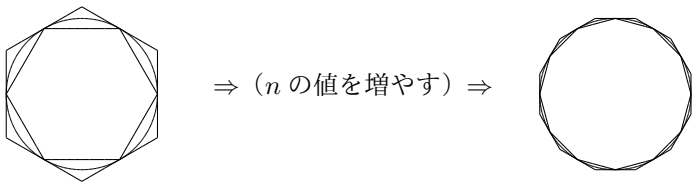
第3週の庭いじり

3.1 円周率

円周率は不思議なもので、古くから多くの人々を魅了してきた。いちばん簡単に円周率を表す数は 3 である。また、もっとも効率のよい近似値なら 3.14 というところだろうか。円周率は、小数点以下に不規則な数が並ぶので、昔から有効桁数を高める競争が行われてきた。コンピュータが発達した現代でも、それは続けられている。

円周率を求める古典的な方法は、アルキメデス[1]が考案した。円に内接する正 n 角形の周長と円に外接する正 n 角形の周長を計算し、それらにはさまれた値を円周率とすることである。はさまれた値といっても、ある範囲にはさまれてるわけだから、値が確定しない。したがって、内接 n 角形の周長と外接 n 角形の周長で、一致している部分までが円周率の正しい値を示していることになる。



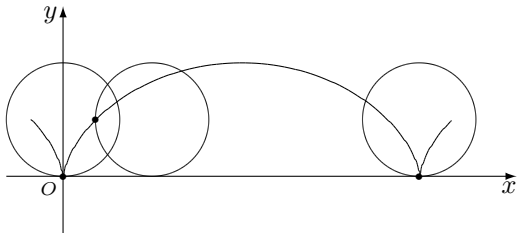
これらの計算をするには、三角比に関する知識があるとよいのだが、ここではアルキメデスの方法で円周率の近似をすることが目的ではないので省略させてもらおう。

円周率はギリシア文字 π で代用される。円周率が通常の分数で表せないので当然の処置だろう。**Haskell** には円周率の値が組み込まれていて、いつでも自由に呼び出せる。`pi` で円周率 16 桁の値が使えるので、たとえば半径 5 の円の面積などはすぐ分かる。

(ghci env.)
```
Prelude> pi
3.141592653589793
Prelude> pi*5*5
78.53981633974483
```

[1] シラクサのアルキメデス（287?B.C.–212B.C.）: 古代ギリシアの数学・物理学者。

繰り返すが円周率は通常の分数で表すことができない。つまり、半径が有理数—つまり通常の分数—ならば、円周の長さは分数値にならないことを意味する。なぜ通常の分数で表せないかを説明するのは思ったほど簡単なことではない。円周は曲がっているから、というのは理由にならない。なぜなら、曲がっていても整数値や有理数で表せるものはいくらでもあるからだ。たとえば、円が一回転するときにできる円周上のある点の軌跡は例のひとつになっている。



この軌跡は**サイクロイド**と呼ばれ、たとえば半径 1 の円なら軌跡の長さはきっちり 8 であるから、曲がった線という理由で曲線の長さが有理数でないとは言えない。ここでは、詳しい計算方法について述べることはしないが、一般に半径 a の円が描くサイクロイドの長さは $8a$ であることが知られている。

円周率は通常の分数で表せないけれど

$$\frac{\pi}{4} = 1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \frac{1}{9} - \frac{1}{11} + \cdots$$

といった、**無限級数**で表すことは可能だ。これはグレゴリー[2]もしくはライプニッツ[3]の功績だ。式が $\pi = \cdots$ ではなく $\frac{\pi}{4} = \cdots$ と書いてあるのは、その求め方に由来しているが、詳しいことは別の書物を参考にしてもらいたい。右辺を無限に計算することができるなら、その結果はぴったり円周率の値となる。しかし、これは計算機向きの式ではない。なぜなら**収束**が非常に遅いからだ。収束が遅いことは、計算機の処理速度がいかに速くても致命的なものである。

収束が遅いことは式を見ているだけでも理解できると思う。たとえば級数を 5 億項先まで加えても、分母は 10 億程度の大きさである。これでは小数点以下 8 桁の精度にしかならない。

しかし悲観ばかりしていても仕方ない。収束が遅いことを承知の上で、この方法で **Haskell** に計算させてみよう。その際、両辺を 4 倍して

$$\pi = 4 - \frac{4}{3} + \frac{4}{5} - \frac{4}{7} + \frac{4}{9} - \frac{4}{11} + \cdots$$

で計算していることに注意してもらいたい。

[2]ジェームス・グレゴリー（1638–1675）: イギリスの数学者・発明家。
[3]ゴットフリード・ウィルヘルム・ライプニッツ（1646–1716）: ドイツの哲学・数学者。

[hs script]
```
pivalue :: Integer -> Integer
pivalue 1 = 4*10^10
pivalue n
    | odd  n = pivalue (n-1) + (4*10^10 `div` (2*n-1))
    | even n = pivalue (n-1) - (4*10^10 `div` (2*n-1))
```

級数の計算は正直に分数の和を計算するのではなく、もちろん再帰を用いている。何度も言うことであるが、**Haskell** には再帰がよく似合う。

1 行目の `pivalue :: Integer -> Integer` と 2 行目の `pivalue 1 = 4*10^10` の記述は、e の値を求めたときと同じである。関数は整数を引数にとり、整数を返す。ただし、この級数で計算しても精度が高まるわけではないので 10 桁分の計算に止めている。それが初期値の `4*10^10` である。

さて、式を第 $n-1$ 項と第 n 項が分かるように書いてみると

$$\pi = 4 - \frac{4}{3} + \frac{4}{5} - \frac{4}{7} + \frac{4}{9} - \frac{4}{11} + \cdots \mp \frac{4}{2n-3} \pm \frac{4}{2n-1} + \cdots$$

であるから、第 n 項までの和を $f(n)$ とすると $f(n) = f(n-1) \pm \dfrac{4}{2n-1}$ であることが分かる。ただし、n 項目が $+$ か $-$ かが問題になる。第 n 項が奇数番目の項なら $+$ であり、偶数番目の項なら $-$ である。こういうときのプログラミングは `if`〜`else` で対処するのが常套手段だろうが、**Haskell** では**ガード**の機能を使う方が見やすいし、よく使われる記述なのでそうした。

ガードは “|” を用いて場合分けを記述する。`odd n` は `n` が奇数であることを調べる関数、`even n` は `n` が偶数であることを調べる関数である。そして、どちらの関数も真のときに `True` を返す。ガードは場合分けの記述を上の行から調べ、`True` のときにはその行に定義された処理を、`False` のときには次の行に移る。

式の作り方は e の値を求めたときと同様であるが、繰り返して説明すると、たとえば $n = 10$ が与えられると `n` は偶数なので、`pivalue 9 - 4*10^10 `div` 19` が計算される。しかし、`pivalue 9` が自分自身を呼び出して `pivalue 8` を計算することになる。これはいずれ `pivalue 1` へ行き着くので、そこまでに溜め込んだ計算を一気に行い π の計算とするのである。実際に 10 万項先までの和を求めると

(ghci env.)
```
*Main> pivalue 100000
31415826542
```

となる。

11 桁分の近似値が小数点を除いて示されたわけだが、`31415` までが正しい値であるから 3.1415 までは近似できたことになる。残念なことに、このスクリプトでは大きな桁数を求めるのは現実的

ではない。引数の値を大きくすれば精確さは増すが、その分、信じられないくらい処理に時間がかかるからだ。でも、処理が終わるまで我慢強く待っても無駄である。なぜなら、`stack overflow`で処理が中断するのがオチなのだから。