Project 1 Report

Taylor Berger, Zachary Friedland, Jianyu Yang

October 13, 2014

1 Language Decisions

We decided to write our project in Python due to the ease of expressing high level concepts and the removal of memory management from the project. We found it to be an effective choice since we could focus more on algorithmic nature of the project instead of the minutae of memory manipulation and management. At the time of writing this paper, we were using version 2.7.2 on a unix based system.

2 Alphabets

All alphabets in this project are restricted to the printable set of ASCII characters although a general alphabet may be completely unrestricted. An alphabet is defined as a set of unique symbols and are formally represented as:

$$\Sigma = \{\sigma_1, \sigma_2, ..., \sigma_n\}$$

In is project, alphabets are given to us in text only format. Each obect in the alphabet is a pre-quoted symbol preceeded by the word `alphabet` and followed by the word `end`. An example alphabet containing the symbols a, b, c would be:

```
alphabet 'a 'b 'c end
```

The former example can be abstracted into a formal definition:

```
Alphabet -> alphabet_keyword AlphabetList end_keyword
AlphabetList ->
AlphabetList -> Sigma AlphabetList
alphabet_keywork -> 'alphabet
end_keyword -> 'end
```

To form an alphabet in our project, we parse the file using pyparsing (see appendix). The following code was be used to parse any arbitrary set of pre-quoted symbols into a list of symbols that we assumed make up an alphabet.

```
1  from pyparsing import *
2
3  # Alphabet definition
4  alphabet_keyword = Keyword("alphabet").suppress()
5  alphabet_end_keyword = Keyword("end;").suppress() |\
6                         Keyword("end").suppress()
7  Symbol = Combine(Literal("\'").suppress() +\
8                   Optional(Literal("\\")) +\
9                   Word(printables + " ", exact=1))
10
11 Symbol.setParseAction(decodeEscapes)
12 SymbolList = OneOrMore(Symbol)
13 Alphabet = alphabet_keyword + SymbolList + alphabet_end_keyword
```

Pyparsing's utility function `suppress` allows the parser to expect the value and remove it from the output completely after a successful parse is compelted. Also, the `setParseAction(decodeEscapes)` function was used on line 10 to amake sure single characters that needed to be escaped in printable ASCII (looking at you `\n`) were parsed into their correct form instead of a two character sequence starting with the backslash.

3 Description of Non-Deterministic Finite State Automata (NFA)

A formal definition for the NFA can be viewed as the following:

```
Nfa -> nfa_keywork States InitialStates AcceptingStates Transitions
States -> states_keywork StateList end_keyword
StateList ->
StateList -> State StateList
InitialState -> initial_keyword State
AcceptingStates -> accept_keyword StateList end_keyword
Transitions -> transitions_keyword TransitionList end_keyword
TransitionList ->
TransitionList -> Transition TransitionList
Transition -> State SymbolList arrow State
SymbolList ->
SymbolList -> symbol SymbolList
dfa_keyword -> 'dfa
states_keyword -> 'states
initial_keyword -> 'initial
```

```
accept_keyword -> 'accept
transitions_keyword -> 'transitions
end_keyword -> 'end;
arrow -> '-->
```

It should be mentioned here that `symbol` $\in \Sigma$, where Σ is the alphabet that corresponds to this NFA.

i

4 Algorithms and Data Structures

Since the algorithms depend on how the data structure it operates on is constructed, we will cover the three main data structures (regular expressions, and DFA/NFAs) used in this project first.

4.1 Regular Expressions

4.1.1 Production

We begin by defining a base class in which all other types of regular expressions inherit from. This class is meaningless except to give the rest of the classes a common ancestor.

```python
class Production():
    '''
    Defines the base class that all Regex inherit from.
    '''
    def __init__(self):
        pass

    def matches(self, string):
        pass

    def consume(self, string):
        pass
```

We define two methods for this class, `matches` and `consume` where they will, respectively, return true if they match the parameterized string and consume as much off the string as possible as long as they match what they consume. These two functions vary depending on the implementation so they must be overridden in their subclasses.

4.1.2 Sigma

The first regular expression we define is the simplest, but most important, Sigma production. This is a regular expression that is responsible for matching to a single character and is a terminal regular expression for a language and is defined as `E -> ` σ where σ is a part of a predefined alphabet described in section 2.

```
class Sigma(Production):
    def __init__(self, sigma):
        self.sigma = sigma

    def __str__(self):
        return str(self.sigma)

    def matches(self, string):
        return self.sigma == string

    def consume(self, string):
        if len(string) >= 1 and string[0] == self.sigma:
            return string[0:1], string[1:]
        else:
            return '', string
```

4.1.3 Repetition

The next type of regular expression to implement was the repetition expression, or Kleene closure defined as `E -> * E`.

```
class Repetition(Production):
    def __init__(self, expr):
        self.expr = expr

    def __str__(self):
        return "* " + str(self.expr)

    def matches(self, string):
        if string == '':
            return True

        return self.expr.matches(string[0:1]) and self.matches(string[1:])

    def consume(self, string):
        consumed = 'default'
        total_consumed = ''
        leftover = string

        while consumed != '':
            consumed, leftover = self.expr.consume(leftover)
            total_consumed += consumed
```

```
        return total_consumed, leftover
```

4.1.4 Alternative

Implementint the alternative production required the composition of multiple regular expressions since it is a binary operator. Alternative regular expressions take the form:

```
E -> | E E
```

and are represented in our code as:

```
class Alternative(Production):
    def __init__(self, left, right):
        self.left = left
        self.right = right

    def __str__(self):
        return '| ' + str(self.left) + ' ' +  str(self.right)

    def matches(self, string):
        return self.left.matches(string) or \
               self.right.matches(string)

    def consume(self, string):
        left_consume, leftover = self.left.consume(string)
        # he he he. rightover.... I crack myself up.
        right_consume, rightover = self.right.consume(string)

        if len(left_consume) >= len(right_consume):
            return left_consume, leftover
        else:
            return right_consume, rightover
```

4.1.5 Concatenation

Concatenations are formally represented in the form `E -> + E E` and are the last meaningful regular expression we need to be able to construct. We represented them as follows.

```
class Concatenation(Production):
    def __init__(self, left, right):
        self.left = left
        self.right = right

    def __str__(self):
```

```
        return '+ ' + str(self.left) + ' ' + str(self.right)

    def matches(self, string):
        return self.left.matches(string[0:1]) and self.right.matches(string[1:])

    def consume(self, string):
        left_match, leftover = self.left.consume(string)
        right_match, leftover = self.right.consume(leftover)

        if left_match == '':
            return '', string

        left_match += right_match

        return ''.join(left_match), leftover
```

4.1.6 Nil Expression

For completeness, a regular expression has one other semantically correct production, the Nil Expression. This expression recognizes regular expressions of the type `E -> _` where the underscore represents the empty string.

Team Composition

Appendix: Libraries Used

We tried to stay away from using any libraries that would make tour required tasks trivial except for reading input files. For that particular task we opted to use a Python text parser called Pyparsing. You can download and install via the python egg from their website.

Source Files

regex.py

```
class Production:
    '''
    Defines the base class that all Regex inherit from.
    '''
    def __init__(self, alphabet):
        self.alphabet = alphabet

    def matches(self, string):
        pass
```

```python
    def consume(self, string):
        pass

class Sigma(Production):
    def __init__(self, sigma):
        self.sigma = sigma

    def __str__(self):
        return str(self.sigma)

    def matches(self, string):
        return self.sigma == string

    def consume(self, string):
        if len(string) >= 1 and string[0] == self.sigma:
            return string[0:1], string[1:]
        else:
            return '', string

class Repetition(Production):
    def __init__(self, expr):
        self.expr = expr

    def __str__(self):
        return "* " + str(self.expr)

    def matches(self, string):
        if string == '':
            return True

        return self.expr.matches(string[0:1]) and self.matches(string[1:])

    def consume(self, string):
        consumed = 'default'
        total_consumed = ''
        leftover = string

        while consumed != '':
            consumed, leftover = self.expr.consume(leftover)
            total_consumed += consumed

        return total_consumed, leftover


class Alternative(Production):
    def __init__(self, left, right):
```

```
        self.left = left
        self.right = right

    def __str__(self):
        return '| ' + str(self.left) + ' ' +  str(self.right)

    def matches(self, string):
        return self.left.matches(string) or \
               self.right.matches(string)

    def consume(self, string):
        left_consume, leftover = self.left.consume(string)
        # he he he. rightover.... I crack myself up.
        right_consume, rightover = self.right.consume(string)

        # This could be a potential problem due to there
        # being multiple parses include the left or right
        # side... We'll go with the longer parse for now
        if len(left_consume) >= len(right_consume):
            return left_consume, leftover
        else:
            return right_consume, rightover


class Concatenation(Production):
    def __init__(self, left, right):
        self.left = left
        self.right = right

    def __str__(self):
        return '+ ' + str(self.left) + ' ' + str(self.right)

    def matches(self, string):
        return self.left.matches(string[0:1]) and self.right.matches(string[1:])

    def consume(self, string):
        left_match, leftover = self.left.consume(string)
        right_match, leftover = self.right.consume(leftover)

        if left_match == '':
            return '', string

        left_match += right_match

        return ''.join(left_match), leftover
```

```python
class NilExpression(Production):
    def __str__(self):
        return ''

    def matches(self, string):
        return string == ''

    def consume(self, string):
        return '', string


def BuildExpression(tokens):
    """Builds an expression from a list of tokens using a one token look ahead
       strategy.

       tokens: Expected to be a list of string tokens (ie: ['+', 'a', 'a'])
    """
    t = tokens[0]

    # E -> + E E
    if t == '+':
        # TODO: Clean this up
        #        return BuildConcatenation(tokens[1:])a
        # build the appropriate expression for the left argument to the concat
        # operation and return the leftover tokens
        leftSide, leftover = BuildExpression(tokens[1:])

        # Make sure we have tokens to consume, otherwise an error
        if len(leftover) == 0:
            raise Error('''No more tokens found after building the left hand side of
                           a ConcatExpression''')
        # Build the right hand side of the ConcatExpression
        rightSide, leftover = BuildExpression(leftover)
        return Concatenation(leftSide, rightSide), leftover
    # E -> | E E
    elif t == '|':

        leftSide, leftover = BuildExpression(tokens[1:])

        # Make sure we have tokens to consume, otherwise an error
        if len(leftover) == 0:
            raise Error('''No more tokens found after building the left hand side of
                           a ConcatExpression''')

        rightSide, leftover = BuildExpression(leftover)
        return Alternative(leftSide, rightSide), leftover
```

```
        # E -> * E
        elif t == '*':
            e, leftover = BuildExpression(tokens[1:])
            return Repetition(e), leftover
        # E -> _ (empty, not underscore)
        elif t == '':
            return NilExpression(), tokens[1:]
        # E -> sigma (where sigma is some symbol that doesn't match the previous
        # values
        else:
            #these value are quoted so we must remove the quote
            return Sigma(t[1:]), tokens[1:]


if __name__ == "__main__":
    a = Sigma('a')
    print a
    print a.matches('a')
    print a.consume('a')
    print a.consume('b')
    print

    r = Repetition(a)
    print r
    print r.matches('aaaaa')
    print r.consume('aaaaab')
    print

    b = Sigma('b')
    a = Alternative(b, r)
    print a
    print a.matches('b')
    print a.matches('aaaa')
    print a.matches('c')
    print a.consume('aaaaabaaa')
    print a.consume('dasdf')
    print

    c = Concatenation(b, a)
    print c
    print c.matches('ba')
    print c.matches('ab')
    print c.matches('baa')
    print c.consume('baba')
    print c.consume('bsa')
    print c.consume('baaaaaaa')
```

automata.py

```
class Automata:
    """Represents a finite automaton."""

    def __init__(self, states=None, start="", accepts=None, transitions=None, alphabet=None)
        # Take care of default arguments - avoids issue of defaults being mutable/evaluated
        if states is None: states = []
        if accepts is None: accepts = []
        if transitions is None: transitions = []
        if alphabet is None: alphabet = []

        self.start = start
        """Starting state for this automata. This is the name of the node as a string."""

        self.accepts = accepts
        """List of names of accepting states for the automata. These are strings."""

        self.nodes = {name: AutomataNode(name) for name in states}
        """Dictionary of nodes in the automata. The key is the state name, the value is the

        self.alphabet = set(alphabet)
        """A set of symbols that comprise the alphabet for this automaton."""

        self.transitions = transitions
        """The transitions in raw, lexed form. This is necessary for some algorithms (Hopcro

        # Set the accept state flag for all nodes in the accept list.
        for accState in accepts:
            self.nodes[accState].accept = True

        # Populate the nodes' transition dictionaries.
        for trans in transitions:
            fromState, symbols, toState = trans[0], trans[1], trans[2]
            for symbol in symbols:
                self.nodes[fromState].addTransition(toState, symbol)

    def __str__(self):
        ret = "Automaton:\n"
        ret += "Start:  " + self.start + '\n'
        ret += "Accept: " + str(self.accepts) + '\n'
        ret += "States: " + str([val.name for val in self.nodes.values()])
        return ret

    def getAllStates(self):
        return self.nodes
```

```
    def getStartState(self):
        return self.start

    def isAcceptState(self, state):
        return state in self.accepts

    def hasTransition(self, fromState, toState):
        return not (self.nodes[fromState].getTransitions(toState) is None)

    def addNodes(self, nodes):
        """Adds supplied nodes (already constructed) to the automata. Expects a list."""
        for node in nodes:
            self.nodes[node.name] = node


class AutomataNode:
    def __init__(self, name, accept=False):
        self.name = name
        """Name of this state. This should uniquely identify this state."""

        self.transitions = {}
        """Dictionary of symbol keys that returns lists of states."""

        self.accept = accept
        """Indicates that this node is an accept state."""

    def __str__(self):
        return self.name

    def __repr__(self):
        ret = "Node:    " + self.name + '\n'
        ret += "Accept: " + str(self.accept) + '\n'
        ret += "Transitions:\n"
        for key in self.transitions.keys():
            ret += "   " + key + " -> " + str([state for state in self.transitions[key]]) +

        return ret

    def __hash__(self):
        # Hash based entirely off of state name, as names *should* be unique.
        return hash(self.name)

    def __eq__(self, other):
        # Equality based entirely off of state name, as names *should* be unique.
        return self.name == other.name
```

```
    def getTransitionState(self, input_string):
        """Returns the state traversed to on a given input symbol, or none if no such trans
        if input_string in self.transitions:
            return self.transitions[input_string]
        else:
            return None

    def addTransition(self, toState, transSymbol):
        """Adds a transition to this state."""
        if transSymbol in self.transitions:
            self.transitions[transSymbol].append(toState)
        else:
            self.transitions[transSymbol] = [toState]
```

description_reader.py

```
#!/usr/bin/env python

"""description_reader.py: Contains scanner/lexer/tokenizer functions for reading from a fil

from pyparsing import *
from automata import Automata
from regex import *
from scanner import LexicalDesc

#######################
# General definitions #
#######################
arrow = Keyword("-->").suppress()
end_keyword = Keyword("end;").suppress()
def decodeEscapes(tokens):
    token = tokens[0]
    tokens[0] = token.decode('string_escape') if "\\" in token else token
    return tokens


# Alphabet definition
alphabet_keyword = Keyword("alphabet").suppress()
alphabet_end_keyword = Keyword("end;").suppress() | Keyword("end").suppress()
Symbol = Combine(Literal("\'").suppress() + Optional(Literal("\\")) + Word(printables + " ",
Symbol.setParseAction(decodeEscapes)
SymbolList = OneOrMore(Symbol)
Alphabet = alphabet_keyword + SymbolList + alphabet_end_keyword
# example: ['a, 'b, 'c]
```

```
# Regex definition:
RegexSymbol = Combine(Literal("\'") + Optional(Literal("\\")) + Word(printables + " ", exact
RegexSymbol.setParseAction(decodeEscapes)
Regex = ZeroOrMore(Literal('*') ^
                   Literal('|') ^
                   Literal('+') ^
                   RegexSymbol)
# example: ['b, +, *, |, 'a, 'o, 'r]


######################
# DFA/NFA definition #
######################
states_keyword = Keyword("states").suppress()
State = ~end_keyword + Word(alphanums)
StateList = ZeroOrMore(State)
States = states_keyword + StateList + end_keyword
# example: [s1, s2, s3]

initial_keyword = Keyword("initial").suppress()
InitialState = initial_keyword + State
# example: [s1, s2, s3]

accept_keyword = Keyword("accept").suppress()
AcceptingStates = accept_keyword + StateList + end_keyword
# example: [s1, s2, s3]

transitions_keyword = Keyword("transitions").suppress()
Transition = Group(State + Group(SymbolList) + arrow + State)
TransitionList = ZeroOrMore(Transition)
Transitions = transitions_keyword + TransitionList + end_keyword
# example: [ [s1, ['a, 'b], s2], [s2, ['a], s3] ]

automata_keyword = Keyword("dfa").suppress() ^ Keyword("nfa").suppress()
FiniteAutomata = automata_keyword \
                 + Group(States).setResultsName("States") \
                 + InitialState.setResultsName("Start") \
                 + Group(AcceptingStates).setResultsName("Accept") \
                 + Group(Transitions).setResultsName("Transitions") \
                 + Group(Alphabet).setResultsName("Alphabet")


###############################
# Complete Lexical Definition #
###############################
identifier = Word(printables)
```

```python
relevant_keyword = Keyword("relevant")
irrelevant_keyword = Keyword("irrelevant")
discard_keyword = Keyword("discard")
SemanticRelevance = relevant_keyword ^ irrelevant_keyword ^ discard_keyword

ClassDescription = Group(Regex)
# example: ["'b", "'a", "'o", "'r", "'i", "'n", "'g", "' ", "'\\n"]

class_keyword = Keyword("class").suppress()
is_keyword = Keyword("is").suppress()
Class = class_keyword + identifier + is_keyword + ClassDescription + SemanticRelevance + enc
# example: ['base', ['+', "'b", '+', '*', '|', "'a", "'o", "'r"], 'relevant'],

ClassList = ZeroOrMore(Group(Class))

language_keyword = Keyword("language").suppress()
LexicalDescription = language_keyword \
                     + identifier.setResultsName("Name") \
                     + Group(Alphabet).setResultsName("Alphabet") \
                     + Group(ClassList).setResultsName("Classes") \
                     + end_keyword
# example: [z++, [ [class 1], [class 2], [class 3] ] ]


def ConstructAutomata(file):
    """Parses the supplied automata file, then constructs and returns an Automata object.

       :param str | file file: File object or URI.
       :rtype: Automata
    """
    fa = FiniteAutomata.parseFile(file)
    # Note on fa.Start: parseResult objects always return values in lists, so this must be c
    return Automata(fa.States, fa.Start[0], fa.Accept, fa.Transitions, fa.Alphabet)


def ConstructRegex(file):
    """Parses the supplied regex, and constructs the appropriate Regex data structure found

       :param str | file file: File object or URI.
       :rtype: Regex
    """
    regex_tokens = Regex.parseString(file)
    return BuildExpression(regex_tokens)
```

```python
def ConstructLexicalDescription(file):
    """Parses the supplied lexical description file, then constructs and returns a lexical d

       :param str | file file: File object or URI.
       :rtype: LexicalDesc
    """
    lexDesc = LexicalDescription.parseFile(file)
    print lexDesc
    return LexicalDesc(lexDesc.Name, lexDesc.Alphabet, lexDesc.Classes)

if __name__ == "__main__":
    # ConstructLexicalDescription("testdata/lexdesc2.txt")
    test = ConstructAutomata("testdata/dfa4.txt")
    for symbol in test.alphabet:
        print symbol
```

dfa_read.py

```python
from description_reader import ConstructAutomata

def dfa_valid_string(automata, testing_string, current_state, current_step):
    if current_step == len(testing_string):
        for x in automata.accepts:
            if current_state == x: return True
        return False
    else:
        next_state = dfa.nodes[current_state].getTransitionState(testing_string[current_step
        if next_state is None: return False
        else: return dfa_valid_string(automata, testing_string, next_state, current_step + 1

if __name__ == "__main__":
    dfa = ConstructAutomata("testdata/dfa2.txt")
    start = dfa.getStartState()
    string = "aaaa"
    print dfa_valid_string(dfa, string, start, 0)
```

scanner.py

```python
from regex import BuildExpression
import description_reader

class LexicalDesc:
    """Encapsulates a complete lexical description."""

    def __init__(self, name, alphabet, classes):
        """
```

```python
        :param name: The name of this description as a string.
        :param alphabet A list of backquoted symbols as produced by description_reader.py.
        :param classes: A list of parsed classes. Each class is a list of tokens as produced
        """
        self.name = name
        self.alphabet = alphabet
        self.classes = [LexicalClass(c[0], c[1], c[2]) for c in classes]

    def scan(self, string_to_scan, tokens=[]):
        '''
        Scans a string and produces a list of Tokens parsed from
        the string.

        :param string_to_scan the string that will be turned into a list
               of Token objects.
        :return a list of Token object recognized by the string. If the
                string cannot be fully recognized (ie, there is part of the
                string left after scanning completely) an empty list will be
                returned.
        '''
        if string_to_scan == '':
            return tokens

        '''
           Represents the leftmost parse of the parse tree.
        '''
        for c in self.classes:
            matched, leftover = c.regex.consume(string_to_scan)

            #if we do get a match using the regex
            if matched != '':
                if c.relevance != 'discard':
                    new_tokens = tokens + [Token(matched, c.name, c.relevance)]
                else:
                    new_tokens = tokens
                try:
                    return self.scan(leftover, new_tokens)
                except Exception as e:
                    continue

        '''
            If we get here, that means there was no logical parse
            using the regexes we were given. We should return an error
            at this point.
        '''
        raise Exception(string_to_scan)
```

```python
class LexicalClass:
    """Describes a lexical class using a regular expression."""

    def __init__(self, name, class_tokens, relevance):
        """
        :param name: The name of this class.
        :param class_tokens: The tokens comprising this class regex, in a list.
        :param relevance: The semantic relevance of this class.
        """
        self.name = name
        self.regex, ignored = BuildExpression(class_tokens)
        self.relevance = relevance

    def __str__(self):
        return "Name: " + self.name + ", Regex: " + str(self.regex) + ", Relevance: " + self


class Token:
    '''Essentially a tuple of a String, LexicalClass, and a relevance as
       defined in the lexical desciption of the grammar
    '''

    def __init__(self, string, lex_class_name, relevance):
        self.string = string
        self.lexical_class = lex_class_name
        self.relevance = relevance

    def __str__(self):
        return "Class: " + str(self.lexical_class) + "\n\tString: " + str(self.string)

if __name__ == "__main__":
     desc = description_reader.LexicalDescription.parseFile("./testdata/tiny_basic_lex_desc
     tiny_basic = LexicalDesc(desc.Name, desc.Alphabet, desc.Classes)
     f = open('./testdata/tinyBasicProgram.txt')
     basic_program = f.read()

     print basic_program
     '''
     num = None
     for c in tiny_basic.classes:
         if c.name == 'number':
             num = c.regex

     print num
```

```
    print num.matches('2')
    print num.consume('2')
    r, over =  num.right.consume('2')
    l, over = num.left.consume('2')
    print len(l), len(r)
'''
    #TODO -- TEB: fails when scanning due to newline characters not being
    #             recongized.
    for c in tiny_basic.scan(basic_program):
        print c
```

thomsons_construction.py

subset_construction.py

```
from automata import *

EPSILON = '\0'


def epsilonClosure(state, visitedStates, nfa):
    """ Calculates the epsilon closure over a state. In English, this
        function returns the set of all states reachable via epsilon-transitions
        from the initially provided state. Note that this returned set will
        include the original state as any state can reach itself via an epsilon-
        transition. This function can traverse over multiple epsilon transitions
        and thus can return states further than one transition.

        :param AutomataNode state: Initial state.
        :param set[AutomataNode] visitedStates: Set of states already visited.
        :param Automata nfa: The automaton that state belongs to.
        :rtype: set[AutomataNode]
    """

    reachableStates = set()     # Stores all states that can be reached via epsilon transit
    reachableStates.add(state)  # A state can always reach itself via epsilon transition.
    visitedStates.add(state)    # Visit this node so that further recursion doesn't re-add

    # For each state reachable by epsilon transition that has not ben visited yet, calculat
    # of that state and add it to the reachableStates set.
    if EPSILON in state.transitions:
        for nextState in state.transitions[EPSILON]:
            if nfa.nodes[nextState] not in visitedStates:
                # Set data structure automatically discards duplicates.
                reachableStates |= epsilonClosure(nfa.nodes[nextState], visitedStates, nfa)
```

```python
    return reachableStates


def move(states, symbol, nfa):
    """ For a given set of states, returns all states reachable on a given input.

        :param set[AutomataNode] states: Set of states to transition from.
        :param str symbol: Input symbol over which to transition.
        :param Automata nfa: The automaton that states belongs to.
        :rtype: set[AutomataNode]
    """

    reachableStates = set()

    for state in states:
        if symbol in state.transitions:
            # List comprehension so that we can get state objects back instead of strings.
            reachableStates |= set([nfa.nodes[name] for name in state.transitions[symbol]])

    return reachableStates


def convertNfaToDfa(nfa):
    """ Converts an NFA into a DFA via epsilon closure and subset construction.
        This code is based off of this pseudocode:

        ===============================================================
        D-States = EpsilonClosure(NFA Start State) and it is unmarked
        while there are any unmarked states in D-States
        {
            mark T
            for each input symbol a
            {
                U = EpsilonClosure(Move(T, a));
                if U is not in D-States
                {
                    add U as an unmarked state to D-States
                }
            DTran[T,a] = U
            }
        }
        ===============================================================

        :param Automata nfa: The non-deterministic finite automata to convert.
        :rtype: Automata
    """
```

```python
# The DFA being constructed.
dfa = Automata()

# Maps new state names to the collection of states they encompass. During the construct
# new DFA, composite states such as {s0,s1,s2} can be created, which is itself just one
# needs to point to each of the three states in the NFA it was constructed from.
nameToStates = dict()

# Sets to keep track of states that are waiting to be processed, and that have been pro
markedStates = set()
unmarkedStates = set()

# Create initial DFA state from epsilon closure over NFA initial state.
initStateClosure = epsilonClosure(nfa.nodes[nfa.start], set(), nfa)
dfaInitState = AutomataNode(stateSetName(initStateClosure))

# Add the initial composite state to the new DFA, the unmarked list, and the name map.
dfa.addNodes([dfaInitState])
dfa.start = dfaInitState.name
unmarkedStates.add(dfaInitState)
nameToStates[dfaInitState.name] = initStateClosure

while unmarkedStates:
    # Mark the new state
    state = unmarkedStates.pop()
    markedStates.add(state)

    # Generate set of states from epsilon closures over every state returned from this
    for symbol in nfa.alphabet:
        moveStates = move(nameToStates[state.name], symbol, nfa)

        # Only proceed if move produced states. Empty set means we don't bother with eps
        if moveStates:
            closureSet, visitedStates = set(), set()
            for moveState in moveStates:
                closureSet |= epsilonClosure(moveState, visitedStates, nfa)

            # Generate new DFA state from combined epsilon closures.
            newDfaState = AutomataNode(stateSetName(closureSet))

            # If this new state is actually new, add new state to DFA, the unmarked lis
            if (newDfaState not in unmarkedStates) and (newDfaState not in markedStates)
                dfa.addNodes([newDfaState])
                unmarkedStates.add(newDfaState)
                nameToStates[newDfaState.name] = closureSet.copy()
```

```
                        # Add this node to the accept list if it is based off of an accept node.
                        for node in closureSet:
                            if node.accept:
                                newDfaState.accept = True
                                dfa.accepts.append(newDfaState.name)
                                break

                    # Add the transition to this (new) state.
                    state.addTransition(newDfaState, symbol)

    return dfa


def stateSetName(states):
    """ Creates a name for a state derived from a supplied set of states. The name is order-
        :param set[AutomataNode] states: The states to derive a name from.
        :rtype: str
    """

    # Note: used list comprehension here instead of just printing list because list uses rep
    namesList = [str(item) for item in states]  # set(['c', 'b', 'a']) -> list['c', 'b', 'a
    namesList = str(sorted(namesList))          # list['c', 'b', 'a']  -> ['a', 'b', 'c']
    namesList = namesList.replace('\'', '')     # ['a', 'b', 'c']      -> [a, b, c]
    namesList = namesList.replace(' ', '')      # [a, b, c]            -> [a,b,c]
    namesList = (namesList[1:])[:-1]            # [a,b,c]              -> a,b,c
    return namesList


if __name__ == "__main__":
    # Recreate NFA from website
    s1 = AutomataNode('s1')
    s2 = AutomataNode('s2')
    s3 = AutomataNode('s3')
    s4 = AutomataNode('s4')
    s5 = AutomataNode('s5')

    s1.addTransition('s2', EPSILON)
    s1.addTransition('s4', EPSILON)
    s2.addTransition('s3', 'a')
    s3.addTransition('s3', 'b')
    s4.addTransition('s4', 'a')
    s4.addTransition('s5', 'b')

    s3.accept = True
    s5.accept = True
```

```
    testNFA = Automata()
    testNFA.start = s1.name
    testNFA.alphabet = ['a', 'b']
    testNFA.addNodes([s1, s2, s3, s4, s5])

    testDFA = convertNfaToDfa(testNFA)
    for node in testDFA.nodes.values():
        print repr(node)
```

hopcrofts_algorithm.py

```
from automata import Automata
from description_reader import ConstructAutomata

def hopcroftMinimize(file):
    dfa = ConstructAutomata(file)
    P = [set(dfa.accepts),set(dfa.nodes.keys()).difference(set(dfa.accepts))]
    W = []
    if len(set(dfa.accepts)) > 1 : W.append(set(dfa.accepts))
    if len(set(dfa.nodes.keys()).difference(set(dfa.accepts))): W.append(set(dfa.nodes.keys(
    #print "P is :",P
    #print "W is :",W
    while len(W) != 0:
        A = W[0]
        W.remove(A)
        for c in dfa.alphabet:
            X = []
            for from_state in A:
                to_state = dfa.nodes[from_state].getTransitionState(c)
                if to_state is not None: to_state = to_state[0]  # Only one possible state f
                #print "from_state: ",from_state," using ",c," TO ",to_state
                if len(X) == 0 : X.append(from_state)
                else:
                    x_to_state = dfa.nodes[X[0]].getTransitionState(c)
                    for p in P:
                        if to_state in p and x_to_state in p: X.append(from_state)
            if not len(X) == len(A):
                X1 = set(X)
                X2 = set(A).difference(X1)
                P.remove(set(A))
                P.append(X1)
                P.append(X2)
                if len(X1) > 1 : W.append(X1)
                if len(X2) > 1 : W.append(X2)
                    #print "------------------------------"
```

```
            break
new_states = []
for s in dfa.nodes.keys():
    new_states.append(s)
new_accept = []
for s in dfa.accepts:
    new_accept.append(s)
new_transitions = []
for s in dfa.transitions:
    new_transitions.append(s)

#print dfa
#print P

for p in P:
    if len(p) > 1:
        this_state = None
        if dfa.start in p: this_state = dfa.start
        for current_state in p:
            if this_state == None: this_state = current_state
            elif not this_state == current_state:
                #print new_states
                #print current_state
                if current_state in new_states:
                    new_states.remove(current_state)
                #print new_states
                if current_state in new_accept:
                    new_accept.remove(current_state)
                    new_accept.append(this_state)
                remove_transitions = []
                for transition in new_transitions:
                    if transition[0] == current_state:
                        remove_transitions.append(transition)
                    elif transition[2] == current_state:
                        transition[2] = this_state
                for p in remove_transitions:
                    new_transitions.remove(p)

return Automata(new_states, dfa.start, new_accept, new_transitions,dfa.alphabet)
```