Efficiency in
Parsing Arbitrary Grammars

Parsing using CYK Algorithm

1) Transform any grammar to Chomsky Form, **in this order**, to ensure:

1. terminals t occur alone on the right-hand side: X:=t
2. no unproductive non-terminals symbols
3. no productions of arity more than two
4. no nullable symbols except for the start symbol
5. no single non-terminal productions X::=Y
6. no non-terminals unreachable from the starting one

Have only rules X ::= Y Z, X ::= t

Questions:

- What is the worst-case increase in grammar size in each step?
- Does any step break the property established by previous ones?

2) Apply CYK dynamic programming algorithm

A CYK for Any Grammar Would Do This

input: grammar G, non-terminals $A_1,\ldots,A_K$, tokens $t_1,\ldots.t_L$

word: $\mathbf{w} \equiv \mathbf{w}_{(0)}\mathbf{w}_{(1)} \ldots \mathbf{w}_{(N-1)}$

notation: $w_{p..q} = w_{(p)}w_{(p+1)} \ldots w_{(q-1)}$

output: P set of **(A, i, j)** implying $\mathbf{A} \Rightarrow^* \mathbf{w}_{i..j}$, A can be: A_k, t_k, or ε

$P = \{(w_{(i)},i,i+1) \mid 0 \leq i < N-1\}$

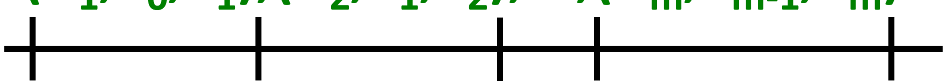
repeat {

choose rule $(A::=B_1\ldots B_m) \in G$

if $((A,k_0,k_m) \notin P$ && (for some $k_1,\ldots,k_{m-1}$:

$((m=0$ && $k_0=k_m)$ || $(B_1,k_0,k_1),(B_2,k_1,k_2),\ldots,(B_m,k_{m-1},k_m) \in P)))$

$P := P \cup \{(A,k_0,k_m)\}$



} until no more insertions possible

What is the maximal number of steps?

How long does it take to check step for a rule?

} for a given grammar

Observation

- How many ways are there to split a string of length Q into m segments?
  - number of {0,1} words of length Q+m with m zeros

$$\binom{Q+m}{m} = \frac{(Q+m)!}{Q!\,m!}$$

- Exponential in m, so algorithm is exponential.
- For binary rules, m=2, so algorithm is efficient.
  - this is why we use at most binary rules in CYK
  - transformation into Chomsky form is polynomial

CYK Parser for Chomsky form

input: grammar G, non-terminals $A_1,...,A_K$, tokens $t_1,....t_L$

word: $\mathbf{w} \equiv \mathbf{w}_{(0)}\mathbf{w}_{(1)} \ldots \mathbf{w}_{(N-1)}$

notation: $w_{p..q} = w_{(p)}w_{(p+1)} \ldots w_{(q-1)}$

output: P set of **(A, i, j)** implying $\mathbf{A} \Rightarrow^* \mathbf{w}_{i..j}$, A can be: A_k, t_k, or ε

P = {(A,i,i+1) | 0 ≤ i < N-1 && ((A ::= $w_{(i)}$)∈G)} // unary rules

repeat {

choose rule ($A::=B_1B_2$)∈G

if ((A,k_0,k_2)∉P && for some k_1: (B_1,k_0,k_1),(B_2,k_1,k_2) ∈ P)

P := P U {(A,k_0,k_2)}

} until no more insertions possible

return (S,0,N-1) ∈ P

Give a bound on the number of elements in **P**: $K(N+1)^2/2+LN$

Next: not just **whether** it parses, but compute the **trees**!

Computing Parse Results
Semantic Actions

A CYK Algorithm Producing Results

Rule **$(A::=B_1...B_m , f) \in G$** with **semantic action f**

$f : (R \cup T)^m \rightarrow R$ **R – results (e.g.trees)** **T - tokens**

Useful parser: returning a set of result (e.g. syntax trees)

$((A, p, q),\mathbf{r})$: $A \Rightarrow^* w_{p..q}$ **and** the result of parsing is **r**

$P = \{((A,i,i+1), f(w_{(i)})) \mid 0 \leq i < N-1$ && $((A ::=w_{(i)}),f) \in G)\}$ // unary
repeat {
choose rule $(A::=B_1B_2 , f) \in G$
if $((A,k_0,k_2) \notin P$ && for some k_1: $((B_1,k_0,k_1),r_1), ((B_2,p_1,p_2),r_2) \in P$
$P := P \cup \{((A,k_0,k_2), f(r_1,r_2))\}$
} until no more insertions possible

Compute parse trees using identity functions as semantic actions:

$((A ::=w_{(i)})$, $x:R \Rightarrow x$**)** **$((A::=B_1B_2)$,** $(r_1,r_2):R^2 \Rightarrow Node_A(r_1,r_2)$ **)**

A bound on the number of elements in **P**? 2^N : squared in each level

Computing Abstract Trees for Ambiguous Grammar

```
abstract class Tree
case class ID(s:String) extends Tree
case class Minus(e1:Tree,e2:Tree) extends Tree
```

Ambiguous grammar: E ::= E – E | Ident

type R = Tree

Chomsky normal form:	semantic actions:
E ::= E R	(e_1,e_2) => Minus(e_1,e_2)
R ::= M E	(_,e_2) => e_2
E ::= Ident	x => ID(x)
M ::= –	_ => Nil

Input string:

a – b – c

0 1 2 3 4

P:

((E,0,1),ID(a)) ((M,1,2),Nil) ((E,2,3),ID(b)) ((M,3,4),Nil) ((E,4,5),ID(c))

((R,1,3),ID(b)) ((R,3,5),ID(c))

((E,0,3),Minus(ID(a),ID(b)))

((E,2,5),Minus(ID(b),ID(c)))

((R,1,5),Minus(ID(b),ID(c)))

((E,0,5),Minus(Minus(ID(a),ID(b)), ID(c)))

((E,0,5),Minus(ID(a), Minus(ID(b),ID(c))))

A CYK Algorithm with Constraints

Rule **$(A::=B_1...B_m\ , f) \in G$** with **partial function** semantic action **f**

$f : (R \cup T)^m$ -> Option[R] **R – results** **T - tokens**

Useful parser: returning a set of results (e.g. syntax trees)

$((A, p, q),\mathbf{r})$: $A =>^* w_{p..q}$ **and** the result of parsing is **$r \in R$**

$P = \{((A,i,i+1), f(w_{(i)}).get) \mid 0 \leq i < N-1$ && $((A ::= w_{(i)}), f) \in G)\}$

repeat {

choose rule $(A::=B_1B_2\ , f) \in G$

if $((A,k_0,k_2) \notin P$ && for some k_1: $((B_1,k_0,k_1),r_1), ((B_2,p_1,p_2),r_2) \in P$

and $f(r_1,r_2)$!= None //apply rule only if **f** is defined

$P := P \cup \{((A,k_0,k_2), f(r_1,r_2).get)\}$

} until no more insertions possible

Resolving Ambiguity using Semantic Actions

In Chomsky normal form: semantic action:

In Chomsky normal form:	semantic action:
E ::= E R	~~(e_1,e_2) => $Minus(e_1,e_2)$~~ mkMinus
R ::= M e	$(\_,e_2)$ => e_2
E ::= Ident	x => ID(x)
M ::= –	_ => Nil

```
def mkMinus(e1 : Tree, e2: Tree) : Option[Tree] = (e1,e2) match {
  case (_,Minus(_,_)) => None
  case _ => Some(Minus(e1,e2))
}
```

Input string:

a – b – c

0 1 2 3 4

P:

((e,0,1),ID(a)) ((M,1,2),Nil) ((e,2,3),ID(b)) ((M,3,4),Nil) ((e,4,5),ID(c))

((R,1,3),ID(b)) ((R,3,5),ID(c))

((e,0,3),Minus(ID(a),ID(b)))

((e,2,5),Minus(ID(b),ID(c)))

((R,1,5),Minus(ID(b),ID(c)))

((e,0,5),Minus(Minus(ID(a),ID(b)), ID(c)))

~~((e,0,5),Minus(ID(a), Minus(ID(b),ID(c))))~~

Expression with More Operators: All Trees

```
abstract class T
case class ID(s:String) extends T
case class BinaryOp(e1:T,op:OP,e2:T) extends T
```

Ambiguous grammar: E ::= E (–|^) E | (E) | Ident

Chomsky form:	semantic action f:	type of **f** (can vary):
E ::= E R	$(e_1,(op,e_2))$=>BinOp(e_1,op,e_2)	(T,(OP,T)) => T
R ::= O E	(op,e_2)=>(op,e_2)	(OP,T) => (OP,T)
E ::= Ident	x => ID(x)	Token => T
O ::= –	_ => MinusOp	Token => OP
O ::= ^	_ => PowerOp	Token => OP
E ::= P Q	(_,e) => e	(Unit,T) => T
Q ::= E C	(e,_) => e	(T,Unit) => T
P ::= (	_ => ()	Token => Unit
C ::=)	_ => ()	Token => Unit

Priorities

- In addition to the tree, return the priority of the tree
  - usually the priority is the top-level operator
  - parenthesized expressions have high priority, as do other 'atomic' expressions (identifiers, literals)
- Disallow combining trees if the priority of current right-hand-side is higher than priority of results being combining
- Given: x - y * z with priority of * higher than of -
  - disallow combining x-y and z using *
  - allow combining x and y*z using -

Priorities and Associativity

abstract class T
case class ID(s:String) **extends** T
case class BinaryOp(e1:T,op:OP,e2:T) **extends** T

Ambiguous grammar: E ::= E (–|^) E | (E) | Ident

Chomsky form: semantic action f: type of **f**

E ::= E R (T',(OP,T')) => Option[T']
R ::= O E type T' = (Tree,Int) tree,priority
E ::= Ident
O ::= –
O ::= ^
E ::= P Q
Q ::= E C
P ::= (
C ::=)

Priorities and Associativity

Chomsky form: semantic action f: type of **f**

E ::= E R mkBinOp (T',(OP,T')) => T'

def mkBinOp((e_1,p_1):T', (op:OP,(e_2,p_2):T')) : Option[T'] = {
  val p = priorityOf(op)
  if ((p < p_1 || (p==p_1 && isLeftAssoc(op)) &&
      (p < p_2 || (p==p_2 && isRightAssoc(op)))
    Some((BinaryOp(e_1,op,e_2),p))
  else None // there will another item in P that will apply instead
}

cf. middle operator: a*b**+**c*d a+b*****c*d a–b**–**c–d a^b**^**c^d

Parentheses get priority p larger than all operators:

E ::= P Q (_,(e,p)) => Some((e,MAX))

Q ::= E C (e,_) => Some(e)

Efficiency of Dynamic Programming

Chomsky normal form:	semantic action:
E ::= E R	mkMinus
R ::= M e	(_,e_2) => e_2
E ::= Ident	x => ID(x)
M ::= –	_ => Nil

Input string:

a – b – c

0 1 2 3 4

P:

((e,0,1),ID(a)) ((M,1,2),Nil) ((e,2,3),ID(b)) ((M,3,4),Nil) ((e,4,5),ID(c))

((R,1,3),ID(b)) ((R,3,5),ID(c))

((e,0,3),Minus(ID(a),ID(b)))

((e,2,5),Minus(ID(b),ID(c)))

((R,1,5),Minus(ID(b),ID(c)))

((e,0,5),Minus(Minus(ID(a),ID(b)), ID(c)))

~~((e,0,5),Minus(ID(a), Minus(ID(b),ID(c))))~~

Naïve dynamic programming: derive all tuples (X,i,j) increasing j-i

Instead: derive only the needed tuples, first time we need them

Start from top non-terminal

Result: **Earley's parsing algorithm** (also needs no normal form!)

Other efficient algos for LR(k),LALR(k) – not handle all grammars

Dotted Rules Like Non-terminals

$$X ::= Y_1\ Y_2\ Y_3$$

Chomsky transformation is (a simplification of) this:

$X ::= W_{123}$
$W_{123} ::= W_{12}\ Y_3$
$W_{12} ::= W_1\ Y_2$
$W_1 ::= W_\varepsilon\ Y_1$
$W_\varepsilon ::= \varepsilon$

Early parser: dotted RHS as names of fresh non-terminals:

$X ::= [Y_1Y_2Y_3.]$
$[Y_1Y_2Y_3.] ::= [Y_1Y_2.Y_3]\ \ Y_3$
$[Y_1Y_2.Y_3] ::= [Y_1.Y_2Y_3]\ \ Y_2$
$[Y_1.Y_2Y_3] ::= [.Y_1Y_2Y_3]\ \ Y_1$
$[.Y_1Y_2Y_3] ::= \varepsilon$

Earley Parser

- group the triples by last element: S(q) ={(A,p) | (A,p,q)∈P}
- dotted rules effectively make productions at most binary

Steps of Earley Parsing Algorithm

Initially, let $S(0) = \{(D' ::= .D \text{ EOF},\ 0)\}$

When scanning input at position j, parser does the following operations (p, q, r are sequences of terminals and non-terminals):

Prediction

If $(X ::= p.Yq, i) \in S(j)$ and $Y ::= r$ is a grammar rule, then

$$S(j) = S(j) \cup \{(Y ::= .r, j)\}$$

Scanning

If $w(j) = a$ and $(X ::= p.aq, i) \in S(j)$ then (we can skip a):

$$S(j+1) = S(j+1) \cup \{(X ::= pa.q, i)\}$$

Completion

If $(X ::= p., i) \in S(j)$ and $(Y ::= q.Xr, k) \in S(i)$ then

$$S(j) = S(j) \cup \{(Y ::= qX.r, k)\}$$

sketch of completion:

```
w(0) ... w(k)    ...    w(i)    ...    w(j)
           |      q       |      p       |
                      Y::=q.Xr         X::=p.
                                       Y::=qX.r
```

		ID s_1	- s_2	ID s_3	==	ID	EOF
	ε .e EOF .ID .e-e .e==e	ID ID. e.EOF e.-e e.==e	ID- e-.e	ID-ID e-e. e.EOF e.-e e.==e	ID-ID== e==.e	ID-ID==ID e==e. e-e.	e.EOF
ID		ε	-	-ID	-ID==	-ID==ID	
-			ε .ID .e-e .e==e	ID ID. e.-e e.==e	ID== e==.e	ID==ID e==e.	
ID				ε	==	==ID	
==					ε .ID .e-e .e==e	ID ID. e.-e e.==e	
ID						ε	
EOF							ε

S :: . e EOF ; e . EOF ; e EOF .

e :: .ID ; ID.

| .e – e ; e. – e ; e –. e ; e – e.

| .e == e ; e. == e ; e ==. e ; e == e.

Attribute Grammars

- They extend context-free grammars to give parameters to non-terminals, have rules to combine attributes
- Attributes can have any type, but often they are trees
- Example:
  - context-free grammar rule:

    A ::= B C

  - attribute grammar rules:

    A ::= B C **{ Plus($1, $2) }**

    or, e.g.

    A ::= B**:x** C**:y {: RESULT := new Plus(x.v, y.v) :}**

Semantic actions indicate how to compute attributes

- attributes computed bottom-up, or in more general ways

Parser Generators:
Attribute Grammar -> Parser

1) Embedded: parser combinators (Scala, Haskell)

They ***are*** code in some (functional) language

implicit conversion: string s to skip(s)

concatenation

```
def ID : Parser = "x" | "y" | "z"
def expr : Parser = factor ~ (( "+" ~ factor | "-" ~ factor )
                            | epsilon)
def factor : Parser = term ~ (( "*" ~ term | "/" ~ term )
                             | epsilon)
def term : Parser = ( "(" ~ expr ~ ")" | ID | NUM )
```

<- often not really LL(1) but "try one by one", must put first non-empty, then epsilon

implementation in Scala: use **overloading** and **implicits**

2) Standalone tools: JavaCC, Yacc, ANTLR, CUP

– typically ***generate*** code in a conventional programming languages (e.g. Java)

Example in CUP - LALR(1) (not LL(1))

```
precedence left PLUS, MINUS;
precedence left TIMES, DIVIDE, MOD;  // priorities disambiguate
precedence left UMINUS;


expr ::= expr PLUS expr          // ambiguous grammar works here
       | expr MINUS expr
       | expr TIMES expr
       | expr DIVIDE expr
       | expr MOD expr
       | MINUS expr %prec UMINUS
       | LPAREN expr RPAREN
       | NUMBER ;
```

Adding Java Actions to CUP Rules

```
expr ::= expr:e1 PLUS expr:e2
        {: RESULT = new Integer(e1.intValue() + e2.intValue()); :}
        | expr:e1 MINUS expr:e2
        {: RESULT = new Integer(e1.intValue() - e2.intValue()); :}
        | expr:e1 TIMES expr:e2
        {: RESULT = new Integer(e1.intValue() * e2.intValue()); :}
        | expr:e1 DIVIDE expr:e2
        {: RESULT = new Integer(e1.intValue() / e2.intValue()); :}
        | expr:e1 MOD expr:e2
        {: RESULT = new Integer(e1.intValue() % e2.intValue()); :}
        | NUMBER:n   {: RESULT = n; :}
        | MINUS expr:e
      {: RESULT = new Integer(0 - e.intValue()); :} %prec UMINUS
        | LPAREN expr:e RPAREN {: RESULT = e; :} ;
```

Which Algorithms Do Tools Implement

- Many tools use LL(1)
  - easy to understand, similar to hand-written parser
- Even more tools use LALR(1)
  - in practice more flexible than LL(1)
  - can encode priorities without rewriting grammars
  - can have annoying shift-reduce conflicts
  - still does not handle general grammars
- Today we should probably be using more parsers for general grammars, such as Earley's (optimized CYK)