Cryptol Crib Sheet

1 To Use Cryptol:

1. From the linux command line: prompt> cryptol to get this:

    Cryptol version 1.8.4, Copyright (C) 2004-2008 Galois, Inc.
    www.cryptol.net

    Type :? for help
    Cryptol>

2. To load a source file, in this case tests.cry, do this:

    Cryptol> :l tests.cry
    Loading "tests.cry".. Checking types.. Processing.. Done!
    tests>

3. To set the base to 10 do this:

    tests> :s base=10

4. To invoke a function, in this case rev(..) (reverse a list) do this:

    tests> rev([1 2 3])
    [3 2 1]
    tests>

5. To enter symbolic mode (for proving assertions) and be verbose do this:

    tests> :s symbolic
    tests> :s +v

6. To switch to a different backend, in this case yices, do this:

    tests> :s yices
    :set sbv
    :set sbv_solver=yices
    tests>

7. All source files are edited using any simple text editor. The following examples are assumed to be written to file then loaded as above. Where prompt> shows up, the functions in the file are invoked.

2 Data Structures

Some variables:

<table>
<thead>
<tr>
<th>cryptol</th>
<th>c</th>
</tr>
</thead>
<tbody>
<tr>
<td>v:[32];</td>
<td>unsigned int v = 45;</td>
</tr>
<tr>
<td>v = 45;</td>
<td></td>
</tr>
<tr>
<td>x:[64];</td>
<td>unsigned long x = 72625;</td>
</tr>
<tr>
<td>x = 72625;</td>
<td></td>
</tr>
</tbody>
</table>

1These are notes collected from experiments by John Franco and do not necessarily represent solutions as they would be coded by experts in cryptol.
Arrays:

```plaintext
cryptol
c

<table>
<thead>
<tr>
<th>v: [8][32];</th>
<th>unsigned int v[] = {1, 2, 3, 4, 5, 6, 7, 8};</th>
</tr>
</thead>
<tbody>
<tr>
<td>v = [1 2 3 4 5 6 7 8];</td>
<td>for (int i=0 ; i &lt; 8 ; i++) cout &lt;&lt; v[i] &lt;&lt; &quot; &quot;;</td>
</tr>
<tr>
<td>prompt&gt; v</td>
<td>printf(&quot;%d&quot;,v[1]);</td>
</tr>
<tr>
<td>[1 2 3 4 5 6 7 8]</td>
<td></td>
</tr>
<tr>
<td>v = [1..8];</td>
<td></td>
</tr>
<tr>
<td>prompt&gt; v</td>
<td></td>
</tr>
<tr>
<td>[1 2 3 4 5 6 7 8]</td>
<td></td>
</tr>
<tr>
<td>prompt&gt; v@1;</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
</tr>
<tr>
<td>x: [8];</td>
<td></td>
</tr>
<tr>
<td>x = 23;</td>
<td></td>
</tr>
<tr>
<td>prompt&gt; :s base=2</td>
<td></td>
</tr>
<tr>
<td>prompt&gt; x</td>
<td></td>
</tr>
<tr>
<td>0b00010111</td>
<td></td>
</tr>
<tr>
<td>prompt&gt; x@3</td>
<td></td>
</tr>
<tr>
<td>False</td>
<td></td>
</tr>
<tr>
<td>prompt&gt; x@4</td>
<td></td>
</tr>
<tr>
<td>True</td>
<td></td>
</tr>
<tr>
<td>prompt&gt; x@6</td>
<td></td>
</tr>
<tr>
<td>False</td>
<td></td>
</tr>
<tr>
<td>prompt&gt; x@@[0..3]</td>
<td></td>
</tr>
<tr>
<td>0b0111</td>
<td></td>
</tr>
</tbody>
</table>

Operations:

```plaintext
cryptol
c

<table>
<thead>
<tr>
<th>x: [8];</th>
<th>unsigned char x = 23;</th>
</tr>
</thead>
<tbody>
<tr>
<td>x = 23;</td>
<td>unsigned char y = 65;</td>
</tr>
<tr>
<td>y: [8];</td>
<td></td>
</tr>
<tr>
<td>y = 65;</td>
<td></td>
</tr>
<tr>
<td>prompt&gt; :s base=2</td>
<td></td>
</tr>
<tr>
<td>prompt&gt; x+y</td>
<td></td>
</tr>
<tr>
<td>0b001011000</td>
<td></td>
</tr>
<tr>
<td>prompt&gt; x-y</td>
<td></td>
</tr>
<tr>
<td>0b11010110</td>
<td></td>
</tr>
<tr>
<td>prompt&gt; x*y</td>
<td></td>
</tr>
<tr>
<td>0b11010111</td>
<td></td>
</tr>
<tr>
<td>prompt&gt; y/x</td>
<td></td>
</tr>
<tr>
<td>0b00000010</td>
<td></td>
</tr>
<tr>
<td>prompt&gt; y%x</td>
<td></td>
</tr>
<tr>
<td>0b00010011</td>
<td></td>
</tr>
<tr>
<td>prompt&gt; y&amp;x</td>
<td></td>
</tr>
<tr>
<td>0b00000001</td>
<td></td>
</tr>
<tr>
<td>prompt&gt; y</td>
<td>x</td>
</tr>
<tr>
<td>0b01010111</td>
<td></td>
</tr>
<tr>
<td>prompt&gt; x&gt;&gt;1</td>
<td></td>
</tr>
<tr>
<td>0b01010111</td>
<td></td>
</tr>
<tr>
<td>prompt&gt; x&lt;&lt;1</td>
<td></td>
</tr>
<tr>
<td>0b00101110</td>
<td></td>
</tr>
</tbody>
</table>

x: [8][32];                  |
| x = [2 3 6 4 3 2 7 8];      |
| y: [8][32];                |
| y = [1 8 3 4 2 1 1 9];      |
| prompt> x+y                |
| [3 11 9 8 5 3 8 17]         |
Function examples:

Simple for loop, using [1..11..1] operator. Type of function is inferred by cryptol to be

\[ \text{sf:} \{ \text{a b} \} (\text{fin a}, \text{b} \geq 2) \Rightarrow [\text{a}] [\text{b}] \rightarrow [\text{a}][\text{b}] \]  
(map any finite list of a numbers of width b \geq 2 to a list of the same type). Symbol @ allows array indexing.

\[
\begin{align*}
\text{cryptol} & \quad \text{c} \\
p: [6][8]; & \quad \text{typedef unsigned char u_int_8} \\
p = [1 \ 2 \ 3 \ 4 \ 5 \ 6]; & \quad u_int_8 *sf(u_int_8 z[], \text{int sz}) \{ \\
sf(z) = [1 \ 2*(z@i)] & \quad \text{u_int_8 *q = new u_int_8[sz];} \\
\mid i <- [0..\text{width}(z)-1] \mid; & \quad \text{for (int i=0; i < 6; i++)} \\
prompt> sf(p) & \quad q[i] = 2*z[i]; \\
[2 \ 4 \ 6 \ 8 \ 10 \ 12] & \quad \text{return q;} \\
\end{align*}
\]

Reverse the elements of an array. Construct \texttt{where} is used to establish a value \texttt{ln} that is used twice in \texttt{rev}. Inferred type is \texttt{rev:} \{ \texttt{a b} \} (\texttt{fin a}) \Rightarrow [\texttt{a}] [\texttt{b}] \rightarrow [\texttt{a}][\texttt{b}]. Observe that \texttt{b} does not have to be a number. Note: there is a built-in function called \texttt{reverse} with the same functionality as \texttt{rev}.

\[
\begin{align*}
\text{cryptol} & \quad \text{c} \\
\texttt{rev (x) =} & \quad \texttt{void **rev(void *x[], \text{int sz}) \{} \\
\mid (x@(\texttt{ln}-i)) & \quad \texttt{void **q = new void*}[\texttt{sz}]; \\
\mid i <- [0..\texttt{ln}] \mid; & \quad \texttt{for (int i=\texttt{sz}-1; i >= 0; i--)} \\
\texttt{where} & \quad \texttt{q[\texttt{sz}-i-1] = x[i];} \\
\{ & \quad \texttt{return q;} \\
\texttt{ln = width(x)-1;} & \quad \texttt{void **p = new void*}[\texttt{sz}]; \\
prompt> rev(p) & \quad \texttt{for (int i=0; i < \texttt{sz}; i++) p[i] = new u_int_8(i);} \\
[6 \ 5 \ 4 \ 3 \ 2 \ 1] & \quad \texttt{u_int_8 **s = (u_int_8**)rev(p,6);} \\
\end{align*}
\]

Membership in a list. The function’s input type is explicitly stated to be any length integer list of at least one 32 bit number and a 32 bit number. The output is a single \texttt{Bit} which is \texttt{True} if \texttt{n} is a member of list \texttt{x}. A list \texttt{s} is initialized to \texttt{False}. As \texttt{n} is tested against elements of \texttt{x} either \texttt{True} or \texttt{False} is appended to \texttt{s}. If \texttt{True} is appended, all following elements of \texttt{s} will be \texttt{True}. The last element of \texttt{s} is the output. This is a simple example using concatenation (\#) and if-then-else.

\[
\begin{align*}
\text{cryptol} & \quad \text{c} \\
\text{member:} \{ \texttt{a} \} (\texttt{fin a}, \texttt{a} \geq 1) \Rightarrow & \quad \texttt{typedef unsigned int u_int_32;} \\
\texttt{([a][32],[32])} \rightarrow \texttt{Bit;} & \quad \texttt{bool member(u_int_32 x[], u_int_32 n, \text{int sz}) \{} \\
\texttt{member(x,n)} = s@\text{width}(x) & \quad \texttt{bool *s = new bool}[\text{sz}+1]; \\
\quad \texttt{where} & \quad \texttt{s}[0] = \text{false;} \\
\quad \texttt{\{ & \quad \texttt{for (int i=0; i < \text{sz}; i++)} \\
\quad \quad \texttt{s[i]} \quad & \quad \texttt{if (x[i] == n)} \quad \texttt{s[i+1]} \quad \texttt{= \text{true;} } \\
\quad \quad \quad \quad \texttt{else} & \quad \texttt{s[i+1]} \quad \texttt{= s[i];} \\
\quad \quad \| i <- [0..\text{ln}] \mid; & \quad \texttt{return s[\text{sz}];} \\
\quad \} & \quad \texttt{\};} \\
prompt> member(p,4); & \quad \texttt{True}
\end{align*}
\]
Merge two infinite increasing streams of integers. This is defined recursively which cryptol does not object to because argument types always match (due to inf). First example of tail.

cryptol

```plaintext
cryptol c++
mrg:([inf][32],[inf][32]) -> [inf][32];
mrg(x,y) =
  if ((x@0) < (y@0))
    then [(x@0)]#mrg(tail(x),y)
  else [(y@0)]#mrg(x,tail(y));

class Stream {
  public:
    int first;
    virtual Stream *rest() { return this; } // this;
    Stream () { first = -1; }
    bool isNull() { return first == -1 }
};
```

A recursive specification of mergesort involving finite lists is possible by coercing the lists into infinite lists then stripping off the required number of tokens at the front using take. Notice that the role of width changes from finding bits in x@0 to finding the number of elements in x and y. For simplicity, we use 0 as a list marker - this prevents 0 from being a legal element. A more complicated solution eliminates this need.

cryptol

```plaintext
cryptol c++ (continued)
merge:{a b c} (fin a, fin b, fin c,
b+1>=width(a),b+1>=width(c)) =>
([a][b],[c][b])->[a+c][b];
merge(x,y) = take(lx+ly, mrg(ax,ay))
  where {
    mrg(px,py) =
      if ((px@0) == 0) then py
      else if ((py@0) == 0) then px
      else if ((px@0) < (py@0)) then
        [(px@0)]#mrg(tail(px),py)
      else
        [(py@0)]#mrg(px,tail(py));
    m = width(x@0);
    ax = x#zero:[inf][m];
    ay = y#zero:[inf][m];
    lx:[m+1];
    ly:[m+1];
  }

prompt> merge([3 6 8 10],[1 4 5 9 11])
[1 3 4 5 6 8 9 10 11]
```

Sum all numbers in an arbitrarily long list. This is a recursive solution so lists are padded with zero:[inf][m]. Function f is made tail recursive to allow cryptol to complete the sum of the necessary elements.

cryptol

```plaintext
cryptol
sum1:{a b} (fin a, fin b, a==b, b>=a) =>
  [a][b] -> [b];
sum1(x) = take(width(x),f(ax,0,0))
  where {
    f(y,acc,n) =
      if (n == width(x))
        then acc
      else f(tail(x),acc+y@0,n+1);
    m = width(x@0);
    ax = x#zero:[inf][m];
  }

prompt> sum1([5 3 4 2])
14
```

c++

```plaintext
c
u_int_32 sum2(u_int_32 x[],
             u_int_32 acc, int n) {
  if (n == 0) return acc;
  return sum2(x, acc+x[n-1], n-1);
}
u_int_32 sum1(u_int_32 x[], int n) {
  return sum2(x,0,n);
}
u_int_32 p[] = { 1,2,3,4,5,6 }; printf("%d",sum1(p,6));```
Sort a list of numbers using mergesort. Mergesort splits a list into two roughly equal sized lists - the odd indexed elements go to one list and the even indexed elements go to the other, recursively sorts both lists, then merges the two now sorted lists. The variable \( i \) in \texttt{srt} is needed to allow cryptol to show termination.

```
cryptol
mrgsrt:{a b} (fin a, fin b, a>=1, b>=1) => [a][b] -> [a][b];
mrgsrt(x) = take(width(x),srt(ax,0))
  where {
    srt(x,i) = 
      if (((x@0) == 0)|(i >= lx))
        then x
      else mrg(srt(splite(x,0),i+1),srt(splito(x,0),i+1));
    splite(x,i) = 
      if (((x@0) == 0)|(i >= lx))
        then x
      else if ((x@1) == 0)
        then drop(1,x)
      else [[(x@1)]#splite(drop(2,x),i+1); nmr(x,y) =
        if ((px@0) == 0) then py
      else if ((px@1) == 0) then px
        else ([(px@0)]#mrg(tail(px),py)
      else
        [(py@0)]#mrg(px,tail(py));
    m = width(x@0); ax = x#zero:[inf][m];
    lx = width(x);
  };
prompt> mrgsrt([7 3 4 2 9 6 2 1 10])
[1 2 2 3 4 6 7 9 10]

cryptol
ordered:{a b} (fin a, fin b, a>=1, b>=1) => [a][b] -> Bit;
ordered x = s@0(width(x)-1)
  where {
    s = [True]#
      [|| if ((x@i) <= (x@(i+1)))
        then (s@i & True)
      else False
      || i <- [0..(width(x)-2)] ||];
  };
prompt> ordered([4 8 10 23 66])
True
prompt> ordered([4 8 10 23 4 66])
False
```

```cpp
class Split : public Stream {
  Stream *s;
public:
  Split(Stream *str) {
    first = str->first; s = str;
  }
  Stream *rest() {
    if (isNull() ||
      s->rest()->rest()->isNull())
      return new Stream();
    return new Split(s->rest()->rest());
  }
};
class MergeSort : public Stream {
  Stream *s;
public:
  MergeSort(Stream *str) {
    Stream *s1 = new Split(str);
    Stream *s2 = new Split(str->rest());
    if (s1->isNull() && s2->isNull()) {
      s = new Stream(); first = -1;
    } else if (s2->isNull() &&
      s1->rest()->rest()->isNull())
      s = new Stream(); first = s->first;
    else {
      s = new Merge(new MergeSort(s1),
                   new MergeSort(s2));
      first = s->first;
    }
  }
};

Stream *rest() { return s->rest(); }
```

```
c
bool ordered (u_int_32 x[], int sz) {
  bool *s = new bool[sz+1];
  s[0] = true;
  for (int i=0 ; i < sz-1 ; i++)
    s[i+1] = (x[i] < x[i+1]) ?
      s[i] & true : false;
  return s[sz-1];
}
```
Returns True if and only if list \( x \) contains only positive numbers.

```
cryptol
valid_list:{a b} (fin a, fin b, a>=1, b>=1) => [a][b] -> Bit;
valid_list x = val(0)
where {
    val(i) =
        if (i == lx) then True
        else if ((x@i)<0) then False
        else val(i+1);
    lx = width(x);
};
prompt> valid_list([4 5 2 3 9])
True
prompt> valid_list([4 5 2 0 3 9])
False
```

Returns list \( x \) with one occurrence of number \( n \) removed.

```
cryptol
remove:{a b c} (fin a, fin b, fin c, a>=1, b>=1, c>=1, c==a-1, b>=width(a)) =>
([a][b],[b]) -> [c][b];
remove(x,n) = take(lx-1, rem(ax,lx,lx-1))
where {
    rem(x,l,i) =
        if ((x@i) == n) then ins(x@(l-1),i,l,0)
        else if (i == 0) then x
        else rem(x,l,i-1);
    ins(p,i,l,j) =
        if (j == l-1) then zero:[inf][w]
        else if (j == i) then [p]#ins(p,i,l,j+1)
        else [(x@j)]#ins(p,i,l,j+1);
    w = width(x@0);
    lx = width(x);
    ax = x#zero:[inf][w];
};
prompt> remove([8 4 3 5 6 4 2], 4)
[8 4 3 5 6 2]
```

Returns True if and only if list \( x \) is a permutation of list \( y \). Note: the member function used in perm is as above but typed as: member:{a b} (fin a, fin b,a>=1,b>=1) => ([a][b],[b]) -> Bit;

```
cryptol
perm:{a b} (fin a, fin b, a>=1, b>=1, b>=width(a+1)) =>
([a][b],[a][b]) -> Bit;
perm(x,y) = if (lx != ly) then False else isperm(x,y,0)
where {
    isperm(x,y,i) = if (i == lx) then True
        else if (~member(y,(x@0))) then False
        else isperm(tail(x)#[0],remove(y#[0],(x@0)),i+1);
    lx = width(x);
    ly = width(y);
};
prompt> perm([1 2 3],[3 2 1])
True
prompt> perm([1 1 2],[2 2 1])
False
```
Correctness of \texttt{mrgsrt}:

Prove that any list of four 5 bit numbers is correctly sorted by \texttt{mrgsrt}. A list \(x\) is correctly sorted to list \(y\) if \(y\) is a permutation of \(x\) and all elements of \(y\) are in increasing order.

\begin{verbatim}
cryptol
prompt> :s symbolic
prompt> :sat (\(x, y\) -> ((((y:[4][5]) == mrgsrt(x:[4][5])) & ~ (perm(y,x) & ordered(y)) & valid_list(x)))
No variable assignment satisfies this function

Alternative check - the theorems are placed in the file.

\end{verbatim}

\begin{verbatim}
cryptol
mergeSortIsCorrect : [3][5] -> Bit;
theorem mergeSortIsCorrect: \{x\}. (ordered(y) & perm(x,y)) | ~ valid_list(x)
  where y = mrgsrt(x);
prompt> :s +v
prompt> :s sbv
prompt> :s symbolic
prompt> :prove mergeSortIsCorrect
Q.E.D.
prompt> :sat mergeSortIsCorrect
mergeSortIsCorrect [0x00 0x00 0x00] = True

msCorrectBySat : [3][5] -> Bit;
theorem msCorrectBySat: \{x\}. ~ (ordered(y) & perm(x,y)) & valid_list(x)
  where y = mrgsrt(x);
prompt> :sat msCorrectBySat
No variable assignment satisfies this function
\end{verbatim}