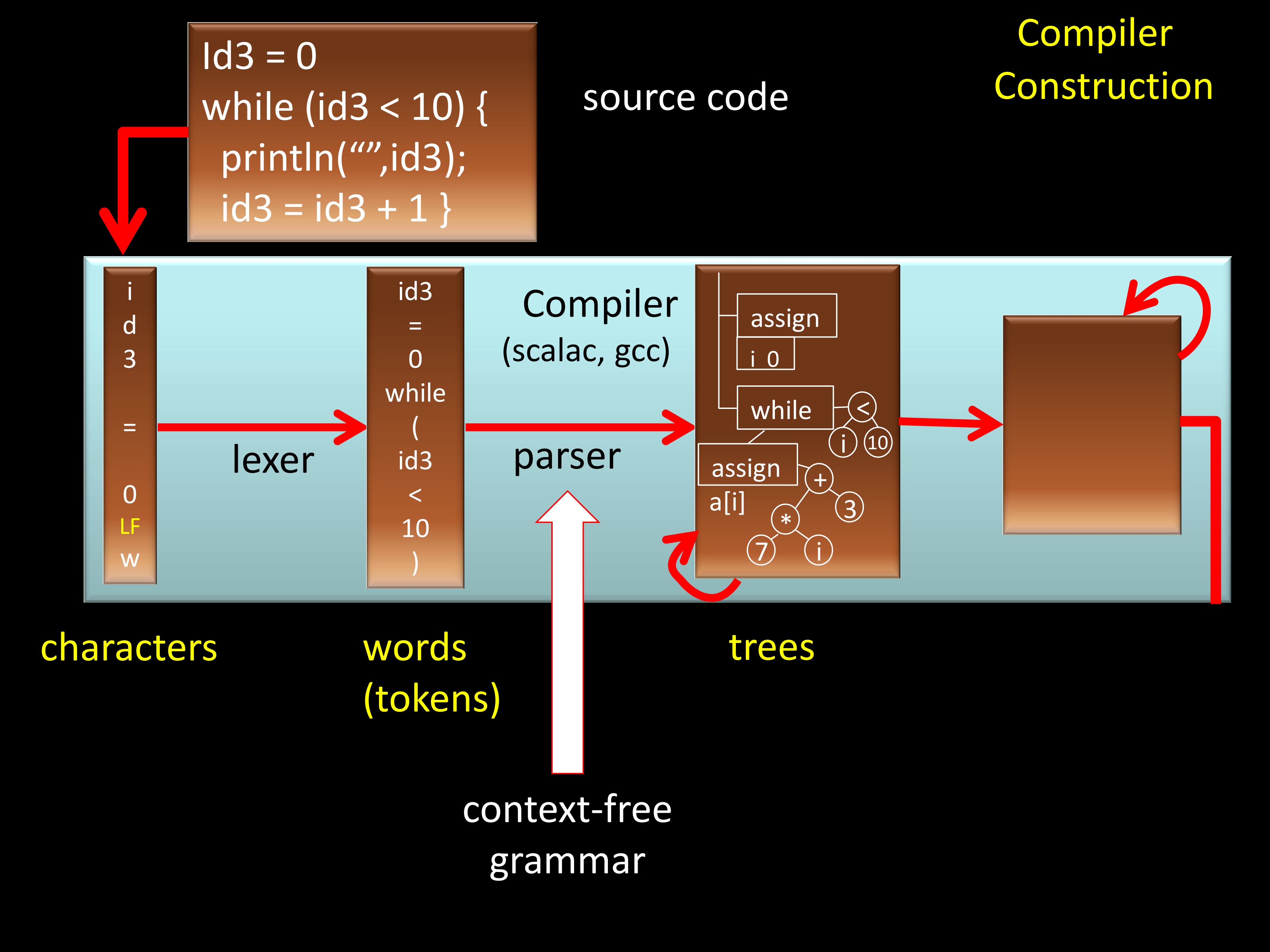
http://lara.epfl.ch

Compiler Construction 2010, Lecture 4

Parsing General Context-Free Grammars


Compiler
Construction
Id3 = 0
while (id3 < 10) {
println(“”,id3);
id3 = id3 + 1 }
source code
i
d
3
=
0
LF
w
lexer
id3
=
0
while
(
id3
<
10
)
Compiler
(scalac, gcc)
parser
assign
i 0
while
<
i
10
assign
a[i]
+
3
*
7
i
characters
words
(tokens)
trees
context-free
grammar

Today

- CYK parsing algorithm
  - Examples
  - Chomsky normal form for grammars
  - CYK Algorithm
  - Transformations to Chomsky form
- Earley's parsing algorithm
  - Example
  - Earley's Algorithm
- Examples of completed projects from 2009

Why Parse General Grammars

- Can be difficult or impossible to make grammar unambiguous
  - thus LL(k) and LR(k) methods cannot work, for such ambiguous grammars
- Some inputs are more complex than simple programming languages
  - mathematical formulas:
    x = y /\ z ? (x=y) /\ z x = (y /\ z)
  - natural language:
    I saw the man with the telescope.
  - future programming languages

Ambiguity

1)

2)

I saw the man with the telescope.

CYK Parsing Algorithm

John **Cocke** and Jacob T. Schwartz (1970). Programming languages and their compilers: Preliminary notes. Technical report, Courant Institute of Mathematical Sciences, New York University.

T. **Kasami** (1965). An efficient recognition and syntax-analysis algorithm for context-free languages. Scientific report AFCRL-65-758, Air Force Cambridge Research Lab, Bedford, MA.

Daniel H. **Younger** (1967). Recognition and parsing of context-free languages in time n^3. *Information and Control* 10(2): 189–208.

Two Steps in the Algorithm

1) Transform grammar to normal form
   called Chomsky Normal Form
   (Noam Chomsky, mathematical linguist)

2) Parse input using transformed grammar
   dynamic programming algorithm

“a method for solving complex problems by breaking them down into simpler steps.
It is applicable to problems exhibiting the properties of overlapping subproblems” (>WP)

Balanced Parentheses Grammar

Exercise:
- copy normal form grammar
- for each rule of type (1) in normal form indicate rules in original grammar

Original grammar G

$S \rightarrow$ “” | (S) | S S

1 2 3

Modified grammar in Chomsky Normal Form:

$S \rightarrow$ “” | S’ ← if "" $\in L(G)$

(0)

3 min

$S' \rightarrow N_(N_{S)} \mid N_(N_) \mid S' S'$

2 2 3

$N_{S)} \rightarrow S'_2 N_)$

} Rules (1) $N \rightarrow N_1 N_2$ nonterminals

$N_(\rightarrow$ (

$N_) \rightarrow$)

} Rules (2) $N \rightarrow t$ nonterminal, terminal

- Terminals: ( ) Nonterminals: S S' $N_{S)}$ $N_)$ $N_($

nonterminal with funny name

Idea How We Obtained the Grammar

$S \rightarrow (\; S \;)$

because S can be empty but S' cannot

$S' \rightarrow N_(\; N_{S)} \mid N_(\; N_)$

$N_(\rightarrow ($

$N_{S)} \rightarrow S' \; N_)$

$N_) \rightarrow)$

Chomsky Normal Form transformation can be done fully mechanically

Dynamic Programming to Parse Input

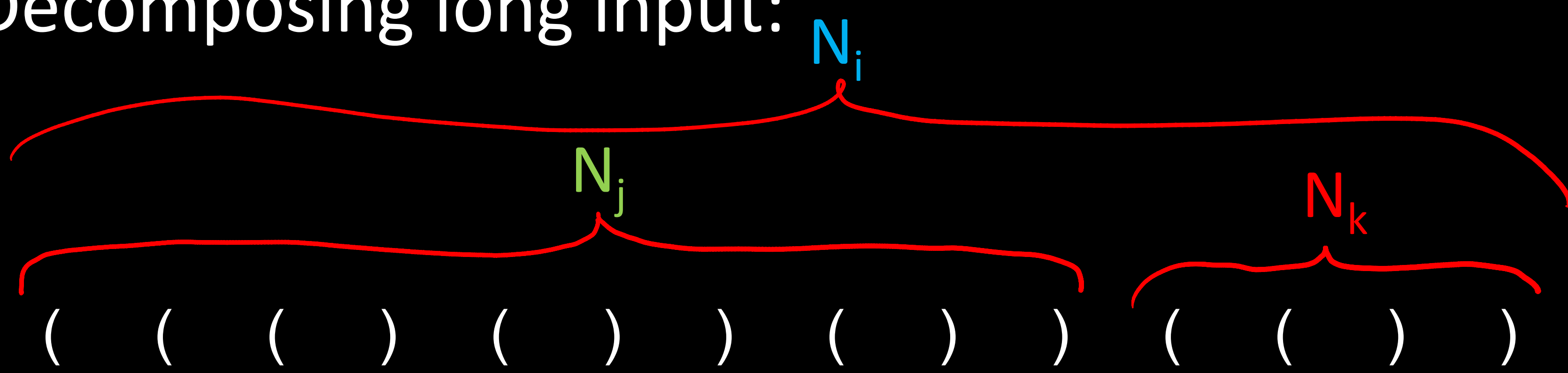
Assume Chomsky Normal Form, 3 types of rules:

$S \rightarrow$ “” | S’ (only for the start non-terminal)

$N_j \rightarrow t$ (names for terminals)

$N_i \rightarrow N_j \; N_k$ (just **2** non-terminals on RHS)

Decomposing long input:



find all ways to parse substrings of length 1,2,3,…

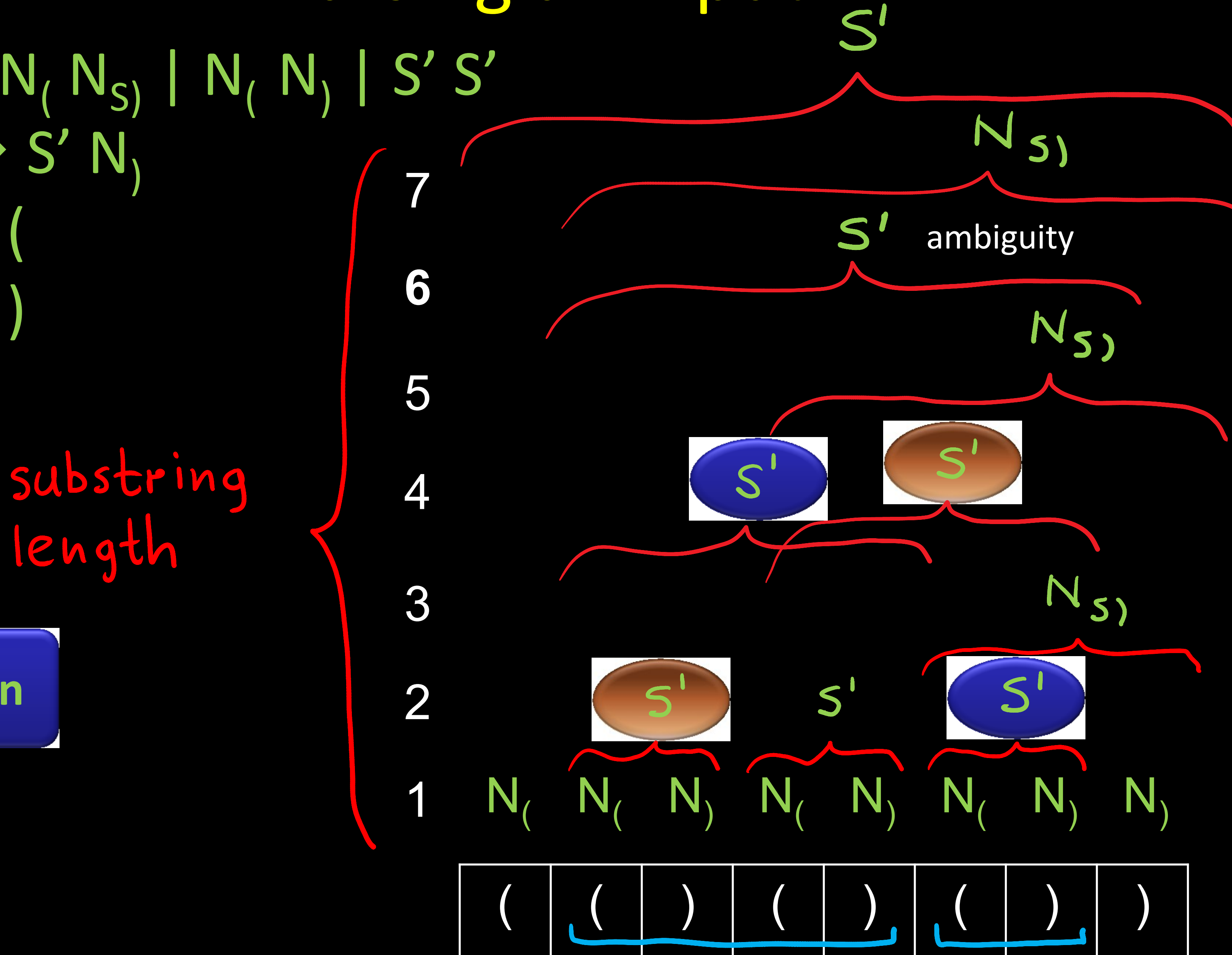
Parsing an Input

$S' \rightarrow N_(\; N_{S)} \mid N_(\; N_) \mid S' \; S'$

$N_{S)} \rightarrow S' \; N_)$

$N_(\rightarrow ($

$N_) \rightarrow)$



2 min

Algorithm Idea

$S' \rightarrow S' \, S'$

w_{pq} – substring from p to q

d_{pq} – all non-terminals that could expand to w_{pq}

Initially d_{pp} has $N_{w(p,p)}$

key step of the algorithm:

if $X \rightarrow Y\,Z$ is a rule,

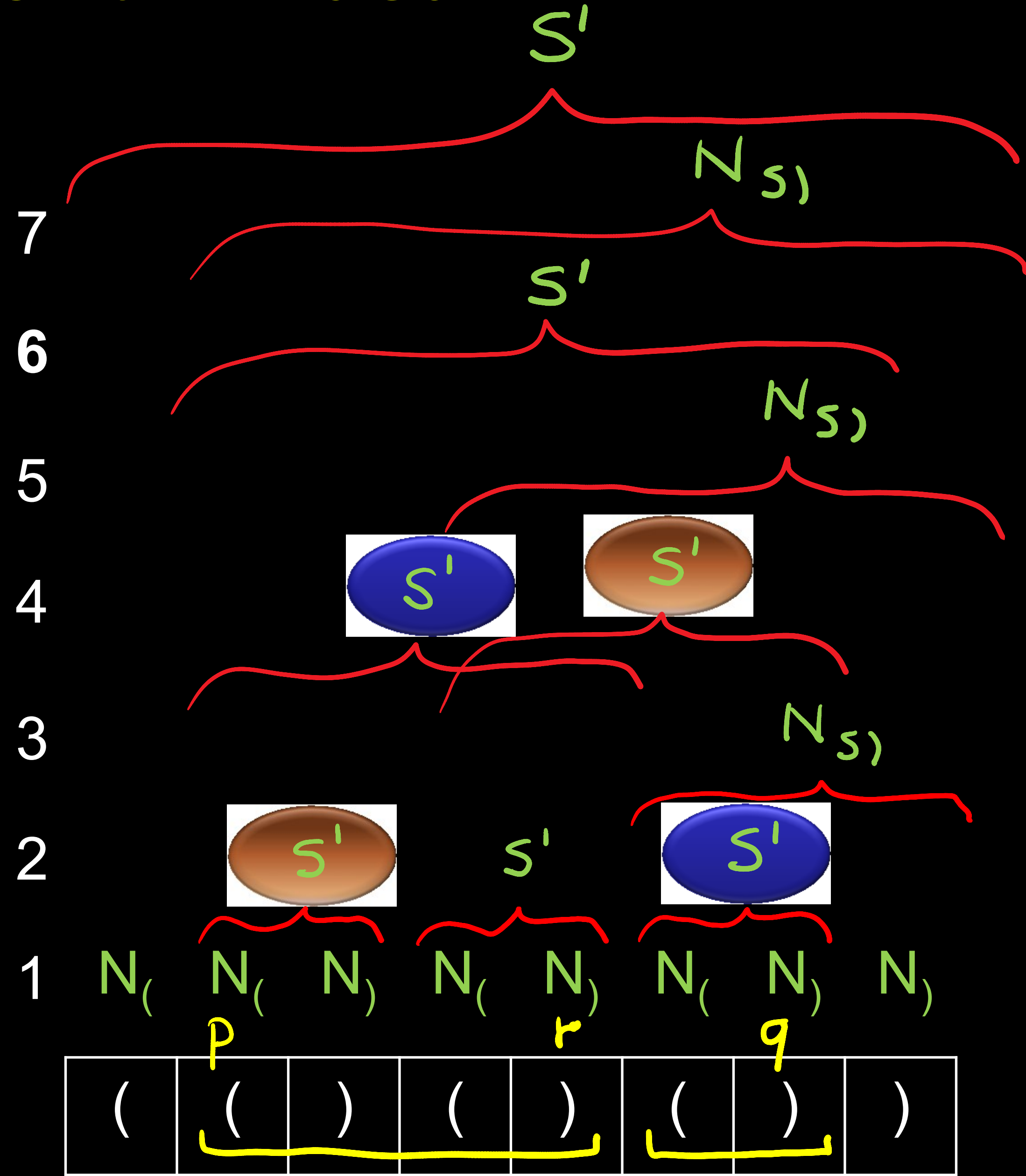
Y is in $d_{p\,r}$, and

Z is in $d_{(r+1)q}$

then put X into d_{pq}

($p \leq r < q$),

in increasing value of (q-p)



Algorithm

INPUT: grammar G in Chomsky normal form
word w to parse using G

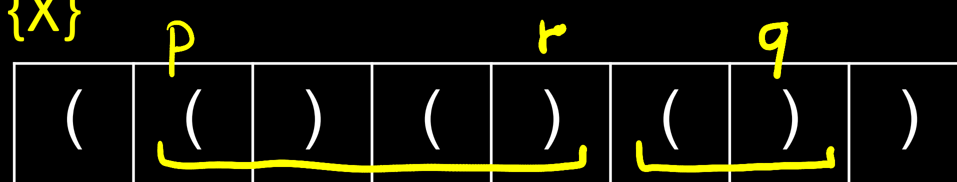
OUTPUT: true iff (w in L(G))

```
N = |w|
var d : Array[N][N]
for p = 1 to N {
  d(p)(p) = {X | G contains X->w(i)}
  for q in {p + 1 .. N} d(p)(q) = {} }
for k = 2 to N // substring length
 for p = 0 to N-k // initial position
  for j = 1 to k-1 // length of first half
   val r = p+j-1; val q = p+k-1;
   for (X::=Y Z) in G
    if Y in d(p)(r) and Z in d(p+r+1)(q)
      d(p)(q) = d(p)(q) union {X}
return  S in d(0)(N-1)
```

What is the running time as a function of grammar size and the size of input?

O(   )

2 min



Parsing another Input

$S' \rightarrow N_(\, N_{S)} \mid N_(\, N_) \mid S' \, S'$

$N_{S)} \rightarrow S' \, N_)$

$N_(\rightarrow ($

$N_) \rightarrow)$

substring length

7

6

5

4

3

2

1 $N_($ $N_)$ $N_($ $N_)$ $N_($ $N_)$ $N_($ $N_)$

(	)	(	)	(	)	(	)

5 min

Number of Parse Trees

- Let w denote word ()()()
  - it has two parse trees
- Give a lower bound on number of parse trees of the word w^n (n is positive integer)

  w^5 is the word

  ()()() ()()() ()()() ()()() ()()()

  2 min

- CYK represents all parse trees compactly
  - can re-run algorithm to extract first parse tree, or enumerate parse trees one by one

Algorithm Idea

$S' \rightarrow S'\ S'$

w_{pq} – substring from p to q

d_{pq} – all non-terminals that could expand to w_{pq}

Initially d_{pp} has $N_{w(p,p)}$

key step of the algorithm:

if $X \rightarrow Y\ Z$ is a rule,

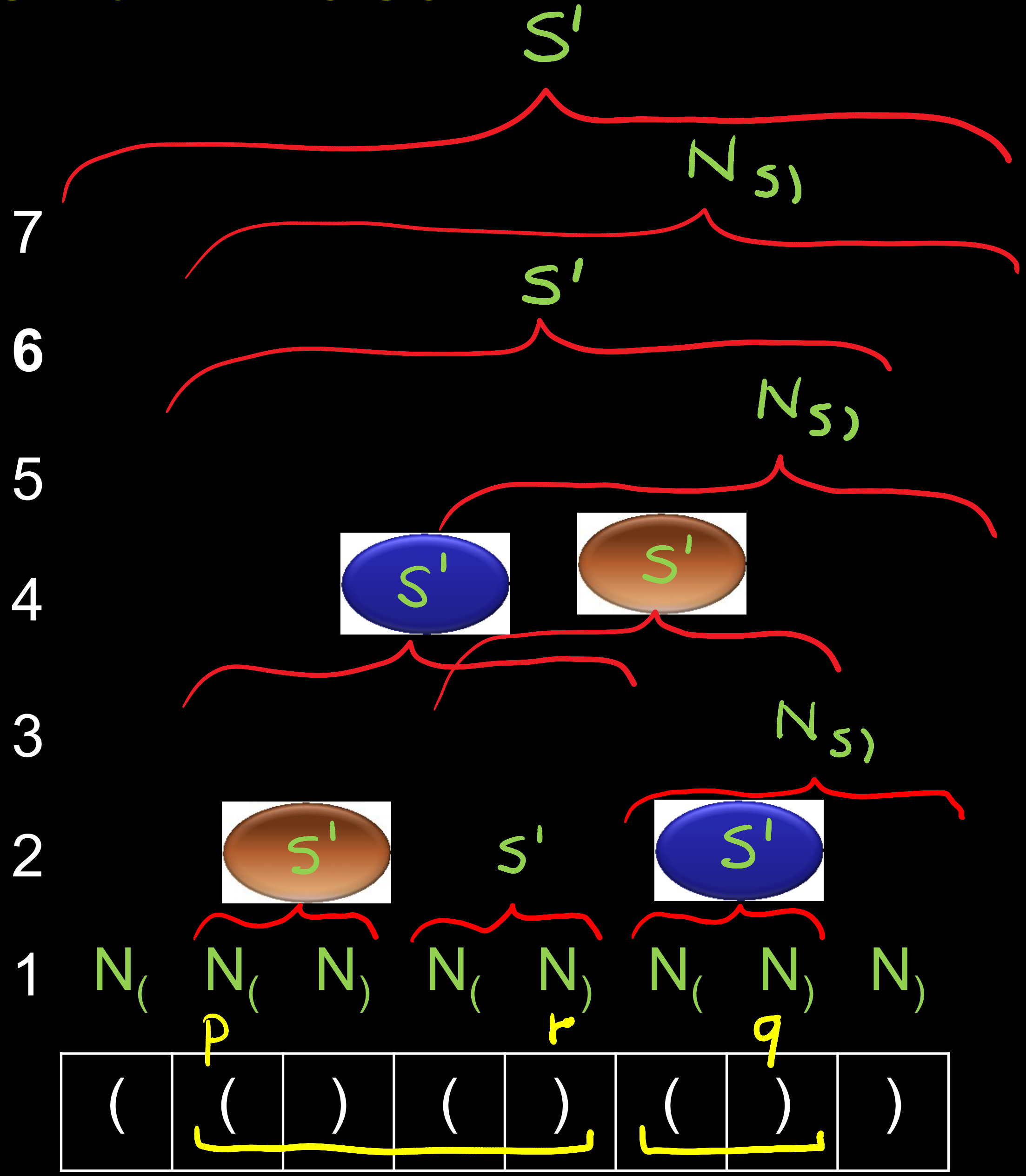
Y is in $d_{p\,r}$, and

Z is in $d_{(r+1)q}$

then put X into d_{pq}

$(p \leq r < q)$,

in increasing value of (q-p)



Transforming to Chomsky Form

- Steps:
  1. remove unproductive symbols
  2. remove unreachable symbols
  3. remove epsilons (no non-start nullable symbols)
  4. remove single non-terminal productions X::=Y
  5. transform productions of arity more than two
  6. make terminals occur alone on right-hand side

1) Unproductive non-terminals

How to compute them?

What is funny about this grammar:

stmt ::= identifier := identifier
| while (expr) stmt
| if (expr) stmt else stmt
expr ::= term + term | term – term
term ::= factor * factor
factor ::= (expr)

2 min

There is no derivation of a sequence of tokens from expr

Why? In every step will have at least one expr, term, or factor

If it cannot derive sequence of tokens we call it *unproductive*

1) Unproductive non-terminals

- Productive symbols are obtained using these two rules (what remains is unproductive)
  - Terminals are productive
  - If $X ::= s_1\ s_2 \ldots s_n$ is rule and each s_i is productive then X is productive

```
stmt ::= identifier := identifier
       | while (expr) stmt
       | if (expr) stmt else stmt
expr ::= term + term | term – term
term ::= factor * factor
factor ::= ( expr )
program ::= stmt | stmt program
```

Delete unproductive symbols.

Will the meaning of top-level symbol (program) change?

2) Unreachable non-terminals

What is funny about this grammar with starting terminal 'program'

program ::= stmt | stmt program

stmt ::= assignment | whileStmt

assignment ::= expr = expr

ifStmt ::= if (expr) stmt else stmt

whileStmt ::= while (expr) stmt

expr ::= identifier

2 min

No way to reach symbol 'ifStmt' from 'program'

2) Unreachable non-terminals

What is funny about this grammar with starting terminal 'program'

```
program ::= stmt | stmt program
stmt ::= assignment | whileStmt
assignment ::= expr = expr
ifStmt ::= if (expr) stmt else stmt
whileStmt ::= while (expr) stmt
expr ::= identifier
```

What is the general algorithm?

2) Unreachable non-terminals

- Reachable terminals are obtained using the following rules (the rest are unreachable)
  - starting non-terminal is reachable (program)
  - If $X ::= s_1\ s_2 \dots s_n$ is rule and X is reachable then each non-terminal among $s_1\ s_2 \dots s_n$ is reachable

Delete unreachable symbols.

Will the meaning of top-level symbol (program) change?

2) Unreachable non-terminals

What is funny about this grammar with starting terminal ‘program’

program ::= stmt | stmt program
stmt ::= assignment | whileStmt
assignment ::= expr = expr
~~ifStmt ::= if (expr) stmt else stmt~~
whileStmt ::= while (expr) stmt
expr ::= identifier

3) Removing Empty Strings

Ensure only top-level symbol can be nullable

program ::= stmtSeq
stmtSeq ::= stmt | stmt ; stmtSeq
stmt ::= “” | assignment | whileStmt | blockStmt
blockStmt ::= { stmtSeq }
assignment ::= expr = expr
whileStmt ::= while (expr) stmt
expr ::= identifier

5 min

How to do it in this example?

3) Removing Empty Strings - Result

program ::= “” | stmtSeq
stmtSeq ::= stmt| stmt ; stmtSeq |
| ; stmtSeq | stmt ; | ;
stmt ::= assignment | whileStmt | blockStmt
blockStmt ::= { stmtSeq } | { }
assignment ::= expr = expr
whileStmt ::= while (expr) stmt
whileStmt ::= while (expr)
expr ::= identifier

3) Removing Empty Strings - Algorithm

- Compute the set of nullable non-terminals
- Add extra rules
  - If $X ::= s_1\ s_2\ \ldots\ s_n$ is rule then add new rules of form
    $X ::= r_1\ r_2\ \ldots\ r_n$

  where r_i is either s_i or, if s_i is nullable then r_i can also be the empty string (so it disappears)
- Remove all empty right-hand sides
- If starting symbol S was nullable, then introduce a new start symbol S' instead, and add rule S' ::= S | ""

3) Removing Empty Strings

- Since stmtSeq is nullable, the rule

  blockStmt ::= { stmtSeq }

  gives

  blockStmt ::= { stmtSeq } | { }

- Since stmtSeq and stmt are nullable, the rule

  stmtSeq ::= stmt | stmt ; stmtSeq

  gives

  stmtSeq ::= stmt | stmt ; stmtSeq
  | ; stmtSeq | stmt ; | ;

4) Eliminating single productions

- Single production is of the form

  X ::=Y

where X,Y are non-terminals

```
program ::= stmtSeq
stmtSeq ::= stmt
          | stmt ; stmtSeq
stmt ::= assignment | whileStmt
assignment ::= expr = expr
whileStmt ::= while (expr) stmt
```

4) Eliminate single productions - Result

- Generalizes removal of epsilon transitions from non-deterministic automata

```
program ::= expr = expr | while (expr) stmt
          | stmt ; stmtSeq
stmtSeq ::= expr = expr | while (expr) stmt
          | stmt ; stmtSeq
stmt ::= expr = expr | while (expr) stmt
assignment ::= expr = expr
whileStmt ::= while (expr) stmt
```

} now unreachable

4) "Single Production Terminator"

- If there is single production

  X ::=Y   put an edge (X,Y) into graph

- If there is a path from X to Z in the graph, and there is rule Z ::= s_1 s_2 … s_n then add rule

  X ::= s_1 s_2 … s_n

1 min

At the end, remove all single productions.

```
program ::= expr = expr | while (expr) stmt
          | stmt ; stmtSeq
stmtSeq ::= expr = expr | while (expr) stmt
          | stmt ; stmtSeq
stmt ::= expr = expr | while (expr) stmt
```

5) No more than 2 symbols on RHS

stmt ::= while (expr) stmt

becomes

stmt ::= while $stmt_1$

$stmt_1$ ::= ($stmt_2$

$stmt_2$ ::= expr $stmt_3$

$stmt_3$ ::=) stmt

6) A non-terminal for each terminal

stmt ::= while (expr) stmt

1 2 3

becomes

$stmt ::= N_{while}\ stmt_1$

$stmt_1 ::= N_{(}\ stmt_2$

$stmt_2 ::= expr\ stmt_3$

$stmt_3 ::= N_{)}\ stmt$

$N_{while} ::= while$

$N_{(} ::= ($

$N_{)} ::=)$

Parsing using CYK Algorithm

- Transform grammar into Chomsky Form:
  1. remove unproductive symbols
  2. remove unreachable symbols
  3. remove epsilons (no non-start nullable symbols)
  4. remove single non-terminal productions X::=Y
  5. transform productions of arity more than two
  6. make terminals occur alone on right-hand side

  Have only rules X ::= Y Z, X ::= t, and possibly S ::= “”
- Apply CYK dynamic programming algorithm

Algorithm Idea

$S' \rightarrow S' S'$

w_{pq} – substring from p to q

d_{pq} – all non-terminals that could expand to w_{pq}

Initially d_{pp} has $N_{w(p,p)}$

key step of the algorithm:

if $X \rightarrow Y Z$ is a rule,

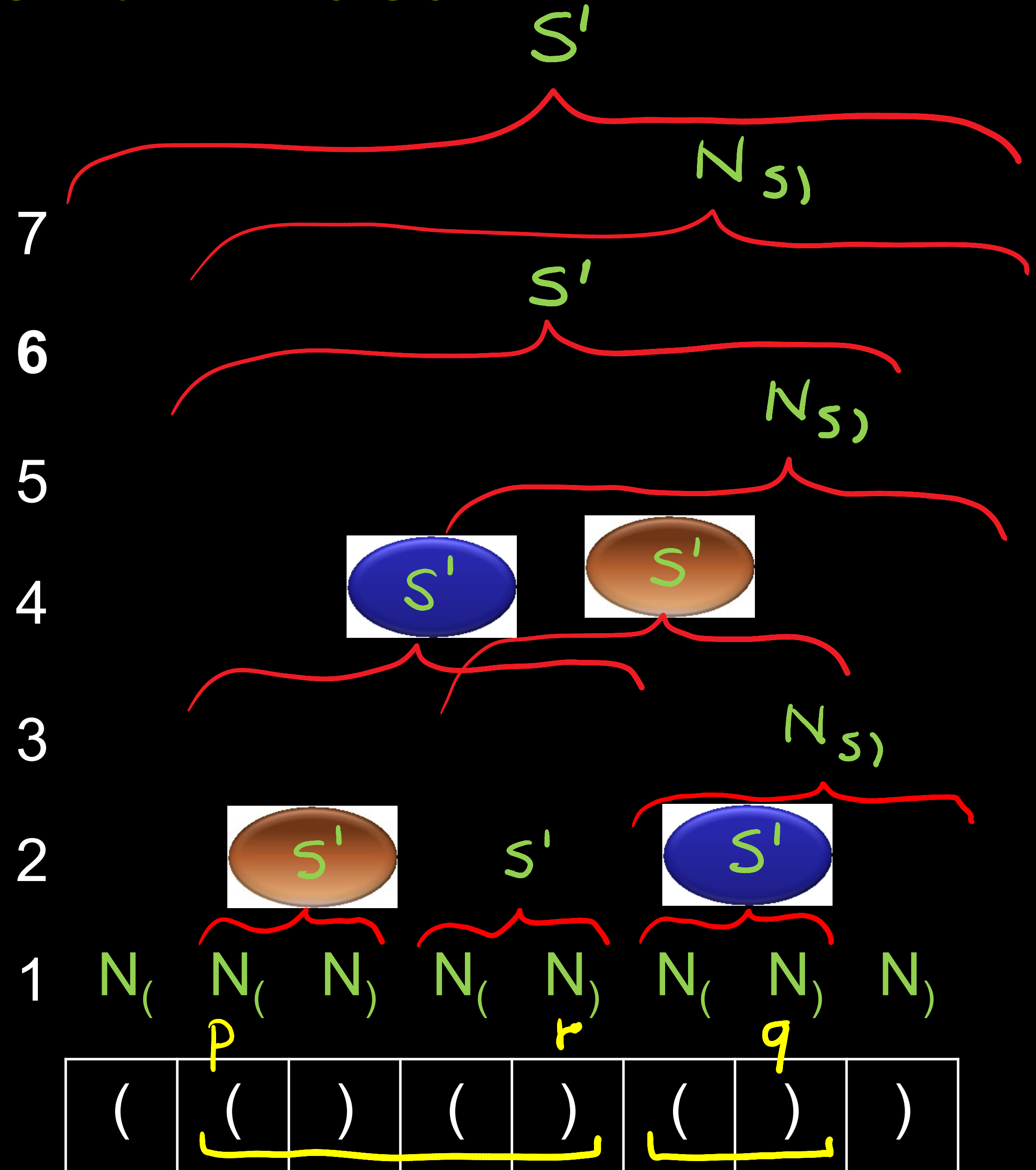
Y is in $d_{p r}$, and

Z is in $d_{(r+1)q}$

then put X into d_{pq}

$(p \leq r < q)$,

in increasing value of (q-p)



Earley's Algorithm
(wiki)

J. Earley, "An efficient context-free parsing algorithm", *Communications of the Association for Computing Machinery*, **13**:2:94-102, 1970.

Examples of Completed 2009 Projects

- Implemented all phases, then added an extension of the language and/or compiler:
  - Type Inference and Implicit Type Conversion
    - Added type inference
    - Added implicit conversions, as in Scala
  - Static garbage collection and C back-end
    - Emitted C instructions to automatically de-allocate memory, based on static analysis of source code
  - Support for exceptions in the language
  - Adding generic types (templates)

ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

http://lara.epfl.ch

Compiler Construction 2010, Lecture 4
- end -

Parsing General Context-Free Grammars