labels or the lower ones.

Given a list of labels, either the upper ones from a list of tiles or the lower ones, we can write productions to describe the set of strings that can be assembled. It’s easiest to express these productions as a function that takes the list of labels and returns a parser.

[2] The problem *tiles*$_1$ is taken from an example in the same book.

```
assembly :: MonadPlus m ⇒ [String] → Parser m ()
assembly tags = p where
  p = alt [ (pString (show i) ≫
                 ((pChar ',' ≫ p) ⊕ pChar '=') ≫ pString tag)
                                         | (i, tag) ← zip [0 . .] tags ]
```

For the list *tags* = ["b", "a", "ca", "abc"], this parser corresponds to the productions,

$$A \rightarrow 0\, A'\, b \mid 1\, A'\, \mathsf{a} \mid 2\, A'\, \mathsf{c\,a} \mid 3\, A'\, \mathsf{a\,b\,c}$$
$$A' \rightarrow ,\, A \mid =.$$

The reason for the reversed list of tile indices emerges here, because the grammar generates the list of indices and the string of tags simultaneously by growing them outwards from the middle, one in each direction.

Now, given a tile set *tiles*, if a string like "abcaaabc" can be generated as the top row of a layout, then some string like "3,1,2,0,1=abcaaabc" will be accepted by the parser *assembly* (*map fst tiles*); and if the string can be generated as the bottom row of a layout, then a similar string will be accepted by *assembly* (*map snd tiles*). Crucially, if a string appears on both the top and bottom of the *same* layout, then there will be a string that is accepted by both these parsers.

So we can put the two parsers together in alternation to get a parser that accepts at least one string in two different ways exactly if the set of tiles has one or more solutions. For present purposes, we can settle on the list monad, written [ ] in the type of the parser:

```
ambiguous :: [Tile] → Parser [ ] Int
ambiguous tiles =
  (assembly (map fst tiles) ≫ pChar '!' ≫ return 1)
   ⊕ (assembly (map snd tiles) ≫ pChar '!' ≫ return 2)
```

I've used '!' as an end-of-file marker to make sure the whole string is matched, and added the return values 1 and 2 to make it easier to see what's happening.

Expressing the same construction as a grammar, we would have one set of productions for the top labels in the tiles, similar to those for A and A' shown earlier, a second set for the bottom labels using non-terminals B and B', and two productions $S \rightarrow A\,! \mid B\,!$ to join them together. Let's try the parser on some examples:

```
*Main> runParser (ambiguous tiles1) "3,1=aabc!"
[(1, "")]
*Main> runParser (ambiguous tiles1) "3,1=abc!"
[(2, "")]
```

```
*Main> runParser (ambiguous tiles1) "3,1=babc!"
[]
*Main> runParser (ambiguous tiles1) "3,1,2,0,1=abcaaabc!"
[(1, ""), (2, "")]
```

The top labels on tiles 1 and 3 concatenate to give "aabc", and the string that encodes this fact is accepted with the result 1; similarly, the string "abc" is made from the bottom labels on the same two tiles, and a corresponding string is accepted with the result 2. On the other hand, the string "babc" does not represent either the top or the bottom labels on these tiles, so the next test string is not accepted at all. Finally, the string "abcbacba" can be obtained from either the top or the bottom labels of the tiles 1, 0, 2, 1, 3, so the string "3,1,2,0,1=abcaaabc!" is accepted in *two* ways by the parser, showing that the grammar is ambiguous.

For any set of tiles, we can form the parser *ambiguous tiles*, and that parser will return multiple results on exactly those strings that encode a solution to the correspondence problem. So the parser is capable of returning multiple results (and the underlying grammar is ambiguous) exactly if the original set of tiles has a solution. Since this question is undecidable, we have shown that it is not in general decidable whether a context-free grammar is ambiguous.

In the next section, we will use a similar construction to show that it is not decidable whether a grammar is correctly parsed by a *Maybe*-based parser.

6 Freedom from backtracking

In place of the parser *ambiguous* from the previous section, let us consider now another parser assembled from two instances of *assembly*. This time, we leave the monad *m* unspecified:

```
backtrack :: MonadPlus m ⇒ [Tile] → Parser m Int
backtrack tiles =
  inner ≫= (λx → pChar '!' ≫ return x)
  where
    inner =
      ((assembly (map fst tiles) ≫ return 1)
       ⊕ (assembly (map snd tiles) ≫ pChar '?' ≫ return 2))
```

Again, I've used '!' as an end-of-file marker, but I've carefully factored the grammar, bearing in mind that the distributive law (∗) is not satisfied by *Maybe*. Note too the presence of the character '?' in one alternative of the *inner* parser.

Let's examine what happens when we apply this parser to a typical input string:

```
∗Main> runParser (backtrack tiles1) "3,1,2,0,1=abcaaabc?!"
  Ambiguous type variable ‘m’ in the constraint: ‘MonadPlus m’
  Probable fix: add a type signature that fixes these type variable(s)
```

Oops! Because the monad *m* is undetermined in the type of *backtrack*, we'll have to specify carefully what type of answer we want before GHC can show us the result.

```
∗Main> runParser (backtrack tiles1) "3,1,2,0,1=abcaaabc?!"
                                                :: [(Int, String)]
[(2, "")]
```

Because the input string consists of a correct layout followed by "?!", the parsing goes as follows:

- The parser *assembly* (*map fst tiles*) succeeds, causing *inner* to produce the result 1 and the remainder "?!".
- On this remainder, the parser *pChar* '!' fails, causing backtracking.

- Now the parser *assembly* (*map snd tiles*) succeeds, producing again the remainder "?!". After this, the parser *pChar* '?' consumes the '?', causing *inner* to produce the result 2 and the remainder "!".
- This time the parser *pChar* '!' succeeds, and the overall outcome is success.

But what happens if we use a parser based on *Maybe* instead?

```
∗Main> runParser (backtrack tiles1) "3,1,2,0,1=abcaaabc?!"
                                               :: Maybe (Int, String)
Nothing
```

This time, the story is different. The parser *assembly* (*map fst tiles*) succeeds as before without the need for backtracking, guided by the indices embedded in the input, and the parser *pChar* '!' subsequently fails. But this time there is no possibility of backtracking to try the other branch, and the whole parse fails, yielding *Nothing*.

We should also check the behaviour of the parser for strings that can be generated only from the top or only from the bottom of a layout. For convenience, let's first give names to specialized versions of the parser:

```
∗Main> let backtrackL = backtrack :: [Tile] -> Parser [ ] Int
∗Main> let backtrackM = backtrack :: [Tile] -> Parser Maybe Int
```

Now a string that can only be generated from the top labels gives no result with either parser, because there is no way to consume the '?' character after matching with the top:

```
∗Main> runParser (backtrackL tiles1) "3,1=aabc?!"
[]
∗Main> runParser (backtrackM tiles1) "3,1=aabc?!"
Nothing
```

In this case, the *Maybe*-based parser gives a result consistent with the list-based one. Again, if a string can only be generated from the bottom labels, it gives a positive result from both parsers:

```
∗Main> runParser (backtrackL tiles1) "3,1=abc?!"
[(2, "")]
∗Main> runParser (backtrackM tiles1) "3,1=abc?!"
Just (2, "")
```

So it is only when a string represents a solution to the correspondence problem that the list-based and *Maybe*-based parsers disagree; in that case, it is the *Maybe*-based parser that gives the wrong answer, failing to recognize a string that is generated by the underlying grammar. Whether such a string exists is, as before, undecidable given the set of tiles, so we conclude that it is undecidable, given a grammar, whether the grammar is correctly parsed by the *Maybe*-based parser that is derived from it.

7 Previous work

The result presented in this article has been known for many years, and in fact for many years before parser combinators were invented. In a set of notes written for a course given in 1967, and subsequently published as (Knuth 1971), Donald Knuth describes an abstract *parsing machine*. This machine runs programs in which the instructions either recognize and consume a token from the input, or call a subroutine to recognize an instance of a non-terminal. Each instruction has two continuations for success and failure, and part of the subroutine mechanism is that if a subroutine returns with failure, then the input pointer is reset to where it was when the subroutine was called. A subroutine that returns successfully, however, deletes the record of the old position of the input pointer. There is a natural translation of context-free grammars into programs for this machine, which behave exactly like combinator parsers based on *Maybe*. Knuth proves that the correct functioning of a program for the parsing machine is undecidable, using the same reduction presented in this article, though I have changed the details in order to work within a fixed alphabet.

Acknowledgements

The author thanks Doaitse Swierstra and ... for their suggestions for improving the presentation.

References

Knuth, D. E. (1971) “Top-down syntax analysis”, *Acta Informatica* **1**, pp. 79–110. Reprinted as Chapter 14 of (Knuth 2003).

Knuth, D. E. (2003) *Selected papers on computer languages*, CSLI Publications.

Sipser, M. F. (2005) *Introduction to the theory of computation*, second edition, Course Technology.