What makes a good program?

- Qualitative factors:
  - Correctness
  - Maintainability, readability, understandability, portability, flexibility, ...
  - Use of appropriate abstractions and idioms
  - ...
- Quantitative factors:
  - Performance, Predictability, ...
  - Time, Memory, Disk, Bandwidth, ...

Understanding Program Behavior:

- High-level languages abstract away from the underlying machine
- This can make it very difficult to understand what is happening when a program executes
- Analytic techniques can predict asymptotic trends
- Hard to model complexities of memory, timing, stack, cache, disk, buffers, network, latencies, bandwidth, concurrency, branch prediction, ...

Profiling Tools:

- Two broad approaches:
  - Instrumentation
  - Sampling
- Standard Advice:
  - Focus on writing qualitatively good code first
  - Once that’s working, use profiling tools to identify performance hot-spots and obtain quantitatively good code

Case Study: Profiling PPM

```haskell
close :: Double -> Double -> Double -> Bool
close epsilon x y = abs(x - y) <= epsilon

circle1 epsilon size x y =
  if close epsilon (x*x + y*y) size
  then red
  else yellow

go1 = mapDouble "circlePlain.ppm"
  (circle1 0.05 4) (-3,-3) (3,3) (420,420)
```
Making Circles in Hugs:

Main> main
^C{Interrupted!}
Main> :set +s +g
Main> main

(6164225 reductions, 8422432 cells, 9 garbage collections)

Making Circles with GHC:

prompt$ ./Main +RTS -sstderr

Bigger Circles with GHC:

prompt$ ./Main +RTS -sstderr

Preparing to Profile:

prompt$ ghc --make Main -prof -auto-all -fforce-recomp -o Main
prompt$ ./Main +RTS -sstderr -p

Inside Main.prof:

<table>
<thead>
<tr>
<th>COST CENTRE</th>
<th>MODULE</th>
<th>no. entries</th>
<th>%time</th>
<th>%alloc</th>
</tr>
</thead>
<tbody>
<tr>
<td>quant8</td>
<td>PPM6</td>
<td>1</td>
<td>47.2</td>
<td>62.1</td>
</tr>
<tr>
<td>lift</td>
<td>PPM6</td>
<td>35.8</td>
<td>38.1</td>
<td>29.3</td>
</tr>
<tr>
<td>cmap</td>
<td>PPM6</td>
<td>3.8</td>
<td>3.8</td>
<td>0.1</td>
</tr>
<tr>
<td>map</td>
<td>DemoPPM</td>
<td>3.8</td>
<td>3.8</td>
<td>0.0</td>
</tr>
<tr>
<td>g1</td>
<td>DemoPPM</td>
<td>1.9</td>
<td>1.9</td>
<td>0.0</td>
</tr>
<tr>
<td>clip</td>
<td>DemoPPM</td>
<td>1.9</td>
<td>1.9</td>
<td>0.0</td>
</tr>
<tr>
<td>close</td>
<td>DemoPPM</td>
<td>1.9</td>
<td>1.9</td>
<td>0.0</td>
</tr>
<tr>
<td>circle1</td>
<td>DemoPPM</td>
<td>1.9</td>
<td>1.9</td>
<td>0.0</td>
</tr>
<tr>
<td>iterDouble</td>
<td>PPM6</td>
<td>0.0</td>
<td>0.0</td>
<td>3.1</td>
</tr>
</tbody>
</table>

... continued:
quant8 and lift:

quant8 :: RealFrac n => n -> Word8
quant8 x = floor $ x * 0xFF

lift :: IOUArray Int Word8 -> (Double -> Double -> Colour) -> Double -> Double -> Int -> IO Int
lift arr f x y next =
  case cclip (f x y) of
    (Colour r g b) -> do writeArray arr next (quant8 r)
       writeArray arr (next+1) (quant8 g)
       writeArray arr (next+2) (quant8 b)
    return (next +3)

We run quant8 3 times for every pixel on the grid!

Inside the Colour library:

module Colour where

data Colour = Colour {redPart, greenPart, bluePart :: Double}
deriving (Eq, Show)

cmap :: (Double -> Double) -> Colour -> Colour
cmap f (Colour r g b) = Colour (f r) (f g) (f b)

clip :: (Num n, Ord n) => n -> n
clip n = max 0 (min 1 x)

cclip :: Colour -> Colour
cclip = cmap clip

black   = Colour 0 0 0
blue    = Colour 0 0 1
green   = Colour 0 0.5 0
...

Refactor the Colour library:

module Colour where

type Colour  = Color Double
data Color n = Color {redPart, greenPart, bluePart :: n } 
deriving (Eq, Show)

cmap :: (n -> n) -> Color n -> Color n
cmap f (Color r g b) = Color (f r) (f g) (f b)

clip :: (Num n, Ord n) => n -> n
clip n = max 0 (min 1 x)

cclip :: (Num n, Ord n) => Color n -> Color n
cclip = cmap clip

black, blue, green :: Colour
black   = Colour 0 0 0
blue    = Colour 0 0 1
green   = Colour 0 0.5 0
...

Update the definition of lift:

lift :: IOUArray Int Word8 -> (Double -> Double -> Color Word8) -> Double -> Double -> Int -> IO Int
lift arr f x y next =
  case cclip (f x y) of
    (Color r g b) -> do writeArray arr next r
       writeArray arr (next+1) g
       writeArray arr (next+2) b
    return (next +3)

Eliminates calls to quant8 ...

Adjust definition of circle1:

circle1 epsilon size x y =
  if close epsilon (x^2 + y^2) size
     then colquant red
     else colquant yellow

colquant :: Color Double -> Color Word8
colquant (Color r g b) = Color (quant8 r) (quant8 g) (quant8 b)

... which get moved here instead

Time to Run!

prompt$ ./Main +RTS -s -derr -p
4,724,860,788 bytes allocated in the heap
2,892,388 bytes copied during GC (scavenged)
1,241,516 bytes copied during GC (not scavenged)
3,153,920 bytes maximum residency (2 sample(s))
MUT    time    9.13s  (  9.23s elapsed)
GC      time    0.06s  (  0.09s elapsed)
Total   time    9.19s  (  9.32s elapsed)
prompt$

Disappointment: (slightly) worse than before 😞
... continued:

<table>
<thead>
<tr>
<th>COST CENTRE</th>
<th>MODULE</th>
<th>no.</th>
<th>entries</th>
<th>time</th>
<th>alloc</th>
<th>time</th>
<th>alloc</th>
</tr>
</thead>
<tbody>
<tr>
<td>MAIN</td>
<td>MAIN</td>
<td>1</td>
<td>0</td>
<td>0.0</td>
<td>0.0</td>
<td>100.0</td>
<td>100.0</td>
</tr>
<tr>
<td>CAF</td>
<td>Main</td>
<td>222</td>
<td>2</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>main</td>
<td>Main</td>
<td>228</td>
<td>1</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>gpl</td>
<td>DemoPPM</td>
<td>250</td>
<td>0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>mapDouble</td>
<td>PPM6</td>
<td>251</td>
<td>0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>CAF</td>
<td>DemoPPM</td>
<td>149</td>
<td>1</td>
<td>0.0</td>
<td>0.0</td>
<td>100.0</td>
<td>100.0</td>
</tr>
<tr>
<td>gpl</td>
<td>DemoPPM</td>
<td>229</td>
<td>1</td>
<td>0.0</td>
<td>0.0</td>
<td>100.0</td>
<td>100.0</td>
</tr>
</tbody>
</table>

circle1 epsilon size x y =
  if close epsilon (x*x + y*y) size
  then qred
  else qyellow

qred = colquant red -- CAFs
qyellow = colquant yellow -- (Constant Applicative Forms)
colquant :: Color Double -> Color Word8
colquant (Color r g b) = Color (quant8 r) (quant8 g) (quant8 b)

---

Re-adjust definition of circle1:

circle1 epsilon size x y =
  if close epsilon (x*x + y*y) size
  then qred
  else qyellow

qred = colquant red -- CAFs
qyellow = colquant yellow -- (Constant Applicative Forms)
colquant :: Color Double -> Color Word8
colquant (Color r g b) = Color (quant8 r) (quant8 g) (quant8 b)

---

Run Again ...

<table>
<thead>
<tr>
<th>COST CENTRE</th>
<th>MODULE</th>
<th>no.</th>
<th>entries</th>
<th>time</th>
<th>alloc</th>
<th>time</th>
<th>alloc</th>
</tr>
</thead>
<tbody>
<tr>
<td>MAIN</td>
<td>MAIN</td>
<td>1</td>
<td>0</td>
<td>0.0</td>
<td>0.0</td>
<td>100.0</td>
<td>100.0</td>
</tr>
<tr>
<td>CAF</td>
<td>Main</td>
<td>222</td>
<td>2</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>main</td>
<td>Main</td>
<td>228</td>
<td>1</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>gpl</td>
<td>DemoPPM</td>
<td>250</td>
<td>0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>mapDouble</td>
<td>PPM6</td>
<td>251</td>
<td>0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>CAF</td>
<td>DemoPPM</td>
<td>149</td>
<td>3</td>
<td>0.0</td>
<td>0.0</td>
<td>100.0</td>
<td>100.0</td>
</tr>
<tr>
<td>gpl</td>
<td>DemoPPM</td>
<td>229</td>
<td>1</td>
<td>0.0</td>
<td>0.0</td>
<td>100.0</td>
<td>100.0</td>
</tr>
<tr>
<td>circle1</td>
<td>DemoPPM</td>
<td>239</td>
<td>786432</td>
<td>0.0</td>
<td>6.0</td>
<td>0.0</td>
<td>9.8</td>
</tr>
<tr>
<td>close</td>
<td>DemoPPM</td>
<td>240</td>
<td>786432</td>
<td>0.0</td>
<td>3.8</td>
<td>0.0</td>
<td>3.8</td>
</tr>
<tr>
<td>mapDouble</td>
<td>PPM6</td>
<td>230</td>
<td>1</td>
<td>0.0</td>
<td>0.0</td>
<td>100.0</td>
<td>90.2</td>
</tr>
<tr>
<td>lift</td>
<td>PPM6</td>
<td>236</td>
<td>786432</td>
<td>38.7</td>
<td>28.1</td>
<td>38.7</td>
<td>28.9</td>
</tr>
</tbody>
</table>
cclip Colour | 247      | 0    | 0.0     | 0.0  | 0.0   | 0.0  | 0.0   |
cmap Colour  | 249      | 0    | 0.0     | 0.0  | 0.0   | 0.0  | 0.0   |
cmap Colour  | 244      | 0    | 0.0     | 0.0  | 0.0   | 0.0  | 0.0   |
cmap Colour  | 248      | 0    | 0.0     | 0.0  | 0.0   | 0.0  | 0.0   |
cmap Colour  | 245      | 0    | 0.0     | 0.0  | 0.0   | 0.0  | 0.0   |
cmap Colour  | 235      | 1    | 3.2  | 3.1  | 3.2  | 3.1  |
| new         | PPM6     | 231 | 1       | 6.5  | 0.1   | 6.5  | 0.1   |

---

Another look at lift:

lift :: IOUArray Int Word8 -> (Double -> Double -> Color Word8)
  -> Double -> Double -> Int -> IO Int
lift arr f x y next =
case cclip (f x y) of
  (Color r g b) -> do writeArray arr next r
  writeArray arr (next+1) g
  writeArray arr (next+2) b
  return (next+3)

---

Another look at lift:

lift :: IOUArray Int Word8 -> (Double -> Double -> Color Word8)
  -> Double -> Double -> Int -> IO Int
lift arr f x y next =
case cclip (f x y) of
  (Color r g b) -> do writeArray arr next r
  writeArray arr (next+1) g
  writeArray arr (next+2) b
  return (next+3)
Not much of a circle ...

Oops, clipping broke the program!

Eliminate Clipping:

```
lift :: IOUArray Int Word8 -> (Double -> Double -> Color Word8)
    -> Double -> Double -> Int -> IO Int
lift arr f x y next =
    case (f x y) of
        (Color r g b) -> do writeArray arr next     r
                            writeArray arr (next+1) g
                            writeArray arr (next+2) b
                            return (next +3)
```

Now what happens to performance?

Another Run:

```
prompt$ ./Main +RTS -sstderr -p
1,805,257,208 bytes allocated in the heap
467,876 bytes copied during GC (scavenged)
232,576 bytes copied during GC (not scavenged)
3,153,920 bytes maximum residency (2 sample(s))
MUT time 3.01s ( 3.20s elapsed)
GC time 0.02s ( 0.03s elapsed)
Total time 3.03s ( 3.23s elapsed)
prompt$
```

improvements, but modest

Adding Cost Centers:

```
lift :: IOUArray Int Word8 -> (Double -> Double -> Color Word8)
    -> Double -> Double -> Int -> IO Int
lift arr f x y next =
    case (f x y) of
        (Color r g b) -> do writeArray arr next     r
                            writeArray arr (next+1) g
                            writeArray arr (next+2) b
                            return (next +3)
```

Adding More Cost Centers:

```
lift :: IOUArray Int Word8 -> (Double -> Double -> Color Word8)
    -> Double -> Double -> Int -> IO Int
lift arr f x y next =
    case (f x y) of
        (Color r g b) -> do writeArray arr next     r
                             writeArray arr (next+1) g
                             writeArray arr (next+2) b
                             return (next +3)
```

More Profiling Data:

```

<table>
<thead>
<tr>
<th>COST CENTRE</th>
<th>MODULE</th>
<th>no.</th>
<th>entries</th>
<th>individual time</th>
<th>allocated time</th>
</tr>
</thead>
<tbody>
<tr>
<td>MAIN</td>
<td>MAIN</td>
<td>1</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>CAF</td>
<td>Main</td>
<td>222</td>
<td>2.0</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>main</td>
<td>Main</td>
<td>228</td>
<td>1.0</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>gil</td>
<td>DemoPPM 235</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>mapDouble</td>
<td>PPM6</td>
<td>254</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>CAF</td>
<td>DemoPPM 149</td>
<td>3.0</td>
<td>0.0</td>
<td>100.0</td>
<td>100.0</td>
</tr>
<tr>
<td>gil</td>
<td>DemoPPM 229</td>
<td>1.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>circle1</td>
<td>DemoPPM 238</td>
<td>786432</td>
<td>0.0</td>
<td>6.4</td>
<td>0.8</td>
</tr>
<tr>
<td>close</td>
<td>DemoPPM 235</td>
<td>786432</td>
<td>9.8</td>
<td>4.1</td>
<td>3.9</td>
</tr>
<tr>
<td>mapDouble</td>
<td>PPM6</td>
<td>250</td>
<td>1.0</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>lift</td>
<td>PPM6</td>
<td>236</td>
<td>786432</td>
<td>0.0</td>
<td>80.5</td>
</tr>
<tr>
<td>act</td>
<td>PPM6</td>
<td>243</td>
<td>786432</td>
<td>80.5</td>
<td>80.2</td>
</tr>
<tr>
<td>iterDouble</td>
<td>PPM6</td>
<td>235</td>
<td>4.9</td>
<td>4.9</td>
<td>4.9</td>
</tr>
<tr>
<td>new</td>
<td>PPM6</td>
<td>231</td>
<td>4.9</td>
<td>4.9</td>
<td>4.9</td>
</tr>
</tbody>
</table>
```
## Even More Profiling Data:

<table>
<thead>
<tr>
<th>COST CENTRE</th>
<th>MODULE</th>
<th>no. entries</th>
<th>time</th>
<th>alloc</th>
<th>time</th>
<th>alloc</th>
</tr>
</thead>
<tbody>
<tr>
<td>MAIN</td>
<td>MAIN</td>
<td>1</td>
<td>0.0</td>
<td>0.0</td>
<td>100.0</td>
<td>100.0</td>
</tr>
<tr>
<td>CAF</td>
<td>Main</td>
<td>222</td>
<td>2</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>main</td>
<td>Main</td>
<td>228</td>
<td>1</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>gpl</td>
<td>DemoPPM</td>
<td>256</td>
<td>0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>mapDouble</td>
<td>PPM6</td>
<td>257</td>
<td>0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>CAF</td>
<td>DemoPPM</td>
<td>149</td>
<td>3</td>
<td>0.0</td>
<td>0.0</td>
<td>100.0</td>
</tr>
<tr>
<td>gpl</td>
<td>DemoPPM</td>
<td>229</td>
<td>1</td>
<td>0.0</td>
<td>0.0</td>
<td>100.0</td>
</tr>
<tr>
<td>circle1</td>
<td>DemoPPM</td>
<td>238</td>
<td>786432</td>
<td>0.0</td>
<td>6.4</td>
<td>10.0</td>
</tr>
<tr>
<td>close</td>
<td>DemoPPM</td>
<td>239</td>
<td>786432</td>
<td>10.0</td>
<td>10.0</td>
<td>10.0</td>
</tr>
<tr>
<td>lift</td>
<td>PPM6</td>
<td>236</td>
<td>786432</td>
<td>0.0</td>
<td>60.0</td>
<td>80.2</td>
</tr>
<tr>
<td>act</td>
<td>PPM6</td>
<td>243</td>
<td>786432</td>
<td>10.1</td>
<td>60.0</td>
<td>80.2</td>
</tr>
<tr>
<td>act1</td>
<td>PPM6</td>
<td>247</td>
<td>786432</td>
<td>23.9</td>
<td>20.0</td>
<td>22.0</td>
</tr>
<tr>
<td>act2</td>
<td>PPM6</td>
<td>244</td>
<td>786432</td>
<td>22.2</td>
<td>20.0</td>
<td>22.0</td>
</tr>
<tr>
<td>act</td>
<td>PPM6</td>
<td>237</td>
<td>786432</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>new</td>
<td>PPM6</td>
<td>235</td>
<td>1</td>
<td>0.0</td>
<td>9.1</td>
<td>9.1</td>
</tr>
</tbody>
</table>

And on we (could) go ...

## Running without Profiling:

```bash
prompt$ ghc --make -fforce-recomp Main
prompt$ ./Main +RTS -sstderr
```

- 1,222,949,592 bytes allocated in the heap
- 217,640 bytes copied during GC (scavenged)
- 91,700 bytes copied during GC (not scavenged)
- 3,153,920 bytes maximum residency (2 sample(s))

- MUT time 1.96s (1.99s elapsed)
- GC time 0.01s (0.02s elapsed)
- Total time 1.97s (2.01s elapsed)

## Form Follows Function:

```haskell
expr, term, atom :: Parser Int

expr = do l <- term; string "+"; r <- expr; return (l+r)
| | | do l <- term; string ";"; r <- expr; return (l-r)
| || term

term = do l <- atom; string "*"; r <- term; return (l*r)
| | | do l <- atom; string "/"; r <- term; return (l\`div`r)
| ||| atom

atom = do string ")"; x <- atom; return (negate x)
| | | do string ";n <- expr; string ");n return n
| || number
```

## Case Study: Profiling a Parser

Form Follows Function:

```haskell
data Parser a = P { applyP :: String -> [(a, String)] }

parse :: Parser a -> String -> [a]
parse p s = 
  v = (applyP (applyP p s) s)

instance Monad Parser where
  return x = P [(x, s)]
  P p >>= f = P [(y, s) | (x, s1) <- applyP p s, (y, s2) <- applyP (f x) s1 ]

(|||) :: Parser a -> Parser a -> Parser a
p ||| q = Parser \s -> applyP p s ++ applyP q s

string :: String -> Parser String
string "" = return ""
string (c:cs) = ...
```

The Parser Monad:
Parsing Examples:

```
Parsing> parse expr "1+2"
[3]
Parsing> parse expr "(1+2) * 3"
[1]
Parsing> parse expr "((1+2)*3)+1"
[10]
Parsing> parse expr "(((1+2)*3)+1)*8"
[80]
```

Execution Statistics in Hugs:

- **Mechanisms:**
  - Enable the collection of execution statistics using `:set +s`
  - Turn on messages when garbage collection occurs using `:set +g`
  - Change total heap size (when loading Hugs) using `hugs -hSize`

- **Measures:**
  - **Cells:** a chunk of memory
  - **Reductions:** a single rewrite step

Collecting Statistics:

```
Parsing> :set +s
1
(22 reductions, 32 cells)
```

Observing Garbage Collection:

```
$ hugs -h100000 +gs
Hugs> length [1..200000]
(4200054 reductions, 5598125 cells, 64 garbage collections)
{{Gc:868666}}Hugs> :q
```

```
$ hugs -h26378
ERROR "/Users/user/local/lib/hugs/packages/hugsbase/Hugs/Prelude.hs"
- Garbage collection fails to reclaim sufficient space
FATAL ERROR: Unable to load Prelude
```

```
$ hugs -h26379
... Hugs> :set +sg
Hugs> length [1..200000]
(4200054 reductions, 5598125 cells, 424 garbage collections)
{{Gc:132450}}Hugs>:
```
Observations:

- Note that: $100000 - 86866 = 13134 = 26379 - 13245$
- So we can conclude that Hugs:
  - uses 13134 cells for internal state
  - needs at least 26379 cells to load
- Possible profile of memory usage during startup:

![Memory Profile Graph]

Heap size, Residency, Allocation:

- **Heap size** measures maximum capacity
- **Residency** measures amount of memory that is actually in use at any given time
- Haskell programs allocate constantly (and, simultaneously, create garbage)
- **Total allocation** may exceed heap size

Back to Parsing:

Parentheses seem to be part of the problem, so let's stress test:

```haskell
addParens n s = if n == 0
  then s
  else "(" ++ addParens (n-1) s ++ ")"
```

```
Parsing> [ addParens n "1" | n <-[0..5] ]
["1","(1)","((1))","(((1)))","((((1))))","((((((1)))))")
Parsing>
```

Analysis (1):

<table>
<thead>
<tr>
<th>Parentheses</th>
<th>Reductions</th>
<th>Cells</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>15060</td>
<td>20628</td>
</tr>
<tr>
<td>2</td>
<td>137062</td>
<td>187767</td>
</tr>
<tr>
<td>3</td>
<td>1234954</td>
<td>1691736</td>
</tr>
<tr>
<td>4</td>
<td>11115840</td>
<td>15227127</td>
</tr>
<tr>
<td>5</td>
<td>100043656</td>
<td>137045268</td>
</tr>
</tbody>
</table>

Memory is not the problem here:
Why Exponential Behavior?

expr, term, atom :: Parser Int

Recall this grammar ...

```
expr  = do l <- term; string "+"; r <- expr; return (l+r) ||| do l <- term; string "-"; r <- expr; return (l-r) ||| term

term  = do l <- atom; string "+"; r <- term; return (l+r) ||| do l <- atom; string "-"; r <- term; return (l-r) ||| atom

atom  = do string "+"; x <- atom; return (negate x) ||| do string "-"; x <- atom; return (x) ||| number
```

A Step Forward:

```
Parsing> :set +s
Parsing> parse expr (addParens 10 "1")
[1]
(3624 reductions, 6091 cells)
Parsing> parse expr (addParens 100 "1")
[1]
(42414 reductions, 83491 cells)
Parsing> parse expr (addParens 1000 "1")
[1]
(1321314 reductions, 3530491 cells, 3 garbage collections)
Parsing> parse expr (addParens 10000 "1")
(3899701 reductions, 11445375 cells, 12 garbage collections)
ERROR - Control stack overflow
Parsing>
```
$ ./altParsing +RTS -p -stderr
[1]
848,494,732 bytes allocated in the heap
1,556,284 bytes copied during GC (not scavenged)
0 bytes copied during GC (not scavenged)
24,576 bytes maximum residency (1 sample(s))
1619 collections in generation 0 { 0.02s}
1 collections in generation 1 { 0.00s}
1 Mb total memory in use
INIT time 0.00s ( 0.00s elapsed)
MUT time 1.01s ( 1.03s elapsed)
GC time 0.02s ( 0.02s elapsed)
EXIT time 0.00s ( 0.00s elapsed)
Total time 1.03s ( 1.06s elapsed)
%GC time 1.7% (2.3% elapsed)
Alloc rate 836,673,373 bytes per MUT second
Productivity 98.2% of total user, 96.0% of total elapsed

$ Profiling Options:
- For more serious work, compile with the –prof flag
  
  
  ghc --make -prof altParsing.lhs

  
  Opens up possibilities for:
  - Time and allocation profiling
  - Memory profiling
  - Coverage Profiling
  - ...

  
  Profiling code has overheads; not for production use

Cost Center Profiling:
- A technique for distributing costs during program execution
- Programmer creates “cost centers”: 
  - by hand ({-# SCC “name” #-})
  - for all top-level functions: -auto-all
- Program maintains runtime stack of cost centers
- RTS samples behavior at regular intervals
- Produce a summary report of statistics at the end of execution

In Practice:
$ ghc --make -prof -auto-all altParsing.lhs
$ ./altParsing +RTS -p
[1]
$ ls
altParsing* altParsing.hi altParsing.lhs
altParsing.o altParsing.prof
$

Time and Allocation Profiling Report (Final)
altParsing +RTS -p -RTS

  total time = 0.54 secs (27 ticks @ 20 ms)
  total alloc = 803,275,236 bytes (excludes profiling overheads)

  COST CENTRE MODULE     %time %alloc
    CAF                    Main         100.0  100.0

  COST CENTRE MODULE no. individual inherited
    CAF Main 194        19 100.0 100.0 100.0 100.0
    CAF GHC.Handle 92     4  0.0  0.0  0.0  0.0

  Alas, not a very insightful report,
  in this case ...

Heap Profiling:
- A technique for measuring heap usage during program execution
- Compile code for profiling and run with argument
  +RTS option where option is:
  -hc by function
  -hm by module
  -hy by type
  -hb by thunk behavior

- Generates output.hp text file
- Produce a graphical version using hp2ps utility
In Practice:

$ ghc --make -prof altParsing.lhs
$ ./altParsing +RTS -hc

$ ls
altParsing* altParsing.hi altParsing.lhs
altParsing.o altParsing.hp
$ hp2ps --c altParsing.hp
$ open altParsing.ps
$
Coverage Profiling:

- Used to determine which parts of a program have been exercised during any given run
- Works by instrumenting code to get exact results
- Provides two kinds of coverage:
  - Source coverage
    - Yellow – not executed
  - Boolean guard coverage
    - Green always true
    - Red always false

In Practice:

$ ghc --make --fhpc altParsing.lhs
$ ./altParsing

In Practice:

$ hpc report altParsing
33% expressions used (138/409)
0% boolean coverage (0/1)
100% guards (0/0)
0% 'if' conditions (0/1), 1 unevaluated
100% qualifiers (0/0)
66% alternatives used (4/6)
0% local declarations used (0/6)
54% top-level declarations used (18/33)

Coverage of altParser:

Coverage of altParser:

Summary:

- Profiling tools help us to understand the complex operational behavior of code
- Expert use of profiling tools requires significant use and experience
- But, even with limited experience, it is still possible to gain some interesting into what our programs really do!