R"""A parser for the FC2 common format for transition systems

Author : Anders Andersen
Created On : Mon Jun 9 01:09:26 1998
Last Modified By:
Last Modified On: Wed Dec 02 21:44:42 1998
Status : Unknown, Use with caution!

This module implements the `FC2` class which is a parser for the FC2 common format for transition systems. The parser does not support the compact format, and it may also be a little bit more strict on newlines than the standard specifies (eg. each label on a separate line). You can create an object of this class initialised with the contents of an FC2 file (`"example.fc2"` in this example) like this:

```
from fc2 import FC2
example = FC2(open("example.fc2"))
```

The internal representation of the FC2 file is now available in the `example.fc2py` attribute which is a mixture of Python dictionaries and lists.

Using the Python built-in function `str` on an object of this class will generate a string in the FC2 format. It is generated from the internal representation of the parsed FC2 format and it may not be identical to the original string or file contents.

"""

```


# String manipulation, type information and regular expressions
from string import atoi, replace
from types import *
import re


# Exceptions in this module
class FC2Exception(Exception):
    pass


# Functions to check and access elements in the FC2 python representation

def isit(net, tablab, type, exp="", val=""):
```

    R"""Is it in the table/label?

    Check if a table or label contains the given element. If value (`val`) is not given only check the given type of expression. If expression (`exp`) is not given only check the given type table/label.

    """

```
    try:
        for (e, v) in net[tablab][type]:
            if not exp:
                return 1
            elif e == exp:
                if not val:
                    return 1
                elif v == val:
                    return 1
        return 0
    except KeyError:
        return 0

```

```
def getit(net, tablab, type, exp=""):
    R"""Get contents of table/label
```

Get the contents of a given table or label. It returns a list of all expressions with the given expression type. If expression is not given the list of all table elements or labels with the given type is returned.

```
    """
    vlist = []
    try:
        if not exp:
            return net[tablab][type]
        for (e, v) in net[tablab][type]:
            if e == exp:
                vlist.append(v)
        return vlist
    except KeyError:
        return vlist

def islist(exp, type):
    R"""Is it a list (or a single element) of the given type?
```

The "infix2" expression type (two expressions seperated by a comma) can be intepreted as a list, where the leftmost exprssion is the first element of the list and the rightmost expression is the rest of the list (either another "infix2" expression or a single element of the given type). You can use the getlist function below to actually create a Python list from these expression.

```
    """
    if exp[0] == type:
        return 1
    elif exp[0] == "infix2":
        if exp[1][1][0][0] == type:
            if exp[1][1][1][0] == type:
                return 1
            else:
                return islist(exp[1][1][1], type)
    return 0

def getlist(exp):
    R"""Create a Python list from an "infix2" expression
```

This function will create a Python list from an "infix2" expression. If the given expression is not an "infix2" expression, then a list with the given expression as a single element is returned.

```
    """
    if exp[0] != "infix2":
        return [exp]
    elif exp[1][1][0] == "infix2":
        return [exp[1][1][0][1]] + getlist(exp[1][1][1])
    else:
        return [exp[1][1][0][1], exp[1][1][1][1]]


class FC2:
    R"""Parsing the FC2 common format for transition systems
```

This class parses the FC2 common format (not the compact format) for transition systems and generate an internal representation which is a mixture of dictionaries and lists. This version doesn't support declarations in the FC2 common format.

```
    """

    # Our internal (empty) automata representation
    fc2py = {}

```

```
# Print debug information (0 = no, 1 = yes)?
DEBUG = 0

# Identation for each level in the str output.
str_indent = 2

# Whitespace line (ignored)
re_ws_line = re.compile(r'^\s*$')

# Temporary group format (used temporarily in string and opcp)
re_exp_grps = re.compile(r'\$(\d+)')

R"""The FC2 common format
```

The rest of these regular expressions are based on the information found in “FC2: Reference Manual version 1.1” (Madelaine/Simone, 1993). I have tried to follow the naming conventions used in the reference manual in the names below.

```
"""

# Version (see group(2))
re_version = re.compile(r'^\s*(version)\s*"([^"]*)"\s*$')

# Declarations are not supported
re_declarations = re.compile(r'^\s*(declarations)\s*$')

# Net table (digits [group(2)] = number of nets)
re_net_table = re.compile(r'^\s*(nets)\s+(\d+)\s*$')

# Table (digits [group(2)] = number of entries)
re_table = re.compile(r'^\s*(structs|behavs|logics|hooks)\s*(\d+)\s*$')

# Label (the last part [(.*)] is an exp: parsed separately)
re_label = re.compile(r'^\s*(struct|behav|logic|hook)\s*(.*)$')

# Net list (digits [group(2)] = net number)
re_net = re.compile(r'^\s*(net)\s*(\d+)\s*$')

# Expression entry (the last part [(.*)] is an exp: parsed separately)
re_exp_entry = re.compile(r'^\s*:(\d+)\s*(.*)$')

# Expressions
re_exp_constant = re.compile(r'^\s*(tau|quit|_)\s*$')
re_exp_unary = re.compile(r'^([?!~#])(.+)$')
#re_exp_infix = re.compile(r'^([])$')
re_exp_infix0 = re.compile(r'^(.+)([.^])(.+)$')
re_exp_infix1 = re.compile(r'^(.+)(;)(.+)$')
re_exp_infix2 = re.compile(r'^(.+)(,)(.+)$')
re_exp_infix3 = re.compile(r'^(.+)([<>])(.+)$')
re_exp_infix4 = re.compile(r'^(.+)([+])(.+)$')
re_exp_opcp = re.compile(r'^\(([^)]+)\)$')
re_exp_sopcp = re.compile(r'\(([^)]+)\)')
#re_exp_prefix = re.compile(r'^()$')
re_exp_string = re.compile(r'^\s*"(([^"]|\")*)"\s*$')
re_exp_sstring = re.compile(r'"(([^"]|\")*)"')
re_exp_star = re.compile(r'^\s*\*(\d*)\s*$')
re_exp_ref = re.compile(r'^\s*([@']?)(\d+)\s*$')

# Vertice table (digits [group(2)] = number vertice)
re_vertice_table = re.compile(r'^\s*(vertice)\s*(\d+)\s*$')
```

```
    # Vertex (digits [group(2)] = vertex number)
    re_vertex = re.compile(r'^\s*(vertex)\s*(\d+)\s*$')

    # Edge table (digits [group(2)] = number edges)
    re_edge_table = re.compile(r'^\s*(edges)\s+(\d+)\s*$')

    # Edge (digits [group(2)] = edge number)
    re_edge = re.compile(r'^\s*(edge)\s*(\d+)\s*$')

    # Target vertice (the last part [(.*)] is an exp: parsed separatly)
    re_target_vertice = re.compile(r'^\s*(->|result)\s*(.*)$')

    def __init__(self, fc2=None):
        R"""Initialise the object

        Initialise the object. Generate the internal representation if the optional fc2 string or file is given.

        """
        self._return_a_line = 0
        if fc2:
            if type(fc2) is FileType:
                self.readfc2file(fc2)
            elif type(fc2) is StringType:
                self.readfc2str(fc2)
            else:
                raise FC2Exception, "FC2 init argument of unknown type"

    def __str__(self):
        R"""Generate the FC2 format

        Generate the FC2 format from the internal representation. This is the result of using the built-in
        Python function str on an instance of this class.

        """
        return self._fc2_str(self.fc2py, 0)

    def readfc2str(self, fc2):
        R"""Convert from fc2 string to the internal fc2 representation

        This function takes a fc2 description (text string) of an automata and generates the internal fc2
        representation which is a mixture of Python dictionaries and lists.

        """

        # Emulate file IO
        import StringIO
        self.readfc2file(StringIO.StringIO(fc2))

    def readfc2file(self, fc2file):
        R"""Convert from fc2 file to the internal fc2 representation

        This function takes a fc2 file description (a file) of an automata and generates the internal fc2
        representation which is a mixture of Python dictionaries and lists.

        """
        self.fc2file = fc2file
        self.fc2py = self._fc2()

    def _nextline(self):
        if self._return_a_line:
            self._return_a_line = 0
        else:
            self.line = self.fc2file.readline()
            while self.line:
                if self.re_ws_line.match(self.line):
                    self.line = self.fc2file.readline()
                else:
                    break
        return self.line

    def _return_one_line(self):
        self._return_a_line = 1

    def _debug(self, str, eol="\n"):
        if self.DEBUG:
            import sys
            sys.stderr.write("%s%s" % (str, eol))

    def _fc2(self):

        # fc2 is saved in a dictionary
        fc2 = {}

        # Parse version information (optional)
        if self._nextline():
            if self.re_version.match(self.line):
                fc2["version"] = self.re_version.match(self.line).group(2)
                self._debug("Version: %s" % (fc2["version"],))
            else:
                self._debug("No version given")
                self._return_one_line()
        else:
            return fc2

        # Parse declarations
        if self._nextline():
            if self.re_declarations.match(self.line):
                self._debug("Declarations found but ignored!")
            else:
                self._debug("No declarations")
                self._return_one_line()
        else:
            return fc2

        # Parse the net table
        while self._nextline():

            # Table of nets (ignoring the rest)
            if self.re_net_table.match(self.line):
                fc2["net_table"] = self._net_table()

            # Ignoring
            else:
                self._debug("Ignoring: %s" % (self.line,), "")

        # Retur the result
        return fc2

    def _fc2_str(self, fc2, level):
        if fc2.has_key("net_table"):
            return self._net_table_str(fc2["net_table"], level)
    def _net_table(self):

        # Create the empty net table with num nets
        num = atoi(self.re_net_table.match(self.line).group(2))
        net_table = {}
        net_table["net_list"] = [{}] * num
        self._debug("Net table: %d" % (num,))

        # Parse the tables part (zero or more, but maximum one of each type?)
        while self._nextline():

            # A table (structs, behavs, ...)
            self._debug("Is this table: %s" % (self.line), "")
            if self.re_table.match(self.line):
                self._debug("  -> YES")
                (type, table) = self._table()
                if not net_table.has_key("tables"):
                    net_table["tables"] = {}
                net_table["tables"][type] = table

            # Go to next part
            else:
                self._debug("  -> NO")
                self._return_one_line()
                break

        # Parse the label part (zero or more of each type?)
        while self._nextline():

            # A label
            self._debug("Is this label: %s" % (self.line), "")
            if self.re_label.match(self.line):
                self._debug("  -> YES")
                (type, label) = self._label()
                if not net_table.has_key("label"):
                    net_table["label"] = {}
                if not net_table["label"].has_key(type):
                    net_table["label"][type] = []
                net_table["label"][type].append(label)

            # Go to next part
            else:
                self._debug("  -> NO")
                self._return_one_line()
                break

        # Look for num nets
        self._debug("Look for %d nets in net list" % (num,))
        for i in range(num):
            if not self._nextline():
                self._debug("EOF after %d of %d nets" % (i, num))
                break

            # One net
            if self.re_net.match(self.line):
                int = atoi(self.re_net.match(self.line).group(2))
                net_table["net_list"][int] = self._net(int)

            # Ignoring
            else:
                self._debug("Ignoring: %s" % (self.line,), "")

        # Return the net table
        return net_table

    def _net_table_str(self, net_table, level):
        str = ""
        if net_table.has_key("net_list"):
            str = str + (" " * level) + \
                  "nets %d\n" % (len(net_table["net_list"]),)
        level = level + self.str_indent
        if net_table.has_key("tables"):
            for (type, table) in net_table["tables"].items():
                str = str + (" " * level) + "%s %d\n" % (type, len(table))
                str = str + self._table_str(
                    type, table, level + self.str_indent)
        if net_table.has_key("label"):
            for (type, label_list) in net_table["label"].items():
                for label in label_list:
                    str = str + self._label_str(type, label, level)
        if net_table.has_key("net_list"):
            for i in range(len(net_table["net_list"])):
                str = str + (" " * level) + "net %d\n" % (i,)
                str = str + self._net_str(
                    net_table["net_list"][i], level + self.str_indent)
        return str

    def _net(self, int):

        # parse one net
        net = {}
        self._debug("Net %d" % (int,))

        # Parse the tables part (zero or more, but maximum one of each type?)
        while self._nextline():

            # A table (structs, behavs, ...)
            self._debug("Is this table: %s" % (self.line), "")
            if self.re_table.match(self.line):
                self._debug("  -> YES")
                (type, table) = self._table()
                if not net.has_key("tables"):
                    net["tables"] = {}
                net["tables"][type] = table

            # Go to next part
            else:
                self._debug("  -> NO")
                self._return_one_line()
                break

        # Parse the label part (zero or more of each type?)
        while self._nextline():

            # A label
            self._debug("Is this label: %s" % (self.line), "")
            if self.re_label.match(self.line):
                self._debug("  -> YES")
                (type, label) = self._label()
                if not net.has_key("label"):
                    net["label"] = {}
                if not net["label"].has_key(type):
                    net["label"][type] = []
                net["label"][type].append(label)

            # Go to next part
            else:
                self._debug("  -> NO")
                break

        # Parse vertice table (zero or one)
        if self.re_vertice_table.match(self.line):
            net["vertice_table"] = self._vertice_table()
        else:
            self._return_one_line()
            self._debug("No vertice table")

        # Return it
        return net

    def _net_str(self, net, level):
        str = ""
        if net.has_key("tables"):
            for (type, table) in net["tables"].items():
                str = str + self._table_str(type, table, level)
        if net.has_key("label"):
            for (type, label_list) in net["label"].items():
                for label in label_list:
                    str = str + self._label_str(type, label, level)
        if net.has_key("vertice_table"):
            str = str + self._vertice_table_str(net["vertice_table"], level)
        return str

    def _table(self):

        # Table type and number of elements
        type = self.re_table.match(self.line).group(1)
        num = atoi(self.re_table.match(self.line).group(2))
        table = [()] * num
        self._debug("Table: %s %d" % (type, num))

        # Parse each element in the table
        for i in range(num):
            if not self._nextline():
                break

            # Parse an expression entry
            if self.re_exp_entry.match(self.line):
                int = atoi(self.re_exp_entry.match(self.line).group(1))
                exp = self.re_exp_entry.match(self.line).group(2)
                self._debug("  Exp entry %d: %s" % (int, exp))
                table[int] = self._exp(exp)

            # Ignore none expression entries (even compact form)
            else:
                break

        # Return table and type
        return (type, table)

    def _table_str(self, type, table, level):
        str = (" " * level) + "%s %d\n" % (type, len(table))
        level = level + self.str_indent
        for i in range(len(table)):
            str = str + (" " * level) + ":%d " % (i,) + \
                  self._exp_str(table[i]) + "\n"
        return str

    def _label(self):

        # Label type
        type = self.re_label.match(self.line).group(1)
        exp = self.re_label.match(self.line).group(2)
        self._debug("Label %s: %s" % (type, exp))
        label = self._exp(exp)

        # Return table and type
        return (type, label)

    def _label_str(self, type, label, level):
        return (" " * level) + type + " " + self._exp_str(label) + "\n"

    def _vertice_table(self):

        # Create the empty vertice table with num vertice
        num = atoi(self.re_vertice_table.match(self.line).group(2))
        vertice_table = [{}] * num
        self._debug("Vertice table: %d" % (num,))

        # Look for num vertice
        for i in range(num):
            if not self._nextline():
                self._debug("EOF after %d of %d vertice" % (i, num))
                break

            # One vertex
            if self.re_vertex.match(self.line):
                int = atoi(self.re_vertex.match(self.line).group(2))
                vertice_table[int] = self._vertex(int)

            # Ignoring
            else:
                self._debug("Ignoring: %s" % (self.line,), "")

        # Return the vertice table
        return vertice_table

    def _vertice_table_str(self, vertice_table, level):
        str = (" " * level) + "vertice %d\n" % (len(vertice_table),)
        level = level + self.str_indent
        for i in range(len(vertice_table)):
            str = str + (" " * level) + "vertex%d\n" % (i,)
            str = str + self._vertex_str(vertice_table[i],
                                         level + self.str_indent)
        return str

    def _exp(self, exp):
        self._debug("Exp: %s" % (exp,), "")
        (exp, grps) = self._repl_grps(exp)
        self._debug(" -> %s" % (exp,))
        if self.re_exp_constant.match(exp):
            self._debug("  (constant:%s)" % (exp,))
            return ("constant", exp)
        elif self.re_exp_unary.match(exp):
            m = self.re_exp_unary.match(exp)
            self._debug("  (unary:", "")
            nexp = self._exp(self._insrt_grps(m.group(2), grps))
            self._debug(")")
            return ("unary", (m.group(1), nexp))
        elif self.re_exp_infix4.match(exp):
            m = self.re_exp_infix4.match(exp)
            self._debug("  (infix4:", "")
            nexp = self._infix_split(m, grps)
            self._debug(")")
            return ("infix4", nexp)
        elif self.re_exp_infix3.match(exp):
            m = self.re_exp_infix3.match(exp)
            self._debug("  (infix3:", "")
            nexp = self._infix_split(m, grps)
            self._debug(")")
            return ("infix3", nexp)
        elif self.re_exp_infix2.match(exp):
            m = self.re_exp_infix2.match(exp)
            self._debug("  (infix2:", "")
            nexp = self._infix_split(m, grps)
            self._debug(")")
            return ("infix2", nexp)
        elif self.re_exp_infix1.match(exp):
            m = self.re_exp_infix1.match(exp)
            self._debug("  (infix1:", "")
            nexp = self._infix_split(m, grps)
            self._debug(")")
            return ("infix1", nexp)
        elif self.re_exp_infix0.match(exp):
            m = self.re_exp_infix0.match(exp)
            self._debug("  (infix0:", "")
            nexp = self._infix_split(m, grps)
            self._debug(")")
            return ("infix0", nexp)
        elif self.re_exp_opcp.match(exp):
            m = self.re_exp_opcp.match(exp)
            self._debug("  (opcp:", "")
            nexp = self._exp(self._insrt_grps(m.group(1), grps))
            self._debug(")")
            return ("opcp", nexp)
        elif self.re_exp_string.match(exp):
            m = self.re_exp_string.match(exp)
            nexp = self._insrt_grps(m.group(1), grps)
            self._debug("  (string:%s)" % (nexp,))
            return ("string", nexp)
        elif self.re_exp_star.match(exp):
            m = self.re_exp_star.match(exp)
            self._debug("  (star:%s)" % (m.group(1),))
            return ("star", m.group(1))
        elif self.re_exp_ref.match(exp):
            m = self.re_exp_ref.match(exp)
            self._debug("  (ref:%s%s)" % (m.group(1), m.group(2)))
            return ("ref", (m.group(1), atoi(m.group(2))))
        else:
            self._debug("  (unknown:%s)" % (exp,))
            return ("unknown", exp)

    def _exp_str(self, exp):
        if exp[0] == "constant":
            return exp[1]
        elif exp[0] == "unary":
            return exp[1][0] + self._exp_str(exp[1][1])
        elif exp[0] in ["infix4", "infix3", "infix2", "infix1", "infix0"]:
            return self._infix_join(exp[1])
        elif exp[0] == "opcp":
            return "(" + self._exp_str(exp[1]) + ")"
        elif exp[0] == "string":
            return '"' + exp[1] + '"'
        elif exp[0] == "star":
            return '*' + exp[1]
        elif exp[0] == "ref":
            return "%s%d" % (exp[1][0], exp[1][1])
        return exp[0]

    def _eop(self, exp, start):
        i = start; p = 1
        while i < len(exp):
            if exp[i] == "(": p = p + 1
            elif exp[i] == ")": p = p - 1
            i = i + 1
            if p == 0: break
        return i

    def _repl_grps(self, exp):
        nexp = replace(exp, "$", "$D")
        exp = ""; grps = []; num = 0
        while 1:
            if self.re_exp_sstring.search(nexp):
                m = self.re_exp_sstring.search(nexp)
                start = m.start() + 1
                end = m.end()
            elif self.re_exp_sopcp.search(nexp):
                m = self.re_exp_sopcp.search(nexp)
                start = m.start() + 1
                end = self._eop(nexp, start)
            else:
                break
            exp = exp + nexp[:start] + "$%d" % (num,) + nexp[end-1]
            grps.append(nexp[start:end-1])
            nexp = nexp[end:]
            num = num + 1
        return (exp + nexp, grps)

    def _insrt_grps(self, exp, grps):
        nexp = ""
        while self.re_exp_grps.search(exp):
            m = self.re_exp_grps.search(exp)
            nexp = nexp + exp[:m.start()] + grps[atoi(m.group(1))]
            exp = exp[m.end():]
        return nexp + exp

    def _infix_split(self, m, grps):
        return (m.group(2),
                (self._exp(self._insrt_grps(m.group(1), grps)),
                 self._exp(self._insrt_grps(m.group(3), grps))))

    def _infix_join(self, exp):
        return self._exp_str(exp[1][0]) + exp[0] + self._exp_str(exp[1][1])

    def _vertex(self, int):

        # parse one net
        vertex = {}
        self._debug("Vertex %d" % (int,))

        # Parse the label part (zero or more of each type?)
        while self._nextline():

            # A label
            self._debug("Is this label: %s" % (self.line), "")
            if self.re_label.match(self.line):
                self._debug("  -> YES")
                (type, label) = self._label()
                if not vertex.has_key("label"):
                    vertex["label"] = {}
                if not vertex["label"].has_key(type):
                    vertex["label"][type] = []
                vertex["label"][type].append(label)

            # Go to next part
            else:
                self._debug("  -> NO")
                break

        # Parse edge table (zero or one)
        if self.re_edge_table.match(self.line):
            vertex["edge_table"] = self._edge_table()
        else:
            self._return_one_line()
            self._debug("No edge table")

        # Return it
        return vertex

    def _vertex_str(self, vertex, level):
        str = ""
        if vertex.has_key("label"):
            for (type, label_list) in vertex["label"].items():
                for label in label_list:
                    str = str + self._label_str(type, label, level)
        if vertex.has_key("edge_table"):
            str = str + self._edge_table_str(vertex["edge_table"], level)
        return str

    def _edge_table(self):

        # Create the empty edge table with num edges
        num = atoi(self.re_edge_table.match(self.line).group(2))
        edge_table = [{}] * num
        self._debug("Edge table: %d" % (num,))

        # Look for num edges
        for i in range(num):
            if not self._nextline():
                self._debug("EOF after %d of %d edges" % (i, num))
                break

            # One edge
            if self.re_edge.match(self.line):
                int = atoi(self.re_edge.match(self.line).group(2))
                edge_table[int] = self._edge(int)

            # The edge keyword and number is optional
            else:
                self._return_one_line()
                edge_table[i] = self._edge(i)

        # Return the vertice table
        return edge_table

    def _edge_table_str(self, edge_table, level):
        str = (" " * level) + "edges %d\n" % (len(edge_table),)
        level = level + self.str_indent
        for i in range(len(edge_table)):
            str = str + (" " * level) + "edge%d\n" % (i,)
            str = str + self._edge_str(edge_table[i], level + self.str_indent)
        return str

    def _edge(self, int):

        # parse one net
        edge = {}
        self._debug("Edge %d" % (int,))

        # Parse the label part (zero or more of each type?)
        while self._nextline():

            # A label
            self._debug("Is this label: %s" % (self.line), "")
            if self.re_label.match(self.line):
                self._debug("  -> YES")
                (type, label) = self._label()
                if not edge.has_key("label"):
                    edge["label"] = {}
                if not edge["label"].has_key(type):
                    edge["label"][type] = []
                edge["label"][type].append(label)

            # Go to next part
            else:
                self._debug("  -> NO")
                break

        # Parse target vertice
        if self.re_target_vertice.match(self.line):
            exp = self.re_target_vertice.match(self.line).group(2)
            self._debug("Target vertice: %s" % (exp,))
            edge["target_vertice"] = self._exp(exp)
        else:
            self._return_one_line()
            self._debug("Target vertice MISSING!")

        # Return it
        return edge

    def _edge_str(self, edge, level):
        str = ""
        if edge.has_key("label"):
            for (type, label_list) in edge["label"].items():
                for label in label_list:
                    str = str + self._label_str(type, label, level)
        if edge.has_key("target_vertice"):
            str = str + (" " * level) + " -> " + \
                  self._exp_str(edge["target_vertice"]) + "\n"
        return str
```