— LECTURE NOTES —

The Cocke-Kasami-Younger Algorithm

1 The following description of the CKY algorithm follows very closely the presentation in Aho & Ullman (1972), *The Theory of Parsing, Translation and Compiling*. Compare this treatment to the treatment in the Edinburgh/Müller coursenotes.

2 As a matter of notation, let $G = \langle N, \Sigma, P, S \rangle$ refer to a CFG in Chomsky Normal Form, with no ϵ-productions. Let $w = a_1 a_2 \ldots a_n$ be the string to be parsed. Observe that $|w| = n$. We will refer to a_i as occurring at position i, although up until now we would have thought of it as being *between* $i - 1$ and i. So we associate words with their end positions, here.

The algorithm will construct a *parse table*, T. The cell appearing in the ith column and jth row will be denoted t_{ij}. *Note the column-major order of the subscripts!* Note also that row 1 is thought of as the *bottom* row of the table. The parse table is *triangular*: each row is one cell shorter than the last. (Look ahead to Fig. 2 to get an idea of what the table looks like.)

3 The cells of T contain sets of nonterminals.

The essential defining property of T is that if a nonterminal symbol A appears in t_{ij}, then it must be the case that

$$A \stackrel{+}{\Longrightarrow} a_i a_{i+1} \ldots a_{i+j-1} \quad .$$

Furthermore, if A derives $a_i a_{i+1} \ldots a_{i+j-1}$, then A must appear in t_{ij}. These two conditions play roles that might be called soundness and completeness, respectively.

If these two conditions are met, then w is in $L(G)$ if and only if S is in t_{1n} (the sole and only cell in line n of the table).

> Read t_{ij} as containing all those categories that can span a substring of the input starting at position i of length j.

So S being in t_{1n} means that we can analyze the substring of w from position 1 of length n as an S. Of course, that "substring" of w is w itself.

4 Again, you may want to look ahead to the parse table in Fig. 2, which represents all of the parses of the string *the villain heard the hero sneeze*, according to the grammar in Fig. 1. Try and verify that the above condition does indeed hold for all categories appearing in this table.

Note: Because the grammar must be in CNF, the rule $VP \longrightarrow V$ has to be "unfolded" into a collection of rules of the form $VP \longrightarrow \sigma$, for all $\sigma \in \Sigma$ of category V.

$$
\begin{aligned}
S &\longrightarrow NP\ VP \\
VP &\longrightarrow V\ NP \\
VP &\longrightarrow V\ S \\
NP &\longrightarrow Det\ N
\end{aligned}
\qquad
\begin{aligned}
Det &\longrightarrow the \\
N &\longrightarrow hero \\
N &\longrightarrow villain \\
V &\longrightarrow heard \\
V &\longrightarrow sneeze \\
VP &\longrightarrow heard \\
VP &\longrightarrow sneeze
\end{aligned}
$$

Figure 1: A Simple Grammar in CNF.

	6	S					
	5	S	–				
	4	–	–	VP			
	3	S	–	VP	S		
	2	NP	–	–	NP	–	
$j \uparrow$	1	Det	N	V,VP	Det	N	V,VP
	$i \rightarrow$	1 the	2 villain	3 heard	4 the	5 hero	6 sneeze

Figure 2: CYK Table.

5 The algorithm proceeds in two steps: an initialization step, in which we fill the cells in row 1 with all the possible lexical categories of words in w, and a main loop, in which we build up the higher lines of the table.

The table will be represented in Prolog by the definition of the predicate `t/3`. That is, we will build the table by asserting unit clauses for `t/3` into the Prolog database.

The top level of the program is given in Fig. 3.

```
cyk( String ):-
        retractall( t(_,_,_) ),
        step1( String, 0, N ),
        loop( 1, N ).
```

Figure 3: Top level of the Prolog CYK parser.

6 The grammar is implemented as in Fig. 4, like usual.

Step 1 of the algorithm is just a matter of setting t_{i1} equal to $\{A \mid A \longrightarrow a_i\}$. The corresponding Prolog code, for the relation `step1/3`, just looks up the words in the input string in the grammar, using `findall/3`, and asserts the result. As a side effect, `step1/3` returns the length of the input string in its third argument. The Prolog code for `step1/3` is given in Fig. 5. The order of arguments in clauses for `t/3` is `t(Contents,I,J)`.

```
rule( np, [det,n] ).
rule( vp, [v,np] ).
rule( vp, [v,s] ).
rule( s, [np,vp] ).

word( the, det ).
word( hero, n ).
word( villain, n ).
word( heard, v ).
word( heard, vp ).
word( sneeze, v ).
word( sneeze, vp ).
```

Figure 4: The grammar of Fig. 1 in Prolog.

7 The main loop works as follows. Given i and j, our job is to compute t_{ij}. We may assume that we have completed T for all rows $j' < j$. Then

$$\begin{aligned} t_{ij} = \{A \mid & \text{ for some } k, 1 \leq k < j, A \longrightarrow BC \text{ is in } P, \\ & B \text{ is in } t_{ik}, \text{ and } C \text{ is in } t_{i+k,j-k}\} \end{aligned}$$

```
step1( [], N, N ).
step1( [Wi|W], I, N ):-
        J is I+1,
        findall( Cat, word(Wi,Cat), Wi_Cats ),
        assertz( t(Wi_Cats,J,1) ),
        step1( W, J, N ).
```

Figure 5: Prolog code for `step1/3`.

That rather nasty looking subscript at the very end of the formula means the following. We are now in row j. That means we are looking at substrings of w of length j. If B is in t_{ik}, then it starts where we start (which is good) and is of length k. Obviously, k has to be strictly less than j. That means that we still have $j - k$ string positions "left over". So we need to look and see what we have available on row $j - k$.

Furthermore, if B is of length k and starts at position i, then C must start at the end of B, namely at position $i + k$. So the only legitimate candidates for C must be categories that start at position $i + k$. The cell which contains strings starting at $i + k$ and going on to cover $j - k$ positions are to be found in cell $t_{i+k,j-k}$.

For example, look at Fig. 2. In cell $t_{1,3}$ there is an S. How did it get there? Well, we have to consider all the categories starting at the same position on lower rows of the table. Right below our S, in cell $t_{1,2}$, there is an NP (so currently $k = 2$). Because it is on row 2, it spans two words of the input (*the villain*). Because we are currently interested in cells on row 3, we want to know if there is anything that can combine with that NP to make a well-formed substring of length 3. There is only one position left, so we must be looking for something on row 1 ($j - k$), and it must be immediately adjacent to our NP, which ends in position 2, so we have to look in $t_{3,1}$, where we find a V and a VP. There is no rule $S \longrightarrow NP\ V$, so we skip that one, but there is a rule $S \longrightarrow NP\ VP$, so we know there must be an S in $t_{1,3}$.

Try verifying that the table is sound for all the other cells. If you keep your left index finger on B and your right index finger on C, you will see that as your left hand drifts up the page your right hand drifts down and to the right. As B gets longer, C gets shorter, and starts later.

Our implementation will do just what your hands did.

8 The structure of the main loop is actually as a doubly nested loop. You should expect at least one nesting, since we are dealing with a two dimensional table. The main loop will be responsible for j, moving us up the rows from row 2 to row n ($= |w|$). Then for each value of j we will have to move across the current row. The inner loop will be responsible for i, then.

Then for each i, j pair we will have a third loop, which checks all the possible values of k. k stands for the midpoint, the string position between B and

C. In fact, it stands for the position of the first word in C, so it must vary from $i+1$ to j. (Remember, or grammar is assumed to be ϵ-free, so B must be at least one word long.)

In our Prolog code, the outer (j) loop is called `loop`, the middle (i) loop is called `line`, and the inner (k) loop is called `inner_loop`.

The following description of a procedure **line**(j) is taken from Aho & Ullman (1972). Step (1) is an initialization step which we carry out in `loop`. Steps (3–6) constitute the innermost loop.

1. Let $i = 1$ and $j' = n - j + 1$.
2. Let $k = 1$.
3. Let $k' = i + k$ and $j'' = j - k$.
4. Examine t_{ik} and $t_{k'j''}$. Let

$$t_{ij} = t_{ij} \cup \{A \mid A \longrightarrow BC \text{ is in } P, B \text{ in } t_{ik}, \text{ and } C \text{ in } t_{k'j''}\} \quad .$$

5. Increment k by 1.
6. If $k = j$, go to step (7). Otherwise, go to step (3).
7. If $i = j'$, halt. Otherwise do step (8).
8. Increment i by 1 and go to step (2).

The rest of the Prolog implementation is given in Fig. 6. Variable names in the Prolog implementation were chosen to be as close as possible to variable names in this algorithm. Integer suffixes were used instead of primes, so `J2` corresponds to j'', for example.

The main point at which the Prolog version differs from the above description is in that some variables are initialized to 0 instead of to 1 so that loop exit conditions can be checked first (**while** loops) where they are checked last (**repeat** loops) in the description. It is easier to implement **while** loops in Prolog, since the loop index can be checked against the loop bound by unification in the head of the clause. This also requires that loop index variables be incremented earlier in the Prolog code than they are in the description.

9 Who's that lurking behind that variable name? Variable j' (alias `J1`) is the last position on row j. For example, if you are on row 4 of a 6 row table, then the last available start position will be at position $6 - 4 + 1 = 3$. That is, there is a 4-position substring starting at position 3 (including positions 3, 4, 5 and 6), but, obviously, none starting in positions 4 or greater.

Variable k' (alias `K1`) marks the start of the second subconstituent (C).

Variable j'' (alias `J2`) represents the length of the second subconstituent: j is the length of A, k is the length of B, so the length of C must be $j - k$.

So variables t_{ik} (alias `T_i_k`) and $t_{k'j''}$ (alias `T_k1_j2`) represent the cells in which we are currently hoping to find our B and C categories.

10 Exit conditions If $k = j$ then the length of C is one. In an ϵ-free CNF grammar, that is the shortest constituent you can have. Remember that this is a **repeat** loop: the exit condition is checked after the end of the loop body, after we are done processing all cases with $k = j$ (i.e., with C as short as it is possible to be). So at this point we are done with the k-loop: there is nothing we can do with any larger values of k.

If $i = j'$ then we have finished the row (the i-loop). Remember, j' represents the last position in the current row.

```
loop( N, N ):- !.
loop( J0, N ):-
        J is J0 + 1,
        J1 is N - J + 1,
        line( 0, J1, J ),
        loop( J, N ).

line( IeqJ1, IeqJ1, _ ):- !.
line( I0, J1, J ):-
        I is I0 + 1,
        inner_loop( T_i_j, [], I, J, 1 ),
        assertz( t(T_i_j, I, J) ),
        line( I, J1, J ).

inner_loop( T_i_j_0, T_i_j, _I, JeqK, JeqK ):-
        !,
        T_i_j_0 = T_i_j.
inner_loop( T_i_j_0, T_i_j, I, J, K ):-
        K1 is I + K,
        J2 is J - K,
        t( T_i_k, I, K ),
        t( T_k1_j2, K1, J2 ),
        findall( A, core(T_i_k,T_k1_j2,A), T_i_j_0, T_i_j_1 ),
        NewK is K + 1,
        inner_loop( T_i_j_1, T_i_j, I, J, NewK ).


core( T1, T2, A ):-
        member( B, T1 ),
        member( C, T2 ),
        rule( A, [B,C] ).
```

Figure 6: The main loop(s).

11 The predicate `core/3` in Fig. 6 expresses the goal used in the call to `findall/4`, in the second clause of `inner_loop/5`.

The use of `findall/4` is interesting in itself: whereas `findall/3` returns a bag of solutions in a list, `findall/4` returns a bag of solutions in a difference list. This is very convenient in this context, because we want to combine all of the candidates for `A` that we find throughout the entire run of an inner loop.

Note that `findall` in either form does not remove duplicates. Unfortunately there is no 4-place variant of `setof`. If duplicates are a problem (i.e., the same category `A` is being constructed in multiple ways) then a quick and dirty solution is to call the Prolog builtin `sort` on `T_i_j`, once the inner loop is done; `sort` not only sorts a list but removes duplicates at the same time.

However, there is a very good reason to allow duplicates, namely if you want to *parse* with this algorithm. In that case, we need only change the definitions of two predicates—`step1` and `core`—as indicated in Fig. 7.

12 **Output** The programs presented here just build the table. The output can then be obtained by calling `t` as follows, where 6 is the length of the sentence just parsed.

```
?- t( STrees, 1, 6 ).
```

On exit `STrees` will be bound to a list of all the parses of the input sentence. This goal will succeed even if there were no parses of the input, binding `STrees` to `[]`.

```
step1( [], N, N ).
step1( [Wi|W], I, N ):-
        J is I+1,
        findall( Cat/(-Wi), word(Wi,Cat), Wi_Cats ),
        assertz( t(Wi_Cats,J,1) ),
        step1( W, J, N ).

core( T1, T2, A/[B/BSubs,C/CSubs] ):-
        member( B/BSubs, T1 ),
        member( C/CSubs, T2 ),
        rule( A, [B,C] ).
```

Figure 7: Modifications necessary to convert the recognizer to a parser.

```
rule( np, [det,n] ).
rule( vp, [v,np] ).
rule( vp, [v,s] ).
rule( s, [np,vp] ).

word( the, det ).    word( hero, n ).
word( villain, n ).  word( heard, v ).
word( heard, vp ).   word( sneeze, v ).
word( sneeze, vp ).  word( duck, n ).
word( duck, v ).     word( duck, vp ).
word( i, np ).       word( her, np ).
word( her, det ).    word( saw, v ).
word( saw, vp ).     word( saw, n ).
```

Figure 8: A more ambiguous grammar.

```
| ?- cyk([i,saw,her,duck]), t(_STrees,1,4), listing(t),
     nl, nl, print(_STrees).

t([np/ -(i)], 1, 1).
t([v/ -(saw),vp/ -(saw),n/ -(saw)], 2, 1).
t([np/ -(her),det/ -(her)], 3, 1).
t([n/ -(duck),v/ -(duck),vp/ -(duck)], 4, 1).
t([s/[np/ -(i),vp/ -(saw)]], 1, 2).
t([vp/[v/ -(saw),np/ -(her)]], 2, 2).
t([s/[np/ -(her),vp/ -(duck)],np/[det/ -(her),n/ -(duck)]], 3, 2).
t([s/[np/ -(i),vp/[v/ -(saw),np/ -(her)]]], 1, 3).
t([vp/[v/ -(saw),s/[np/ -(her),vp/ -(duck)]],
   vp/[v/ -(saw),np/[det/ -(her),n/ -(duck)]]], 2, 3).
t([s/[np/ -(i),vp/[v/ -(saw),s/[np/ -(her),vp/ -(duck)]]],
   s/[np/ -(i),vp/[v/ -(saw),np/[det/ -(her),n/ -(duck)]]]], 1, 4).


[s/[np/ -(i),vp/[v/ -(saw),s/[np/ -(her),vp/ -(duck)]]],
 s/[np/ -(i),vp/[v/ -(saw),np/[det/ -(her),n/ -(duck)]]]]
```

Figure 9: Running the parser on *I saw her duck*.