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, ...

Form Follows Function:

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

expr = term "+" expr
  | term ":" expr
  | term
  -- return (l+r)

term = atom "*" term
  | atom "/" term
  | atom
  -- return (l*r)

atom = ":" atom
  | "(" expr ")"
  | number
  -- return (negate x)

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 ";"; return n
  || number
```
Parsing Examples:

```hugs
Parsing> parse expr "1+2"
[3]
Parsing> parse expr "(1+2) * 3"
[1]
Parsing> parse expr "((1+2)*3)+1"
[9]
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:

```hugs
Parsing> :set +s
Parsing> length "hello"
5
  (56 reductions, 75 cells)
Parsing> length "world"
5
  (56 reductions, 75 cells)
Parsing> id 1
1
  (22 reductions, 32 cells)
Parsing> (\x -> x) 1
1
  (23 reductions, 32 cells)
Parsing> (\x -> x) 1
1
  (23 reductions, 32 cells)
Parsing> (\x -> x) 1
1
  (23 reductions, 32 cells)
```

Observing Garbage Collection:

```hugs
$ hugs -h100000 +gs
Hugs> length [1..200000]
{(Gc:86831)}{(Gc:86830)}{(Gc:86832)}{(Gc:86833)}{(Gc:86828)}...
{(Gc:86828)}{(Gc:86829)}{(Gc:86828)}{(Gc:86828)} 200000
(4200054 reductions, 5598125 cells, 64 garbage collections)
{(Gc:86866)}Hugs> :q
```

```hugs
$ hugs -h8M +gs
... Hugs> length [1..200000]
200000
(4200054 reductions, 5598125 cells)
{(Gc:7986866)}Hugs> :q
```

```hugs
$ 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]
{(Gc:13208)}{(Gc:13213)}{(Gc:13208)}{(Gc:13205)}{(Gc:13209)}...
{(Gc:13203)}{(Gc:13209)}{(Gc:13208)} 200000
(4200054 reductions, 5598125 cells, 424 garbage collections)
{(Gc:13245)}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)))))")
```

Rapid increases in reductions and cell counts

```haskell
Parsing> :set +s
Parsing> parse expr (addParens 1 "1")
[1]
(15060 reductions, 20628 cells)
Parsing> parse expr (addParens 2 "1")
[1]
(137062 reductions, 187767 cells)
Parsing> parse expr (addParens 3 "1")
[1]
(123494 reductions, 1691736 cells, 1 garbage collection)
Parsing> parse expr (addParens 4 "1")
[1]
(11115840 reductions, 15227127 cells, 15 garbage collections)
Parsing> parse expr (addParens 5 "1")
[1]
(100043656 reductions, 137045268 cells, 139 garbage collections)
```

Memory is not the problem here:

```haskell
$ hugs -h26379 +sg
Hugs> :alt Parsing.lhs
Parsing> :gc
Garbage collection recovered 6462 cells
Parsing> parse expr "1"
[1]
(1367 reductions, 1881 cells)
```

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>123494</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>
Why Exponential Behavior?

<table>
<thead>
<tr>
<th>expr, term, atom :: Parser Int</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

Recall this grammar ...

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 "+"; return n
|| number

Why Exponential Behavior?

Recall this grammar ...

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 "+"; return n
|| number

Why Exponential Behavior?

Recall this grammar ...

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 "+"; return n
|| number

Why Exponential Behavior?

Recall this grammar ...

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 "+"; return n
|| number

Why Exponential Behavior?

Recall this grammar ...

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 "+"; return n
|| number

Why Exponential Behavior?

Recall this grammar ...

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 "+"; return n
|| number

Profiling with GHC:

GHC provides a much broader and more powerful range of profiling tools than Hugs

We have to identify a main program:

module Main where

main = print (parse expr "((((((1)))))"))

Compiling: ghc --make altParsing.lhs

Running: ./altParsing +RTS –stderr

Still slow!
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
$

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
$ ./alt Parsing +RTS -hc
$ ls
altParsing* altParsing.hi altParsing.lhs
altParsing.o altParsing.hp
$ hp2ps –c altParsing.hp
$ open altParsing.ps
$

Biographical Profiling (-hb):

- LAG phase: object created but not yet used
- USE: objects is in use
- DRAG: object has been used for the last time, but is still referenced
- VOID: an object is never used
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 --fphc altParsing.lhs
$ ./altParsing
[1]
$ ls
altParsing* altParsing.hi altParsing.lhs
alt Parsing.o alt Parsing.tix
```

```
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!