This note is meant to give people who are familiar with the functional programming language
Haskell98 a concise overview of Clean language elements. The goal is to support the reader when
reading Clean code. In the table on the other side of this page frequently occurring Clean language
elements are summarized, and their Haskell98 counterpart is given next to it.

Obviously, this summary is not exhaustive. Notable Clean language elements that also occur
frequently in Clean programs, but that do not appear in this summary are:

**Strictness** By default, Clean evaluates expressions lazily. Types can be annotated with strictness
attributes (!) to indicate that they occur within a strict context. This is similar to Haskell98.
Within a function definition, #! can be used to enforce evaluation of expressions. Haskell98
programmers will likely use the `seq` function.

**The uniqueness type system** Briefly, types and type variables can be annotated with a uniqueness
attribute. This attribute can be *, which indicates that the type must occur within a unique context, and u:; which is a uniqueness attribute variable, which can be instantiated
with * or not. The uniqueness type system allows Clean to use the world-as-value paradigm
for side-effective programming: an interactive program is of type `*World -> *World`, where
`World` represent the external environment of the program of which there can be only one, hence
its uniqueness attribute. In Haskell98, such a program would have type `IO ()`. The uniqueness
type system also allows the Clean compiler to generate efficient code because uniquely
attributed data structures can be destructively updated.

**Generic programming** With generic programming, the programmer defines a small set of instances
of a generic function scheme that are used by the Clean compiler to derive for arbitrary data types
the corresponding instance. Generic programming in Clean resembles generic programming in Generic Hskeil.

**Dynamic types** Dynamic types allow the programmer to serialize and de-serialize arbitrary expressions (including functions), resulting in a new value of type Dynamic. Dynamic values
 can be stored on disk.

I hope you enjoy this note and that it will aid you in reading Clean programs.
Clean for Haskell98 Programmers

-- Quick Reference Guide --

Clean

<table>
<thead>
<tr>
<th>Haskell98</th>
</tr>
</thead>
<tbody>
<tr>
<td>(True,False) :: (Bool,Bool)</td>
</tr>
<tr>
<td>42 :: Int</td>
</tr>
<tr>
<td>3.1415926 :: Real</td>
</tr>
<tr>
<td>'A' :: Char</td>
</tr>
<tr>
<td>&quot;Hello&quot; :: String</td>
</tr>
</tbody>
</table>

Basic types

Type definitions

:: T a,...a, := t
data T a,...a = C t1,...,tn
:: T a,...a = { f1 :: t1, ..., fn :: tn }
data T a,...a = T { f1 :: t1, ..., fn :: tn }

Abstract data type definitions

definition module M
:: T a,...a // no implementation module M(T)

Function types

f :: t1 ... tn_tn -> t
class C t | Cm a where f :: t
instance C t | Cm a where ...

Type classes

class C a where f :: t
instance C a where ...

as-patterns

x := p

λ-expressions

\pi ... \pi \pi -> e or \pi ... \pi \pi \pi -> e or \pi ... \pi \pi -> e

Distinction of cases

if c then t else e

List expressions

[1;[2;[3]]] :: [Int]
[\pi pi \pi pi p <~ gi ]
[\pi pi \pi pi pi <~ gi & p ]
[\pi pi \pi pi pi <~ gi & p2 <~ gi2 ]
[\pi pi \pi pi pi <~ gi & p2 <~ gi2 & p <~ gi2 ]

Record expressions

:: R = { f :: t }
data R = R { f :: t }
r.f
r.f
{ r & f = e }
{ r & f = e }

Record patterns

:: R1 = { f1 :: R2 }
data R1 = R1 { f1 :: R2 }
:: R2 = { f2 :: t }
data R2 = R2 { f2 :: t }
g1 { f1 } = e f1
g2 { f1 } = e f2
g1 { f1 } = e f1
g2 { f1 } = e f2

Array expressions

:: A ::= [t]
type A = Array Int t
a = \pi pi p \pi pi a (see list comprehensions)
[0,v0),...,(n-1,vn-1)]
a = array (0,length a-1)
\pi pi (i,a) \pi pi zipWith (,) [0..length a-1] a
a!i
a!i
[a & [i]=e]

Comments

// single line comment
*/ multi-line, */ nested, */ comment */

-- single line comment

Function definitions

f p1

# q1 = e1

= e

f p1

= e[x := x']

where q1[x := x'] = e1 -- for each x e var(q1) \ var(e1)