

Implementation of a Cubic-Time Parser for Tree-Insertion Grammar in Python

By Tomer Filiba

Introduction

The purpose of this work is to implement a parser for Tree Insertion Grammar (TIG), based on an algorithm and analysis found in [SW94]. *Tree Insertion Grammar* is a formalism that branched out of *Tree Adjoining Grammar* (TAG) [Joshi75]; the fundamental idea in both is that grammar rules are expressed as a set of *elementary trees*, instead of the "linear" production rules of Context Free Grammar (CFG). These elementary trees are then embedded into each other to form larger trees, until they cover the entire input.

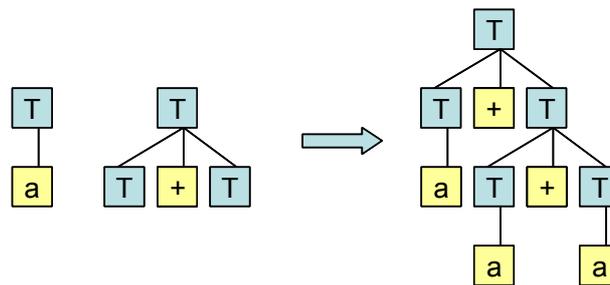


Figure 1: Substitution of two elementary initial trees

TIG and TAG combine trees using two operations: *substitution* and *adjunction*. Substitution takes a non-terminal leaf L of one tree and replaces it with a tree whose root is also L . This mechanism is demonstrated in the figure above, and is very similar to the way production rules are applied in CFG. For instance, the two trees above are equivalent to the following minimal CFG: $T \rightarrow a \mid T + T$.

Adjunction, on the other hand, is more powerful. Conceptually, it "rips open" an existing node in a derivation tree and replaces it with a sub-tree. The contents of the ripped node are then "pushed down" onto a special leaf node of the sub-tree, called the *foot node* (marked by an asterisk in the following diagrams). The process is demonstrated in the following diagram.

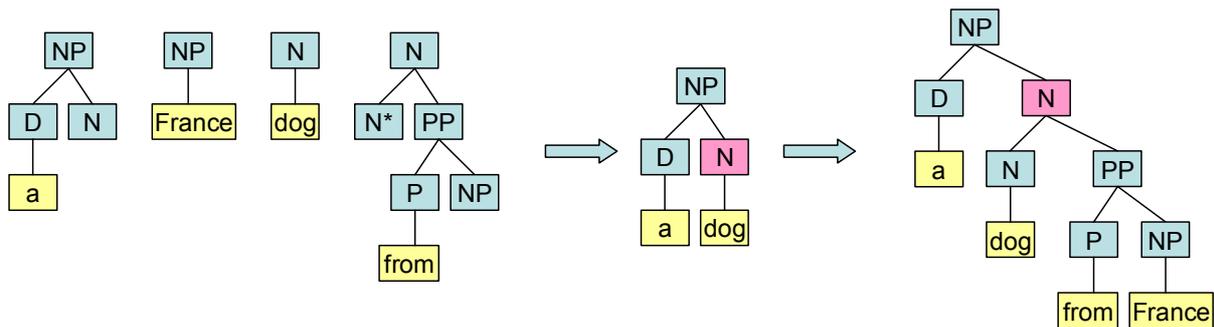


Figure 2: Adjunction of elementary trees

For instance, when parsing an NP such as "a dog from France", we first build $[_{NP} \text{ a dog}]$, and then perform adjunction on it with $[_{PP} \text{ from France}]$ to form the larger NP shown in the diagram above. The node that was "ripped open" is highlighted in pink.

Elementary trees divide into two types: *initial* and *auxiliary*, the difference being auxiliary trees contain *foot nodes* while initial trees do not. Both formalisms require exactly one foot node in each auxiliary tree; however, while TAG allows the foot node to be any leaf node, TIG requires that it would be either the **rightmost** or **leftmost** leaf of the tree (ignoring empty ϵ productions that may occur to the right/left of the foot). This virtually minor difference between the two formalisms has a strong impact on their *expressive power*: while TAGs generate *Mildly-Context-Sensitive* languages (as defined in [VW94]), TIGs only generate *Context-Free* languages (CFLs being a subset of MCSL, of course). In essence, adjunction in TAG allows *two-sided wrapping* of one tree by another, while adjunction in TIG allows only *one-sided embedding*. The equivalence of TIGs and CFGs is given as a theorem in [SW94].

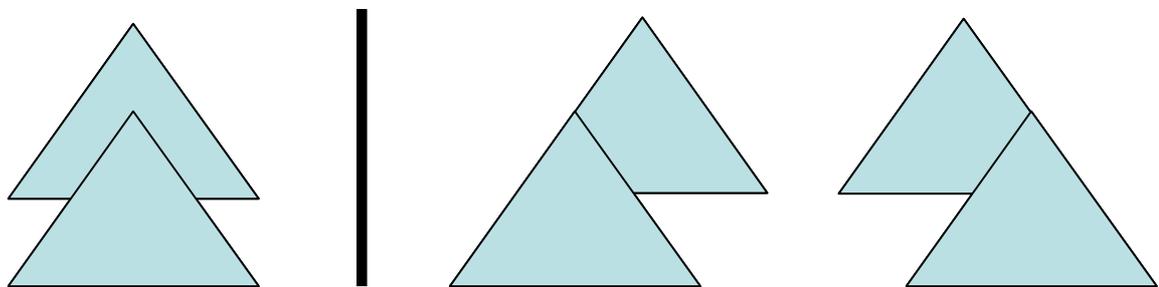


Figure 3: Two-sided wrapping (left) as opposed to one-sided embedding (right)

If CFG and TIG possess the same expressive power, why is the latter of any special interest? The answer has two parts: first, TIG is a more "natural" way to construct grammars for natural languages; it is easier to express long-spanning dependencies this way, where in CFG one would have to add numerous production rules to "propagate" the dependency. Also, according to [SW94], it tends to produce considerably smaller grammars than the equivalent CFGs (elaborate comparison is given at the end of their paper).

Second, it enables *strong lexicalization* of CFG while preserving the parse trees. This means that a CFG can be transformed into an equivalent TIG, producing the same trees, but in a way that each elementary tree is *anchored by a lexical item* (i.e., each tree has a terminal leaf node). It is generally impossible to lexicalize the production rules of CFG while preserving the parse-tree structure, as demonstrated by grammars like $S \rightarrow S S \mid a$; using TIG, this task proves feasible.

Lexicalization is a desired feature of grammars, since it means each production/tree must "consume" at least one input token, so that the number of parsing trees is bounded; this also entails that every lexicalized grammar is finitely-ambiguous. These two properties are required for the meaningful extraction of derivation trees from a grammar.

Schabes and Waters go to great lengths to prove that TIG is indeed equivalent to CFG in power and construct a method for converting finitely-ambiguous CFGs into equivalent lexicalized TIGs; however, lexicalization is beyond the scope of this work, and we constrain ourselves to parsing. Since TIG is a special case of TAG, a general TAG parser will handle them properly, but this will require a parsing time of $O(n^6)$ [JS97], while alternative CFG parsers (CYK, Earley) require only $O(n^3)$ time.

The Algorithm

In their paper, Schabes and Waters describe an $O(n^3)$ Earley-style recognizer for TIG, given as a set of *inference rules* (instead of the more conventional pseudocode). Like [Earley70], this algorithm uses a set called *the chart* (borrowing from Earley, even though it does not consist of columns), into which intermediate parsing states are inserted; the chart begins empty and fills as the algorithm progresses. It is important to note that the chart cannot contain duplicates, and that it is monotonically non-decreasing in size during the entire run of the algorithm.

The states of the chart are 4-tuples in the form $\langle \text{tree}, \text{dot}, i, j \rangle$, where *tree* is an elementary tree (or a part of one), *dot* is the index of the node in the tree we've reached (only immediate children of the root are considered), and *i* and *j* are indexes representing the span of input tokens that this state covers, ranging from 0 to *n* (*n* being the length of the input).

Initially, all states of the form $\langle \mu_S \rightarrow \bullet \alpha, \theta, \theta \rangle$ are inserted into the chart, where μ_S is an initial tree whose root is the start symbol. Next, rules (2)-(12) are invoked so long as the chart keeps growing, and when it reaches a stable size, the occurrence of a state of the form $\langle \mu_S \rightarrow \alpha \bullet, \theta, n \rangle$ in the chart (*n* being the length of the input and μ_S being an initial tree), marks acceptance of the recognizer.

The inference rules of the algorithm are given below:

$$\begin{array}{c}
 \text{Initialization} \\
 \hline
 \text{Init}(\mu_S) \vdash [\mu_S \rightarrow \bullet \alpha, 0, 0] \quad (1) \\
 \\
 \text{Left Adjunction} \\
 \hline
 [\mu_A \rightarrow \bullet \alpha, i, j] \wedge \text{Left Aux}(\rho_A) \vdash [\rho_A \rightarrow \bullet \gamma, j, j] \quad (2) \\
 [\mu_A \rightarrow \bullet \alpha, i, j] \wedge [\rho_A \rightarrow \gamma \bullet, j, k] \wedge \text{Left Aux}(\rho_A) \vdash [\mu_A \rightarrow \bullet \alpha, i, k] \quad (3) \\
 \\
 \text{Scanning} \\
 \hline
 [\mu_A \rightarrow \alpha \bullet \nu_a \beta, i, j] \wedge a = a_{j+1} \vdash [\mu_A \rightarrow \alpha \nu_a \bullet \beta, i, j+1] \quad (4) \\
 [\mu_A \rightarrow \alpha \bullet \nu_a \beta, i, j] \wedge a = \varepsilon \vdash [\mu_A \rightarrow \alpha \nu_a \bullet \beta, i, j] \quad (5) \\
 [\mu_A \rightarrow \alpha \bullet \nu_B \beta, i, j] \wedge \text{Foot}(\nu_B) \vdash [\mu_A \rightarrow \alpha \nu_B \bullet \beta, i, j] \quad (6) \\
 \\
 \text{Substitution} \\
 \hline
 [\mu_A \rightarrow \alpha \bullet \nu_B \beta, i, j] \wedge \text{Subst}(\nu_B) \wedge \text{Init}(\rho_B) \vdash [\rho_B \rightarrow \bullet \gamma, j, j] \quad (7) \\
 [\mu_A \rightarrow \alpha \bullet \nu_B \beta, i, j] \wedge [\rho_B \rightarrow \gamma \bullet, j, k] \wedge \text{Subst}(\nu_B) \wedge \text{Init}(\rho_B) \vdash [\mu_A \rightarrow \alpha \nu_B \bullet \beta, i, k] \quad (8) \\
 \\
 \text{Subtree Traversal} \\
 \hline
 [\mu_A \rightarrow \alpha \bullet \nu_B \beta, i, j] \vdash [\nu_B \rightarrow \bullet \gamma, j, j] \quad (9) \\
 [\mu_A \rightarrow \alpha \bullet \nu_B \beta, i, j] \wedge [\nu_B \rightarrow \gamma \bullet, j, k] \vdash [\mu_A \rightarrow \alpha \nu_B \bullet \beta, i, k] \quad (10) \\
 \\
 \text{Right Adjunction} \\
 \hline
 [\mu_A \rightarrow \alpha \bullet, i, j] \wedge \text{Right Aux}(\rho_A) \vdash [\rho_A \rightarrow \bullet \gamma, j, j] \quad (11) \\
 [\mu_A \rightarrow \alpha \bullet, i, j] \wedge [\rho_A \rightarrow \gamma \bullet, j, k] \wedge \text{Right Aux}(\rho_A) \vdash [\mu_A \rightarrow \alpha \bullet, i, k] \quad (12) \\
 \\
 \text{Final Recognition} \\
 \hline
 [\mu_S \rightarrow \alpha \bullet, 0, n] \wedge \text{Init}(\mu_S) \vdash \text{Acceptance} \quad (13)
 \end{array}$$

Figure 4: Inference rules of the Earley-style TIG recognizer, taken from [SW94]

The inference rules (2)-(12) can be grouped into 4 groups:

- **Scanning** – the rules (4)-(6) attempt to match the current input token with the terminal that the current state expects. If the token is matched, a new state is added, whose dot is advanced. Note that unlike Earley, ϵ -productions are handled inherently.
- **Substitution** – the rules (7)-(8) look for states waiting for a sub-tree, while a state with the required sub-tree exists in the chart. If so, they add a new state where the expected sub-tree has been matched.
- **Subtree-Traversal** – the rules (9)-(10) look for a state whose next production is a sub-tree, and add a new state to the chart that's expecting this sub-tree.
- **Adjunction** – the rules (2)-(3) and (11)-(12) handle left- and right-adjunction respectively. They basically look for an auxiliary tree and state that is at a point where adjunction may apply, and "predict" new states where the adjunction might have taken place.

Analysis

The correctness of the algorithm outlined in the paper is somewhat intuitive – the authors define a "correctness condition" that must hold, and because the algorithm is expressed as inference rules, they make little effort to show that each state added to the chart satisfies this condition.

The computational bounds (worst case) of the algorithm are $O(|G| \cdot n^2)$ for space and $O(|G|^2 \cdot n^3)$ for time, where $|G|$ is the size of the grammar and n is the length of the input. The exact definition of "grammar size" is quite complicated, but it's constant nonetheless. The space is bound by the number of possible different chart states: there are n^2 options for i and j , and $|G|$ combinations for dotted trees (the dot ranges from 0 to the width of the widest sub-tree, a constant of the grammar).

The analysis of the time complexity is as follows: the completion rules ((3), (8), (10) and (12)) are the most complex ones, as they apply to **pairs** of chart states: there are $O(|G|)$ possibilities for each of the **two** trees, thus $O(|G|^2)$, and $O(n^3)$ combinations for i , j and k . Note that we only consider adjacent states (i.e., $\text{state1.j} = \text{state2.i}$), so there are only 3 indexes involved, instead of 4.

The Implementation

From Inference Rules to Procedural Code

The authors claim they've developed a deductive parser which operates directly on such inference rules; however, I must admit that I found their notation very confusing and hard to understand at first. It also tends to hide the complexity involved, as each new variable in an inference rule means an implicit nested for-loop over the entire chart.

Here follows an interpretation of the inference rules as procedural pseudocode; this is only a sketch, of course, leaving out most of the details:

```
procedure recognize(G, tokens)
  chart = {< $\mu_s \rightarrow \bullet \alpha, \theta, \theta$ > |  $\mu_s \in \text{InitTrees}(G)$  and  $S = \text{StartSymbol}(G)$ }

  while chart keeps growing:
    for each state st in chart:
      apply_inference_rules_2_to_12(G, chart, st, tokens)

  for each state st in chart:
    if st = < $\mu_s \rightarrow \alpha \bullet, \theta, \text{len}(\text{tokens})$ > and  $\mu_s \in \text{InitTrees}(G)$ :
      return True
  return False
```

From Recognizer to Parser

Another gem of the paper was the statement "*... the algorithm is a recognizer. However, it can be straight-forwardly converted to parser by keeping track of the reasons why states are added to the chart*".

This may have been straight-forward to the authors, but it wasn't the case for me. Adding back-pointers to the chart states increased the number of states exponentially (because two states that were previously considered identical may now derive from different parents, and are thus distinct). Associating with each state a list of parents also proved futile: since the algorithm operates on the entire chart with inference rules (instead of progressing in "columns" as in Earley), there is no notion of "time" in the process; state X may cause the inference of state Y, and this state Y may later cause the inference of state X. Thus a list of parents is bound to have cycles, and attempting to back-track in this list is not possible.

On top of this "straight-forwardness", there's also an inherent problem with the way the algorithm works: the chart was designed for recognition, but it loses some information required to reconstruct adjunction.

Consider the parsing of the following simple NP, "the tasty banana", according to these elementary trees:

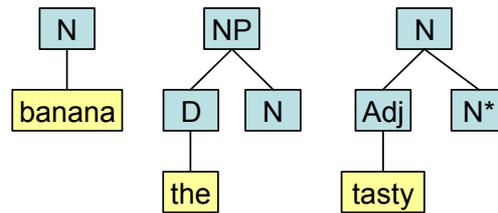


Figure 5: A simple grammar for "the tasty banana"

After algorithm is run, the chart will contain (listing relevant states only):

Tree	Span (i:j)	Inferred by rule
NP→•D N	0:0	Init (1)
D→•the	0:0	Subtree (9)
D→the•	0:1	Scan (4)
NP→D •N	0:1	Subtree (10)
N→•banana	1:1	Subst (7)
N→•Adj N*	1:1	LeftAdj (2)
Adj→•tasty	1:1	Subtree (9)
Adj→tasty•	1:2	Scan (4)
N→Adj •N*	1:2	Subtree (10)
N→Adj N*•	1:2	Scan (6)
N→•banana	1:2	LeftAdj (3)
N→banana•	1:3	Scan (4)
NP→D N•	0:3	Subst (8)

Figure 6: The chart after the parsing of "the tasty banana"

The matching state is of course the last one, which represents the full tree. However, tracing back we see that D matched "the" (spanning 0:1), while N matched "banana" (spanning 1:3) – meaning a **single** lexical item spans two input tokens. This is the result of the left-adjunction rule (3), which neglects the reason why $\langle N \rightarrow \bullet \text{banana}, 1, 2 \rangle$ is added to the chart: we would have wanted it to add $\langle N \rightarrow \text{Adj}(\text{"tasty"}) \bullet N(\text{"banana"}), 1, 2 \rangle$, but this is neither an

elementary tree nor a part of one. Adding such new states to the chart would break the time and space bounds entirely.

In lack of any better solution, I resorted to using this method: whenever substitution or adjunction was performed, a modified copy of the tree (where the operation was carried out) was created and added to the chart as a new state.

According to the analysis, the number of states in the chart is $O(|G| \cdot n^2)$, and it remains so after this change as well. However, once we add all intermediate derivation trees as states in the chart, $|G|$ is no longer of constant size. In fact, it becomes **unbound** in the general case, and the length of the input becomes negligible; a parsing time of $O(|G|^2 \cdot n^3)$ becomes impractical.

Tree Builder Combinators

Ultimately, after three weeks without progress, I've found a solution I call *Tree-Builder Combinators*. As keeping back-pointers proved futile and generating all intermediate parse trees proved impractical, I realized I should associate with each chart state a set of "tree builders"; these are basically functions that generate the parse trees on demand. The use of combinators (functions with no free variables) instead of general functions is important, as it allows us to eliminate duplicates: we can define equivalence for tuples such as $\langle \text{combinator}, \text{arg1}, \text{arg2} \rangle$, which we cannot do for "black box" functions.

The tree-builder combinators are added to the chart alongside with their arguments. It is important to note that the **number of states** in the chart remains unaltered; however, each state now holds a set of (unique) tree builders. Also, during parsing, tree builders are only added to the chart (but never invoked), so this change will not affect the time complexity.

In the implementation, I used four such combinators; the most complex ones bind two arguments, which are chart states, so in total, there could be at most $O(4 \cdot (|G| \cdot n^2) \cdot (|G| \cdot n^2)) = O(|G|^2 \cdot n^4)$ tree builders associated with each chart state; in practice, of course, this number is very small. The number of states in the chart is $O(|G| \cdot n^2)$ so seemingly, the memory-footprint of the chart is $O(|G|^3 \cdot n^6)$.

This would of course affect the time complexity as well, as we could theoretically add each such tree builder. However, this upper bound is very loose. In fact, since exactly one tree builder is added each time a state is added to the chart, the total number of tree builders is bound by the number of times we add

states. As shown in the analysis, the number of additions is bound by $O(|G|^2 \cdot n^3)$, and therefore the space requirements of the table are also bound by $O(|G|^2 \cdot n^3)$.

Extracting the parse trees from the chart is trivial now, as we just need to run the recognizer, find the matching states, and invoke their tree builders. These would, in turn, invoke the next level's tree builders, and so on; at the end of each such chain, a complete derivation tree is formed. We cannot discuss the computational bounds of extracting the parse trees, of course, as their number might be unbound.

I also employed *memoization* during the process of tree extraction, so that once a state's subtrees are computed, they are stored in the chart; future attempts to extract the subtrees of this state are an $O(1)$ operation. This works much like memoization in a Fibonacci number generator, where it lowers the computational complexity from $\Theta(\varphi^n)$ to $\Theta(n)$. Here, memoization ensures that each tree is extracted in amortized linear time.

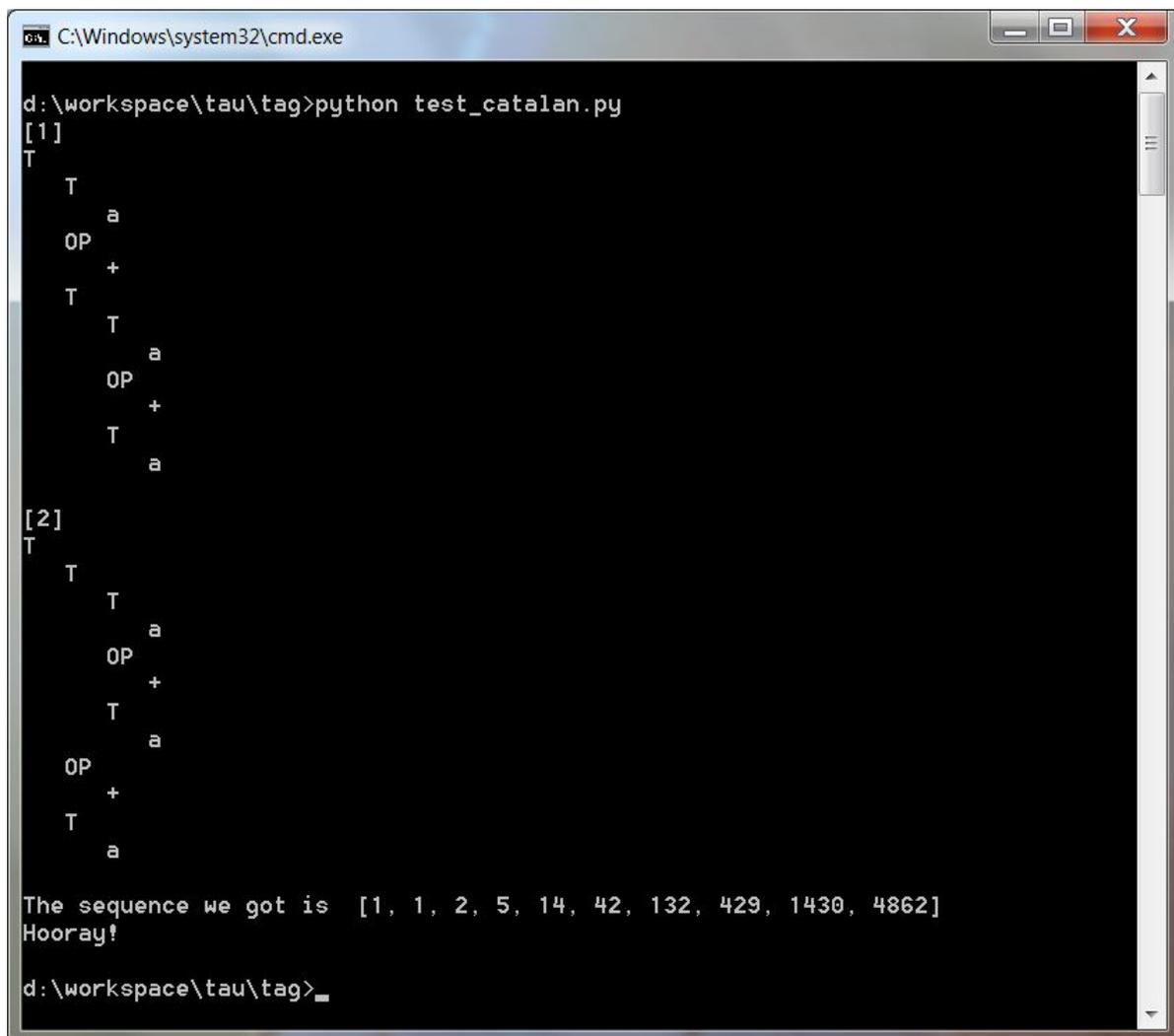
References

- [Earley70] – Jay Earley; 1970. *An Efficient Context-Free Parsing Algorithm*
- [JLT75] – Aravind K. Joshi, L. S. Levy, M. Takahashi; 1975. *Tree Adjunct Grammars*
- [SW94] – Yves Schabes, Richard C. Waters; 1994. *Tree Insertion Grammar: A Cubic-Time Parsable Formalism that Lexicalizes Context-Free Grammar without Changing the Trees Produced*
- [VW94] – K. Vijay-Shanker, David J. Weir; 1994. *The Equivalence of Four Extensions of Context-Free Grammars*
- [JS97] – Aravind K. Joshi, Yves Schabes; 1997. *Tree-Adjoining Grammars*

Appendix A: The Distribution

Appendix C includes the code of the parser and tree extraction. Note that 230 lines having to do with the formalism itself are omitted.

Attached are the full parser code and two tests. The first test defines an intentionally ambiguous grammar for $(a+a+\dots+a)$ and counts the number of generated parse trees. If all goes well, it should generate the Catalan sequence, as a form of sanity-check for the parser.

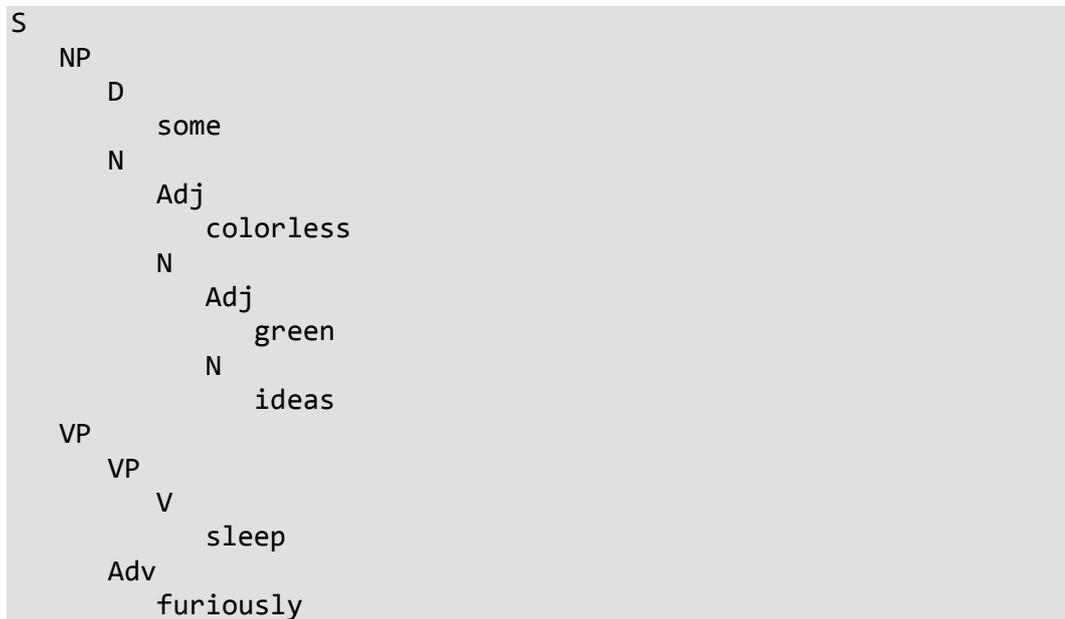


```
C:\Windows\system32\cmd.exe
d:\workspace\tau>tag>python test_catalan.py
[1]
T
  T
    a
  OP
  +
  T
    T
      a
    OP
    +
    T
    a

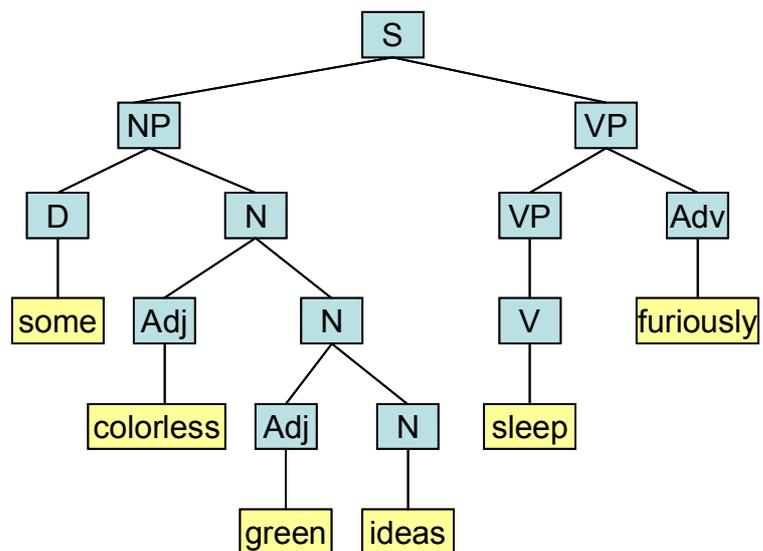
[2]
T
  T
    T
      a
    OP
    +
    T
    a
  OP
  +
  T
  a

The sequence we got is [1, 1, 2, 5, 14, 42, 132, 429, 1430, 4862]
Hooray!
d:\workspace\tau>tag>
```

The second test defines a mock grammar for English and prints out the parse trees of 12 complex-adjunction sentences, 2 of which are ambiguous. Brought here is a sample parse tree generated for "some colorless green ideas sleep furiously":



Or graphically:



Appendix B: Python from Java Perspective

Python is an object-oriented, garbage-collected, dynamically-typed (duck-typed) interpreted language that makes use of indentation for code-level nesting and borrows numerous features from functional languages. This is a short introduction to Python from a Java perspective.

Syntax

<pre> if cond: then-code [elif cond: then-code] [else: else-code] </pre>	<pre> for var in coll: loop-code </pre>
<pre> while cond: loop-code </pre>	<pre> def name (args...): func-body </pre>
<p>"pass" is a keyword that denotes an empty body (required because of the indentation-based nature of the syntax)</p>	
<pre> class name (bases...): [class-level attributes] def name (self, args...): method-body </pre> <p>Note that methods in Python take an explicit "self" parameter (AKA "this"), which represents the method's instance.</p>	
<pre> [1,2,3] → list (mutable) (1,2,3) → tuple (immutable) {1,2,3} → set (no duplicates, quick lookup) {1:2, 3:4} → dictionary (AKA "map", quick lookup) {} → empty dictionary (not set!) set([]) → empty set </pre>	
<pre> [x for x in coll [if <cond>]] </pre> <p>List comprehension, like set-builder notation, but the result is ordered and may contain duplicates. For instance,</p> <pre> >>> [x % 3 for x in range(10) if x > 4] [2, 0, 1, 2, 0] </pre>	
<pre> (x for x in coll [if <cond>]) </pre> <p>Generator expressions, like list-comprehensions, but done lazily, i.e., builds a generator (iterator) object that produces values only when consumed.</p> <pre> >>> (x * 2 for x in range(10) if x % 3 == 0) <generator object <genexpr> at 0x0157A878> >>> sum(x * 2 for x in range(10) if x % 3 == 0) 36 </pre>	

Semantics

Python relies on *special methods* (in the form of `__xxx__`) to implement language-level features like operators (`__add__`) and other interfaces (`__hash__`, `__eq__`, `__str__`, ...).

Python classes behave like functions; "calling" a class creates an instance of it, and any arguments given are passed to the special method `__init__`, which is used to initialize the instance. Objects are much like dictionaries; all objects start "empty" and attributes are added to them at runtime.

```
>>> class C(object):
...     def __init__(self, a, b):
...         self.my_name = a + b
...
>>> x = C("foo", "bar")
>>> x
<__main__.C object at 0x01589390>
>>> x.my_name
'foobar'
>>> x.your_name = "spam"
```

Doc-strings may be placed as the first code-element in functions and classes, where they serve for documentation and are retrievable by the interpreter.

```
>>> def foo(a,b):
...     "I am a docstring. This function just returns 5"
...     return 5
...
>>> help(foo)
foo(a, b)
    I am a docstring. This function just returns 5
```

Functions can list their arguments explicitly, but they might also accept "varargs", which means they can take any number of arguments. For instance

```
>>> def foo(a, b, *args):
...     print a, b, args
...
>>> foo(1,2,3,4,5,6)
1 2 (3, 4, 5, 6)
```

The occurrence of `yield` inside a function turns it into a generator which could be consumed by a for-loop

```
>>> def f():
...     yield 1
...     yield 2
...     yield 3
...
>>> list(f())
[1, 2, 3]
```

Appendix C: Parser Code

```
#####
# Chart and Chart States
#####
class State(object):
    """
    The chart state. This is in essence a 4-tuple <tree, dot, i, j>, with some
    helper methods
    """
    def __init__(self, tree, dot, i, j):
        self.tree = tree
        self.dot = dot
        self.i = i
        self.j = j
        self.index = None
        self._hash = None
    def __str__(self):
        prod = ["%s%s" % (c, SYM_DOWN_ARROW)
                if isinstance(c, NonTerminal) else str(c)
                for c in self.tree.children]
        prod.insert(self.dot, SYM_DOT)
        return "%s %s %s, %r:%r" % (self.tree.root, SYM_RIGHT_ARROW,
                                   " ".join(prod), self.i, self.j)
    def __eq__(self, other):
        return (self.tree, self.dot, self.i, self.j) == (
            other.tree, other.dot, other.i, other.j)
    def __ne__(self, other):
        return not (self == other)
    def __hash__(self):
        if self._hash is None:
            self._hash = hash((self.tree, self.dot, self.i, self.j))
        return self._hash
    def is_complete(self):
        """
        Returns True iff the dot is past the last child (thus the state
        is complete)
        """
        return self.dot >= len(self.tree.children)
    def next(self):
        """
        Return the next (first-level only) production of this tree,
        or None if we've reached the end
        """
        if self.is_complete():
            return None
        return self.tree.children[self.dot]
```

```

class ChartItem(object):
    """
    A helper object, associated with each chart state, that holds the reasons
    for adding this state and the state's subtree builders
    """
    UNPROCESSED = 1
    PROCESSING = 2
    PROCESSED = 3

    def __init__(self, reason, subtreefunc):
        self.reasons = {reason}
        self.subtreefuncs = {subtreefunc}
        self.subtrees = set()
        # `stage` serves as a marker for get_subtrees()
        self.stage = self.UNPROCESSED

    def add(self, reason, subtreefunc):
        """
        Adds a reason and a subtree-builder to this chart item
        """
        self.reasons.add(reason)
        self.subtreefuncs.add(subtreefunc)

class Chart(object):
    """
    Represents the parser chart. It comprises of states (without duplicates),
    but it preserves the ordering relations for debugging purposes. With each
    state we associates a ChartItem, to hold some extra info.
    States are add()ed to the chart, but they don't actually become part of
    it until commit()ted. This prevents some issues with dictionary iteration.
    """

    def __init__(self):
        self._states = {}
        self._ordered_states = []
        self._changes = []
    def __iter__(self):
        return iter(self._ordered_states)
    def __len__(self):
        return len(self._ordered_states)
    def __getitem__(self, index):
        return self._ordered_states[index]

    def add(self, state, reason, subtreefunc = None, *args):
        """
        Adds a new state to the chart, including the state's reason and
        subtree-builder. Note that it's not actually added to the chart
        until commit() is called
        """
        if subtreefunc is None:
            subtreefunc = BUILD_CONST
            args = (state.tree,)
        self._changes.append((state, reason, (subtreefunc, args)))

```

```

def commit(self):
    """
    commits the changes to the chart -- returns True if the chart
    has grew, False otherwise
    """
    added = False
    while self._changes:
        st, reason, subtreefunc = self._changes.pop(0)
        if st not in self._states:
            st.index = len(self._ordered_states)
            self._ordered_states.append(st)
            self._states[st] = ChartItem(reason, subtreefunc)
            added = True
        else:
            self._states[st].add(reason, subtreefunc)

    return added

def get_subtrees(self, st):
    """
    Gets the set of subtrees for a given state; this is memoized (cached)
    so once the subtrees of some state have been built,
    future calls are O(1)
    """
    item = self._states[st]
    if item.stage == ChartItem.PROCESSED:
        return item.subtrees
    # make sure we're not accidentally reentrant
    assert item.stage == ChartItem.UNPROCESSED
    item.stage = ChartItem.PROCESSING
    for func, args in item.subtreefuncs:
        item.subtrees.update(func(self, *args))
    item.stage = ChartItem.PROCESSED
    return item.subtrees

def show(self, only_completed = False):
    """
    Print the chart in a human-readable manner
    """
    for st in self._ordered_states:
        if only_completed and not st.is_complete():
            continue
        print "%3d | %-40s | %s" % (st.index, st,
            " ; ".join(self._states[st].reasons))
    print "-" * 80

```

```

=====
# Tree extraction combinators:
#
# Whenever we add a new state to the chart, we associate with it a
# subtree-builder, which serves us later (we get_subtrees() is called).
# These builders combine partial trees to form bigger ones, according to
# the rules of the grammar
=====
def BUILD_CONST(chart, t):
    return [t]

def BUILD_PROPAGATE(chart, st):
    return chart.get_subtrees(st)

def BUILD_SUBSTITUTION(chart, st, st2):
    return [t1.deep_substitute(st.dot, t2)
            for t1 in chart.get_subtrees(st) for t2 in chart.get_subtrees(st2)]

def BUILD_AUX(chart, st, st2):
    return [t2.substitute_foot(t1)
            for t1 in chart.get_subtrees(st) for t2 in chart.get_subtrees(st2)]

=====
# Parser
=====
def handle_left_adj(grammar, chart, st):
    """
    handles the case of left-adjunction rules (2) and (3)
    """
    if st.dot != 0:
        return

    # (2)
    for t in grammar.get_left_aux_trees_for(st.tree.root):
        chart.add(State(t, 0, st.j, st.j), "[2]/%d" % (st.index,))

    # (3)
    for st2 in chart:
        if (st2.tree.type == Tree.LEFT_AUX and st.tree.root == st2.tree.root
            and st.j == st2.i and st2.is_complete()):
            chart.add(State(st.tree, 0, st.i, st2.j),
                      "[3]/%d,%d" % (st.index, st2.index),
                      BUILD_AUX, st, st2)

```

```

def handle_scan(grammar, chart, st, token):
    """
    handles the case of scanning rules (4), (5) and (6)
    """
    prod = st.next()
    if isinstance(prod, str):
        if prod == token:
            # (4)
            chart.add(State(st.tree, st.dot+1, st.i, st.j+1),
                "[4]/%d" % (st.index,),
                BUILD_PROPAGATE, st)
        elif prod == "":
            # (5)
            chart.add(State(st.tree, st.dot+1, st.i, st.j),
                "[5]/%d" % (st.index,),
                BUILD_PROPAGATE, st)
        elif isinstance(prod, Foot):
            # (6)
            chart.add(State(st.tree, st.dot+1, st.i, st.j),
                "[6]/%d" % (st.index,),
                BUILD_PROPAGATE, st)

def handle_substitution(grammar, chart, st):
    """
    handles the case of substitution rules (7) and (8)
    """
    prod = st.next()
    if isinstance(prod, NonTerminal):
        # (7)
        for t in grammar.get_init_trees_for(prod):
            chart.add(State(t, 0, st.j, st.j), "[7]/%d" % (st.index,))

        # (8)
        for st2 in chart:
            if (st2.tree.root == prod and st.j == st2.i and st2.is_complete()
                and st2.tree.type == Tree.INIT_TREE):
                chart.add(State(st.tree, st.dot + 1, st.i, st2.j),
                    "[8]/%d,%d" % (st.index, st2.index),
                    BUILD_SUBSTITUTION, st, st2)

def handle_subtree_traversal(grammar, chart, st):
    """
    handles the case of subtree-traversal rules (9) and (10)
    """
    prod = st.next()
    if isinstance(prod, Tree):
        # (9)
        chart.add(State(prod, 0, st.j, st.j), "[9]/%d" % (st.index,))

        # (10)
        for st2 in chart:
            if st2.tree == prod and st.j == st2.i and st2.is_complete():
                chart.add(State(st.tree, st.dot + 1, st.i, st2.j),
                    "[10]/%d,%d" % (st.index, st2.index),
                    BUILD_SUBSTITUTION, st, st2)

```

```

def handle_right_adj(grammar, chart, st):
    """
    handles the case of right-adjunction rules (11) and (12)
    """
    if not st.is_complete():
        return

    # (11)
    for t in grammar.get_right_aux_trees_for(st.tree.root):
        chart.add(State(t, 0, st.j, st.j), "[11]/%d" % (st.index,))

    # (12)
    for st2 in chart:
        if (st2.tree.type == Tree.RIGHT_AUX and st2.tree.root == st.tree.root
            and st.j == st2.i and st2.is_complete()):
            chart.add(State(st.tree, len(st.tree.children), st.i, st2.j),
                "[12]/%d,%d" % (st.index, st2.index),
                BUILD_AUX, st, st2)

def parse(grammar, start_symbol, tokens, debug = False):
    """
    The actual parser: it takes a TIG grammar object, a start symbol
    (NonTerminal) of that grammar, and a list of tokens, and returns
    (hopefully) all possible parse trees for them.

    It works by first applying the initialization rule (1),
    then applying rules (2)-(12) for as long as the chart keeps changing,
    and once it's stable, it looks for matching states according to
    acceptance rule (13).

    It then takes all matching states (normally there should be only one),
    extracts the trees of each state, and returns a set of them.

    Note that TIG is assumed to be lexicalized, or at least finitely-ambiguous,
    so we know the number of trees is bounded.

    The parsing is done in  $O(|G|^2 * n^3)$ , as discussed in the paper,
    and tree extraction is performed in amortized linear time, per each tree.
    """
    if isinstance(tokens, str):
        tokens = tokens.split()
    chart = Chart()
    tokens = list(tokens)
    padded_tokens = [None] + tokens

    # (1)
    for t in grammar.get_init_trees_for(start_symbol):
        chart.add(State(t, 0, 0, 0), "[1]")

```

```

# main loop: run (2)-(12) until no more changes occur
while True:
    for st in chart:
        handle_left_adj(grammar, chart, st)
        tok = padded_tokens[st.j+1] if st.j+1 < len(padded_tokens) else None
        handle_scan(grammar, chart, st, tok)
        handle_substitution(grammar, chart, st)
        handle_subtree_traversal(grammar, chart, st)
        handle_right_adj(grammar, chart, st)

    if not chart.commit():
        # no more changes, we're done
        break

# (13)
matches = [st for st in chart if st.is_complete() and st.i == 0
            and st.j == len(tokens) and st.tree.root == start_symbol
            and st.tree.type == Tree.INIT_TREE]
if debug:
    chart.show()
    print "Matches:", [st.index for st in matches]
    print

# fail if no matching state was found
if not matches:
    raise ParsingError("Grammar does not derive the given sequence")

# extract trees, drop ones that do not generate the correct token sequence
trees = set(t for m in matches for t in chart.get_subtrees(m)
            if list(t.leaves()) == tokens)

# and make sure we didn't lose all trees, for then it's our fault
assert trees
return trees

```