Finite State Automata are Limited

Let us use (context-free) **grammars**!

Context Free Grammar for a^nb^n

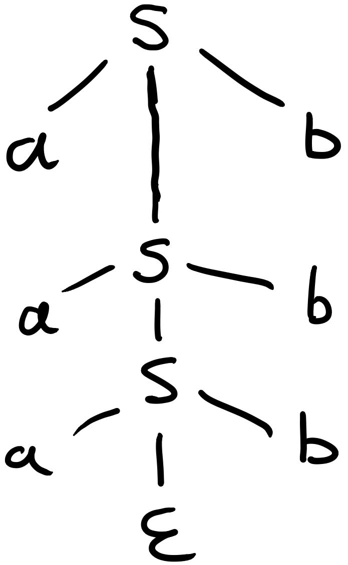
S ::= ε - a grammar rule

S ::= a S b - another grammar rule

Example of a derivation

S => aSb => a aSb b => aa aSb bb => aaabbb

Parse tree:



leaves give us the result

aaabbb

Context-Free Grammars

← all sets are finite A, N, R

G = (A, N, S, R)

- A - terminals (alphabet for generated words w ∈ A*)
- N - non-terminals – symbols with (recursive) definitions
- Grammar rules in R are pairs (n,v), written

  n ::= v where

  n ∈ N is a non-terminal

  v ∈ (A U N)* - sequence of terminals and non-terminals

A derivation in G starts from the starting symbol S ∈ N

- Each step replaces a non-terminal with one of its right hand sides

Example from before: G = ({a,b}, {S}, S, {(S,ε), (S,aSb)})

Parse Tree

Given a grammar G = (A, N, S, R), t is a **parse tree** of G iff t is a node-labelled tree with ordered children that satisfies:

- root is labeled by S
- leaves are labelled by elements of $A \cup \{\varepsilon\}$
- each non-leaf node is labelled by an element of N
- for each non-leaf node labelled by n whose children left to right are labelled by $p_1 \ldots p_n$, we have a rule $(n ::= p_1 \ldots p_n) \in R$

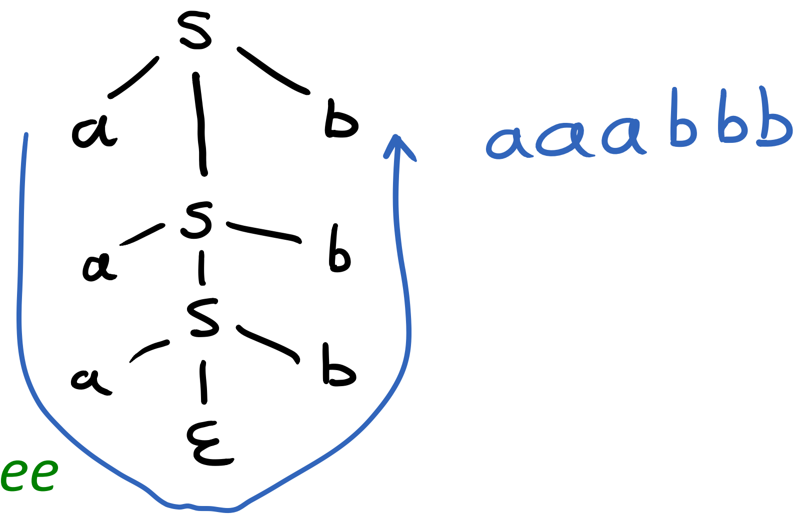
Yield of a parse tree t is the unique word in A* obtained by reading the leaves of t from left to right

Language of a grammar G = words of all yields of parse trees of G

L(G) = {yield(t) | isParseTree(G,t)}

isParseTree - **easy** to check condition

Harder: know *if a word has a parse tree*



Grammar Derivation

A **derivation** for G is any sequence of words $p_i \in (A \cup N)^*$,whose:

- first word is S
- each subsequent word is obtained from the previous one by replacing one of its letters by right-hand side of a rule in R :
  $p_i = unv$, $(n::=q) \in R$,
  $p_{i+1} = uqv$
- Last word has only letters from A

Each parse tree of a grammar has one or more derivations, which result in expanding tree gradually from S

- Different orders of expanding non-terminals may generate the same tree

Remark

We abbreviate

S ::= p

S ::= q

as

S ::= p | q

Example: Parse Tree vs Derivation

Consider this grammar G = ({a,b}, {S,P,Q}, S, R) where R is:

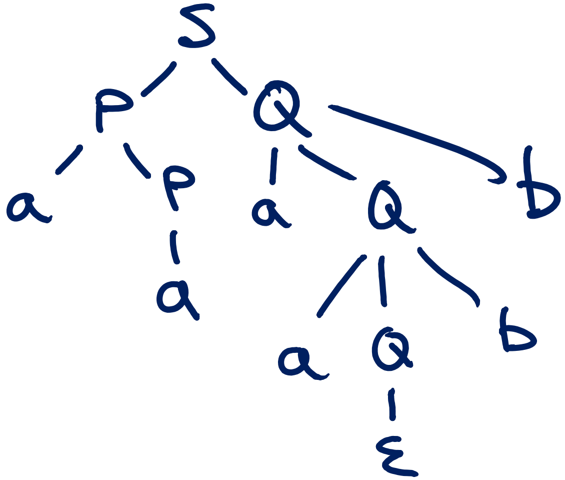
S ::= PQ
P ::= a | aP
Q ::= ε | aQb

$\{a^m a^n b^n \mid m \geq 1, n \geq 0\}$

Show a derivation tree for aaaabb

Show at least two derivations that correspond to that tree.

$S \Rightarrow PQ \Rightarrow aPQ \Rightarrow aaQ \Rightarrow aa\,aQb \Rightarrow aa\,aaQbb \Rightarrow aaaabb$



$S \Rightarrow PQ \Rightarrow PaQb$
$\Rightarrow P\,aaQbb$
$\Rightarrow aP\,aaQbb$
$\Rightarrow aP\,aabb$
$\Rightarrow aa\,aabb$

Balanced Parentheses Grammar

Consider the language L consisting of precisely those words consisting of parentheses “**(**“ and “**)**” that are balanced (each parenthesis has the matching one)

- Example sequence of parentheses

((()) ()) - balanced, belongs to the language

( )) (( ) - not balanced, does not belong

Exercise: give the grammar and example derivation for the first string.

Balanced Parentheses Grammar

$B ::= BB \mid (B) \mid \varepsilon$

$C ::= C\,(C)\,C \mid \varepsilon$

$D ::= (D)D \mid \varepsilon$

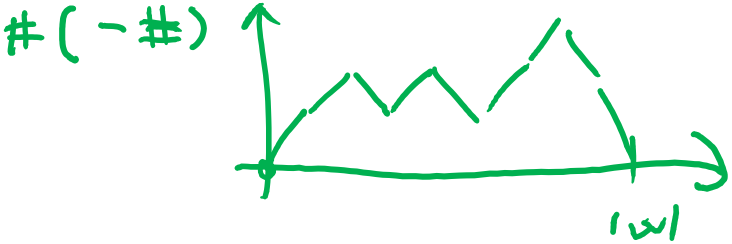
Proving that a Grammar Defines a Language

Grammar G: S ::= ε | **(**S**)**S

defines language L(G)

Theorem: $L(G) = L_b$

where L_b = { w | for every pair *u*,*v* of words such that *uv*=*w*, the number of **(** symbols in *u* is greater or equal than the number of **)** symbols in *u* . These numbers are equal in w }



$L(G) \subseteq L_b$: If $w \in L(G)$, then it has a parse tree. We show $w \in L_b$ by induction on size of the parse tree deriving w using G.

If tree has one node, it is ε, and $\varepsilon \in L_b$, so we are done.

Suppose property holds for trees up size n. Consider tree of size n. The root of the tree is given by rule **(**S**)**S . The derivation of sub-trees for the first and second S belong to L_b by induction hypothesis. The derived word w is of the form (p)q where $p,q \in L_b$. Let us check if $(p)q \in L_b$. Let (p)q = uv and count the number of **(** and **)** in u. If $u = \varepsilon$ then it satisfies the property. If it is shorter than |p|+1 then it has at least one more **(** than **)**. Otherwise it is of the form $(p)q_1$ where q_1 is a prefix of q. Because the parentheses balance out in p and thus in (p), the difference in the number of **(** and **)** is equal to the one in q_1 which is a prefix of q so it satisfies the property. Thus u satisfies the property as well.

$L_b \subseteq L(G)$: If $w \in L_b$, we need to show that it has a parse tree. We do so by induction on $|w|$. If $w = \varepsilon$ then it has a tree of size one (only root). Otherwise, suppose all words of length $<n$ have parse tree using G. Let $w \in L_b$ and $|w|=n>0$. (Please refer to the figure counting the difference between the number of **(** and **)**. We split w in the following way: let p_1 be the shortest non-empty prefix of w such that the number of **(** equals to the number of **)**. Such prefix always exists and is non-empty, but could be equal to w itself. Note that it must be that p_1 = **(**p**)** for some p because p_1 is a prefix of a word in L_b , so the first symbol must be **(** and, because the final counts are equal, the last symbol must be **)**. Therefore, w = **(**p**)**q for some shorter words p,q. Because we chose p to be the shortest, prefixes of **(**p always have at least one more **(**. Therefore, prefixes of p always have at greater or equal number of **(**, so p is in L_b. Next, for prefixes of the form **(**p**)**v the difference between **(** and **)** equals this difference in v itself, since (p) is balanced. Thus, v has at least as many **(** as **)**. We have thus shown that w is of the form (p)q where p,q are in L_b. By IH p,q have parse trees, so there is parse tree for w.

Exercise: Grammar Equivalence

Show that each string that can be derived by grammar G_1

$$B ::= \varepsilon \mid \mathbf{(}\ B\ \mathbf{)} \mid B\ B$$

can also be derived by grammar G_2

$$B ::= \varepsilon \mid \mathbf{(}\ B\ \mathbf{)}\ B$$

and vice versa. In other words, $L(G_1) = L(G_2)$

Remark: there is no algorithm to check for equivalence of *arbitrary* grammars. We must be clever.

Grammar Equivalence

G_1: B ::= ε | **(** B **)** | B B

G_2: B ::= ε | **(** B **)** B

(Easy) Lemma: Each word in alphabet A={(,)} that can be derived by G_2 can also be derived by G_1.

Proof. Consider a derivation of a word w from G_2. We construct a derivation of w in G_1 by showing one or more steps that produce the same effect. We have several cases depending on steps in G_2 derivation:

uBv => uv	replace by (same)	uBv => uv
uBv => u(B)Bv	replace by	uBv => uBBv => u(B)Bv

This constructs a valid derivation in G_1.

Corollary: $L(G_2) \subseteq L(G_1)$

Lemma: $L(G_1) \subseteq L_b$ (words derived by G_1 are balanced parentheses).

Proof: very similar to proof of $L(G_2) \subseteq L_b$ from before.

Lemma: $L_b \subseteq L(G_2)$ – this was one direction of proof that $L_b = L(G_2)$ before.

Corollary: $L(G_2) = L(G_1) = L(L_b)$

Regular Languages and Grammars

Exercise: give grammar describing the same language as this regular expression:

(a|b)(ab)*b*

S ::= P Q R

P ::= a
P ::= b

Q ::= ab Q
Q ::= ε

R ::= b R
R ::= ε

Translating Regular Expression into a Grammar

- Suppose we first allow regular expression operators * and | within grammars
- Then R becomes simply

  S ::= R

- Then give rules to remove *, | by introducing new non-terminal symbols

$$N ::= R_1 \mid R_2 \quad \Longrightarrow \quad \begin{array}{l} N ::= R_1 \\ N ::= R_2 \end{array}$$

$$N ::= R^* \quad \Longrightarrow \quad \begin{array}{l} N ::= \varepsilon \\ N ::= R\,N \end{array}$$

Eliminating Additional Notation

- Alternatives

s ::= P | Q   becomes   s ::= P
s ::= Q

- Parenthesis notation
  - introduce fresh non-terminal

expr (&& | < | == | + | - | * | / | %) expr

- Kleene star

{ statmt* }

- Option – use an alternative with epsilon

if (expr) statmt (else statmt)$^?$

Grammars for Natural Language

→ can also be used to automatically generate essays

Statement = Sentence "."

Sentence ::= Simple | Belief

Simple ::= Person liking Person

liking ::= "likes" | "does" "not" "like"

Person ::= "Barack" | "Helga" | "John" | "Snoopy"

Belief ::= Person believing "that" Sentence but

believing ::= "believes" | "does" "not" "believe"

but ::= "" | "," "but" Sentence

Exercise: draw the derivation tree for:

John does not believe that
Barack believes that Helga likes Snoopy,
but Snoopy believes that Helga likes Barack.

While Language Syntax

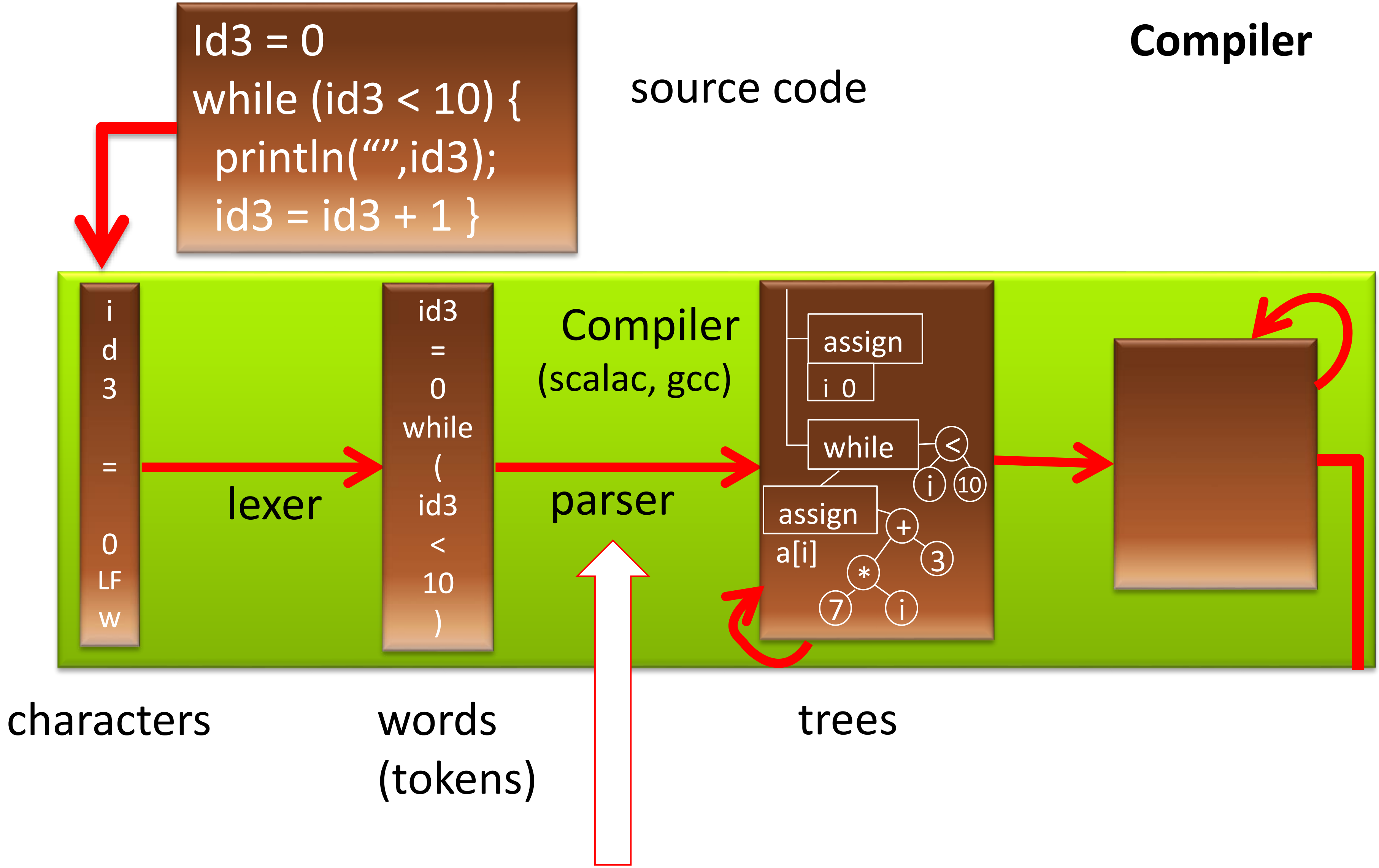
This syntax is given by a context-free grammar:

program ::= statmt*

statmt ::= println(stringConst , ident)
| ident = expr
| **if** (expr) statmt (else statmt)$^?$
| **while** (expr) statmt
| { statmt* }

expr ::= intLiteral | ident
| expr (&& | < | == | + | - | * | / | %) expr
| ! expr | - expr

Compiler



```
Id3 = 0
while (id3 < 10) {
  println(“”,id3);
  id3 = id3 + 1 }
```

source code

Compiler (scalac, gcc)

lexer

parser

characters

words (tokens)

trees

Recursive Descent Parsing - Manually

- weak, but useful parsing technique
- to make it work, we might need to transform the grammar

Recursive Descent is Decent

descent = a movement downward

decent = adequate, good enough

Recursive descent is a decent parsing technique

- can be easily implemented manually based on the grammar (which may require transformation)
- efficient (linear) in the size of the token sequence

Correspondence between grammar and code

- concatenation → ;
- alternative (|) → if
- repetition (*) → while
- nonterminal → recursive procedure

A Rule of While Language Syntax

// Where things work very nicely for recursive descent!

```
statmt ::=
          println ( stringConst , ident )
        | ident = expr
        | if ( expr ) statmt (else statmt)?
        | while ( expr ) statmt
        | { statmt* }
```

Parser for the statmt (rule -> code)

```
def skip(t : Token) = if (lexer.token == t) lexer.next
  else error("Expected"+ t)
// statmt ::=
def statmt = {
  // println ( stringConst , ident )
  if (lexer.token == Println) { lexer.next;
    skip(openParen); skip(stringConst); skip(comma);
    skip(identifier); skip(closedParen)
  // | ident = expr
  } else if (lexer.token == Ident) { lexer.next;
    skip(equality); expr
  // | if ( expr ) statmt (else statmt)?
  } else if (lexer.token == ifKeyword) { lexer.next;
    skip(openParen); expr; skip(closedParen); statmt;
    if (lexer.token == elseKeyword) { lexer.next; statmt }
  // | while ( expr ) statmt
```

Continuing Parser for the Rule

```
// | while ( expr ) statmt

} else if (lexer.token == whileKeyword) { lexer.next;
  skip(openParen); expr; skip(closedParen); statmt


// | { statmt* }

} else if (lexer.token == openBrace) { lexer.next;
  while (isFirstOfStatmt) { statmt }
  skip(closedBrace)

} else { error(“Unknown statement, found token ” +
                    lexer.token)  }
```

How to construct **if** conditions?

statmt ::= println (stringConst , ident)
| ident = expr
| if (expr) statmt (else statmt)$^{?}$
| while (expr) statmt
| { statmt* }

- Look what each alternative starts with to decide what to parse
- Here: we have terminals at the beginning of each alternative
- More generally, we have 'first'computation, as for regular expresssions
- Consider a grammar G and non-terminal N

L_G(N) = { set of strings that N can derive }

e.g. L(statmt) – all statements of while language

first(N) = { a | aw in L_G(N), a – terminal, w – string of terminals}

first(statmt) = { **println**, **ident**, **if**, **while**, **{** }

first(**while** (expr) statmt) = { **while** }