Introduction to PX: Using XSB in Python

This notebook provides some examples of how to run XSB from a Python interpreter, and by extension, how to embed XSB into Python applications using an XSB package tentatively called **px**. The material presented here does not cover all **px** functionality, and does not even begin to cover all XSB functionality. See the XSB manuals for background in and full details of XSB, and Volume 2 Chapter 18 for details of **px**.

To start, we need to **px** directory is on our Python path. After that we can simply import the **px** module like any other Python module.

In [6]:
```
import sys
sys.path.insert(0,'../../packages/xsbpy/px')
from px import *
```

```
[xsb_configuration loaded]
[sysinitrc loaded]
[xsbbrat loaded, cpu time used: 0.001 seconds]
[xsbrc loaded]
[xsbpy loaded]
xsbpy_initted_with_python(
```

```
auto(python3.8))[px loaded]
[px_test loaded]
```

Note that importing **px** automatically starts up XSB within the same process as your Python session. You can get basic information on commands via help(), but see the XSB manual for full details (Volume 2, Chapter 18, remember)

In [7]:
```
help('px')
```

```
Help on module px:

NAME
    px

FUNCTIONS
    add_prolog_path(List)
        Convenience function to add one or more XSB library paths designated as a list of strings.

        This function calls XSB's equivalent of Python's sys.path.append()} and is defined as:
        px_cmd('consult','add_lib\dir',Paths).

    consult(File)
        Convenience function for compiling a Prolog 'file' as necessary, and loading it.

        Defined as px_cmd('consult','consult',File)

    ensure_loaded(File)
        Convenience function for loading and/or compiling a Prolog 'file' as necessary.

        Defined as px_cmd('consult','ensure_loaded',File)

    pp_px_cmd(Module, Pred, *args)
        Pretty print px_cmd() and its return

    pp_px_comp(Module, Pred, *args, **kwargs)
        Pretty print px_comp() and its return

    pp_px_qdet(Module, Pred, *args)
        Pretty print px_qdet() and its return

    prolog_paths()
        Convenience function to return a list of all current XSB library paths (XSB's equivalent of Python's sy
s.path).

    px_close(...)
        Close XSB

    px_cmd(...)
        XSB command execution from Python

    px_comp(...)
        Set comprehehsion using XSB from Python

    px_get_error_message(...)
        Find the XSB error message

    px_init(...)
        Init XSB

    px_qdet(...)
        XSB query execution from Python

DATA
    DELAY_LISTS = 1
    NO_TRUTHVALS = 2
    PLAIN_TRUTHVALS = 4

FILE
    /home/tswift/xsb-repo/xsb-code/XSB/packages/xsbpy/px/px.py
```

Deterministic Queries and Commands

Le's get started with a simple query. He're we're asking XSB to reverse a list containing a integers, a tuple and a dictionary. 'basics' is the XSB module, and 'reverse' is the XSB predicate.

In [8]:
```
pp_px_qdet('basics','reverse',[1,2,3,('mytuple'),{'a':{'b':'c'}}])
```

```
?- basics:reverse(([1, 2, 3, 'mytuple', {'a': {'b': 'c'}}],),Answer).

   Answer  = [{'a': {'b': 'c'}}, 'mytuple', 3, 2, 1]
   TV = True
```

The pp_px_qdet function calls px_qdet and pretty prints the call and return in a style like that used in XSB''s command line inferface. Note that while the Python call was variable-free, XSB infers that there is an additional variable in the call -- the variable is shown as ''Answer'' but that''s just for display here. XSB also passes back the truth value of an answer, which can be

- 1 (*true*) which means the query succeeded and that the answer is true in the Well-Founded Model of the program.
- 0 *false* which means that the query failed and that the query has no answers in the Well-Founded Model of the program.
- 2 *undefined* which means that the query succeeded, but the answer is neither *true* nor *false* in the Well-Funded Mdel of the program.

To understand a little better why a truth value is needed, lets make a query that fails (is false):

In [9]:
```
 pp_px_cmd('px_test','one_ary_fail','p')
```

```
?- px_test:one_ary_fail(p)

   TV = False
```

In this case, there is no answer to return, but the truth value indicates that the query failed (in Prolog, a failure is different than an error condition, as we'll see below). Meanwhile, lets see how the query to reverse/2 would usually look to a Python programmer, when where is no pretty printing: the answer and truth value are returned in a tuple

In [10]:
```
Answer,TV = px_qdet('basics','reverse',[1,2,3,('mytuple'),{'a':{'b':'c'}}])
print(Answer)
print(TV)
```

```
[{'a': {'b': 'c'}}, 'mytuple', 3, 2, 1]
1
```

Remember that px_qdet() adds an extra argument to the Prolog call and that the bindings to the variable in this argument is the answer passed back to Prolog. But what if you don't want that behavior, say you want to call the Prolog goal consult(px) or p(a,b,c). In this case, just use px_command (or pp_px_command)

In [13]:
```
pp_px_cmd('px_test','win',0)
```

```
?- px_test:win(0)

   TV = Undefined
```

which also shows the *undefined* truth value. Using px_cmd() to consult XSB files was mentioned above; this can be done fully interactively with **px**. Let's say you made a change to **px_test.P**. There's no need to leave your session -- just (re-)consult it.

In [12]:
```
consult('px_test')
```

```
[px_test loaded]
```

One least aspect of querying is exception handling. If an exception occurs in XSB execution, it is caught by **px**, a Python exception of the general **Exception** class is then raised for Python to handle. However, by another call to XSB the actual XSB exception can also be examined. Within the pretty-print display format this looks like:

In [11]:
```
   pp_px_qdet('usermod','open','missing_file','read')
```

```
?- usermod:open(('missing_file', 'read'),Answer).

Exception Caught from XSB: 
      ++Error[XSB/Runtime/P]: [Permission (Operation) open[mode=r,errno= ENOENT: No such file or directory] on
file: missing_file]  in /(open,3)
```

Collection Comphrehensions with PX

The above queries were determinstic, but you can collect all solutions of a non-deterministic query using a construct similar to list or set comprehension in Python.By default the collection is a Python list, but it can also be a Python set.

The Python function call

px_comp(Module,Predicate,*args,kwargs)**

returns a collection of answers such that the Prolog goal

Module:Predicate(input_args,output_variables)

The number of variable arguments in the Python call (i.e., the number of elements in the *args tuple) corresponds to the number of **input_arguments** in the XSB goal. The number of **output_variables** is by default 1, but can be set by the **vars** keyword argument.

Let's see how this works:

In [14]:
```
px_comp('px_test','nondet_query')
```

Out[14]:
```
[(('d',), 1), (('c',), 1), (('b',), 1), (('a',), 1)]
```

NNote that when translating from Prolog to Python a prolog term like **nondet_query(a)** gets translated to a Python tuple **(plgTerm,nondet_query,a)**. The **plgTerm** argument is needed to so that the Python can distinguish whether the tuple represents the Prolog structure **nondet_query(a)** and the tuple **(nondet_query,a)**. This distinction can be useful if Python wants to repreent a Prolog structure in a call to XSB.

By default the collection passed back to Python is a list, but it can also be a Python set if the keyword argument **list_collect** is set to True.

Non-monotonic Programming with **px**

XSB has strong support non-monotonic programming, one apsect of which are *delay lists* which indicate why an answer is neither true nor false in the well-founded model of a program. These are explained in detail in the XSB manual (and in various papers) but for now, we show a non-deterministic query for which some answers are true and some undefined (i.e., the second argument of their tuple is something other than '1').

In [15]:
```
px_comp('px_test','test_comp')
```

Out[15]:
```
[(('e',), 2), (('e',), 2), (('d',), 1), (('c',), 1), (('b',), 1), (('a',), 1)]
```

Going Forward

As mentioned, a short notebook is only a sampler of what XSB can do. The files in the directory **XSB_ROOT/xsbtests/python_tests** contain a number of other examples. These examples include constraint-based reasoning, timed calls, and stress tests. Probabilistic reasoning and virtually all other XSB functionality is also supported -- well, pretty much. See Volume 2 chapter 18 for a list of current limitiations, most all of which are in the process of being addressed.

In [ ]:
```
```