Project 2 Report

Taylor Berger, Zachary Friedland, Jianyu Yang

November 9, 2014

1 Language Decisions

Our group continued to use Python for the ease of expressing high level concepts and removal of memory management from the project. We also used Git for version control the collaboration tools. The repository can be found at Taylor's Github.

We continued our use of PyParsing to read in the grammar rules defined in external files.

2 Data Structures for a Context Free Grammar

The complete defintions for our data structures can be found in section 10.1. However, we will break them down in the order presented in the project specification and outline the reason they were created and the features implemented.

2.1 Productions

The first data structure we implemented is a structures for a grammar production. The implementation of the data structure is a decorator for a pair of symbols. It simply groups the symbols together into a left and right half. The right half is allowed to be a list of symbols whereas the left half must only be one symbol.

```python
class Production:
    """ Represents a production. """

    def __init__(self, lhs, rhs):
        """
        :param str lhs: Left-hand-side symbol.
        :param list[str] rhs: List of right-hand-side symbols.
        """
        self.lhs = lhs
        self.rhs = rhs

#... property functions excluded for brevity ...
```

2.2 Grammar

A grammar object is the logical extension of having a single production. In our implementation, grammars are a dictionary of symbol to list of list of symbols. The nested list returned represents all productions that can come from the non-terminal on the left hand side of a production. Therefore if the set of productions looks like:

```
A -> aaa
A -> b
```

Then, the resulting dictionary lookup for `A` would return the list `['aaa','b']`.

When constructing the grammar, the first left hand side of a production encountered is considered the start symbol. So upon parsing and reading that symbol an internal pointer to the start symbol is maintained by the grammar object. The definition for the constructor of a grammar is as follows:

```
1  class Grammar:
2      """ Data structure for context-free grammar. """
3
4      def __init__(self, grammar=None):
5
6          self.productions = dict()
7          self.nonTerminals = set()
8          self.start = ""
9
10         if grammar is not None:
11             # Convert ParseResult objects into Productions.
12             parse = pyp_Grammar.parseFile(grammar)
13             productions = [Production(p.lhs, p.rhs.asList()) for p in parse]
14
15             # First production's left-hand-side is start symbol.
16             self.start = productions[0].lhs
17
18             # Hygiene checks.
19             productions = Generating(productions)
20             if self.start not in set(p.lhs for p in productions):
21                 raise Exception("Starting production is non-generating!")
22             productions = Reachable(productions, self.start)
23
24             # Add all of the productions to the grammar.
25             for production in productions:
26                 self.addProduction(production.lhs, production.rhs)
```

As you can see, on lines 19 and 22, there are grammar hygiene checks that are performed that remove non-generating and non-reachable productions. Otherwise, a grammar is constructed from the string parameter which points to a relative file location. Line 12 is the only location in which PyParsing is used in the project.

3 Context Free Grammar Representation

We chose to stick with the representation given in the project specification document. We found there were no changes need to be made to successfully represent a grammar. For completeness, the representation is below:

```
Grammar -> Grammar Production
Grammar ->
Production -> symbol Arrow List
Arrow -> '->
List -> symbol List
List ->
```

We note that a `symbol` in this context is any whitespace delimited string of printable ascii characters.

4 Parser for CFG Representation

Using PyParsing, the following is the representation for the representation given in section 3:

```python
# pyparsing definitions for above-specified context-free grammar.
pyp_Arrow = Keyword("->").suppress()
pyp_Symbol = Word(alphanums)
pyp_List = ZeroOrMore(~LineStart().leaveWhitespace() + Word(printables))
pyp_Production = Group(pyp_Symbol.setResultsName("lhs") +
                       pyp_Arrow.suppress() +
                       Group(pyp_List).setResultsName("rhs"))
pyp_Grammar = ZeroOrMore(pyp_Production)
```

5 Grammar Hygiene Checks

After reading in a grammar, certain hygiene checks must be made. We must ensure that we can reach all non-terminals and we also must ensure that we can generate a terminal string with every production.

5.1 Generating

We first check for non-generating productions as follows:

```python
def Generating(productions):
    """
      ... comment excluded for brevity ...
    """
    nonTerms = set(production.lhs for production in productions)
    productive = set()

    # Repeat until no new results are yielded.
```

```
    previousSize = -1
    while previousSize < len(productive):
        previousSize = len(productive)

        for production in productions:
            # Don't bother with productions that are already marked.
            if production.lhs not in productive:

                # Empty rhs is productive.
                if not production.rhs:
                    productive.add(production.lhs)

                # If every symbol in the RHS is productive, lhs is
                # productive.
                else:
                    isProductive = True
                    for symbol in production.rhs:
                        if symbol in nonTerms and symbol not in productive:
                            isProductive = False
                            break

                    if isProductive:
                        productive.add(production.lhs)

    # Build list of all productions with generating left-hand-sides,
    # but remove any that have non-generating variables in their
    # right-hand-side.
    unproductive = nonTerms - productive
    return [p for p in productions if
            p.lhs in productive
            and set(p.rhs).isdisjoint(unproductive)]
```

5.2 Unreachables

We also remove any non-terminals that are unreachable and, consequently, any productions including those unreachable non-terminals.

```
def Reachable(productions, start):
    """
       ... comment ommited for brevity ...
    """
    nonTerms = set(production.lhs for production in productions)
    reachable = {start}
    marked = set(filter((lambda p: p.lhs == start), productions))
    unmarked = set(productions) - marked

```

```python
    # Repeat until no new results are yielded.
    previousSize = -1
    while previousSize < len(reachable):
        previousSize = len(reachable)

        # Find all new reachable non-terminals in current marked set. Generate
        # a new marked set from all newly reachable non-terminals.
        nextMarked = set()
        for production in marked:
            for symbol in production.rhs:
                if symbol in nonTerms and symbol not in reachable:
                    nextMarked |= set(filter((lambda p: p.lhs == symbol), unmarked))
                    reachable.add(symbol)

        marked = nextMarked
        unmarked -= marked

    # Create new list of only reachable rules.
    return [p for p in productions if p.lhs in reachable]
```

6 Nullable, First and Follows

To construct the LL1 parse table, we first must be able to compute First and Follows. Both of those also use the set of all non-termianls that are nullable. We'll start with nullable since it is the stand alone computation.

6.1 Nullable

The set of all non-terminals that are nullable (can be removed through a series of productions ending with the null character) are:

```python
def nullable(grammar):
    '''
        ...excluded for brevity...
    '''

    nullable = set()
    productions = grammar.productions

    cardinality = -1
    while cardinality < len(nullable):
        cardinality = len(nullable)

        for non_term in productions:
            #if epsilon is in the rhs already,
            #the production is nullable
            if [] in productions[non_term]:
```

```python
                nullable.add(non_term)
            else:
                isNullable = False
                for N in productions[non_term]:
                    #check to see if this specific production is
                    # nullable by checking to see if the join set of
                    # all symbols in the production are nullable.
                    # If they are not, that production is not nullable
                    isProductionNullable = True
                    for symbol in N:
                        isProductionNullable &= symbol in nullable
                    # This is the disjoint set of all nullable
                    # productions the correspond to this lhs
                    isNullable |= isProductionNullable

                # if any of the disjoint productions are nullable,
                # then this is true. Therefore, add this production's
                # lhs to the nullable set
                if isNullable:
                    nullable.add(non_term)
    return nullable
```

6.2 First

Now, to compute the set of terminals that could be the first terminals after a non-terminal is replaced with a production, we calculate the closure as:

```python
def first(grammar):
    '''
       excluded for brevity
    '''
    productions = grammar.productions
    #define the nullables
    nullable_non_terms = nullable(grammar)

    #initially set our table to the empty set of terminals
    prev_table = {non_term : set() for non_term in grammar.nonTerminals}
    #add just the productions of the form:
    #   non_terminal -> terminal
    #to start the algorithm off
    for non_term in grammar.nonTerminals:
        for N in productions[non_term]:
            if [] == N: continue
            if N[0] not in grammar.nonTerminals:
                prev_table[non_term].add(N[0])

```

```
    has_changed = True

    while has_changed:
        has_changed = False
        #construct the new table so we don't interfere with the
        #previous one by adding things we wouldn't add this iteration
        new_table = prev_table.copy()

        for non_term in grammar.nonTerminals:
            for rhs in productions[non_term]:
                # if we have an epsilon, ignore it. It is the
                # equivalent of adding the empty set.
                if [] == rhs or rhs[0] not in grammar.nonTerminals:
                    continue

                first = rhs[0]
                # Now, we need to check to see if adding anything from
                # the first item in the rhs changes the current set of
                # terminals we have for this non_term

                # this is done by seeing if the rhs's first
                # non-terminal's first set can add anything to the
                # current set of terminals for non_term
                if len(prev_table[first] - prev_table[non_term]) > 0:
                    new_additions = prev_table[first] - prev_table[non_term]
                    #skip past all the nullables until we hit the end
                    #or we find a non-terminal. Add all the first sets
                    #for the next item in the production as we come
                    #across them
                    i = 1
                    while i+1 < len(rhs) and rhs[i] in nullable_non_terms:
                        if rhs[i + 1] in grammar.nonTerminals:
                            new_additions |= prev_table[rhs[i + 1]]
                        else:
                            #must list-ify to match the rest in the
                            #set
                            new_additions |= set([rhs[i+1]])
                            break
                        i += 1

                    new_table[non_term] |= new_additions
                    has_changed = True

        prev_table = new_table.copy()
    return prev_table
```

6.3 Follows

To finally wrap up the closures needed to compute the LL1 parse table, we construct the set of terminals that follow any non-terminal by:

```
def follows(grammar):
    '''
       excluded for brevity
    '''
    nullable_non_terms = nullable(grammar)
    first_table = first(grammar)
    productions = grammar.productions

    #initalize the table to contain only the empty sets
    follow_table = {non_term : set() for non_term in grammar.nonTerminals}
    #add the EOF symbol for the start state
    #TODO: Correct symbol for eof? What should we do here?
    follow_table[grammar.start].add('$')

    has_changed = True
    #iterate until all sets have not changed
    while has_changed:
        has_changed = False

        #construct the new table for us to put additions into
        new_table = follow_table.copy()

        #construct all the follow sets for every non-terminal
        for non_term in grammar.nonTerminals:
            #get the dictionary of all {lhs : 'beta' values} (lists of
            #expressions following the non-terminal)n
            betas_following_term = betas_following(non_term, productions)

            #Get the lhs of the production, call it M (like in the book)
            for M in betas_following_term.keys():
                #For every beta, calculate the following...
                for beta in betas_following_term[M]:
                    i = 0
                    while i < len(beta):
                        #iterating from the first to the last term in the list of
                        #symbols
                        beta_term = beta[i]

                        # Case where beta is a non-terminal:
                        # Follows(non_term) = first(beta) U follows(non_term)
                        if beta_term in grammar.nonTerminals:
                            # if we see a value that's not in follows(non_term), add
```

```
                            # it and set the changed flag to True
                            if not first_table[beta_term] <= follow_table[non_term]:
                                has_changed = True
                                new_table[non_term] |= first_table[beta_term]
                        # Case where beta term is a terminal and we
                        # haven't seen it before add the non-terminal
                        # to the follows set and set the changed
                        # flag to True
                        elif beta_term not in follow_table[non_term]:
                            has_changed = True
                            new_table[non_term] |= set([beta_term])

                        # if the beta_term is not nullable, we are
                        # done with this list of symbols
                        if beta_term not in nullable_non_terms:
                            break

                        i += 1

                    # case where all of the symbol list is nullable, in which
                    # we need to say follows(M) = follows(M) U follows(non_term)
                    if i == len(beta):
                        if not follow_table[M] <= follow_table[non_term]:
                            has_changed = True
                            new_table[M] |= follows_table[non_term]

        #update our table to point to the new one
        follow_table = new_table.copy()
    return follow_table
```

Where the function `betas_following_term` can be found in the full code for ll1_tools.py in section 10.2. With all three closures calculated, we can now construct the parse table (defined in the following section 7).

7 LL1 Parse Table

Constructing the Parse Table using the first, follows, and nullable definitions becomes a straigtforward calculation of three situations. With an intially empty table T, an end of file character, #, and for each production of the form $A \to \alpha$:

$$T[A,t] = \alpha, \quad \text{where, } t \in \text{First}(\alpha) \tag{1}$$

$$T[A,t] = \alpha, \quad \text{where, } \epsilon \in \text{First}(\alpha) \text{ and } t \in \text{Follows}(A) \tag{2}$$

$$T[A,\#] = \alpha, \quad \text{where, } \epsilon \in \text{First}(\alpha) \text{ and } \# \in \text{Follows}(A) \tag{3}$$

We implemented the parse table as a separate class and is created by passing in a grammar into the class initializer. The table is built during the construction of the object and a flag is set inside the parse table if the grammar is ll1. The construction of the table is as follows:

```python
class ParseTable:
    """ Represents a parse table. """

    def __init__(self, grammar):
        self.table = dict()
        self.isLl1 = True
        self.grammar = grammar

        first_dict = first(grammar)
        follows_dict = follows(grammar)
        nullables_set = nullable(grammar)

        # Initialize all table cells to the empty list.
        for nonTerminal in grammar.nonTerminals:
            self.table[nonTerminal] = dict()
            # also initialize the epsilon and eof characters
            # to have empty cells as well
            self.table[nonTerminal][EPSILON] = []
            self.table[nonTerminal][EOF] = []
            for terminal in grammar.terminals:
                self.table[nonTerminal][terminal] = []

        # Build the table.
        for lhs in grammar.productions:
            # All right hand sides of a production in the
            # form: A -> alpha1 | alpha2 | alpha3 as a list
            alphas = grammar.productions[lhs]

            # For every right hand side above
            for alpha in alphas:

                #calculate First(alpha) using first dictionary and nullables
                #this is First(alpha[0]) U First(alpha[i]) where i = 1 : len(alpha)
                #and stop i when alpha[i] is not in nullables
                first_of_alpha = create_first_from_list(first_dict, nullables_set, alpha)

                # for all 't' that exist in First(alpha), where 't' is always
                # a terminal
                for t in first_of_alpha:
                    #add this to the table for 't'
                    self.table[lhs][t].append(alpha)

                    # Check for multiple entries in cell - LL1 check.
```

```
44                  if len(self.table[lhs][t]) > 1:
45                      self.isLl1 = False
46
47              # Now, if EPSILON exists in First(alpha), then we have a couple
48              # other additions to make
49              if EPSILON in first_of_alpha:
50                  # For every terminal, t, in Follows(A), add an entry from [A][t] = alpha
51                  for t in follows_dict[lhs]:
52                      self.table[lhs][t].append(alpha)
53
54                      # Check for multiple entries in cell - LL1 check.
55                      if len(self.table[lhs][t]) > 1:
56                          self.isLl1 = False
57
58                  #Also, if EOF exists in Follows(A), then add entry: [A][EOF] = alpha
59                  if EOF in follows_dict[lhs]:
60                      self.table[lhs][EOF].append(alpha)
61                      # Check for multiple entries in cell - LL1 check.
62                      if len(self.table[lhs][EOF]) > 1:
63                          self.isLl1 = False
```

As you can see on lines 45, 56, and 64, if at any point, the table entry is a list of length greater than 1 the grammar is not LL(1) and we set the flag to false.

7.1 LL(1) versus Strong LL(1)

We have tried to find the difference between Strong LL(1) and LL(1) grammars in multiple references but there seems to be no consesus what it means to be a Strong LL(1) grammar. We chose to follow the definition in chapter 8 of the parsing book *Parsing Techniques* by Grune and Jacobs. The definition given is: if your language is LL(1) it is also Strong LL(1). This implies that if your language is Strong LL(1), it is also LL(1), meaning there is no difference between the two.

We're not sure if there is a more formal definition somewhere, but we took a definition out of a recent textbook to be formal enough.

8 Parser Construction

Regardless of whether or not a grammar is LL(1) or not, we still construct a parser for the grammar. However, if at any point the parser identifies the possibility of having two different parses by having more than one entry in the parse table, it fails fast by raising an error. The parser builds a Rose Tree as it consumes input where the children of any particular node are the sequence of tokens put on the parse stack (identified by the parse table).

The parser and LL(1) parse method are as follows:

```
class Parser:

    def __init__(self, grammar=None):
        #storing this just in case
        self.grammar = grammar
        #start the stack with just the start node
        self.parse_stack = [grammar.start]
        #construct the parse table
        self.table = ParseTable(grammar).table

    def ll1_parse(self, rose_tree, token_list):
        ''' excluded for brevity '''

        # if the stack is empty, return the current parse and leftover input
        if not self.parse_stack:
            return rose_tree, token_list

        # if there's no tokens left and we are here, parse error
        if not token_list:
            raise ValueError("Unexpected EOF, current parse stack is:" + str(self.parse_stack.reverse()

        # Get the first symbol
        current_symbol = self.parse_stack.pop()
        # one token look-ahead
        token = token_list[0]

        #if we have a matching symbol and token, we can consume the input
        if current_symbol in self.grammar.terminals:
            leaf = Rose_Tree(current_symbol, token)
            return leaf, token_list[1:]

        #else we have a non-terminal, we must continue with the rewrite
        #returns a list of possible productions that should follow
        production_to_follow = self.table[current_symbol][token]

        # if empty production to follow, unexpected terminal found:
        if not production_to_follow:
            raise ValueError("Unexpected terminal, " + str(token) + ", found.")

        # we can't handle this in LL1 style parsing
        if len(production_to_follow) > 1:

            print production_to_follow
            raise ValueError("Too many possible parses for LL1, this is non-deterministic. "
                             "Please check your grammar. Current parse: " +
                             str(self.parse_stack[0:-1]) + " on terminal " + str(token))
```

```

        # push all symbols onto the stack
        for symbol in production_to_follow[0]:
            self.parse_stack.append(symbol)


        #construct a node in the tree and attach all children parsed
        current_rose_tree_node = Rose_Tree(current_symbol, "")

        leftover = token_list
        for symbol in production_to_follow[0]:
            #this can never just return from an empty stack since we place
            #all symbols found on the stack before this. See lines 54 & 55
            parsed_tree_node, leftover = self.ll1_parse(current_rose_tree_node, leftover)

            # append as a child
            current_rose_tree_node.children.append(parsed_tree_node)

            # point back to parent
            parsed_tree_node.parent = current_rose_tree_node

        #finished parsing for this node completely, return the node and any leftover input
        return current_rose_tree_node, leftover
```

9 Team Composition

Work on this project was divided into four main parts: Reading and cleaning grammars, algorithms for LL1 languages, parsetable construction/parsing, and writing the report.

Reading and cleaning grammars was primarily worked on by Zach, with bug-fixes and modifications by Taylor. As the program evolved, necessary changes to these utilities were made apparent and a group effort was made to modify this section.

Work on the LL1 algorithms and tools were discussed and planned by the whole group. The final implementation is mostly the product of Taylor's hard work. Collective group effort was involved in late changes to this code.

The third section was initially drafted via pair (triplet?)-programming between Taylor, Justin, and Zach. Evolution of this code to the final product was mostly effected by Taylor with assistance from Zach via a Google Hangout, pair-programming session.

Finally, the report was written almost entirely by Taylor.

10 Full Code

10.1 cfg.py

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
```

```python
""" Data structures for representing context-free grammars over ASCII alphabets
    and parsing functions to read grammar descriptions in files. The grammars
    are specified formally by:

    Grammar -> Grammar Production
    Grammar -> Production
    Production -> Symbol Arrow List
    List -> List Symbol
    List ->

    Where any symbols appearing left of the arrow are non-terminals, and the
    non-terminal of the first production is starting production. Productions are
    separated by lines.
"""

from pyparsing import *


# pyparsing definitions for above-specified context-free grammar.
pyp_Arrow = Keyword("->").suppress()
pyp_Symbol = Word(printables)
pyp_List = ZeroOrMore(~LineStart().leaveWhitespace() + Word(printables))
pyp_Production = Group(pyp_Symbol.setResultsName("lhs") +
                       pyp_Arrow.suppress() +
                       Group(pyp_List).setResultsName("rhs"))
pyp_Grammar = ZeroOrMore(pyp_Production)

EOF = '\0'
EPSILON = ''

class Production:
    """ Represents a production. """

    def __init__(self, lhs, rhs):
        """
        :param str lhs: Left-hand-side symbol.
        :param list[str] rhs: List of right-hand-side symbols.
        """
        self.lhs = lhs
        self.rhs = rhs

    def __str__(self):
        return str(self.lhs) + " -> " + str(self.rhs)

    def __repr__(self):
```

```
        return str(self)

    def __hash__(self):
        return hash(str(self))

    def __eq__(self, other):
        return isinstance(other, Production)\
               and self.lhs == other.lhs\
               and self.rhs == other.rhs


class Grammar:
    """ Data structure for context-free grammar. """

    def __init__(self, grammar=None):

        self.productions = dict()
        self.nonTerminals = set()
        self.terminals = set()
        self.start = ""

        if grammar is not None:
            # Convert ParseResult objects into Productions.
            parse = pyp_Grammar.parseFile(grammar)
            productions = [Production(p.lhs, p.rhs.asList()) for p in parse]

            # First production's left-hand-side is start symbol.
            self.start = productions[0].lhs

            # Hygiene checks.
            productions = Generating(productions)
            if self.start not in set(p.lhs for p in productions):
                raise Exception("Starting production is non-generating!")
            productions = Reachable(productions, self.start)

            # Add all of the productions to the grammar.
            for production in productions:
                self.addProduction(production.lhs, production.rhs)

            # Determine set of terminals by finding set of all symbols, then
            # subtracting the intersection of all symbols with non-terminals.
            for righthandsides in self.productions.values():
                for rhs in righthandsides:
                    self.terminals |= set(rhs)

            self.terminals = self.terminals - self.nonTerminals
```

```python
    def addProduction(self, lhs, rhs):
        """Adds a production to the grammar. If the production's LHS already
        exists in the dictionary, the RHS is appended to the value
        list. If the LHS is not in the dictionary, it is added, and
        the RHS is added as a list.

        :param str lhs: Left-hand-side of the production.
        :param list[str] rhs: Right-hand-side of the production.

        """
        # Convert empty lists into epsilon character.
        if not rhs:
            rhs = EPSILON

        if lhs in self.productions:
            self.productions[lhs].append(rhs)
        else:
            self.productions[lhs] = [rhs]

        self.nonTerminals.add(lhs)


def Generating(productions):
    """Returns a list of generating rules from a list of productions.

    This algorithm first identifies all initially-productive rules: -
    Rules with only terminals on the right-hand-side.  - Rules with
    the empty string (epsilon) on the right-hand-side.  Productions
    are then marked productive if their right-hand-side consists of
    only terminals and non-terminals marked as productive. This
    process is repeated until no new results are yielded. Any
    productions not marked as productive at this point are
    unproductive.

    :param list[Production] productions: List of productions to examine.
    :rtype: list[Production]

    """
    nonTerms = set(production.lhs for production in productions)
    productive = set()

    # Repeat until no new results are yielded.
    previousSize = -1
    while previousSize < len(productive):
        previousSize = len(productive)
```

```python
        for production in productions:
            # Don't bother with productions that are already marked.
            if production.lhs not in productive:

                # Empty rhs is productive.
                if not production.rhs:
                    productive.add(production.lhs)

                # If every symbol in the RHS is productive, lhs is
                # productive.
                else:
                    isProductive = True
                    for symbol in production.rhs:
                        if symbol in nonTerms and symbol not in productive:
                            isProductive = False
                            break

                    if isProductive:
                        productive.add(production.lhs)

    # Build list of all productions with generating left-hand-sides,
    # but remove any that have non-generating variables in their
    # right-hand-side.x+y
    unproductive = nonTerms - productive
    return [p for p in productions if
            p.lhs in productive
            and set(p.rhs).isdisjoint(unproductive)]


def Reachable(productions, start):
    """
    Returns a list of reachable rules from a list of productions.

    This algorithm initially marks the start productions as reachable. For every
    reachable state, all non-terminals in the reachable states' right-hand-sides
    are then marked as reachable (with a bit of checking to make sure it doesn't
    repeatedly check the same rules). This process is repeated until no new
    reachable states are found.

    :param list[Production] productions: List of productions to examine.
    :param str start: The starting non-terminal.
    :rtype: list[Production]
    """
    nonTerms = set(production.lhs for production in productions)
    reachable = {start}
```

```
marked = set(filter((lambda p: p.lhs == start), productions))
unmarked = set(productions) - marked

# Repeat until no new results are yielded.
previousSize = -1
while previousSize < len(reachable):
    previousSize = len(reachable)

    # Find all new reachable non-terminals in current marked set. Generate
    # a new marked set from all newly reachable non-terminals.
    nextMarked = set()
    for production in marked:
        for symbol in production.rhs:
            if symbol in nonTerms and symbol not in reachable:
                nextMarked |= set(filter((lambda p: p.lhs == symbol), unmarked))
                reachable.add(symbol)

    marked = nextMarked
    unmarked -= marked

# Create new list of only reachable rules.
return [p for p in productions if p.lhs in reachable]
```

10.2 ll1_tools.py

```python
from cfg import Grammar, EOF, EPSILON


def nullable(grammar):
    '''
    Returns a list of the all non-terminals that are nullable
    in the given grammar. A nullable non-terminal is calculated
    as a closure using the following rules:

      nullable(A -> Epsilon) -> True
      nullable(A -> a) -> False
      nullable(A -> AB) -> nullable(A) AND nullable(B)
      nullable(A -> A1 | A2 | ... | AN) -> nullable(A1) OR .. OR nullable(A_n)

    :param Grammar grammar: the set of productions to use and
                            wrapped in the Grammar object
    :return a set of all non-terminals that can be nullable
    '''

    nullable = set()
    productions = grammar.productions

    cardinality = -1
    while cardinality < len(nullable):
        cardinality = len(nullable)

        for non_term in productions:
            #if epsilon is in the rhs already,
            #the production is nullable
            if EPSILON in productions[non_term]:
                nullable.add(non_term)
            else:
                isNullable = False
                for N in productions[non_term]:
                    #check to see if this specific production is
                    # nullable by checking to see if the join set of
                    # all symbols in the production are nullable.
                    # If they are not, that production is not nullable
                    isProductionNullable = True
                    for symbol in N:
                        isProductionNullable &= symbol in nullable
                    # This is the disjoint set of all nullable
                    # productions the correspond to this lhs
                    isNullable |= isProductionNullable
```

```python
            # if any of the disjoint productions are nullable,
            # then this is true. Therefore, add this production's
            # lhs to the nullable set
            if isNullable:
                nullable.add(non_term)

    return nullable

def first(grammar):
    '''A first set calucation for a grammar returns a dictionary of all
    the first terminals that can proceed the rest of a parse given a
    non-termial symbol. First is a closure that is calculated as
    follows:

      first(Epsilon) -> EmptySet
      first(A -> a) -> { a }

      first(A -> A B) -> { first(A) U first( B),   if nullable(A)
                         { first(A),               otherwise
      first(A -> A1 | A2 | ... | AN) -> first(A1) U first(A2 U ... U first(AN)

    :param Grammar grammar: the set of productions to use wrapped in a
                            Grammar object
    :return dict{Non-Terminal : set(Terminal)}: a table of all terminals that
                                                could come from a given
                                                non-terminal
    '''
    productions = grammar.productions
    #define the nullables
    nullable_non_terms = nullable(grammar)

    #initially set our table to the empty set of terminals
    prev_table = {non_term : set() for non_term in grammar.nonTerminals}
    #add just the productions of the form:
    #   non_terminal -> terminal
    #to start the algorithm off
    for non_term in grammar.nonTerminals:
        for rhs in productions[non_term]:
            if [] == rhs: continue
            if rhs[0] not in grammar.nonTerminals:
                prev_table[non_term].add(rhs[0])

    has_changed = True

    while has_changed:
        has_changed = False
```

```python
#construct the new table so we don't interfere with the
#previous one by adding things we wouldn't add this iteration
new_table = prev_table.copy()

for non_term in grammar.nonTerminals:
    for rhs in productions[non_term]:
        # if we have an epsilon, add it to the first set of this
        if len(rhs) == 0:
            if EPSILON not in prev_table[non_term]:
                has_changed = True
                new_table[non_term] |= set([EPSILON])
            continue

        # we already handled this above
        if rhs[0] in grammar.terminals:
            continue

        first = rhs[0]
        # Now, we need to check to see if adding anything from
        # the first item in the rhs changes the current set of
        # terminals we have for this non_term

        # this is done by seeing if the rhs's first
        # non-terminal's first set can add anything to the
        # current set of terminals for non_term
        if len(prev_table[first] - prev_table[non_term]) > 0:
            new_additions = prev_table[first] - prev_table[non_term]
            #skip past all the nullables until we hit the end
            #or we find a non-terminal. Add all the first sets
            #for the next item in the production as we come
            #across them
            i = 1
            while i+1 < len(rhs) and rhs[i] in nullable_non_terms:
                if rhs[i + 1] in grammar.nonTerminals:
                    new_additions |= prev_table[rhs[i + 1]]
                else:
                    #must list-ify to match the rest in the
                    #set
                    new_additions |= set([rhs[i+1]])
                    break
                i += 1

            new_table[non_term] |= new_additions
            has_changed = True

prev_table = new_table.copy()
```

```python
    return prev_table

def create_first_from_list(first_table, nullables, symbols):
    if len(symbols) == 0: return set()
    #if it starts with a terminal, return the singleton set
    if symbols[0] not in first_table.keys():
        return set([symbols[0]])

    first_set = first_table[symbols[0]]

    i = 1
    while i < len(symbols) and symbols[i] in nullables:
        first_set |= first_table[symbols[i]]
        i += 1

    return first_set

def betas_following(non_terminal, productions):
    ret_set = {}

    for lhs,all_rhs in productions.iteritems():
        for rhs in all_rhs:
            if non_terminal in rhs:
                symbol_list = rhs
                while non_terminal in symbol_list:
                    idx = symbol_list.index(non_terminal)
                    beta = []

                    if idx + 1 < len(symbol_list):
                        beta = symbol_list[idx + 1:]

                        if lhs in ret_set:
                            ret_set[lhs].append(beta)
                        else:
                            ret_set[lhs] = [beta]

                    symbol_list = beta
    return ret_set

def follows(grammar):
    '''Calculates all terminals that can follow a given non terminal.
    Follows is a closure calculated by the following rules:

      given [M -> ANB] -> follows(N) = follows(N) U first(B)
                          if nullable(B) then
                            follows(M) = follows(M) U follows(N)
```

```
  given [M -> A N B1...A N B2...A N BX]
             -> follows(N) = first(B1) U first(B2) U ...
                               U first(BX)
                            if nullable(B_i) then
                              follows(M) = follows(M) U follows(N)

:param Grammar grammar: the set of productions to use as a Grammar
                        object
:return dict{non-Terminal : set(terminals)}: the set of terminal characters
                                             that can follow any given
                                             non-terminal
'''
#add the EOF symbol for the start state
# S' -> S EOF
previous_start = grammar.start
rhs = [grammar.start, EOF]
lhs = grammar.start + "'"
grammar.addProduction(lhs, rhs)
grammar.start = lhs

nullable_non_terms = nullable(grammar)
first_table = first(grammar)
productions = grammar.productions

#initalize the table to contain only the empty sets
follow_table = {non_term : set() for non_term in grammar.nonTerminals}

has_changed = True
#iterate until all sets have not changed
while has_changed:
    has_changed = False

    #construct the new table for us to put additions into
    new_table = follow_table.copy()

    #construct all the follow sets for every non-terminal
    for non_term in grammar.nonTerminals:
        #get the dictionary of all {lhs : 'beta' values} (lists of
        #expressions following the non-terminal)
        betas_following_term = betas_following(non_term, productions)

        #Get the lhs of the production, call it M (like in the book)
        for M in betas_following_term.keys():
            #For every beta, calculate the following...
            for beta in betas_following_term[M]:
                m = create_first_from_list(first_table, nullable_non_terms, beta)
```

```
                if not m <= follow_table[non_term]:
                    has_changed = True
                    new_table[non_term] |= m

                is_nullable = True

                for term in beta:
                    if term not in nullable_non_terms:
                        is_nullable = False

                if is_nullable:
                # case where all of the symbol list is nullable, in which
                # we need to say follows(M) = follows(M) U follows(non_term)
                    if not follow_table[M] <= follow_table[non_term]:
                        has_changed = True
                        new_table[non_term] |= follow_table[M]

    #update our table to point to the new one
    follow_table = new_table.copy()

#restore grammar to previous state
follow_table.__delitem__(grammar.start)
grammar.nonTerminals.remove(grammar.start)
grammar.productions.__delitem__(grammar.start)
grammar.start = previous_start

return follow_table
```

10.3 parsetable.py

```
#!/usr/bin/env python
from ll1_tools import *
from cfg import EOF, EPSILON


class ParseTable:
    """ Represents a parse table. """

    def __init__(self, grammar):
        self.table = dict()
        self.isLl1 = True
        self.grammar = grammar

        first_dict = first(grammar)
        follows_dict = follows(grammar)
        nullables_set = nullable(grammar)

        # Initialize all table cells to the empty list.
        for nonTerminal in grammar.nonTerminals:
            self.table[nonTerminal] = dict()
            # also initialize the epsilon and eof characters
            # to have empty cells as well
            self.table[nonTerminal][EPSILON] = []
            self.table[nonTerminal][EOF] = []
            for terminal in grammar.terminals:
                self.table[nonTerminal][terminal] = []

        # Build the table.
        for lhs in grammar.productions:
            # All right hand sides of a production in the
            # form: A -> alpha1 | alpha2 | alpha3 as a list
            alphas = grammar.productions[lhs]

            # For every right hand side above
            for alpha in alphas:

                #calculate First(alpha) using first dictionary and nullables
                #this is First(alpha[0]) U First(alpha[i]) where i = 1 : len(alpha)
                #and stop i when alpha[i] is not in nullables
                first_of_alpha = create_first_from_list(first_dict, nullables_set, alpha)

                # for all 't' that exist in First(alpha), where 't' is always
                # a terminal
                for t in first_of_alpha:
                    #add this to the table for 't'
```

```
                    self.table[lhs][t].append(alpha)

                    # Check for multiple entries in cell - LL1 check.
                    if len(self.table[lhs][t]) > 1:
                        self.isLl1 = False

                # Now, if EPSILON exists in First(alpha), then we have a couple
                # other additions to make
                if EPSILON in first_of_alpha:
                    # For every terminal, t, in Follows(A), add an entry from [A][t] = alpha
                    for t in follows_dict[lhs]:
                        self.table[lhs][t].append(alpha)

                        # Check for multiple entries in cell - LL1 check.
                        if len(self.table[lhs][t]) > 1:
                            self.isLl1 = False

                    #Also, if EOF exists in Follows(A), then add entry: [A][EOF] = alpha
                    if EOF in follows_dict[lhs]:
                        self.table[lhs][EOF].append(alpha)
                        # Check for multiple entries in cell - LL1 check.
                        if len(self.table[lhs][EOF]) > 1:
                            self.isLl1 = False

    def __str__(self):
        """ Output to CSV format for viewing with a spreadsheet program. """
        ret = ""

        # Print row headers
        for header in self.grammar.terminals:
            ret += ',' + header
        ret += '\n'

        # Print rows
        for row in self.table:
            ret += row
            for col in self.table[row]:
                ret += ',' + str(self.table[row][col]).replace(',', ' ')
            ret += "\n"

        return ret
```

10.4 parser.py

```
__author__ = 'Taylor'

from parsetable import ParseTable
from cfg import Grammar
from ll1_tools import first, follows

class Parser:
    '''
    Data structure wrapping a grammar and parse table together
    '''

    def __init__(self, grammar=None):
        #storing this just in case
        self.grammar = grammar
        #start the stack with just the start node
        self.parse_stack = [grammar.start]
        #construct the parse table
        self.table = ParseTable(grammar).table

    def ll1_parse(self, rose_tree, token_list):
        '''
        :param rose_tree: a Rose_Tree of the current node we are parsing
        :param token_list: a list of terminal tokens to parse into a tree structure
        :return: RoseTree, [tokens]: the RoseTree is the AST of the parse and the
                                     list of tokens are the unconsumed tokens
        '''

        # if the stack is empty, return the current parse and leftover input
        if not self.parse_stack:
            return rose_tree, token_list

        # if there's no tokens left and we are here, parse error
        if not token_list:
            raise ValueError("Unexpected EOF, current parse stack is:" + str(self.parse_stack.reverse()

        # Get the first symbol
        current_symbol = self.parse_stack.pop()
        # one token look-ahead
        token = token_list[0]

        #if we have a matching symbol and token, we can consume the input
        if current_symbol in self.grammar.terminals:
            leaf = Rose_Tree(current_symbol, token)
            return leaf, token_list[1:]
```

```
        #else we have a non-terminal, we must continue with the rewrite
        #returns a list of possible productions that should follow
        production_to_follow = self.table[current_symbol][token]

        # if empty production to follow, unexpected terminal found:
        if not production_to_follow:
            raise ValueError("Unexpected terminal, " + str(token) + ", found.")

        # we can't handle this in LL1 style parsing
        if len(production_to_follow) > 1:

            print production_to_follow
            raise ValueError("Too many possible parses for LL1, this is non-deterministic. "
                             "Please check your grammar. Current parse: " +
                             str(self.parse_stack[0:-1]) + " on terminal " + str(token))

        # push all symbols onto the stack
        for symbol in production_to_follow[0][::-1]:
            self.parse_stack.append(symbol)

        #construct a node in the tree and attach all children parsed
        current_rose_tree_node = Rose_Tree(current_symbol, "")

        leftover = token_list

        for symbol in production_to_follow[0]:
            print self.parse_stack[::-1]
            #this can never just return from an empty stack since we place
            #all symbols found on the stack before this. See lines 54 & 55
            parsed_tree_node, leftover = self.ll1_parse(current_rose_tree_node, leftover)

            # append as a child
            current_rose_tree_node.children.append(parsed_tree_node)

            # point back to parent
            parsed_tree_node.parent = current_rose_tree_node

        #finished parsing for this node completely, return the node and any leftover input
        return current_rose_tree_node, leftover

class Rose_Tree:
    def __init__(self, symbol, node_value):
        '''
        :param node_name:
        :param node_value:
        :param children:
```

```python
        :return:
        '''
        self.symbol = symbol
        self.value = node_value
        self.children = []
        self.parent = None

    def __str__(self):
        ret = "Symbol: " + str(self.symbol) + ", val=" + str(self.value) + '\n'

        for child in self.children:
            ret += str(child)

        return ret
```