Simple CDS Statistics
A Kleisli/CPL Tutorial

Limsoon Wong

BioInformatics Centre &
Institute of Systems Science
Heng Mui Keng Terrace, Singapore 119597
email: limsoon@iss.nus.sg

26 May 1997

Kleisli/CPL Version D Release 6 April 1997
BioKleisli Experimental Version Release 23 April 1997

©Copyright 1997
by Limsoon Wong, BioInformatics Centre, and Institute of Systems Science

Contents

1 Introduction

Kleisli/CPL [4, 1] is a powerful query system for broad-scale integration of heterogeneous distributed databanks. It is particularly suitable for dealing with the complexity of the many existing biological databanks.

In this tutorial, I will focus on using Kleisli/CPL for extracting some interesting statistics about human complete cds from Entrez data. In particular, I will look at how to get information such as exon count, exon gaps, and so on. This tutorial is inspired by some questions from Bob Lipshutz of Affymetrix.

This tutorial primarily concerns the use of a small part of the Kleisli/CPL system. You will need to have some background understanding of CPL, the Collection Programming Language supported by Kleisli, to fully appreciate this tutorial. You should consult [7] for a detailed description of CPL and [2, 3] for an introduction to the underlying theory. (We will help you by putting some footnotes on the syntax and such, whenever we remember.)

2 Downloading Human CDS

We are interested in certain statistics on full-length human genomic sequences. These are DNA sequences that are complete CDS. We do not want any mRNA, cDNA, etc. We want to look at various statistics involving the exons and also promoters in these CDS.

We use Entrez [6] as our data source. So we need to connect to Entrez. This is done by executing the following command in CPL:

```
entrez-add (#name: "entrez", #level: 2);
```

To prepare for the analysis to come, these sequences must be downloaded first. This should done in two steps. In the first step, summaries are downloaded. In the second step, the full sequence report is downloaded.

Example 2.1 *An ideal program[1] to select summaries of human complete CDS from Entrez.*

```
primitive affy-get-human-completecds-presummaries ==
  { (#accession: x.#caption,
     #uid: x.#uid,
     #title: x.#title)
  | \x <- process (
            #db: "na",
            #select: "(word complete) and (word cds) and (organism human)",
            #path: "Seq-entry",
            #args: "-D")
```

[1] This program uses the CPL comprehension syntax. In general, $\{A(x) \mid \backslash x$ `<-` $B,\ C(x)\}$ means the set of $A(x)$ such that the condition $C(x)$ is true for those x in the set B.

```
        using entrez,
    x.#title string-islike "%complete%cds%",
    not (x.#title string-islike "%cDNA%"),
    not (x.#title string-islike "%mRNA%") };
```

It defines a function to access Entrez using its `word` *index and its* `organism` *index.*[2] *The last three lines*[3] *of this function specifically checks that the words "complete cds" appear in the title field*[4] *of each selected summary and that neither "cDNA" nor "mRNA" appear in the title field.* □

If for some reason, the `organism` index of Entrez is not available, as it happened when I ran the above, don't panic. Here is a slightly less efficient but still pretty good way to get the same information.

Example 2.2 *A more clumsy way to extract human complete cds.*

```
primitive affy-get-human-completecds-presummaries ==
  { (#accession: x.#caption,
     #uid: x.#uid,
     #title: x.#title)
  | \x <- process (
            #db: "na",
            #select: "(word complete) and (word cds) and (word human)",
            #path: "Seq-entry",
            #args: "-D")
          using entrez,
     (x.#title string-islike "%human%") orelse
     (x.#title string-islike "%sapien%"),
     x.#title string-islike "%complete%cds%",
     not (x.#title string-islike "%virus%"),
     not (x.#title string-islike "%cDNA%"),
     not (x.#title string-islike "%mRNA%") };
```

This time only the `word` *index is used. As such, the variable* `x` *can get instantiated to everything in Entrez that contains the word* `complete`*,* `cds`*, and* `human`*. Thus, some non-human records may be included. To reduce such false hits, we added three lines to make sure that the title field of each summary explicitly mention the word "human" or the word "sapien", but not the word "virus."* □

Next we need to fetch the full reports associated with these summaries. However, we do not want those reports that are segmented into multiple pieces (which usually means some portions of the associated sequences could not be properly assembled.)

Example 2.3 *Download full sequence reports using sequence uids*[5] *from the already downloaded summaries.*

[2] `process` A `using` B is CPL's syntax for sending a request A to a server connection B.
[3] `String-islike` is an operator in CPL that tests if one string looks like the other.
[4] Field names in CPL are prefixed by a `#` symbol.
[5] A "uid" is a number that uniquely identifies a sequence report in Entrez.

```
primitive affy-get-human-completecds-reports ==
  { (#summary: x, #report: r)
  | \x <- affy-get-human-completecds-presummaries,
    <#seq: \r> <- process (
                    #db: "na",
                    #select: "",
                    #path: "Seq-entry",
                    #args: "-g " ^ x.#uid.num-stringify)
                  using entrez };
writefile affy-get-human-completecds-reports
to "cds-reports" using machineout;
```

The function `affy-get-human-completecds-reports` *gets only non-segmented reports* `r`*. How do we tell a report is segmented or not? Easy: A non-segmented report carries the tag* `#seq`*, while a segmented report carries the tag* `#set`*. We also save the downloaded reports to a file.*[6] □

Since only some of our summaries give rise to non-segmented reports, let us keep only these summaries.

Example 2.4 *We read the full reports back in.*[7] *Then we extract the summaries from the reports. Finally, we "materialize" them in memory.*[8]

```
readfile affy-get-human-completecds-reports
from "cds-reports" using stdin;
primitive affy-get-human-completecds-summaries ==
   { x.#summary | \x <- affy-get-human-completecds-reports };
materialize "cds-summaries";
```

□

3 Generating Indices

Having collected the relevant summaries and reports, we need to create various index files that make subsequent querying much more efficient. We will create an index on promoter regions and an index on cds regions of our sequences.

Promoter and CDS regions can be found in the feature tables of full GenBank reports. A feature table is a rather compex structure containing information on positions and annotations of a feature, as well as other information. So we need functions to extract these two types of information for promoter and CDS regions.

Example 3.1 *Here is a function that, given a feature entry in a GenBank feature table, extracts all the end points from it.*

[6] `writefile` A `to` B `using` C is CPL command for writing the object A to the file B using the output server C.

[7] `readfile` A `from` B `using` C is CPL command for reading the contents of the file B using the input server C and setting the value of the variable A to whatever is read.

[8] `materialize` is CPL's function to keep data in memory, as opposed to constructing them on the fly.

```
primitive affy-get-feat-location ==
  let \get-loc == fix (\f => \loc =>
         case loc
         of <#int: \I> =>
               { (#from: I.#from, #to: I.#to, #length: (I.#to - I.#from) + 1)}
         or <#pnt: \I> => { (#from: I.#point, #to: I.#point, #length: 1)}
         or <#equiv: \E> => E.set-head.f
         or <#mix: \M> => { l | \loc <--- M, \l <- loc.f }
         or <#packed_int: \M> => { l | \loc <--- M, \l <- loc.f }
         or <#packed_pnt: \M> =>
              { (#from: x, #to: x, #length: 1) | \x <--- M.#points }
         otherwise {})
  in get-loc o #location;
```

The location of a feature is recursively composed of multiple sublocations or fragments. In the case of a promoter, each fragment corresponds to a region of that promoter. In the case of a CDS, each fragment corresponds to an coding exon of that CDS. In the case of a mRNA, each fragment corresponds either to a coding exon, a non-coding exon, or the 5' UTR, or the 3' UTR.

Each fragment can either be an interval (and thus carries the `#int` *tag), a point (and thus carries the* `#pnt` *tag), a set of equivalent locations (and thus carries the* `#equiv` *tag), a mixed list of points and intervals (and thus carries the tag* `#mix`*), a homogeneous list of intervals (and thus carries the tag* `#packed_int`*), or a homogeneous list of points (and thus carries the tag* `#packed_pnt`*).*

The function `get-loc` *uses a fixpoint computation to recursively traverse the fragments and extract their endpoints and lengths into a set.*[9] *The function* `affy-get-feat-location` *is composed*[10] *of a projection on the* `#location` *field of the input feature, followed by a call to the endpoint extraction function* `get-loc`*.* □

Example 3.2 *Here is a function that, given a feature entry in a GenBank feature table, extracts its primary annotations. We consider product name, gene name, and qualifiers to be primary annotations. This function relies on Kleisli/CPL's wildcard structural matching capability.*[11]

```
primitive affy-get-feat-qual == \feat =>
  let \S == { s.het-km
            | \s <- het-multimatch ("*%#prot.##name.?", het-mk (feat))} in
  let \T == { s.het-km
            | \s <- het-multimatch ("*%#gene.##locus.%#some", het-mk(feat))} in
  let \Q == { q.het-km.#val
            | \q <- het-multimatch ("*##qual.?", het-mk (feat)),
              (q.het-km.#qual = "gene") orelse (q.het-km.#qual = "product") }
  in set-unique (Q {+} S {+} T);
```

□

Now we are ready to extract promoter region and CDS region information and save them into two files.

[9] `fix` is a CPL function that takes a non-recursive functional and returns its least fixpoint. Thus $\texttt{fix}(f) = f(\texttt{fix}(f))$.

[10] The syntax $f \circ g$ defines a function h such that $h(x) = f(g(x))$.

[11] `het-mk` is a CPL function that turns any object into one that wildcard structural matching can be performed. `het-multimatch` is a CPL function that performs wildcard structural matching. See [7, 5] for more information.

Example 3.3 *Extract promoter region information.*

```
primitive affy-get-promoter-index ==
  { (#uid: x.#summary.#uid, #promoter: (#loc: P, #qual: Q))
  | \x <- affy-get-human-completecds-reports,
    (#data: <#ftable: \F>, ...) <- x.#report.#annot,
    \f <- F,
    (#data: <#imp: (#key: "promoter", ...)>, ...) == f,
    \P == affy-get-feat-location (f),
    \Q == affy-get-feat-qual (f) };
writefile affy-get-promoter-index
to "promoter-index" using stdout;
```

□

Example 3.4 *Extract CDS region information.*

```
primitive affy-get-cds-index ==
  { (#uid: x.#summary.#uid, #cds: (#loc: P, #qual: Q))
  | \x <- affy-get-human-completecds-reports,
    (#data: <#ftable: \F>, ...) <- x.#report.#annot,
    \f <- F,
    (#data: <#cdregion: _>, ...) == f,
    \P ==  affy-get-feat-location (f),
    \Q == affy-get-feat-qual (f) };
writefile affy-get-cds-index
to "cds-index" using stdout;
```

□

Having created the two index files, We can now form the in-memory indices. We use the uid associated with a GenBank report as the index key in both cases.

Example 3.5 *Form in-memory index of promoter regions.* `promoter-index` *becomes a function that given a uid, returns all promoter regions coming from the associated records.*

```
readfile promoter from "promoter-index" using stdin;
primitive promoter-index == (set-index' (#uid, promoter)).#eq;
materialize "promoter-index";
```

□

Example 3.6 *Form in-memory index of CDS regions.* `cds-index` *becomes a function that given a uid, returns all promoter regions coming from the associated records.*

```
readfile cds from "cds-index" using stdin;
primitive cds-index == (set-index' (#uid, cds)).#eq;
materialize "cds-index";
```

□

In the remaining sections, we show how to use these indices to obtain various statistics of our sequences.

4 Promoter Count

The first statistics I want to get are (a) the mean, (b) the variance, (c) the minimum, and (d) the maximum of the number of regions that promoters have. I would also like to generate a distribution graph on the number of regions.

Recall that `affy-get-human-completecds-summaries` is a set containing the summaries of all relevant sequences. Recall also that `promoter-index` is a function that gives us the promoter regions of a sequence given its uid.

Example 4.1 *For each sequence* `x` *and for each promoter* `G` *in* `x`*, we count*[12] *the number of regions in* `G`*. The bag* `affy-get-promoter-count` *contains these count.*

```
primitive affy-get-promoter-count ==
 {| set-roughcount (G)
  | \x <- affy-get-human-completecds-summaries,
    \G <- { c.#promoter.#loc | \c <- x.#uid.promoter-index } |};
materialize "affy-get-promoter-count";
```

□

Now we generate some numbers needed to produce our statistics.

Example 4.2 *Count the number of promoters (a). Sum their number of regions (b). Sum the squares of their number of regions (c).* `Affy-get-promoter-sums` *defined below evaluates to a triple* (a, b, c) *where* a*,* b*, and* c *are the desired numbers.*

```
primitive affy-get-promoter-sums ==
  let \incr == (\i, (\c, \x, \y)) => (c + 1, x + i, y + (i * i))
  in bag-sri @ (incr, (0, 0, 0)) @ affy-get-promoter-count;
materialize "affy-get-promoter-sums";
```

□

To get the minimum and maximum number of regions that promoters can have, we just sort the counts and extract the two ends of the sorted list.[13]

Example 4.3 `Affy-get-promoter-minmax` *is a pair* (a, b) *where* a *is the minimum and* b *is the maximum number of regions our promoters have.*

[12] `set-numitems` is CPL's function to obtain the cardinality of a set.

[13] `list-sort` is CPL's function to sort a list in ascending order. `list-head` extracts the first item in a list. `list-rev` reverses a list.

```
primitive affy-get-promoter-minmax ==
  let \p == list-sort [ x | \x <-- affy-get-promoter-count ]
  in (p.list-head, p.list-rev.list-head);
materialize "affy-get-promoter-minmax";
```

□

Let us produce the desired statistics now.

Example 4.4 *The desired mean, variance, min, max, and count of promoter regions.*

```
primitive affy-get-promoter-statistics ==
  let (\pc, \px, \py) == affy-get-promoter-sums in
  let (\pmin, \pmax) == affy-get-promoter-minmax
  in (#no-of-promoters: pc,
      #min-region-count: pmin,
      #max-region-count: pmax,
      #mean-region-count: px / pc,
      #variance: (py - ((px * px) / pc)) / pc);
materialize "affy-get-promoter-statistics";
```

□

One last thing to do. We want a function to generate the distribution of promoter regions count. That is, we want to know how many promoters have n regions each.

Example 4.5 `Affy-get-promoter-dist` *is a list of pairs* (x, y)*. A pair* (x, y) *is in this list if and only if the number of promoters that have* x *regions each is equal to* x*.*

```
primitive affy-get-promoter-dist ==
  (bag-index' (\x => x, affy-get-promoter-count)).#eq before (\idx =>
  [ (x, y)
  | \x <-- bag-unique (affy-get-promoter-count),
    \y == bag-sru @ (+, _ => 1, 0) @ (x.idx) ]);
```

□

5 Promoter Length

We want statistics on promoter length in this section. That is, the mean, min, max, variance, and distribution of promoter lengths.

The first step is to obtain the lengths.

Example 5.1 `Affy-get-promoter-size` *is a bag containing the length of the relevant promoters.*

```
primitive affy-get-promoter-size ==
 {| set-sru @ (+, #length, 0) @ G
  | \x <- affy-get-human-completecds-summaries,
    \G <- {  c.#promoter.#loc | \c <- x.#uid.promoter-index } |};
materialize "affy-get-promoter-size";
```

□

Next we generate the desired statistics. The steps are similar to that of the previous section. To make things more interesting, I will generalize a bit.

Example 5.2 `Affy-get-statistics` *is a function that, given a bag of sample numbers, produces the min, max, mean, and variance of these samples.*

```
primitive affy-get-statistics == \samples =>
  let \get-sums ==
         let \incr == (\i, (\c, \x, \y)) => (c + 1, x + i, y + (i * i))
         in bag-sri @ (incr, (0, 0, 0)) @ samples in
  let \get-minmax ==
         let \p == list-sort [ x | \x <-- samples ]
         in (p.list-head, p.list-rev.list-head) in
  let (\pc, \px, \py) == get-sums in
  let (\pmin, \pmax) == get-minmax
  in (#no-of-samples: pc,
      #min: pmin,
      #max: pmax,
      #mean: px / pc,
      #variance: (py - ((px * px) / pc)) / pc);
```

□

Then the desired statistics is obtained by executing the following expression.

```
affy-get-statistics (affy-get-promoter-size);
```

Let us do the same thing for generating distribution. To make it even more interesting, we allow the user to specify the size of the intervals in the distribution. That is, we allow him to say "How many promoters have size between n and m bp."

Example 5.3 `Affy-get-dist` *is a function. It expects a pair (S, I) as its input. S is a bag of sample numbers. I is to be used as interval size. It returns a list of triples. A triple $(\#from : a, \#to : b, \#count : c)$ is in the output list if and only if there are c numbers in S that whose size is between a and b (inclusive).*

```
primitive affy-get-dist == (\samples, \interval) =>
  {| x div interval | \x <-- samples |} before (\count =>
  (bag-index' (\x => x, count)).#eq before (\idx =>
```

```
  [ (#from: x * interval,
     #to: ((1 + x) * interval) - 1,
     #count: y)
  | \x <-- bag-unique (count),
    \y == bag-sru @ (+, _ => 1, 0) @ (x.idx) ]));
```

□

For example, to get the distribution of promoter sizes at intervals of 200 bp, we can now simply execute the following expression.

```
affy-get-dist (affy-get-promoter-size, 200);
```

6 Exon Count

I want to now obtain statistics on the number of exons our CDS can have. As in the two previous sections, let us first obtain a bag of exon count.

Example 6.1 `Affy-get-exon-count` *is a bag containing the count of the number of exons for each CDS in our collection.*

```
primitive affy-get-exon-count ==
 {| set-roughcount (G)
  | \x <- affy-get-human-completecds-summaries,
    \G <- { c.#cds.#loc | \c <- x.#uid.cds-index } |};
materialize "affy-get-exon-count";
```

□

Example 6.2 *We can now use the function* `affy-get-statistics` *from the previous section to obtain the min, max, mean, and variance of the number of exons in our CDS. Just execute the following expression:*

```
affy-get-statistics (affy-get-exon-count);
```

□

Example 6.3 *We can also use the* `affy-get-dist` *function from the previous section to obtain the distribution of exon counts. Try it out by executing the expression below. (We use an interval size of 1.)*

```
affy-get-dist (affy-get-exon-count, 1);
```

□

7 Exon Length

Statistics on exon length is next.

Example 7.1 `Affy-get-exon-size` *is a bag containing the length of every exon in every CDS in our collection.*

```
primitive affy-get-exon-size ==
 {| z.#length
  | \x <- affy-get-human-completecds-summaries,
    \c <- x.#uid.cds-index,
    \z <- c.#cds.#loc |};
materialize "affy-get-exon-size";
```

□

Example 7.2 *The min, max, mean, and variance of the size of all exons in our collection can be obtained by running this:*

```
affy-get-statistics (affy-get-exon-size);
```

□

Example 7.3 *The distribution of exon size can be obtained by executing the expression below. The interval size is 200bp.*

```
sffy-get-dist (affy-get-exon-size, 200);
```

□

Very easy, isn't it? Let us move on to statistics on more interesting stuff!

8 Exon Gap

Stuff like promoter size, promoter count, exon size, and exon count were explictly recorded in indices `promoter-index` and `cds-index`. We now look at an example that gets statistics of non-explicitly recorded data: Exon gaps.

In a CDS with multiple exons, there are gaps between these exons. That is, those positions from the end of one exon to the start of the next exon. These gaps correspond to introns. We want to know their size statistics. To obtain the desired statistics require many steps.

Example 8.1 *Before we can extract the gaps in a CDS, we have to sort the exons of that CDS according to their positions.* `Affy-sort-loc` *is a function that, given a set of exon locations, sort them according to their positions.*[14]

[14] `list-gensort` is a CPL function for general sorting. It has two arguments. The first is the comparison function to be used. The second is the list to be sorted.

```
primitive affy-sort-loc ==
  let \leq == (\L, \R) => (L.#from < R.#from) orelse
                          ((L.#from = R.#from) andalso (L.#to <= R.#to)) in
  let \cvt == \S => [ x | \x <- S ]
  in (list-gensort @ leq) o cvt;
```

□

Example 8.2 `Affy-gap-loc` *is a function that extract gaps from a linearly ordered list of exons.*

```
primitive affy-gap-loc ==
  let \add == (\f, \t, \a) =>
         (#from: f + 1, #to: t - 1, #length: t - f - 1) +} a in
  let \incr == (\x, (\y, \a)) => case y
         of <#none: ()> => (<#some: x.#from>, {})
         or <#some: \z> => (<#some: x.#from>, add (x.#to, z, a))
  in #2 o (list-irs @ (incr, (<#none: ()>, { }))) o affy-sort-loc;
```

□

Having these two functions, we can now extract exon gaps in our CDS by first sorting the exons in each CDS using `affy-sort-loc`, then generates the gaps using `affy-gap-loc`, and finally extracts the length of these gaps.

Example 8.3 `Affy-get-gap-size` *is a bag containing the size of all the exon gaps in our CDS.*

```
primitive affy-get-gap-size ==
 {| z.#length
  | \x <- affy-get-human-completecds-summaries,
    \c <- x.#uid.cds-index,
    \z <- c.#cds.#loc.affy-gap-loc |};
materialize "affy-get-gap-size";
```

□

Phew! We can get our statistics and distributions easily now.

Example 8.4 *Execute the expression below to get exon gap statistics.*

```
affy-get-statistics (affy-get-gap-size);
```

□

Example 8.5 *Execute the expression below to get exon gap distribution. Set interval size to 200 bp.*

```
sffy-get-dist (affy-get-gap-size, 200);
```

□

9 CDS

I want to obtain statistics on total exon size in our CDS, ignoring any intervening introns or gaps.

Example 9.1 *For this, we need to first compute the total exon size of each of our CDS.* `Affy-get-cds-size` *is a bag containing the total exon size of all our CDS.*[15]

```
primitive affy-get-cds-size ==
 {| set-sru @ (+, #length, 0) @ G
  | \x <- affy-get-human-completecds-summaries,
    \G <- { c.#cds.#loc | \c <- x.#uid.cds-index } |};
materialize "affy-get-cds-size";
```

□

Example 9.2 *To get statistics of total exon size, just execute the following:*

```
affy-get-statistics (affy-get-cds-size);
```

□

Example 9.3 *To get distribution of total exon size at an interval size of 200bp, just execute the following expression:*

```
affy-get-dist (affy-get-cds-size, 200);
```

□

10 CDS and Intervening Introns

The total size of CDS, including all the exon gaps, requires a slightly different treatment. We can't just sum up the size of the exons in the CDS any more. However, we can compute the total size of a CDS including all its exon gaps in a simple way. The difference between the leftmost or minimum endpoint and the rightmost or maximum endpoint of a CDS gives its total size :-)

Example 10.1 `Affy-get-cds-size` *is bag containing the total size of a CDS, including all its introns.*[16]

```
primitive affy-get-cds+intron-size ==
 {| 1 + (max - min)
```

[15] `set-sru` is a CPL function that has two arguments. The first argument is a triple (u, f, i). The second argument is a set S. It returns $f(o_1)\ u\ \cdots\ u\ f(o_n)\ u\ i$, where o_i's are the element sof S.

[16] `list-sortdesc` is a CPL function that sorts its input in a decreasing order.

```
  | \x <- affy-get-human-completecds-summaries,
    \G <- { c.#cds.#loc | \c <- x.#uid.cds-index },
    \min == [ g.#from | \g <- G].list-sort.list-head,
    \max == [ g.#to | \g <- G].list-sortdesc.list-head  |};
materialize "affy-get-cds+intron-size";
```

□

Statistics and distribution of total CDS size is now chicken feed.

Example 10.2 *Execute the following to get statistics.*

```
affy-get-statistics (affy-get-cds+intron-size);
```

□

Example 10.3 *Execute the following to get distribution. Use an interval size of 500bp.*

```
affy-get-dist (affy-get-cds+intron-size, 500);
```

□

11 Promoters, CDS, and Intervening Introns

In all our previous examples, we did not mix promoters and CDS. Let us do something about it in this section. I now want statistics on the total size of promoter, CDS, and exon gaps for those CDS and promoters that "go together."

What does "go together" mean? Well, it means that the promoter and CDS are part of the same gene or product. How to test for this? Ah, remember our promoter and cds indices also carry some useful annotation or qualifier information. This is the key to solving our problem. Given a promoter region and a CDS region, if all the qualifiers of one are consistent with those of the other one, then we can assume that the two should "go together."

Example 11.1 `Affy-go-together` *is a function that tests if two regions "go together."*[17]

```
primitive affy-go-together == (\L, \R) =>
  set-forall
  @ (\l => set-thereis
          @ (\r => (r string-islike ("%" ^ l ^ "%")) orelse
                   (l string-islike ("%" ^ r ^ "%")))
          @ R)
  @ L;
```

[17] `string-islike` is a CPL function that tests if two strings are alike.

□

Now we put everything together to get those total sizes of our sequences.

Example 11.2 `Affy-get-promoter`*cds+intron-size+ is a bag containing the total size of promoter and CDS (with introns) that "go together."*

```
primitive affy-get-promoter+cds+intron-size ==
 {| (1 + (max - min)) + (set-sru @ (+, #length, 0) @ (G.#2))
  | \x <- affy-get-human-completecds-summaries,
    \G <- set-unique { (Z, U)
            | \y <- set-unique (
                     { y.#cds.#qual | \y <- x.#uid.cds-index} {+}
                     { y.#promoter.#qual | \y <- x.#uid.promoter-index}),
              \Z == { z | \c <- x.#uid.cds-index,
                         c.#cds.#qual affy-go-together y,
                         \z <- c.#cds.#loc },
              \U == { z | \c <- x.#uid.promoter-index,
                         c.#promoter.#qual affy-go-together y,
                         \z <- c.#promoter.#loc },
              U.set-isempty.not,
              Z.set-isempty.not },
    \min == [ g.#from | \g <- G.#1].list-sort.list-head,
    \max == [ g.#to | \g <- G.#1].list-sortdesc.list-head  |};
materialize "affy-get-promoter+cds+intron-size";
```

□

The rest is as before.

Example 11.3 *Execute the following to get the desired statistics.*

```
affy-get-statistics (affy-get-promoter+cds+intron-size);
```

□

Example 11.4 *Execute the following to get the desired distribution. Use interval size of 500 bp.*

```
affy-get-dist (affy-get-promoter+cds+intron-size, 500);
```

□

12 A Concluding Exercise

I showed you a lot of ways to do simple statistics on CDS and promoters already. Why don't you try to do one yourself? Here is a suggestion: Get statistics on the total size of promoter and CDS (without introns) in our sequences. Enjoy!

References

[1] Peter Buneman, Susan Davidson, Kyle Hart, Chris Overton, and Limsoon Wong. A data transformation system for biological data sources. In *Proceedings of 21st International Conference on Very Large Data Bases*, pages 158–169, Zurich, Switzerland, August 1995.

[2] Peter Buneman, Leonid Libkin, Dan Suciu, Val Tannen, and Limsoon Wong. Comprehension syntax. *SIGMOD Record*, 23(1):87–96, March 1994.

[3] Peter Buneman, Shamim Naqvi, Val Tannen, and Limsoon Wong. Principles of programming with complex objects and collection types. *Theoretical Computer Science*, 149(1):3–48, September 1995.

[4] Susan Davidson, Christian Overton, Val Tannen, and Limsoon Wong. BioKleisli: A digital library for biomedical researchers. Technical Report 96-215-0, Institute of Systems Science, Heng Mui Keng Terrace, Singapore 119597, August 1996.

[5] Kyle Hart and Limsoon Wong. Pruning nested data values using branch expressions with wildcards. In *Abstracts of 2nd Meeting on the Interconnection of Molecular Biology Databases*, Cambridge, England, July 1995. Available at `http://www.cis.upenn.edu/~khart/mimbd95/branches.html`.

[6] National Center for Biotechnology Information, National Library of Medicine, Bethesda, MD. *ENTREZ: Sequences Users' Guide*, 1992. Release 1.0.

[7] Limsoon Wong. *The Collection Programming Language Reference Manual.* Institute of Systems Science, Heng Mui Keng Terrace, Singapore 119597, October 1995. Available at `http://sdmc.iss.nus.sg/kleisli/psZ/cpl-defn.ps`.