The Simplified SQL Reference Manual

Limsoon Wong

Kent Ridge Digital Labs
21 Heng Mui Keng Terrace, Singapore 119613
`email: limsoon@krdl.org.sg`

10 November 1998
Updated 11 September 2001

Version E Release 11 September 2001

©Copyright 2001

by Limsoon Wong and Kent Ridge Digital Labs

Contents

1 Introduction **5**

2 Introduction to sSQL for Beginners **6**

2.1 Records and Collections 6

2.2 Select-From 6

2.3 Select-From-Where 7

2.4 Select-From-Where-Order-By 7

2.5 Interfacing to Sybase 8

3 Introduction to sSQL for Experts **9**

3.1 Types 9

3.2 Expressions 10

3.3 Other Operations and Shorthands 17

3.4 Commands 18

4 Example Bioinformatics Queries **20**

4.1 Proline at N-Terminal 20

4.2 TPR Domains in Swissprot Sequences 20

4.3 Patented Proteins in Superfamily 21

4.4 Compact Genome of Pufferfish 22

A Switching Interpretors **27**

B Lexical Conventions **27**

B.1 Variables Names 27

B.2 Label Names 27

B.3 Numbers 27

B.4 Comments 27

B.5 Reserved Words 27

B.6 Precedence and Associativity 28
B.7 Embedded ML Codes 28
B.8 Embedded CPL Codes 28

C Available Data Drivers **29**

D Available External Primitives **29**

E Available System Operations **29**

1 Introduction

For the last several years, CPL has been the only high-level query language for the Kleisli Query System. CPL has a syntax that is familiar to functional programming language community. However, this also makes it superficially very different from SQL, the standard query language of the database community.

There are very good reasons from a mathematical and design principles viewpoint that CPL is a good query language. There are also very good reasons from the same viewpoint that SQL is a rather poorly designed query language [4, 3, etc.].

Unfortunately, real-life is not often rational and SQL is popular. So after much consideration, I have decided to make available **simplified SQL** (sSQL for short) as a second high-level query language for the Kleisli Query System. sSQL follows the syntax of SQL very closely and is designed so that it can be translated to the core of CPL easily. (For more information on the core of CPL, see *The Collection Programming Language Reference Manual.*)

Almost all of the most commonly used fragment of SQL has been adopted into sSQL. However, there are certain parts of SQL that I reject from sSQL. I should briefly mention the junks that I reject from sSQL and my reasons for these design decisions. The first thing I get rid of are all the stupid restrictions in SQL that prevent generalization to nested relations, because there is absolutely no reason not to allow for nested relations since Kleisli can handle them. The second thing I get rid of is the part of SQL syntax that is ambiguous, because this part of SQL cannot be interpreted in the absence of schema.[1] The third thing I get rid of is the `GROUP-BY` syntax, because its presence in SQL is due to the fact that SQL cannot handle nested relations and is thus redundant in sSQL, which can handle nested relations. The last thing I get rid of is SQL's special syntax for using aggregate functions, because it is a "side effect" from SQL's irrational `GROUP-BY` syntax.

[1] Alas, I have been forced to reintroduced some of these. But don't blame me if you are surprised by some unexpected behaviours, as the way I resolve some of these ambiguity may not be the same way as you!

2 Introduction to sSQL for Beginners

This section provides an informal account of a few constructs in sSQL.

2.1 Records and Collections

Although sSQL does not limit the matter which may be stored or manipulated, it strictly confines the manner in which its subjects may be held. It insists that it will accept information only if it has been divided into carefully structured pieces.

The kind of structured pieces (or data types) allowed in sSQL is considerably richer than that in SQL: records, variants, sets, bags, and lists are all available and can all be freely nested and composed. We focus on records and sets here; the other data types are described later.

- Records in sSQL corresponds to a row in a relational table in SQL. They looks like this:

  ```
  (#name: "Sun", #distance: 0, #diameter: 865400)
  ```

  `#name`, `#distance`, and `#diameter` are the field names or field labels of the records. `"Sun"`, `0`, `865400` are the values of these fields in the record. In general, the value of a field can be another record, or a set, or other complicated data types.

- Sets in sSQL correspond to relational tables in SQL. They looks like this:

  ```
  { (#name: "Sun", #distance: 0, #diameter: 865400),
    (#name: "Saturn", #distance: 887, #diameter: 75100),
    (#name: "Neptune", #distance: 2795, #diameter: 27700) }
  ```

  In general, the elements of a set can be sets, or records of sets, or other more complicated data types.

Let `SolarSystem` be the set containing Sun, Saturn, and Neptune as above. (Yes, I know Earth, Jupiter, etc. are missing from this set!) The next several subsections illustrate some sSQL query constructs.

2.2 Select-From

- `select x from SolarSystem as x;`

  Returns every element `x` in the set `SolarSystem`. The result is thus:

  ```
  { (#name: "Sun", #distance: 0, #diameter: 865400),
    (#name: "Saturn", #distance: 887, #diameter: 75100),
    (#name: "Neptune", #distance: 2795, #diameter: 27700) }
  ```

- `select x.#name from SolarSystem as x;`

  Returns the name of every element `x` in the set `SolarSystem`. The result is thus:

```
{ "Sun",
  "Saturn",
  "Neptune" }
```

- `select #name: x.#name, #diameter: x.#diameter from SolarSystem as x;`

  Returns the name and diameter of every element in `SolarSystem`. The result is thus:

```
{ (#name: "Sun", #diameter: 865400),
  (#name: "Saturn", #diameter: 75100),
  (#name: "Neptune", #diameter: 27700) }
```

2.3 Select-From-Where

- ```
  select x
  from SolarSystem as x
  where x.#name string-islike "S%";
  ```

  Returns elements of `SolarSystem` whose names start with "S". The result is thus:

```
{ (#name: "Sun", #distance: 0, #diameter: 865400),
  (#name: "Saturn", #distance: 887, #diameter: 75100) }
```

- ```
  select #name: x.#name
  from SolarSystem as x
  where (x.#diameter < 100000) andalso
        (x.#name string-islike "S%");
  ```

  Returns names fo elements in `SolarSystem` whose diameter is small and whose name starts with "S". Thus the result is

```
{ (#name: "Saturn") }
```

2.4 Select-From-Where-Order-By

- ```
  select x
  from SolarSystem as x
  where not(x.#name = "Sun") order by #name;
  ```

  Return everything in `SolarSystem` other than the Sun and sort the output in ascending order by their names. The result is thus:

```
{ (#name: "Neptune", #distance: 2795, #diameter: 27700),
  (#name: "Saturn", #distance: 887, #diameter: 75100)}
```
```

- ```
  select #name: x.#name, #distance: x.#distance
  from SolarSystem as x
  order by #distance desc;
  ```

  Return the name and distance of every element in `SolarSystem` and sort them in descending order of distance. The result is thus:

  ```
  { (#name: "Neptune", #distance: 2795),
    (#name: "Saturn", #distance: 887),
    (#name: "Sun", #distance: 0) }
  ```

2.5 Interfacing to Sybase

The easiest way to connect to Sybase is to do this:

```
sybase-add (
    #name: "gdb56",
    #user: "gdbsqllim",
    #password: "BLAHBLAH",
    #level: 1,
    #host: "JICST",
    #port: "4100");
```

This creates an agent (a set of connections) "gdb56" to the GDB database in Japan. Thus following the above, you can immediately query tables in the "gdb" database. For example, you can check the locations of various GDB servers by looking up the "gdb_servers" table like this:

```
select #gdb_server_location: x.#gdb_server_location
from  gdb_servers as x
using gdb56;
```

```
Result:
{(#gdb_server_location: "Medical Research Council, England (Read-only)"),
 (#gdb_server_location: "Demonstration Genome Database ( Host 1 )"),
 (#gdb_server_location: "Demonstration Genome Database ( Host 2 )"),
 (#gdb_server_location: "Medical Research Council, England (Read-only)"),
 ... }
```
```

3 Introduction to sSQL for Experts

This section more formally defines the semantics of sSQL.

3.1 Types

The ground complex object types, ranged over by s and t, are given by the grammar below. The l_i are labels and are required to be distinct. The b ranges over base types and parameterized user-defined types. Parameterized user-defined types are really a sort of abstract data types, and for the purpose of this report can be treated as base types.

$$s ::= b \mid \{s\} \mid \{\texttt{|}\,s\,\texttt{|}\} \mid \texttt{[}s\texttt{]} \mid \texttt{(}l_1 : s_1, \cdots, l_n : s_n\texttt{)} \mid \texttt{<}l_1 : s_1, \cdots, l_n : s_n\texttt{>}$$

The ground types, over which u and v range, are given by the grammar below.

$$u ::= b \mid \{s\} \mid \{\texttt{|}\,s\,\texttt{|}\} \mid \texttt{[}s\texttt{]} \mid \texttt{(}l_1 : u_1, \cdots, l_n : u_n\texttt{)} \mid \texttt{<}l_1 : u_1, \cdots, l_n : u_n\texttt{>} \mid u \texttt{ -> } v$$

sSQL allows $\texttt{(}u_1, \cdots, u_n\texttt{)}$ as a syntactic sugar for $\texttt{(}\#1 : u_1, \cdots, \#n : u_n\texttt{)}$. Labels in sSQL always start with the #-sign.

Objects of type $\{s\}$ are finite sets whose elements are objects of type s. Objects of type $\{\texttt{|}\,s\,\texttt{|}\}$ are finite bags whose elements are objects of type s. Objects of type $\texttt{[}s\texttt{]}$ are finite lists whose elements are objects of type s. Objects of type $\texttt{(}l_1 : u_1, \cdots, l_n : u_n\texttt{)}$ are records having exactly fields l_1, ..., l_n and whose values at these fields are objects of types u_1, ..., u_n respectively. An object of type $\texttt{<}l_1 : u_1, \cdots, l_n : u_n\texttt{>}$ is called a variant object and is a pair $\texttt{<}l_i : o\texttt{>}$ such that o is an object of type u_i; that is, it is an object of type u_i tagged with the label l_i. (Variants are also called tagged-unions and co-products. See Gunter [7] or Hull and Yap [8] for more information.) Objects of type $u \texttt{ -> } v$ are functions from type u to type v. Included amongst the base types are `num`, `string`, `bool`, and `unit` (which is the type having exactly the empty record `()` as its only object). It is also possible to extend sSQL with new base types; see *The Kleisli Query System Reference Manual.*

sSQL also has nonground types. I only intend to explain how to read a sSQL type expression having nonground types below. The nonground complex object types are ranged over by the symbol σ. Nonground complex object types are obtained from ground complex object types by replacing some subexpressions with complex object type variables of the following forms:

- Unconstrained complex object type variable. It has the form `''n`, where n is a natural number. It can be instantiated to any ground complex object type.
- Record complex object type variable. It has the form $\texttt{(}l_1 : \sigma_1, \cdots, l_n : \sigma_n\texttt{; ''m)}$, where m is a natural number. It can only be instantiated to ground record types having at least the fields l_1, ..., l_n, so that σ_i can be consistently instantiated to the type at field l_i.
- Variant complex object type variable. It has the form $\texttt{<}l_1 : \sigma_1, \cdots, l_n : \sigma_n\texttt{; ''m>}$, where m is a natural number. It can only be instantiated to ground variant types having at least the fields l_1, ..., l_n, so that σ_i can be consistently instantiated to the type at field l_i.

- Numeric tuple object type variable. It has the form ^m>=n, where m and n are natural numbers. It can only be instantiated to ground tuples of numbers of arity at least n.
- Subtyping object type variable. It has the form 'm<b, where m is a natural number and b is the name of a user-defined type. It can only be instantiated to b or to its subtypes. Note that b must be a user-defined type that has no parameters. The subtyping supported by CPL allows only (explicitly declared) single inheritance.

The nonground types are ranged over by the symbol β. Nonground types are obtained from ground types by replacing some subexpressions with a universal type variable of the following forms:

- Unconstrained universal type variable. It has the form 'n, where n is a natural number. It can be instantiated to any ground type.
- Record universal type variable. It has the form ($l_1 : \beta_1, \cdots, l_n : \beta_n$; 'm), where m is a natural number. It can only be instantiated to ground record types having at least the fields l_1, ..., l_n, so that β_i can be consistently instantiated to the type at field l_i.
- Variant universal type variable. It has the form <$l_1 : \beta_1, \cdots, l_n : \beta_n$; 'm>, where m is a natural number. It can only be instantiated to ground variant types having at least the fields l_1, ..., l_n, so that β_i can be consistently instantiated to the type at field l_i.

Hence for example (#age : string; '1) -> string indicates the type of functions whose inputs are records having at least the field #age of string type and producing outputs of string type. Similarly, (#age : string; '1) -> (#age : string; '1) indicates the type of functions whose inputs are records having at least the fields #age of string type and produces outputs of exactly the same type as the inputs.

The distinction between nonground complex object types and nonground types is that the latter types include function types but the former types do not. For example, the nonground complex object type (#input_set: {num}; ''1) cannot be instantiated to a record type such as (#input_set: {num}, #transformer: num->num). On the other hand, the nonground type (#input_set: {num}; '1) can be instantiated to a record type such as (#input_set: {num}, #transformer: num->num).

3.2 Expressions

The expressions are ranged over by e. The variables are ranged over by x. For simplicity, we assign a ground type once-and-forever to all the variables; the type u assigned to a variable x is indicated by superscripting: x^u. Expression formation constructs are based on the structure of types.

For function types, the expression constructs are given in Figure 1. The meaning of apply e_1 to e_2 is the result of applying the function e_1 to the object e_2. The meaning of function that transforms x^u to e is a function f that when applied to an object o of type u produces the object $e[o/x^u]$. (The notation $e[o/x^u]$ means replace all free occurrences of x^u in e by o.)

$$\frac{}{x^u : u} \qquad \frac{e : v}{\texttt{function that transforms } x^u \texttt{ to } e : u \texttt{ -> } v} \qquad \frac{e_1 : u \texttt{ -> } v \quad e_2 : u}{\texttt{apply } e_1 \texttt{ to } e_2 : v}$$

Figure 1: The constructs for function types in sSQL.

For record types, the expression constructs are given in Figure 2. I have already mentioned that `()` is the unique object having type `unit`. The construct $(l_1 : e_1, \cdots, l_n : e_n)$ forms a record having fields l_1, ..., l_n whose values are e_1, ..., e_n respectively. A label when used as an expression stands for the obvious projection function. The construct `modify` l `of` e_1 `to` e_2 forms a new record having the same value as e_1, except at the l field which is updated to e_2.

$$\frac{}{\texttt{()} : \texttt{unit}} \qquad \frac{e_1 : u_1 \quad \cdots \quad e_n : u_n}{(l_1 : e_1, \cdots, l_n : e_n) : (l_1 : u_1, \cdots, l_n : u_n)}$$

$$\frac{}{l^{(l:u,\ l_1:u_1, \cdots, l_n:u_n)} : (l : u,\ l_1 : u_1, \cdots, l_n : u_n) \texttt{ -> } u}$$

$$\frac{e_1 : (l : u,\ l_1 : u_1, \cdots, l_n : u_n) \qquad e_2 : u}{\texttt{modify } l \texttt{ of } e_1 \texttt{ to } e_2 : (l : u,\ l_1 : u_1, \cdots, l_n : u_n)}$$

Figure 2: The constructs for record types in sSQL.

For variant types, the expression constructs are given in Figure 3. The construct `<`$l : e$`>` forms a variant object by tagging the object e with the label l. The case-expression evaluates to $e_i[o/x_i^{u_i}]$ if e evaluates to `<`$l_i : o$`>`. The case-otherwise-expression evaluates to $e_i[o/x_i^{u_i}]$ if e evaluates to `<`$l_i : o$`>` where $1 \leq i \leq n$; otherwise it evaluates to e'.

For the base type `bool`, there are the usual constructs given in Figure 4.

For set types, the expression constructs are given in Figure 5. The meaning of $\{e_1, \ldots, e_n\}^s$ is the set containing exactly e_1, ..., e_n as elements. The meaning of e_1`union` e_2 is the set union of e_1 and e_2. The numerous `select-from-where` constructs are inherited from SQL. The very fact that there are so many of them points to the irrationality of the design of SQL. This is in clear contrast to CPL where a single `ext` construct suffices; see the *Collection Programming Language Reference Manual*. Here are the definitions of these clumsy constructs inherited from SQL. We give these definitions in terms of CPL's comprehension syntax. `select` e `from` e_1 `as` x_1, ..., e_n `as` x_n `where` e' is the same set as given by the CPL expression $\{e \mid \backslash x_1$ `<-` e_1, ..., $\backslash x_n$ `<-` $e_n, e'\}$. `select` e `from` e_1 `as` x_1, ..., e_n `as` x_n is the same set as given by the CPL expression $\{e \mid \backslash x_1$ `<-` e_1, ..., $\backslash x_n$ `<-` $e_n\}$. `select distinct` e `from` e_1 `as` x_1, ..., e_n `as` x_n `where` e' is semantically equal to `select` e `from` e_1 `as` x_1, ..., e_n `as` x_n `where` e', but has the side effect of eliminating duplicates. `select distinct` e `from` e_1 `as` x_1, ..., e_n `as` x_n is semantically

$$\frac{e : u}{\texttt{<}l : e\texttt{>}^{\texttt{<}l_1:u_1,\cdots,l_n:u_n\texttt{>}} : \texttt{<}l : u,\ l_1 : u_1, \cdots, l_n : u_n\texttt{>}}$$

$$\frac{e : \texttt{<}l_1 : u_1, \cdots, l_n : u_n\texttt{>} \quad e_1 : u \quad \cdots \quad e_n : u}{\begin{array}{c}\texttt{case } e \texttt{ of } \ l_1 \texttt{ then apply function that transforms } x_1^{u_1} \texttt{to } e_1 \\ \texttt{|} \ \cdots \ \texttt{|} \ l_n \texttt{ then apply function that transforms } x_n^{u_n} \texttt{ to } e_n : u\end{array}}$$

$$\frac{e : \texttt{<}l_1 : u_1, \cdots, l_{n+m} : u_{n+m}\texttt{>} \quad e_1 : u \quad \cdots \quad e_n : u \quad e' : u}{\begin{array}{c}\texttt{case } e \texttt{ of } l_1 \texttt{ then apply function that transforms } x_1^{u_1} \texttt{to } e_1 \ \texttt{|} \ \cdots \\ \texttt{|} \ l_n \texttt{ then apply function that transforms } x_n^{u_n} \texttt{ to } e_n \texttt{ else } e' : u\end{array}}$$

Figure 3: The constructs for variant types in sSQL.

$$\frac{}{\texttt{true} : \texttt{bool}} \qquad \frac{}{\texttt{false} : \texttt{bool}} \qquad \frac{e_1 : \texttt{bool} \quad e_2 : u \quad e_3 : u}{\texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3 : u}$$

Figure 4: The constructs for the Boolean type in sSQL.

equal to `select` e `from` e_1 `as` x_1, ..., e_n `as` x_n, but has the side effect of eliminating duplicates. `select` e `from` e_1 `as` x_1, ..., e_n `as` x_n `where` e' `order by` l_1 O_1, ..., l_k O_k is semantically equal to `select` e `from` e_1 `as` x_1, ..., e_n `as` x_n `where` e', but has the side effect of sorting the set according to fields l_1 (most important) to l_m (least important), a field l_i is sorted ascendingly if O_i is `asc` and is sorted descendingly if O_i is `desc`. `select distinct` e `from` e_1 `as` x_1, ..., e_n `as` x_n `where` e' `order by` l_1 O_1, ..., l_k O_k is same as `select` e `from` e_1 `as` x_1, ..., e_n `as` x_n `where` e' `order by` l_1 O_1, ..., l_k O_k, but has the additional side effect of eliminating duplicates. `select` e `from` e_1 `as` x_1, ..., e_n `as` x_n `order by` l_1 O_1, ..., l_k O_k is same as `select` e `from` e_1 `as` x_1, ..., e_n `as` x_n `where true order by` l_1 O_1, ..., l_k O_k. `select distinct` e `from` e_1 `as` x_1, ..., e_n `as` x_n `order by` l_1 O_1, ..., l_k O_k is same as `select distinct` e `from` e_1 `as` x_1, ..., e_n `as` x_n `where true order by` l_1 O_1, ..., l_k O_k.

For bag types, the expression constructs are given in Figure 6. The meaning of $\mathtt{b}\{e_1, \ldots, e_n\}^s$ is the bag containing exactly $e_1, \ldots, e_n$ as elements. The meaning of e_1`bunion` e_2 is the bag union of e_1 and e_2. The numerous `bselect-from-where` constructs are based on SQL. We give their definitions in terms of CPL's comprehension syntax. `bselect` e `from` e_1 `as` x_1, ..., e_n `as` x_n `where` e' is the same bag as given by the CPL expression `{|`e | \\$x_1$ `<--` e_1, ..., \\$x_n$ `<--` e_n, e'`|}`. `bselect` e `from` e_1 `as` x_1, ..., e_n `as` x_n is the same bag as given by the CPL expression `{|`e | \\$x_1$ `<--` e_1, ..., \\$x_n$ `<--` e_n`|}`. `bselect distinct` e `from` e_1 `as` x_1, ..., e_n `as` x_n `where` e' is equal to `bselect` e `from` e_1 `as` x_1, ..., e_n `as` x_n `where` e', with duplicates eliminated. `bselect distinct` e `from` e_1 `as` x_1, ..., e_n `as` x_n is equal to `bselect` e `from` e_1 `as` x_1, ..., e_n `as` x_n, with duplicates eliminated. `bselect` e `from` e_1 `as` x_1, ..., e_n `as` x_n `where` e' `order by` l_1 O_1, ..., l_k O_k is semantically equal to `bselect` e `from` e_1 `as` x_1, ..., e_n `as` x_n `where` e', but has the side effect of sorting the bag according to fields l_1 (most important) to l_m (least important), a field l_i is sorted ascendingly if O_i is `asc` and is sorted descendingly if O_i is `desc`. `bselect distinct` e `from` e_1 `as` x_1, ..., e_n `as` x_n `where` e' `order by` l_1 O_1, ..., l_k O_k is same as `bselect` e `from` e_1 `as` x_1, ..., e_n `as` x_n `where` e' `order by` l_1 O_1, ..., l_k O_k, with duplicates eliminated. `bselect` e `from` e_1 `as` x_1, ..., e_n `as` x_n `order by` l_1 O_1, ..., l_k O_k is same as `bselect` e `from` e_1 `as` x_1, ..., e_n `as` x_n `where true order by` l_1 O_1, ..., l_k O_k. `bselect distinct` e `from` e_1 `as` x_1, ..., e_n `as` x_n `order by` l_1 O_1, ..., l_k O_k is same as `bselect distinct` e `from` e_1 `as` x_1, ..., e_n `as` x_n `where true order by` l_1 O_1, ..., l_k O_k.

For list types, the expression constructs are given in Figure 7. The meaning of $\mathtt{l}\{e_1, \ldots, e_n\}^s$ is the list containing exactly $e_1, \ldots, e_n$ as elements in the given order. The meaning of e_1`lunion` e_2 is the concatenation of e_1 and e_2. The numerous `lselect-from-where` constructs are based on SQL. We give their definitions in terms of CPL's comprehension syntax. `lselect` e `from` e_1 `as` x_1, ..., e_n `as` x_n `where` e' is the same list as given by the CPL expression `[`e | \\$x_1$ `<---` e_1, ..., \\$x_n$ `<---` e_n, e'`]`. `lselect` e `from` e_1 `as` x_1, ..., e_n `as` x_n is the same list as given by the CPL expression `[`e | \\$x_1$ `<---` e_1, ..., \\$x_n$ `<---` e_n`]`. `lselect distinct` e `from` e_1 `as` x_1, ..., e_n `as` x_n `where` e' is equal to `lselect` e `from` e_1 `as` x_1, ..., e_n `as` x_n `where` e', with duplicates eliminated. `lselect distinct` e `from` e_1 `as` x_1, ..., e_n `as` x_n is equal to `lselect` e `from` e_1 `as` x_1, ..., e_n `as` x_n, with duplicates eliminated. `lselect` e `from` e_1 `as` x_1, ..., e_n `as` x_n `where` e' `order by` l_1 O_1, ..., l_k O_k is equal to `lselect` e `from` e_1 `as` x_1, ..., e_n `as` x_n `where` e', but is sorted according to fields l_1 (most important) to l_m (least important), a field l_i is sorted ascendingly if O_i is `asc` and is sorted descendingly if O_i is `desc`. `lselect distinct` e `from` e_1 `as` x_1, ..., e_n `as` x_n `where` e' `order by` l_1 O_1, ..., l_k O_k is same as `lselect` e `from` e_1 `as` x_1, ..., e_n `as` x_n `where` e' `order by` l_1 O_1, ..., l_k O_k, with duplicates eliminated. `lselect` e `from` e_1 `as` x_1, ..., e_n `as` x_n `order by` l_1 O_1, ..., l_k O_k is same as `lselect` e `from` e_1 `as` x_1, ..., e_n `as` x_n `where true order by` l_1 O_1, ..., l_k O_k. `lselect distinct` e `from` e_1 `as` x_1, ..., e_n `as` x_n `order by` l_1 O_1, ..., l_k O_k is same as `lselect distinct` e `from` e_1 `as` x_1, ..., e_n `as` x_n `where true order by` l_1 O_1, ..., l_k O_k.

$$\frac{e_1 : s \ \cdots \ e_n : s}{\{e_1, \ldots, e_n\} : \{s\}} \qquad \frac{e_1 : \{s\} \quad e_2 : \{s\}}{e_1 \texttt{union } e_2 : \{s\}}$$

$$\frac{e : s \quad e' : bool \quad e_1 : \{s_1\} \ \cdots \ e_n : \{s_n\}}{\texttt{select } e \texttt{ from } e_1 \texttt{ as } x_1^{\{s_1\}}, \ \ldots, \ e_n \texttt{ as } x_n^{\{s_n\}} \texttt{ where } e' : \{s\}}$$

$$\frac{e : s \quad e' : bool \quad e_1 : \{s_1\} \ \cdots \ e_n : \{s_n\}}{\texttt{select distinct } e \texttt{ from } e_1 \texttt{ as } x_1^{\{s_1\}}, \ \ldots, \ e_n \texttt{ as } x_n^{\{s_n\}} \texttt{ where } e' : \{s\}}$$

$$\frac{e : s \quad e_1 : \{s_1\} \ \cdots \ e_n : \{s_n\}}{\texttt{select } e \texttt{ from } e_1 \texttt{ as } x_1^{\{s_1\}}, \ \ldots, \ e_n \texttt{ as } x_n^{\{s_n\}} : \{s\}}$$

$$\frac{e : s \quad e_1 : \{s_1\} \ \cdots \ e_n : \{s_n\}}{\texttt{select distinct } e \texttt{ from } e_1 \texttt{ as } x_1^{\{s_1\}}, \ \ldots, \ e_n \texttt{ as } x_n^{\{s_n\}} : \{s\}}$$

$$\frac{\begin{array}{c} e : (l_1 : s_1, \ldots, l_m : s_m) \quad e' : bool \quad e_1 : \{s'_1\} \ \cdots \ e_n : \{s'_n\} \\ k \leq m \quad \forall 1 \leq i \leq k.\ O_i \in \{\texttt{asc}, \texttt{desc}\} \end{array}}{\texttt{select } e \texttt{ from } e_1 \texttt{ as } x_1^{\{s'_1\}}, \ \ldots, \ e_n \texttt{ as } x_n^{\{s'_n\}} \texttt{ where } e' \texttt{ order by } l_1\ O_1, \ldots, \ l_k\ O_k : \{s\}}$$

$$\frac{\begin{array}{c} e : (l_1 : s_1, \ldots, l_m : s_m) \quad e' : bool \quad e_1 : \{s'_1\} \ \cdots \ e_n : \{s'_n\} \\ k \leq m \quad \forall 1 \leq i \leq k.\ O_i \in \{\texttt{asc}, \texttt{desc}\} \end{array}}{\texttt{select distinct } e \texttt{ from } e_1 \texttt{ as } x_1^{\{s'_1\}}, \ \ldots, \ e_n \texttt{ as } x_n^{\{s'_n\}} \texttt{ where } e' \texttt{ order by } l_1\ O_1, \ldots, \ l_k\ O_k : \{s\}}$$

$$\frac{\begin{array}{c} e : (l_1 : s_1, \ldots, l_m : s_m) \quad e_1 : \{s'_1\} \ \cdots \ e_n : \{s'_n\} \\ k \leq m \quad \forall 1 \leq i \leq k.\ O_i \in \{\texttt{asc}, \texttt{desc}\} \end{array}}{\texttt{select } e \texttt{ from } e_1 \texttt{ as } x_1^{\{s'_1\}}, \ \ldots, \ e_n \texttt{ as } x_n^{\{s'_n\}} \texttt{ order by } l_1\ O_1, \ldots, \ l_k\ O_k : \{s\}}$$

$$\frac{\begin{array}{c} e : (l_1 : s_1, \ldots, l_m : s_m) \quad e_1 : \{s'_1\} \ \cdots \ e_n : \{s'_n\} \\ k \leq m \quad \forall 1 \leq i \leq k.\ O_i \in \{\texttt{asc}, \texttt{desc}\} \end{array}}{\texttt{select distinct } e \texttt{ from } e_1 \texttt{ as } x_1^{\{s'_1\}}, \ \ldots, \ e_n \texttt{ as } x_n^{\{s'_n\}} \texttt{ order by } l_1\ O_1, \ldots, \ l_k\ O_k : \{s\}}$$

Figure 5: The constructs for set types in sSQL.

$$\frac{e_1 : s \ \cdots \ e_n : s}{\mathtt{b}\{e_1, \ldots, e_n\} : \{\!|s|\!\}} \qquad \frac{e_1 : \{\!|s|\!\} \quad e_2 : \{\!|s|\!\}}{e_1 \mathtt{bunion}\ e_2 : \{\!|s|\!\}}$$

$$\frac{e : s \quad e' : bool \quad e_1 : \{\!|s_1|\!\} \ \cdots \ e_n : \{\!|s_n|\!\}}{\mathtt{bselect}\ e\ \mathtt{from}\ e_1\ \mathtt{as}\ x_1^{\{\!|s_1|\!\}},\ \ldots,\ e_n\ \mathtt{as}\ x_n^{\{\!|s_n|\!\}}\ \mathtt{where}\ e' : \{\!|s|\!\}}$$

$$\frac{e : s \quad e' : bool \quad e_1 : \{\!|s_1|\!\} \ \cdots \ e_n : \{\!|s_n|\!\}}{\mathtt{bselect\ distinct}\ e\ \mathtt{from}\ e_1\ \mathtt{as}\ x_1^{\{\!|s_1|\!\}},\ \ldots,\ e_n\ \mathtt{as}\ x_n^{\{\!|s_n|\!\}}\ \mathtt{where}\ e' : \{\!|s|\!\}}$$

$$\frac{e : s \quad e_1 : \{\!|s_1|\!\} \ \cdots \ e_n : \{\!|s_n|\!\}}{\mathtt{bselect}\ e\ \mathtt{from}\ e_1\ \mathtt{as}\ x_1^{\{\!|s_1|\!\}},\ \ldots,\ e_n\ \mathtt{as}\ x_n^{\{\!|s_n|\!\}} : \{\!|s|\!\}}$$

$$\frac{e : s \quad e_1 : \{\!|s_1|\!\} \ \cdots \ e_n : \{\!|s_n|\!\}}{\mathtt{bselect\ distinct}\ e\ \mathtt{from}\ e_1\ \mathtt{as}\ x_1^{\{\!|s_1|\!\}},\ \ldots,\ e_n\ \mathtt{as}\ x_n^{\{\!|s_n|\!\}} : \{\!|s|\!\}}$$

$$\frac{\begin{array}{c} e : (l_1 : s_1, \ldots, l_m : s_m) \quad e' : bool \quad e_1 : \{\!|s'_1|\!\} \ \cdots \ e_n : \{\!|s'_n|\!\} \\ k \leq m \quad \forall 1 \leq i \leq k.\ O_i \in \{\mathtt{asc}, \mathtt{desc}\} \end{array}}{\mathtt{bselect}\ e\ \mathtt{from}\ e_1\ \mathtt{as}\ x_1^{\{\!|s'_1|\!\}},\ \ldots,\ e_n\ \mathtt{as}\ x_n^{\{\!|s'_n|\!\}}\ \mathtt{where}\ e'\ \mathtt{order\ by}\ l_1\ O_1, \ldots,\ l_k\ O_k : \{\!|s|\!\}}$$

$$\frac{\begin{array}{c} e : (l_1 : s_1, \ldots, l_m : s_m) \quad e' : bool \quad e_1 : \{\!|s'_1|\!\} \ \cdots \ e_n : \{\!|s'_n|\!\} \\ k \leq m \quad \forall 1 \leq i \leq k.\ O_i \in \{\mathtt{asc}, \mathtt{desc}\} \end{array}}{\mathtt{bselect\ distinct}\ e\ \mathtt{from}\ e_1\ \mathtt{as}\ x_1^{\{\!|s'_1|\!\}},\ \ldots,\ e_n\ \mathtt{as}\ x_n^{\{\!|s'_n|\!\}}\ \mathtt{where}\ e'\ \mathtt{order\ by}\ l_1\ O_1, \ldots,\ l_k\ O_k : \{\!|s|\!\}}$$

$$\frac{\begin{array}{c} e : (l_1 : s_1, \ldots, l_m : s_m) \quad e_1 : \{\!|s'_1|\!\} \ \cdots \ e_n : \{\!|s'_n|\!\} \\ k \leq m \quad \forall 1 \leq i \leq k.\ O_i \in \{\mathtt{asc}, \mathtt{desc}\} \end{array}}{\mathtt{bselect}\ e\ \mathtt{from}\ e_1\ \mathtt{as}\ x_1^{\{\!|s'_1|\!\}},\ \ldots,\ e_n\ \mathtt{as}\ x_n^{\{\!|s'_n|\!\}}\ \mathtt{order\ by}\ l_1\ O_1, \ldots,\ l_k\ O_k : \{\!|s|\!\}}$$

$$\frac{\begin{array}{c} e : (l_1 : s_1, \ldots, l_m : s_m) \quad e_1 : \{\!|s'_1|\!\} \ \cdots \ e_n : \{\!|s'_n|\!\} \\ k \leq m \quad \forall 1 \leq i \leq k.\ O_i \in \{\mathtt{asc}, \mathtt{desc}\} \end{array}}{\mathtt{bselect\ distinct}\ e\ \mathtt{from}\ e_1\ \mathtt{as}\ x_1^{\{\!|s'_1|\!\}},\ \ldots,\ e_n\ \mathtt{as}\ x_n^{\{\!|s'_n|\!\}}\ \mathtt{order\ by}\ l_1\ O_1, \ldots,\ l_k\ O_k : \{\!|s|\!\}}$$

Figure 6: The constructs for bag types in sSQL.

$$\frac{e_1 : s \ \cdots \ e_n : s}{\mathtt{b}\{e_1, \ldots, e_n\} : \mathtt{[}s\mathtt{]}} \qquad \frac{e_1 : \mathtt{[}s\mathtt{]} \quad e_2 : \mathtt{[}s\mathtt{]}}{e_1 \mathtt{lunion}\ e_2 : \mathtt{[}s\mathtt{]}}$$

$$\frac{e : s \quad e' : bool \quad e_1 : \mathtt{[}s_1\mathtt{]} \ \cdots \ e_n : \mathtt{[}s_n\mathtt{]}}{\mathtt{lselect}\ e\ \mathtt{from}\ e_1\ \mathtt{as}\ x_1^{\mathtt{[}s_1\mathtt{]}},\ \ldots,\ e_n\ \mathtt{as}\ x_n^{\mathtt{[}s_n\mathtt{]}}\ \mathtt{where}\ e' : \mathtt{[}s\mathtt{]}}$$

$$\frac{e : s \quad e' : bool \quad e_1 : \mathtt{[}s_1\mathtt{]} \ \cdots \ e_n : \mathtt{[}s_n\mathtt{]}}{\mathtt{lselect\ distinct}\ e\ \mathtt{from}\ e_1\ \mathtt{as}\ x_1^{\mathtt{[}s_1\mathtt{]}},\ \ldots,\ e_n\ \mathtt{as}\ x_n^{\mathtt{[}s_n\mathtt{]}}\ \mathtt{where}\ e' : \mathtt{[}s\mathtt{]}}$$

$$\frac{e : s \quad e_1 : \mathtt{[}s_1\mathtt{]} \ \cdots \ e_n : \mathtt{[}s_n\mathtt{]}}{\mathtt{lselect}\ e\ \mathtt{from}\ e_1\ \mathtt{as}\ x_1^{\mathtt{[}s_1\mathtt{]}},\ \ldots,\ e_n\ \mathtt{as}\ x_n^{\mathtt{[}s_n\mathtt{]}} : \mathtt{[}s\mathtt{]}}$$

$$\frac{e : s \quad e_1 : \mathtt{[}s_1\mathtt{]} \ \cdots \ e_n : \mathtt{[}s_n\mathtt{]}}{\mathtt{lselect\ distinct}\ e\ \mathtt{from}\ e_1\ \mathtt{as}\ x_1^{\mathtt{[}s_1\mathtt{]}},\ \ldots,\ e_n\ \mathtt{as}\ x_n^{\mathtt{[}s_n\mathtt{]}} : \mathtt{[}s\mathtt{]}}$$

$$\frac{\begin{array}{c} e : (l_1 : s_1, \ldots, l_m : s_m) \quad e' : bool \quad e_1 : \mathtt{[}s'_1\mathtt{]} \ \cdots \ e_n : \mathtt{[}s'_n\mathtt{]} \\ k \leq m \quad \forall 1 \leq i \leq k.\ O_i \in \{\mathtt{asc}, \mathtt{desc}\} \end{array}}{\mathtt{lselect}\ e\ \mathtt{from}\ e_1\ \mathtt{as}\ x_1^{\mathtt{[}s'_1\mathtt{]}},\ \ldots,\ e_n\ \mathtt{as}\ x_n^{\mathtt{[}s'_n\mathtt{]}}\ \mathtt{where}\ e'\ \mathtt{order\ by}\ l_1\ O_1, \ldots,\ l_k\ O_k : \mathtt{[}s\mathtt{]}}$$

$$\frac{\begin{array}{c} e : (l_1 : s_1, \ldots, l_m : s_m) \quad e' : bool \quad e_1 : \mathtt{[}s'_1\mathtt{]} \ \cdots \ e_n : \mathtt{[}s'_n\mathtt{]} \\ k \leq m \quad \forall 1 \leq i \leq k.\ O_i \in \{\mathtt{asc}, \mathtt{desc}\} \end{array}}{\mathtt{lselect\ distinct}\ e\ \mathtt{from}\ e_1\ \mathtt{as}\ x_1^{\mathtt{[}s'_1\mathtt{]}},\ \ldots,\ e_n\ \mathtt{as}\ x_n^{\mathtt{[}s'_n\mathtt{]}}\ \mathtt{where}\ e'\ \mathtt{order\ by}\ l_1\ O_1, \ldots,\ l_k\ O_k : \mathtt{[}s\mathtt{]}}$$

$$\frac{\begin{array}{c} e : (l_1 : s_1, \ldots, l_m : s_m) \quad e_1 : \mathtt{[}s'_1\mathtt{]} \ \cdots \ e_n : \mathtt{[}s'_n\mathtt{]} \\ k \leq m \quad \forall 1 \leq i \leq k.\ O_i \in \{\mathtt{asc}, \mathtt{desc}\} \end{array}}{\mathtt{lselect}\ e\ \mathtt{from}\ e_1\ \mathtt{as}\ x_1^{\mathtt{[}s'_1\mathtt{]}},\ \ldots,\ e_n\ \mathtt{as}\ x_n^{\mathtt{[}s'_n\mathtt{]}}\ \mathtt{order\ by}\ l_1\ O_1, \ldots,\ l_k\ O_k : \mathtt{[}s\mathtt{]}}$$

$$\frac{\begin{array}{c} e : (l_1 : s_1, \ldots, l_m : s_m) \quad e_1 : \mathtt{[}s'_1\mathtt{]} \ \cdots \ e_n : \mathtt{[}s'_n\mathtt{]} \\ k \leq m \quad \forall 1 \leq i \leq k.\ O_i \in \{\mathtt{asc}, \mathtt{desc}\} \end{array}}{\mathtt{lselect\ distinct}\ e\ \mathtt{from}\ e_1\ \mathtt{as}\ x_1^{\mathtt{[}s'_1\mathtt{]}},\ \ldots,\ e_n\ \mathtt{as}\ x_n^{\mathtt{[}s'_n\mathtt{]}}\ \mathtt{order\ by}\ l_1\ O_1, \ldots,\ l_k\ O_k : \mathtt{[}s\mathtt{]}}$$

Figure 7: The constructs for list types in sSQL.

3.3 Other Operations and Shorthands

sSQL also includes the usual functions listed in Figure 8 for comparing complex objects.

$$\frac{e_1 : s \quad e_2 : s}{e_1 \texttt{ = } e_2 : \texttt{bool}} \qquad \frac{e_1 : s \quad e_2 : s}{e_1 \texttt{ <= } e_2 : \texttt{bool}} \qquad \frac{e_1 : s \quad e_2 : s}{e_1 \texttt{ < } e_2 : \texttt{bool}}$$

$$\frac{e_1 : s \quad e_2 : s}{e_1 \texttt{ => } e_2 : \texttt{bool}} \qquad \frac{e_1 : s \quad e_2 : s}{e_1 \texttt{>} e_2 : \texttt{bool}}$$

Figure 8: The constructs for comparing objects in sSQL.

sSQL has a construct for sending requests to external data servers. This construct is given in Figure 9. The expression `process` e `using` N has the effect of sending the request e to the server N and the reply from N is taken as the meaning of the whole expression. See *The Kleisli Query System Reference Manual* for more information on the behaviour of servers.

$$\frac{e : s \qquad N \text{ is a server whose request type is } s \text{ and result type is } t}{\texttt{process } e_1 \texttt{ using } N : t}$$

Figure 9: The construct for sending a request to a server in sSQL.

sSQL supports some syntactic sugar on expressions:

- The expression $(e_1, ..., e_n)$ is a shorthand for $(\#1 : e_1, ..., \#n : e_n)$.
- The expression `select` $l_1 : e_1, ..., l_n : e_n$ `from` ... `where` e is a shorthand for `select` $(l_1 : e_1, ..., l_n : e_n)$ `from` ... `where` e. Ditto for other variations of `select-from-where`. Moreover, if any of the l_i does not start with the # sign, the # sign is automatically inserted to so that l_i becomes a label.
- $f(e)$ is a shorthand for `apply` f `to` e, provided f is the name of a primitive.
- $e.f$ is a shorthand for `apply` f `to` e, provided f is the name of a primitive. It is treated as $e.\#f$, if f is not the name of a primitive and is not a label.
- $e_1\ f\ e_2$ is a shorthand for $f(e_1, e_2)$, provided f is the name of a binary primitive.
- `function that transforms` $(l_1 : x_1, ..., l_n : x_n)$ `to` e is a shorthand for `function that transforms` x `to` $e[l_1(x)/x_1, ..., l_n(x)/x_n]$.

- `function that transforms` $(x_1, ..., x_n)$ `to` e is a shorthand for `function that transforms` x `to` $e[\#1(x)/x_1, ..., \#n(x)/x_n]$.

- `select` e_1 `from` e_2 `as` $(l_1 : x_1, ..., l_n : x_n)$ `where` e_3 is a shorthand for `select` $e_1[l_1(x)/x_1, ..., l_n(x)/x_n]$ `from` e_2 `as` x `where` $e_3[l_1(x)/x_1, ..., l_n(x)/x_n]$. Ditto for other variations of `select-from-where`.

- `select` e_1 `from` e_2 `as` $(x_1, ..., x_n)$ `where` e_3 is a shorthand for `select` $e_1[\#1(x)/x_1, ..., \#n(x)/x_n]$ `from` e_2 `as` x `where` $e_3[\#1(x)/x_1, ..., \#n(x)/x_n]$. Ditto for other variations of `select-from-where`.

- `select` e_1 `from` e_2 x `where` e_3 is a shorthand for `select`e_1 `from` e_2 `as` x `where` e_3. Ditto for other variations of `select-from-where`.

- `select` * `from` e_2 x `where` e_3 is a shorthand for `select` x `from` e_2 x `where` e_3. Ditto for other variations of `select-from-where`.

3.4 Commands

Besides executing expressions, you can also run the following types of commands at the top level of sSQL.

- `create view` N `as` e

  This is same as executing `primitive` N `==` e at the top level of CPL. It defines N as a "macro" for the expression e.

- `create view` N `from` e `using` M

  This is same as executing `readfile` N `from` e `using` M at the top level of CPL. It defines N as a "macro" for the contents of a "file" e read using the scanner M.

- `create function` F x `as` e

  This is same as executing `primitive` F `==` $\backslash x$ `=>` e at the top level of CPL. It defines F as a function that transforms x to e. Note that we can use the same sort of pattern for x, just as in `function that transforms`x `to` e.

- `create file` e_1 `using` M `as` e_2

  This is same as executing `writefile` e_2 `to` e_1 `using` M in CPL. It writes e_2 to a "file" e_1 using the writer M.

- `select` e_1 `into` e_2 `from` e_3 `where` e_4 `using` M

  This is same as executing `writefile select` e_1 `from` e_3 `where` e_4 `to` e_2 `using` M`-write`. Ditto for other variations of `select-from-where`.

- `create table` $t(l_1 : e_1, \ldots, l_n : e_n)$ `using` M

  This is same as excuting M`-mktable (#table:` t, `#schema:` $(l_1 : e_1, \ldots, l_n : e_n)$ `)`.

- `drop table` t `using` M.

  This is same as executing M-`droptable` t.

- `drop index` t `using` M

  This is same as executing M-`dropindex` t.

- and many more that mirrors SQL update commands in the obvious way.

4 Example Bioinformatics Queries

We provide four examples of bioinformatics queries written in sSQL to close our presentation.

4.1 Proline at N-Terminal

The first of our four examples is this query: *What proportion of mature peptides from human have prolines at their N-terminal?* Its implementation in sSQL is given below.

```
create view mat-peptides as
  select string-span (x.#sequence, f.#start, f.#end)
  from aa-get-seqfeat-general "mature AND peptide AND human[Organism]" as x,
       x.#feature as f
  where f.#name string-islike "%mat_peptide%";
materialize "mat-peptides";

create view prolines as
  select x
  from mat-peptides as x
  where x string-islike "P%";

set-roughcount (prolines) / set-roughcount (mat-peptides);
```

It basically consists of three steps.

1. We create a view `mat-peptides` which is a sSQL that accesses Entrez to obtain GenPept reports of human amino acid sequences of mature peptides. We also materialize this view so that Entrez is not accessed twice.
2. We create a view `prolines` as a subset of the view `mat-peptides` that start with proline.
3. Finally, we just count the number of items in the set `prolines` and the number of items in the set `mat-peptides`. Their ratio gives the desired answer.

4.2 TPR Domains in Swissprot Sequences

It is not unusual for a biologist to "hunt" for a particular domain in large sequence databases. The second of our four examples is typical of domain hunting: *Build a hidden Markov model [5] of TPR domains [9] and use it to recognize TPR domains in Swissprot sequences.*

```
create file "tpr.hmm" using hmm-localtraining as
  select #uid: r.#uid, #title: r.#title, #accession: r.#accession,
         #seq: string-span (r.#sequence, f.#start, f.#end)
```

```
  from aa-get-seqfeat-general "TPR" as r,
       r.#feature as f,
       l2s (f.#anno) as d
  where ((f.#end - f.#start) < 40) andalso
        ((f.#end - f.#start) > 25) andalso
        ((f.#name string-islike "%mat_peptide%") orelse
         (f.#name string-islike "%region%")) andalso
        (d.#descr string-islike "%TPR%") andalso
        (not (d.#descr string-islike "%putative%")) andalso
        (not (d.#descr string-islike "%hypothetical%"));

hmm-localsearch (
  #name: "hmm-tpr", #model: "tpr.hmm", #threshold: 0,
  #level: 1, #dna: false, #window: 1000, #best: false);

create view sprot from "sprot36.fa" using fasta-in;

select #title: x.#title, #tpr-start: r.#seqstart, #tpr-end: r.#seqend
from sprot as x,
     process <#hmmfs: x.#seq> using hmm-tpr as h,
     h.#hits as r
where r.#score > 20;
```

This implementation has four steps.

1. We extract known examples of TPR domains and construct a hidden Markov model (HMM) "tpr.hmm" for it.
2. A HMMER[11] connection `hmm-tpr` is established to the hidden Markov Model `tpr.hmm` using standard parameters.
3. The view `sprot` is set to the contents of the file `sprot36.fa`, which is the FASTA version of our copy of Swissprot.
4. We iterate over every sequence in `sprot`. For each sequence `x`, we check it using our HMMER connection `hmm-tpr`. We examine the reply `h` to see if it contains a hit `r` with a score greater than the threshold 20. If so, we include the title of the sequence `x` and the start and end positions of the hit `r` in the output set as desired.

4.3 Patented Proteins in Superfamily

Assessing the patent potential of a sequence is of practical importance. The third of our four examples is an interesting query on this subject: *Suppose we are given a protein sequence* `SEQ`*. List patented sequences that belong to the same superfamily as* `SEQ`*.* By a superfamily, we mean sequences that have the same tertiary structure and the same function, but they do not have to have similar sequences.

The implementation in sSQL is given below. A full exposition to the complement of this question—list unpatented sequences in the same superfamily as SEQ—was given in [2].

```
localblast-blastp (#name: "scop-blast", #db: "scopseq");
localblast-blastp (#name: "pat-blast", #db: "patseq");
setindex-access (#name: "sid2seq", #file: "scopseq", #key: "#sid");
scop-add "scop";

select #sf: (#desc: xinfo.#desc.#sf, #hit: x.#accession, #pscore: x.#pscore),
       #bridge: (#hit: s, #patent: p.#title, #pscore: p.#pscore)
from process SEQ using scop-blast as x,
     process <#sidinfo: x.#accession> using scop as xinfo,
     process <#numsid: xinfo.#type.#sf> using scop as s,
     process <#key: s> using sid2seq as y,
     process y.#seq using pat-blast as p
where (x.#pscore <= 1E~6) andalso (p.#pscore <= 1E~6);
```

The implementation consists of two steps.

1. We need to use SCOP[10] to obtain superfamily classification. We also need access to patented sequences (we got them in the file `patseq`) and to representative sequences for the various superfamilies in SCOP (we got them in the file `scopseq`). So we establish a WU-BLAST2[1] connection `scop-blast` to `scopseq`, a WU-BLAST2 connection `pat-blast` to `patseq`, and a connection `scop` to SCOP. To provide random access to representative sequences of SCOP superfamilies given their SCOP identifier (`sid`), we also establish the database index `sid2seq`. Having set all these connections up, we are ready to do the real work.

2. We BLAST the given sequence `SEQ` against SCOP superfamily representative sequences to look for a hit `x` at a strong threshold (1^{-6}). `SEQ` is regarded to be in the same superfamily as `x`. We look up `x`'s record from SCOP. From this record, we ask SCOP for all the identifiers of representatives in the superfamily of `x`. Each representative `s` serves as a "bridge" to help us identify the desired patented sequences. For each such representative `s`, we use our index to look up its sequence `y`. We BLAST this sequence `y` against the patented sequences; those hits at the strong threshold (1^{-6}) are the patented sequences we want. The relevant information (the superfamily information of `x`, the bridge `s`, and the patent information `p`) are returned in the output set.

4.4 Compact Genome of Pufferfish

The small size of the pufferfish genome and the close correspondence of pufferfish genes exon-intron structures to homologous human genes make it an ideal shortcut to understanding the human genome[6]. Our last example is: *Given a file of pufferfish (expressed) DNA fragments, identify those that are "interesting."* A fragment is considered interesting if it is not known to be homologous to any known vertebrate proteins and if it has good homology to some human or mouse ESTs. Such a fragment is interesting because it is likely to be part of a pufferfish gene that is homologous to a human or mouse gene that is not yet sequenced. Here is the implementation in sSQL.

```
create file "vertebrates"  using stdout as
  select #uid: x
  from aa-get-uid-general "vertebratea[Organism]" as x;
create file "#uid" using setindex-create as "vertebrates";
setindex-access (#name: "vertebrates", #file: "vertebrates", #key: "#uid");
webblast2-blastx-detail (#name: "nr-blast", #db: "nr", #level: 1);
localblast-tblastn (#name: "est-blast", #db: "man-mouse-est");

create view fugu from "fugu" using stdin;

create file "protein" using stdout as
  select #fugu: f, #homologs: B
  from fugu as f,
       (select #title: x.#title, #uid: x.#uid,
               #pscore: x.#hits.list-head.#pscore
        from process f.#seq using nr-blast as x
        where x.#hits.list-head.#pscore < 1E~6) as B;

create view proteins from "proteins" using stdin;
create file "bridges" using stdout as
  select #fugu: x.#fugu, #bridge: b
  from proteins as x
       x.#homologs as b
  where set-isempty (
          select z
          from x.#homologs as y,
               process <#key: y.#uid> using vertebrates as z);

create view bridges from "bridges" using stdin;
create file "ests" using stdout as
  select #bridge: b, #est: e
  from (select distinct x.#bridge.#uid from bridges as x) as b,
       aa-get-seq-by-uid (b.#uid) as s,
       (select #uid: y.#uid, #title: y.#title, #pscore: y.#pscore
        from process s.#sequence using est-blast as y
        where y.#pscore < 1E~30) as e;
create file "#bridge" using setindex-create as "ests";
setindex-access (#name: "ests", #file: "ests", #key: "#bridge");

select #fugu: x.#fugu, #bridge: x.#bridge,
       #ests: (select e.#est from E as a)
from bridges as x,
     { process x.#bridge using ests} as E
where not(set-isempty (E));
```

The implementation consists of five steps.

1. To determine if a pufferfish DNA fragment is similar to any known proteins, we set up a Gapped BLAST2 connection `nr-blast` to the nonredundant protein sequence database at the National Centre of Biotechnology Information. To determine if a protein is similar to any human or mouse EST, we set up a local WU-BLAST2 connection `est-blast` to our private copy of human and mouse ESTs. To determine if a protein hit by `nr-blast` is vertebrate or not, we copy from Entrez all the identifiers of vertebrate amino acid sequences into a local file. In order to speed up access to this file of vertebrate protein identifiers, we also create an index `vertebrates` on this file.

2. The pufferfish DNA fragments to be tested are given in the FASTA file `fugu`, which is now read in. We iterate over each DNA fragment `f` in the file. For each `f`, we BLAST it using `nr-blast`, keeping only those hits at a strong threshold (1^{-6}) in the set `B`. This pufferfish DNA fragment and the corresponding set of protein homologs `B` are written to a file.

3. We examine this file, as we wish only to keep those DNA fragments whose protein homologs are strictly nonvertebrate. For each DNA fragment `x`, we test each of its protein homologs against our vertebrate index to make sure they are not in the index. If so, then each such protein homolog `b` is considered a "bridge" for the DNA fragment `f` and is recorded in the file `bridges`.

4. We examine this file of "bridges", as we wish to find out those ESTs of human and mouse hit by them. For each "bridge" protein `b`, we get its sequence `s` from Entrez. Then we BLAST this sequence using est-blast and keep those hits at a strong threshold (1^{-30}). Each such hit `e` and the corresponding "bridge" `b` are written to a file. To allow very fast access to the EST hits of a "bridge", we also build an index on this file.

5. All that remains to be done is to iterate over every pufferfish DNA fragment-protein "bridge" pair `x` in the file `bridges`, pull out the already computed EST hits of the "bridge" component of `x` and make sure it is nonempty, extract the relevant portion of these EST hits, and return the relevant information in the output set.

References

[1] S. F. Altschul and W. Gish. Local alignment statistics. *Methods in Enzymology*, 266:460–480, 1996.

[2] Jing Chen, Louxin Zhang, and Limsoon Wong. A protein patent query system powered by Kleisli. In *Proceedings of ACM SIGMOD International Conference on Management of Data*, pages 593–595, Seattle, Washington, June 1998.

[3] C. J. Date. A critique of the SQL database language. *SIGMOD Record*, 14(3):8–52, November 1984.

[4] C. J. Date. Some principles of good language design. *SIGMOD Record*, 14(3):1–7, November 1984.

[5] S.R. Eddy. Hidden Markov models. *Current Opinion in Structural Biology*, 6:361–365, 1996.

[6] Greg Elgar. Quality not quantitiy: The pufferfish genome. *Human Molecular Genetics*, 5:1437–1442, 1996.

[7] Carl A. Gunter. *Semantics of Programming Languages: Structures and Techniques.* Foundations of Computing. MIT Press, 1992.

[8] Richard Hull and Chee K. Yap. The Format model: A theory of database organisation. *Journal of the ACM*, 31(3):518–537, July 1984.

[9] John R. Lamb, Stuart Tugendreich, and Phil Hieter. Tetratrico peptide repeat interactions: to TPR or not to TPR? *Trends in Biochemical Sciences*, 20(7):257–259, July 1995.

[10] A. Murzin, S. E. Brenner, T. Hubbard, and C. Chothia. SCOP: A structural classification of protein database for the investigation of sequences and structures. *Journal of Molecular Biology*, 247:536–540, 1995.

[11] E. L. Sonnhammer, S. R. Eddy, E. Birney, A. Bateman, and R. Durbin. Pfam: Multiple sequence alignments and HMM-profiles of protein domains. *Nucleic Acids Research*, 26(1):320–322, January 1998.

Appendices

A Switching Interpreters

You can switch between the sSQL, CPL, and SML/NJ interpreters. To switch from the sSQL interpreter to the SML/NJ interpreter, you just type control-D at the sSQL prompt. To switch from the CPL interpreter to the SML/NJ interpreter, you just type control-D at the CPL prompt. To switch from the SML/NJ interpreter to the sSQL interpreter, you just type `sSQLControl.RunsSQL();`. To switch from the SML/NJ interpreter to the CPL interpreter, you just type `Control.RunCPL();`.

B Lexical Conventions

This section describes some of the conventions used by the CPL parsers. A few words of general advice: Use white space liberally — it makes programs easier to read and it reduces parsing errors.

B.1 Variables Names

sSQL is extremely liberal when it comes to accepting names of variables. The following are all perfectly acceptable variable names: `X`, `2*x`, `2x-y^3`, `@!$%^'‘`, etc. Note that there is no requirement that the name of a variable must start with a letter. Thus just about anything can appear as part of the name of a variable. The exceptions are white space, brackets, and string quotes.

B.2 Label Names

sSQL is extremely liberal when it comes to accepting names of labels. The only real requirement is that it start with the `#` sign. The following are all perfectly acceptable variable names: `#X`, `#2*x`, `#2x-y^3`, `#@!$%^'‘`, etc. Thus just about anything can appear as part of the name of a variable. The exceptions are white space, brackets, and string quotes.

B.3 Numbers

sSQL uses `~` as the minus sign.

B.4 Comments

The start of a comment is indicated by a bang `!`, followed by a space. A comment extends all the way until the end-of-line character.

B.5 Reserved Words

Here are the reserved words and symbols in sSQL.

```
create view as from using function file case of modify in
| error process that transforms to apply if then else true false
{ } b{ l{ union bunion lunion select where distinct bselect lselect
order by asc desc ; ( ) _ .  = <= >= < > : ,
```

B.6 Precedence and Associativity

All binary operators, including user-defined macros used in infixed form, associate to the left. User-defined macros are lowest in precedence. The precedence of the other operators are listed below in increasing strength, with those having the same strength grouped into the same line.

```
= < <= > >=
union bunion lunion
.
```

B.7 Embedded ML Codes

You can embed ML codes into a sSQL script also. The start of a piece of embedded ML codes is indicated by a semi-colon, followed by a newline, followed by `! beginsml`, followed by a space, followed by a newline. The end of a piece of embedded ML codes is indicated by a newline, followed by `! endsml`, followed by a space, followed by a newline, followed by a semi-colon.

B.8 Embedded CPL Codes

You can embed CPL codes into a sSQL script also. The start of a piece of embedded ML codes is indicated by a semi-colon, followed by a newline, followed by `! begincpl`, followed by a space, followed by a newline. The end of a piece of embedded ML codes is indicated by a newline, followed by `! endcpl`, followed by a space, followed by a newline, followed by a semi-colon.

C Available Data Drivers

All data drivers available in CPL are available in sSQL in exactly the same way.

D Available External Primitives

All external primitives available in CPL are available in sSQL in exactly the same way.

E Available System Operations

All system operations available in CPL are available in sSQL in exactly the same way.

F SSQL Grammar

```
TOP
 : COMMAND (COMMAND)


COMMAND
 : EXPR
   (sSQLRare.Exec (LegalE EXPR))

 | CREATE VIEW SYMBOL AS EXPR
   (sSQLRare.Macro (SYMBOL, LegalE EXPR))

 | CREATE VIEW SYMBOL FROM EXPR USING SYMBOL
   (let
      val E = chooseName EXPR
      val S = (case Server.Peek SYMBOL2
        of SOME SvNo => SYMBOL2
        | NONE => SYMBOL2 ^ "-read")
    in sSQLRare.ReadFile (SYMBOL1, E, S) end)

 | CREATE FUNCTION SYMBOL EXPR AS EXPR
   (sSQLRare.Macro (SYMBOL, LegalE (sSQLRare.Fun (EXPR1, EXPR2))))

 | CREATE FILE EXPR USING SYMBOL AS EXPR
   (sSQLRare.WriteFile (LegalE EXPR2, LegalE EXPR1, SYMBOL))

 | CREATE FILE EXPR AS EXPR USING SYMBOL
   (sSQLRare.WriteFile (LegalE EXPR2, LegalE EXPR1, SYMBOL))

 | SELECT eXPR INTO EXPR FROM TABLES WHERE EXPR USING SYMBOL
   (mkWriteFile
      sSQLRare.Select
      eXPR EXPR1 TABLES EXPR2 SYMBOL
      (fn x => x))

 | SELECT DISTINCT eXPR INTO EXPR FROM TABLES WHERE EXPR USING SYMBOL
   (mkWriteFile
      sSQLRare.SelectDistinct
      eXPR EXPR1 TABLES EXPR2 SYMBOL
      (fn x => x))

 | SELECT eXPR INTO EXPR FROM TABLES USING SYMBOL
   (mkWriteFile
      sSQLRare.Select
      eXPR EXPR TABLES (sSQLRare.Bool true) SYMBOL
      (fn x => x))

 | SELECT DISTINCT eXPR INTO EXPR FROM TABLES USING SYMBOL
   (mkWriteFile
```

```
    sSQLRare.SelectDistinct
    eXPR EXPR TABLES (sSQLRare.Bool true) SYMBOL
    (fn x => x))

| SELECT eXPR INTO EXPR FROM TABLES WHERE EXPR ORDER BY KEYS USING SYMBOL
  (mkWriteFile
    sSQLRare.Select
    eXPR EXPR1 TABLES EXPR2 SYMBOL
    (fn C => sSQLRare.Sort(C, KEYS)))

| SELECT DISTINCT eXPR INTO EXPR FROM TABLES WHERE EXPR ORDER BY KEYS USING SYMBOL
  (mkWriteFile
    sSQLRare.SelectDistinct
    eXPR EXPR1 TABLES EXPR2 SYMBOL
    (fn C => sSQLRare.Sort(C, KEYS)))

| SELECT eXPR INTO EXPR FROM TABLES ORDER BY KEYS USING SYMBOL
  (mkWriteFile
    sSQLRare.Select
    eXPR EXPR TABLES (sSQLRare.Bool true) SYMBOL
    (fn C => sSQLRare.Sort(C, KEYS)))

| SELECT DISTINCT eXPR INTO EXPR FROM TABLES ORDER BY KEYS USING SYMBOL
  (mkWriteFile
    sSQLRare.SelectDistinct
    eXPR EXPR TABLES (sSQLRare.Bool true) SYMBOL
    (fn C => sSQLRare.Sort(C, KEYS)))

| CREATE TABLEDEF SPECEXPR USING SYMBOL
  (let
    val (EXPR1, EXPR2) = SPECEXPR
    val E = chooseName EXPR1
    val D = (case Server.Peek SYMBOL
      of SOME SvNo => SYMBOL
      | NONE => SYMBOL ^ "-mktable")
   in sSQLRare.Exec (sSQLRare.Scanner (D, sSQLRare.Record [
        ("#table", E), ("#schema", EXPR2)]))
   end)

| DROP TABLEDEF EXPR USING SYMBOL
  (let
    val E = chooseName EXPR
    val D = (case Server.Peek SYMBOL
      of SOME SvNo => SYMBOL
      | NONE => SYMBOL ^ "-droptable")
   in sSQLRare.Exec (sSQLRare.Scanner (D, E)) end)

| DROP INDEX EXPR USING SYMBOL
  (let
    val E = chooseName EXPR
    val D = (case Server.Peek SYMBOL
```

```
      of SOME SvNo => SYMBOL
      | NONE => SYMBOL ^ "-dropindex")
   in sSQLRare.Exec (sSQLRare.Scanner (D, E)) end)

| INSERT INTO SPECEXPR USING SYMBOL
  (let
     val (EXPR1, EXPR2) = SPECEXPR
     val E = chooseName EXPR1
     val D = (case Server.Peek SYMBOL
       of SOME SvNo => SYMBOL
       | NONE => SYMBOL ^ "-insert")
     fun b2s B = sSQLRare.Apply (sSQLRare.Symbol "b2s", B)
     fun l2s L = sSQLRare.Apply (sSQLRare.Symbol "l2s", L)
     val cvt = (case EXPR2
       of sSQLRare.Select _ => (fn x => x)
       | sSQLRare.SelectDistinct _ => (fn x => x)
       | sSQLRare.SelectB _ => b2s
       | sSQLRare.SelectBDistinct _ => b2s
       | sSQLRare.SelectL _ => l2s
       | sSQLRare.SelectLDistinct _ => l2s
       | _ => Error.ReportFatal (
            "sSQL.grm: Expecting a select-from-where in insert-into."))
     val U = sSQLRare.Select (
       sSQLRare.Symbol "1y",
       [("1x", cvt EXPR2),
        ("1y", sSQLRare.Scanner (D, sSQLRare.Record [
              ("#table", E), ("#replacement", sSQLRare.Symbol "1x")]))],
       sSQLRare.Bool true)
   in sSQLRare.Exec U end)

| INSERT INTO EXPR VALUES EXPR USING SYMBOL
  (let
     val E = chooseName EXPR1
     val D = (case Server.Peek SYMBOL
       of SOME SvNo => SYMBOL
       | NONE => SYMBOL ^ "-insert")
     val U = sSQLRare.Scanner (D, sSQLRare.Record [
             ("#table", E), ("#replacement", EXPR2)])
   in sSQLRare.Exec U end)

| DELETE FROM SPECEXPR USING SYMBOL
  (let
     val (EXPR1, EXPR2) = SPECEXPR
     val E = chooseName EXPR1
     val D = (case Server.Peek SYMBOL
       of SOME SvNo => SYMBOL
       | NONE => SYMBOL ^ "-delete")
     fun b2s B = sSQLRare.Apply (sSQLRare.Symbol "b2s", B)
     fun l2s L = sSQLRare.Apply (sSQLRare.Symbol "l2s", L)
     val cvt = (case EXPR2
       of sSQLRare.Select _ => (fn x => x)
```

```
      | sSQLRare.SelectDistinct _ => (fn x => x)
      | sSQLRare.SelectB _ => b2s
      | sSQLRare.SelectBDistinct _ => b2s
      | sSQLRare.SelectL _ => l2s
      | sSQLRare.SelectLDistinct _ => l2s
      | _ => Error.ReportFatal (
           "sSQL.grm: Expecting a select-from-where in delete-from."))
    val U = sSQLRare.Select (
      sSQLRare.Symbol "1y",
      [("1x", cvt EXPR2),
       ("1y", sSQLRare.Scanner (D, sSQLRare.Record [
             ("#table", E), ("#selector", sSQLRare.Symbol "1x")]))],
      sSQLRare.Bool true)
   in sSQLRare.Exec U end)

| UPDATE EXPR SET SPECEXPR USING SYMBOL
  (let
    val (EXPR2, EXPR3) = SPECEXPR
    val E = chooseName EXPR1
    val D = (case Server.Peek SYMBOL
      of SOME SvNo => SYMBOL
      | NONE => SYMBOL ^ "-update")
    fun mkSpec (S, F, W) = (
      sSQLRare.Record [
        ("#table", E),
        ("#selector", S),
        ("#replacement", EXPR2)],
      F, W)
    fun b2s B = sSQLRare.Apply (sSQLRare.Symbol "b2s", B)
    fun l2s L = sSQLRare.Apply (sSQLRare.Symbol "l2s", L)
    val cvt = (case EXPR3
      of sSQLRare.Select SFW => sSQLRare.Select (mkSpec SFW)
      | sSQLRare.SelectDistinct SFW => sSQLRare.SelectDistinct (mkSpec SFW)
      | sSQLRare.SelectB SFW => b2s (sSQLRare.SelectB (mkSpec SFW))
      | sSQLRare.SelectBDistinct SFW =>
          b2s (sSQLRare.SelectBDistinct (mkSpec SFW))
      | sSQLRare.SelectL SFW => l2s (sSQLRare.SelectL (mkSpec SFW))
      | sSQLRare.SelectLDistinct SFW =>
          l2s (sSQLRare.SelectLDistinct (mkSpec SFW))
      | _ => Error.ReportFatal (
           "sSQL.grm: Expecting a select-from-where in update-set."))
    val U = sSQLRare.Select (
      sSQLRare.Symbol "1y",
      [("1x", cvt),
       ("1y", sSQLRare.Scanner (D, sSQLRare.Symbol "1x"))],
      sSQLRare.Bool true)
   in sSQLRare.Exec U end)

| SEMICOLON COMMAND
  (COMMAND)
```

```
 | %prec SEMICOLON
   (sSQLRare.Quit)

 | SMLPROG
   (sSQLRare.EmbeddedSML SMLPROG)

 | CPLPROG
   (sSQLRare.EmbeddedCPL CPLPROG)


EXPR
 : UNIT
   (sSQLRare.Unit)

 | INT
   (sSQLRare.Integer INT)

 | BOOL
   (sSQLRare.Bool BOOL)

 | REAL
   (sSQLRare.Real REAL)

 | CHARS
   (sSQLRare.String CHARS)

 | SYMBOL
   (sSQLRare.Symbol SYMBOL)

 | UNDERSCORE
  (sSQLRare.DontCare)

 | ERROR EXPR
   (sSQLRare.Error EXPR)

 | LBRAC EXPR RBRAC
   (EXPR)

 | SCANNER EXPR USING SYMBOL
   (sSQLRare.Scanner (SYMBOL, EXPR))

 | FUNCTION THAT TRANSFORMS EXPR TO EXPR
   (sSQLRare.Fun (EXPR1, EXPR2))

 | EXPR EQUAL EXPR
   (sSQLRare.Eq (EXPR1, EXPR2))

 | EXPR LANGLEEQUAL EXPR
   (sSQLRare.Leq (EXPR1, EXPR2))

 | EXPR RANGLEEQUAL EXPR
```

```
  (sSQLRare.Leq (EXPR2, EXPR1))

| EXPR LANGLE EXPR
  (sSQLRare.Lt (EXPR1, EXPR2))

| EXPR RANGLE EXPR
  (sSQLRare.Lt (EXPR2, EXPR1))

| SYMBOL EXPR
  (sSQLRare.Apply (sSQLRare.Symbol SYMBOL, EXPR))

| EXPR SYMBOL EXPR
  (sSQLRare.Apply (sSQLRare.Symbol SYMBOL, sSQLRare.Tuple [EXPR1, EXPR2]))

| EXPR DOT EXPR
 (sSQLRare.Apply (EXPR2, EXPR1))

| APPLY EXPR TO EXPR
 (sSQLRare.Apply (EXPR1, EXPR2))

| IF EXPR THEN EXPR ELSE EXPR
  (sSQLRare.IfThenElse (EXPR1, EXPR2, EXPR3))

| LCURLY EXPR_LIST RCURLY
  (sSQLRare.Set EXPR_LIST)

| LCURLYB EXPR_LIST RCURLY
  (sSQLRare.Bag EXPR_LIST)

| LCURLYL EXPR_LIST RCURLY
  (sSQLRare.List EXPR_LIST)

| SELECT eXPR FROM TABLES WHERE EXPR
  (sSQLRare.Select (dealWithStar (eXPR, TABLES, EXPR)))

| SELECT DISTINCT eXPR FROM TABLES WHERE EXPR
  (sSQLRare.SelectDistinct (dealWithStar (eXPR, TABLES, EXPR)))

| SELECT eXPR FROM TABLES
  (sSQLRare.Select (dealWithStar (eXPR, TABLES, sSQLRare.Bool true)))

| SELECT DISTINCT eXPR FROM TABLES
  (sSQLRare.SelectDistinct (dealWithStar (eXPR, TABLES, sSQLRare.Bool true)))

| SELECT eXPR FROM TABLES WHERE EXPR ORDER BY KEYS
  (sSQLRare.Sort (sSQLRare.Select (dealWithStar (eXPR, TABLES, EXPR)), KEYS))

| SELECT DISTINCT eXPR FROM TABLES WHERE EXPR ORDER BY KEYS
  (sSQLRare.Sort (
     sSQLRare.SelectDistinct (dealWithStar (eXPR, TABLES, EXPR)),
     KEYS))
```

```
| SELECT eXPR FROM TABLES ORDER BY KEYS
  (sSQLRare.Sort (
     sSQLRare.Select (dealWithStar (eXPR, TABLES, sSQLRare.Bool true)),
     KEYS))

| SELECT DISTINCT eXPR FROM TABLES ORDER BY KEYS
  (sSQLRare.Sort(
     sSQLRare.SelectDistinct(dealWithStar(eXPR, TABLES, sSQLRare.Bool true)),
     KEYS))

| SELECTB eXPR FROM TABLES WHERE EXPR
  (sSQLRare.SelectB (dealWithStar (eXPR, TABLES, EXPR)))

| SELECTB DISTINCT eXPR FROM TABLES WHERE EXPR
  (sSQLRare.SelectBDistinct (dealWithStar (eXPR, TABLES, EXPR)))

| SELECTB eXPR FROM TABLES
  (sSQLRare.SelectB (dealWithStar (eXPR, TABLES, sSQLRare.Bool true)))

| SELECTB DISTINCT eXPR FROM TABLES
  (sSQLRare.SelectBDistinct (dealWithStar (eXPR, TABLES, sSQLRare.Bool true)))

| SELECTB eXPR FROM TABLES WHERE EXPR ORDER BY KEYS
  (sSQLRare.Sort (sSQLRare.SelectB (dealWithStar (eXPR, TABLES, EXPR)), KEYS))

| SELECTB DISTINCT eXPR FROM TABLES WHERE EXPR ORDER BY KEYS
  (sSQLRare.Sort (
     sSQLRare.SelectBDistinct (dealWithStar (eXPR, TABLES, EXPR)),
     KEYS))

| SELECTB eXPR FROM TABLES ORDER BY KEYS
  (sSQLRare.Sort (
     sSQLRare.SelectB (dealWithStar (eXPR, TABLES, sSQLRare.Bool true)),
     KEYS))

| SELECTB DISTINCT eXPR FROM TABLES ORDER BY KEYS
  (sSQLRare.Sort(
     sSQLRare.SelectBDistinct (dealWithStar (eXPR,TABLES,sSQLRare.Bool true)),
     KEYS))

| SELECTL eXPR FROM TABLES WHERE EXPR
  (sSQLRare.SelectL (dealWithStar (eXPR, TABLES, EXPR)))

| SELECTL DISTINCT eXPR FROM TABLES WHERE EXPR
  (sSQLRare.SelectLDistinct (dealWithStar (eXPR, TABLES, EXPR)))

| SELECTL eXPR FROM TABLES
  (sSQLRare.SelectL (dealWithStar (eXPR, TABLES, sSQLRare.Bool true)))

| SELECTL DISTINCT eXPR FROM TABLES
```

```
  (sSQLRare.SelectLDistinct (dealWithStar (eXPR, TABLES, sSQLRare.Bool true)))

| SELECTL eXPR FROM TABLES WHERE EXPR ORDER BY KEYS
  (sSQLRare.Sort (sSQLRare.SelectL (dealWithStar (eXPR, TABLES, EXPR)), KEYS))

| SELECTL DISTINCT eXPR FROM TABLES WHERE EXPR ORDER BY KEYS
  (sSQLRare.Sort (
    sSQLRare.SelectLDistinct (dealWithStar (eXPR, TABLES, EXPR)),
    KEYS))

| SELECTL eXPR FROM TABLES ORDER BY KEYS
  (sSQLRare.Sort (
    sSQLRare.SelectL (dealWithStar (eXPR, TABLES, sSQLRare.Bool true)),
    KEYS))

| SELECTL DISTINCT eXPR FROM TABLES ORDER BY KEYS
  (sSQLRare.Sort(
    sSQLRare.SelectLDistinct(dealWithStar (eXPR,TABLES,sSQLRare.Bool true)),
    KEYS))

| EXPR UNIONS EXPR
  (sSQLRare.UnionS (EXPR1, EXPR2))

| EXPR UNIONB EXPR
  (sSQLRare.UnionB (EXPR1, EXPR2))

| EXPR UNIONL EXPR
  (sSQLRare.UnionL (EXPR1, EXPR2))

| LANGLE LABEL COLON EXPR RANGLE
  (sSQLRare.Variant (LABEL, EXPR))

| CASE EXPR OF NONEMPTY_BRANCHES
 (sSQLRare.Case (EXPR, NONEMPTY_BRANCHES))

| CASE EXPR OF NONEMPTY_BRANCHES ELSE EXPR
 (sSQLRare.Otherwise (EXPR1, NONEMPTY_BRANCHES, EXPR2))

| LABEL
 (sSQLRare.Proj LABEL)

| LBRAC NONEMPTY_FIELD_LIST RBRAC
  (sSQLRare.Record  NONEMPTY_FIELD_LIST)

| LBRAC EXPR COMMA NONEMPTY_EXPR_LIST RBRAC
  (sSQLRare.Tuple (EXPR ::  NONEMPTY_EXPR_LIST))

| MODIFY LABEL OF EXPR TO EXPR
  (sSQLRare.Modify (LABEL, EXPR1, EXPR2))

| MODIFY SYMBOL OF EXPR TO EXPR
```

```
   (sSQLRare.Modify ("#" ^ SYMBOL, EXPR1, EXPR2))


eXPR
 : EXPR
   (EXPR)

 | NONEMPTY_FIELD_LIST
   (sSQLRare.Record  NONEMPTY_FIELD_LIST)

NONEMPTY_BRANCHES
 : BRANCH
   %prec OR
   ([BRANCH])

 | BRANCH OR NONEMPTY_BRANCHES
   (BRANCH :: NONEMPTY_BRANCHES)

BRANCH
 : LABEL THEN APPLY FUNCTION THAT TRANSFORMS EXPR TO EXPR
   (LABEL, EXPR1, EXPR2)

NONEMPTY_EXPR_LIST
 : EXPR
   ([EXPR])

 | EXPR COMMA NONEMPTY_EXPR_LIST
   (EXPR :: NONEMPTY_EXPR_LIST)

NONEMPTY_FIELD_LIST
 : FIELD
   ([FIELD])

 | FIELD COMMA NONEMPTY_FIELD_LIST
   (FIELD :: NONEMPTY_FIELD_LIST)

FIELD
 : LABEL COLON EXPR
   (LABEL, EXPR)

 | SYMBOL COLON EXPR
   ("#" ^ SYMBOL, EXPR)

EXPR_LIST
 : ([])

 | NONEMPTY_EXPR_LIST
   (NONEMPTY_EXPR_LIST)

TABLES
 : TABLE
```

```
   ([TABLE])

 | TABLE COMMA TABLES
   (TABLE :: TABLES)

TABLE
 : EXPR SYMBOL
   (SYMBOL, EXPR)

 | EXPR AS SYMBOL
   (SYMBOL, EXPR)

 | EXPR
   ("_*self*_", EXPR)


KEY
 : LABEL
   (LABEL, true)

 | LABEL ASCENDING
   (LABEL, true)

 | LABEL DESCENDING
   (LABEL, false)

 | SYMBOL
   ("#" ^ SYMBOL, true)

 | SYMBOL ASCENDING
   ("#" ^ SYMBOL, true)

 | SYMBOL DESCENDING
   ("#" ^ SYMBOL, false)

KEYS
 : KEY
   ([KEY])

 | KEY COMMA KEYS
   (KEY :: KEYS)

SPECEXPR
 : SYMBOL EXPR
   (sSQLRare.Symbol SYMBOL, EXPR)

 | EXPR EXPR
   (EXPR1, EXPR2)
```

G More Examples

```
! Kleisli/sSQL demo.
! Copyright (c) 2001 by KRDL
! Created by Limsoon, 22/8/2001
!
! First let us demonstrate Kleisli's
! ability to treat a flat relational
! RDBMS as a nested relational RDBMS
!

! Log on to a MySQL server.
! MySQL is a flat relational DBMS
! This created an ``agent'' scott who
! will handle requests to this MySQL server.

mysql-cplobj-add (
  #name: "scott",
  #host: "oracle.krdl.org.sg", #port: "3306",
  #db: "bic", #user: "limsoon", #password: "limsoon", #level: 1);


! Create a simple table

create table wlstest2 (
  #name: "varchar(255)",
  #age:  "int",
  #addr: "text")
using scott;


! Write some stuff to this table.
! Notice that the address is a list.
! Kleisli automatically takes care of
! such complex objects when they are stored
! to a flat database
!

select x into wlstest2
from { (#name: "limsoon", #age: 34, #addr: { "spore", "msia"}),
       (#name: "brian", #age: 30, #addr: { "uk", "usa"})} as x
using scott;


! Map the wlstest2 table on MySQL into local space

create view wlstest2 from wlstest2 using scott;


! Let's print its contents
```

```
! <a href="file:html/a.html">the result</a>

wlstest2;


! Add another record to it
! <a href="file:html/b.html">the result</a>

insert into wlstest2
values (#name: "chris", #age: 21, #addr: { "taiwan" })
using scott;

wlstest2;


! Delete everyone who has a home in spore
! <a href="file:html/c.html">the result</a>

delete from wlstest2
select x from wlstest2 as x, x.#addr as h where h = "spore"
using scott;

wlstest2;


! drop the table

drop table wlstest2 using scott;


!
! Next, we illustrate access to some bioinformatics sources
!


! GenPept reports at NCBI.
! How many ‘‘small-scale’’ reports are there?
! <a href="file:html/d.html">the result</a>

aa-get-count-general "small-scale";


! Get report 13276888 back
! <a href="file:html/e.html">the result</a>
! are you horrified by its deeply nested structure?

aa-get-seqfeat-by-uid 13276888;


! Get only its functional annotations
! <a href="file:html/f.html">the result</a>
```

```
select
  #title: x.#title,
  #start: f.#start, #end: f.#end,
  #function: a.#descr
from
  (aa-get-seqfeat-by-uid 13276888) as x,
  x.#feature as f,
  f.#anno.l2s as a
where
  a.#anno_name = "function";


! Play with psort to see where 13276888
! might localized to.
! <a href="file:html/g.html">the result</a>

select y
from (aa-get-seqfeat-by-uid 13276888) as x,
     x.#sequence.localpsort.v2r.#summary as y;


! Now let's use our agent scott to keep
! some complex GenPept data


! first allocate a table on the MySQL database
! for these data

create table mygenpept (
  #uid: "int", #title: "varchar(255)", #accession: "varchar(255)",
  #common: "varchar(255)", #organism: "text",
  #feature: "text", #sequence: "text")
using scott;


! Write a whole lot of GenPept records into
! it. See how smoothly these complex data
! go into the underlying flat RDBMS which
! has miraculously been endowed with nested
! relational capability by our system.

select x into mygenpept
from (aa-get-seqfeat-general "small-scale") as x
using scott;


! let's have a look
! <a href="file:html/h.html">the result</a>

create view mygenpept from mygenpept
```

```
using scott;

mygenpept;


! now print only those that have function
! <a href="file:html/i.html">the result</a>

select
  #uid: x.#uid,
  #title: x.#title,
  #start: f.#start, #end: f.#end,
  #function: a.#descr
from
  mygenpept as x,
  x.#feature as f,
  f.#anno.l2s as a
where
  a.#anno_name = "function";


!
! ok, now let's clean up.
!

! ask the system what tables have been created
! <a href="file:html/j.html">the result</a>

scott-info;


! drop it.

drop table mygenpept using scott;
```

H Even More Examples

Here are some examples illustrating the kind of shorthands that Mark Haselup requested.

```
create view A as {(a: {1,2}, b:3), (a: {4,5}, b:6)};

select * from A;

select x.a from A x;

select y from A x, x.a y;

select * from A x where x.b > 3;

select distinct * from A order by b desc;

select  y from A x, x.a y where (x.b > 3) and (y < 5);

select  y from A x, x.a y where (x.b > 3) or (y < 2);
```