FRIL Tutorial

Nick Sorrell

1 Overview

If you're coming from an object-oriented, or imperative, programming background, then logic programming is going to be... different. Gone are those fanciful classes, those control structures, and those pointers. In Fril, your currency is the clause, and the probabilistic power that accompanies it. Your control structure is the bootstrap clause which will stop your recursion (yes, recursion is your looping mechanism).

You may be familiar with functions from your favorite language, and in languages like C++, these functions have names. So our closest analogy to that in Fril would be predicates. Named-functions are to C++ what Predicated-clauses are to Fril. Function parameters are to C++ what terms can be to Fril. So, Fril is based on predicates and terms, which can look like:

```
((predicate term))
```

Note that predicate and term are wrapped in not just one set of parentheses, but two. Why? Well, every Fril program is one big conjunctive clause (technically it's a list of atoms, and I encourage you to look at the Appendix for a layout of how clauses, atoms, terms and other parts of the language connect), and is thus wrapped in its own set of parentheses. Imagine a logical expression like $((p \vee q) \wedge (r \vee q) \wedge ... \wedge (z \vee q))$. So we can view a program with just one atom in the same way, but with a silent *true* clause appended, as follows.

```
((predicate term)(true))
```

Throughout this tutorial, I will keep simple clauses on one line, but more advanced clauses will look like this (for the sake of readability):

```
(
    (predicate_1 term1)
    (predicate_2 term2)
    ...
    (predicate_X termX)
)
```

With that in mind, here are a few key concepts that will make learning Fril easier.

- Terms can be used like function parameters and also as return values.
- Conditional expressions are created with multiple definitions of a single predicate.
- Looping is created with recursion, and by defining bootstrap conditions on a predicate.

So, that's a quick intro. What can you expect from the rest of this short tutorial? Here's a list of the topics we'll explore.

1. **BIPS** - in this section, we'll look at Fril's Built In Predicates (BIPS). Think of these like the "keywords" of the language that provide a handy, basic set of operations.
2. **Recursion** - here, we explore Fril's control structure: recursion.
3. **Conditional Expressions** - in this section, we look at how to form the conditional if-then-else expression you're so accustomed to.
4. **File Handling** - how to load files to build your library.
5. **Debugging** - finally, we explore debugging Fril code.

2 BIPS

Let's start with the classic "Hello World" example. Like other programming languages, there are "keywords" in Fril - but in Fril they are called Built-in Predicates, or "bips." (At the time of this writing, a list of bips is available at http://www.enm.bris.ac.uk/ai/martin/FrilManual/index.html) Let's get started with one! To print to the console, we will use the "pp" bip, which prints all proceeding terms on a new line.

Listing 1: Hello World (pp)

```
1 Fril > ?((pp "Hello World!"))
2 >Hello World!
3 >yes
```

Listing 2: Hello World (pp)

```
1 Fril > ?((pp "hello" " " "world!"))
2 >hello
3 >
4 >world!
5 >yes
```

To elicit this behavior, we used the "?" bip. The "?" bip executes a series of calls, called a goal_list. If the calls are successful, then Fril returns with "yes", as seen above. If we didn't need a new line after each term, we might do something like the following:

Listing 3: Hello World (p)

```
1 Fril > ?((p "Hello")(p " ")(p "World!"))
2 >Hello World!
3 >yes
```

At this point, it would be good to clarify, for the final time, that in *Listing* 2, we had 1 clause, 1 predicate(pp) and 3 terms ("hello" " " "world"). In *Listing* 3, we had 3 clauses, each with 1 predicate(p) and 1 term.

Let's try something a little more complicated with a very handy bip called "read." With this bip, we're going to specify "stdin" as our source to read from. We could also specify a filename to read from a file.

Listing 4: User Input

```
Fril > ?(
            (pp "Hi! What is your name?")
            (read stdin X)
            (p "Hi ")(p X)(p "! My name is FRIL!")
        )
>Hi! What is your name?
>Nick
>Hi Nick! My name is FRIL!
>yes
```

As you can see from this example, we can still work with variables and the command line. We just carry the variable, "X" in this case, around to our next clause to get some usage out of it. The "read" bip can be viewed as a binary predicate, which reads from a source to a destination variable, and can be read as:

Listing 5: The read bip

```
((read FROM_Source TO_Destination))
```

3 Recursion

So you're probably wondering how to exploit the power of recursion in Fril. Well, to do that, you must understand that a list is split between its head (the very first element) and its tail (everything after that first element). So, with that definition, here's a quick test.

1) Given a list (3 1 4 1 5 9 2 7), what's the head and what's the tail?
Answer : Head: 3 Tail: (1 4 1 5 9 2 7)
2) Given a list (abc def ghi jkl), what's the head and what's the tail?
Answer : Head: abc Tail: (def ghi jkl)

Given this definition, let's now create a simple predicate (or function) to return the length of a list. To do this, we will need to handle the bootstrap condition of an empty list. As you can see below in Line 1 of *Listing* 6, we handle the empty list by binding the number 0 to whatever variable has been passed along as a container. Lines 2-6 get into the meat of the recursion.

Listing 6: Length of a list

```
1 ((length () 0))
2 (
3     (length (HEAD | TAIL) LISTSIZE)
4     (length TAIL TEMPSIZE)
5     (sum TEMPSIZE 1 LISTSIZE)
6 )
```

- Line 3 gives us the generic template for calling this predicate. To call "length", one must supply a list and a container (or return) variable ($LISTSIZE$ in this case) to store the length. That list is comprised of anything ($HEAD$ in this case) and its tail ($TAIL$ in this case). The "|" provides the necessary character to denote a split list.
- Line 4 invokes the recursion. It calls "length" with the only the tail of the list from Line 3, and a new container variable, $TEMPSIZE$.
- Line 5 takes $TEMPSIZE$ and adds 1 to it, storing the result in $LISTSIZE$. Since Line 5 modifies $LISTSIZE$, and $LISTSIZE$ belongs to Line 3 (the function template), you can now see how we can return values in the imperative programming sense.

Just to be verbose, let's walk through the action of calling "length," with ((length (3 1 4) LISTSIZE))

Listing 7: Recursion on length of a list

```
1  You Call : ((length (3 1 4) LISTSIZE))
2      Fril Sees:  (length (3 | 1 4) LISTSIZE)
3      Fril Calls: (length (1 4) TEMPSIZE)
4          Fril Sees:  (length (1 | 4) LISTSIZE)
5          Fril Calls: (length (4) TEMPSIZE)
6              Fril Sees:  (length (4 | ) LISTSIZE)
7              Fril Calls: (length () TEMPSIZE)
8                  Fril Sees:  (length () LISTSIZE)    [Bootstrap!  Pop ↩
                       recursion!]
9                  Fril binds 0 to LISTSIZE
10             Fril Calls (sum TEMPSIZE 1 LISTSIZE) [TEMPSIZE is zero from ↩
                   line 9, so LISTSIZE is 1]
11             Fril binds 1 to LISTSIZE
12         Fril Calls (sum TEMPSIZE 1 LISTSIZE)    [TEMPSIZE is 1 from line ↩
               10, so LISTSIZE is now 2]
13         Fril binds 2 to LISTSIZE
14     Fril Calls (sum TEMPSIZE 1 LISTSIZE)    [TEMPSIZE is 2, from line 11, ↩
           so LISTSIZE is now 3]
15     Fril binds 3 to LISTSIZE
```

To see this result in Fril, copy the code from *Listing* 6 into the Fril interpreter, and after it has been entered, just type:
?((length (3 1 4) LISTSIZE)(p "List length is ")(pp LISTSIZE))

As you can see, the recursive power of Fril is also part of the control structure.

4 Conditional Expressions

You want your $if-then-else$ expressions back, don't you? Well, the good news is that they're alive and well in Fril. But before we get to defining them, it's good to revisit the concept behind how Fril executes your commands.

Fril attempts to successfully complete a *goal_list*. What this means is that Fril stops trying to evaluate your command once it succeeds. And if it doesn't succeed, it tries the next definition that you have provided for your predicate. If this doesn't yet make sense, don't worry, the following examples will clarify that for you.

In the following *Listing*, we will define a predicate, *test*, that takes two arguments, compares them with the *less* bip, and depending on that comparison, Fril will print a different value. Have a look.

Listing 8: Creating a Condition Expression

```
1     (   % This will print if X < Y succeeds
2         (test X Y)
3         (less X Y)
4         (p X)(p " is less than ")(p Y)
5     )
6     (   % This will print if Y < X succeeds
7         (test X Y)
8         (less Y X)
9         (p X)(p " is NOT less than ")(p Y)
10    )
```

Notice that in both cases, our first clause is identical. But the second clause is where they differ. Thus, we've handled both *greater* and *less than* cases. Now, let's try them!

Listing 9: Testing a Conditional

```
1     Fril >?((test 4 6))
2     >4 is less than 6
3     >yes
4
5     Fril >?((test 6 4))
6     >6 is NOT less than 4
7     >yes
8
```

```
9     Fril >?((test 4 4))
10    >no
```

Success! Well, almost. We successfully handled the first two cases, but take a look at the last case, where we compare a number to itself. We are met with a mere "no". Well, let's go back and add one more clause to handle that. Here's the we'll add:

Listing 10: Testing a Conditional

```
1     (   % This will print if X = Y succeeds
2         (test X Y)
3         (eq X Y)
4         (p X)(p " is equal to ")(p Y)
5     )
```

Now, when we try to compare a number to itself, we'll have an expression to handle that. Try it! For a more in-depth look at what's going on here, see the section on Debugging.

5 File Handling

At this point, it would be nice to have a library of predicates that we can just load when we want to use them. To do that, let's create a file called *conditional.frl* and place this file in the same directory as our FRIL executable. Open this new file with a text editor, and let's add the conditional tests that we just created to the file. Here they are for convenience:

Listing 11: Conditional.frl

```
(   % This will print if X = Y succeeds
    (test X Y)
    (eq X Y)
    (p X)(p " is equal to ")(p Y)
)
(   % This will print if X < Y succeeds
    (test X Y)
    (less X Y)
    (p X)(p " is less than ")(p Y)
)
(   % This will print if Y < X succeeds
    (test X Y)
    (less Y X)
    (p X)(p " is NOT less than ")(p Y)
)
```

Now save your changes, and let's load this file in FRIL with following command.

Listing 12: Reloading a file

```
Fril > reload conditional
>yes
```

At this point, whenever your exit the FRIL interpreter, you can just reload your libraries!

6 Debugging

To understand how FRIL operates, you must remember that it attempts to satisfy your command. So when you type a *goal_list* preceded by a '?', FRIL tries to make it succeed - thus, as you learned in the "Conditional Expressions" section, FRIL tries every definition of a given predicate until it succeeds. To see how this works, let's load our 'conditional.frl' file and introduce a new bip, *tq*. *tq* will allow us to *trace* the action that Fril takes while trying to satisfy our command. Have a look (and try for yourself) the following trace:

Listing 13: Tracing 'test'

```
1  Fril >tq ((test 7 5))
2
3  Starting Fril Trace - header gives (Clause-name Index Nth-goal-in-body ↩
       Depth)
4
5  (top-level * 1 0) ****** (test 7 5) trace ?
6  (test 1 1 1) ****** (eq 7 5)
7  (test 1 1 1) FAILED (eq 7 5)
8  (test 2 1 1) ****** (less 7 5)
9  (test 2 1 1) FAILED (less 7 5)
10 (test 3 1 1) ****** (less 5 7)
11 (test 3 1 1) SOLVED (less 5 7)
12 (test 3 2 1) ****** (p 7)
13 7(test 3 2 1) SOLVED (p 7)
14 (test 3 3 1) ****** (p ' is NOT less than ')
15  is NOT less than (test 3 3 1) SOLVED (p ' is NOT less than ')
16 (test 3 4 1) ****** (p 5)
17 5(test 3 4 1) SOLVED (p 5)
18 (top-level * 1 0) SOLVED (test 7 5)
```

As you can see on line 7, the first 'test' predicate fails, as the numbers aren't equal. Line 9 shows the second definition failing similarly. Finally, Line 11 indicates a success, and shows the subsequent steps FRIL takes.

The *tq* bip is enormously handy when trying to debug a program that you've constructed, as it will give you detailed info at each level that it encounters.

This is also a good time to introduce the *cut*. A cut is indicated by '(!)', and it tells FRIL to stop trying to satisfy the given predicate. To illuminate

this, let's change our 'conditional.frl' definition by adding a trackback to the equality-test. Here's what your file should look like now:

Listing 14: Conditional.frl

```
    (   % This will print if X = Y succeeds
        (test X Y)
        (!)
        (eq X Y)
        (p X)(p " is equal to ")(p Y)
    )
    (   % This will print if X < Y succeeds
        (test X Y)
        (less X Y)
        (p X)(p " is less than ")(p Y)
    )
    (   % This will print if Y < X succeeds
        (test X Y)
        (less Y X)
        (p X)(p " is NOT less than ")(p Y)
    )
```

What we're saying here is that we don't want the other predicates that we've defined to be evaluated. So now, when we try ?((*test* 7 5)) we get a 'no'. And here is what our trace looks like:

Listing 15: Tracing 'test' with cut

```
Fril >tq ((test 7 5))

Starting Fril Trace - header gives (Clause-name Index Nth-goal-in-body ↩
    Depth)

(top-level * 1 0) ****** (test 7 5) trace ?
(test 1 1 1) ****** (!)
(test 1 1 1) SOLVED (!)
(test 1 2 1) ****** (eq 7 5)
(test 1 2 1) FAILED (eq 7 5)
(test 1 1 1) FAILED (!)
(top-level * 1 0) FAILED (test 7 5) - try again: n(default), y or q ?
```

7 Appendix

About variables... In Fril, a Variable is "a capital letter followed by zero or more digits or a string of letters in which at least the first two are capitals or an underscore followed by a string of symbols." Puke. See the list below for a better idea. A Constant is a character string other than that which falls under Variable's rules. A Number is a positive/negative floating point or integer. A Term is a Variable, Constant, Integer, or List. A List is anything between two parentheses. An Atom is a List whose first element is a predicate and whose remaining elements are terms. A Clause is a List of one or more Atoms. And a Program is a sequence of Clauses. Here's a recap, with '→' denoting "composed of."

Program → Lists of Clauses
Clauses → Lists of Atoms
Atoms → (<predicate> <term 1> ⋯ <term n>)
Predicate → built-in or user-defined predicates (conclusions)
Terms → Variables, Numbers, Constants, Lists
Lists → ()
Variables → X X1 XL LIst JOHN _X5 _12
Numbers → -5 68 42 23.64 -23.64
Constants → Jenny susan "Barry Jerry"