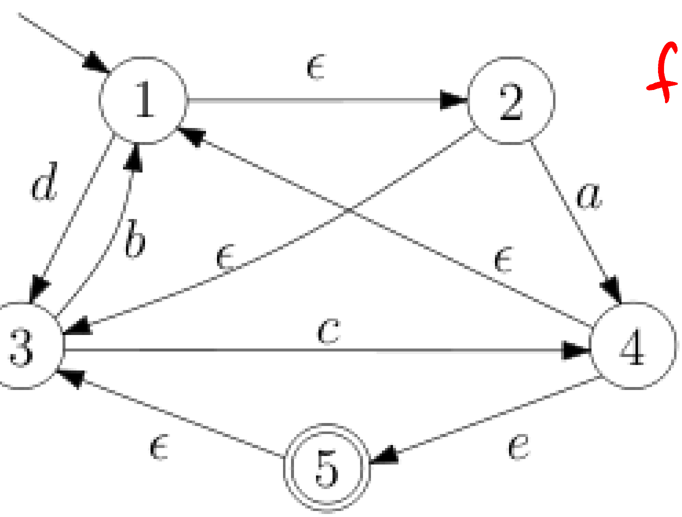
Given NFA A, find first(L(A))

- First symbols of words accepted by non-deterministic finite state machine with epsilon transitions
- Give general technique

A   L(A) = {..., ce, ... }

first(L(A))
= {c, ... }

a, c, d, b



More Questions

- Find automaton or regular expression for:
  - Sequence of open and closed parentheses of even length?
  - as many digits before as after decimal point?
  - Sequence of balanced parentheses

    ((()) ()) - balanced

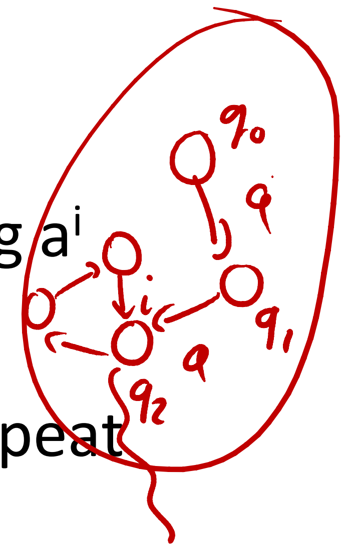
    ( )) (( ) - not balanced
  - Comment as a sequence of space,LF,TAB, and comments from // until LF
  - Nested comments like /* ... /* */ ... */

Automaton that Claims to Recognize $\{ a^n b^n \mid n >= 0 \}$

Make the automaton deterministic

Let the resulting DFA have K states, $|Q|=K$

Feed it a, aa, aaa, ... Let q_i be state after reading a^i



$q_0, q_1, q_2, \ldots, q_K$

This sequence has length K+1 -> a state must repeat

$q_i = q_{i+p}$ $\quad p > 0$

Then the automaton should accept $a^{i+p}b^{i+p}$.

But then it must also accept

$$a^i b^{i+p}$$

because it is in state after reading a^i as after a^{i+p}.

So it does not accept the given language.

Limitations of Regular Languages

- Every automaton can be made deterministic
- Automaton has finite memory, cannot count
- Deterministic automaton from a given state behaves always the same
- If a string is too long, deterministic automaton will repeat its behavior

Pumping Lemma

- Each finite language is regular (why?)
- To prove that an *infinite* L is not regular:
  - suppose it is regular
  - let the automaton recognizing it have ***K*** states
  - long words will make the automaton loop
  - shortest cycle has length ***K*** or less
  - if adding or removing a loop changes if ***w*** is in L, we have contradiction, e.g. uvw in L, uw not in L
- Pumping lemma: a way to do proofs as above

Pumping Lemma

If L is a regular language, then there exists a positive integer p (the pumping length) such that every string $s \in$ L for which $|s| \geq p$, can be partitioned into three pieces, $s = x\,y\,z$, such that

- $|y| > 0$
- $|xy| \leq p$
- $\forall i \geq 0.\ xy^iz \in$ L

Let's try again: { a^nb^n | n >= 0 }

Context-Free Grammars

- Σ - terminals
- Symbols with recursive defs - nonterminals
- Rules are of form
  N ::= v
  v is sequence of terminals and non-terminals
- Derivation starts from a starting symbol
- Replaces non-terminals with right hand side
  - terminals and
  - non-terminals

Context Free Grammars

- S ::= "" | a S b (for $a^n b^n$)

Example of a derivation

S => => aaabbb

Corresponding derivation tree:

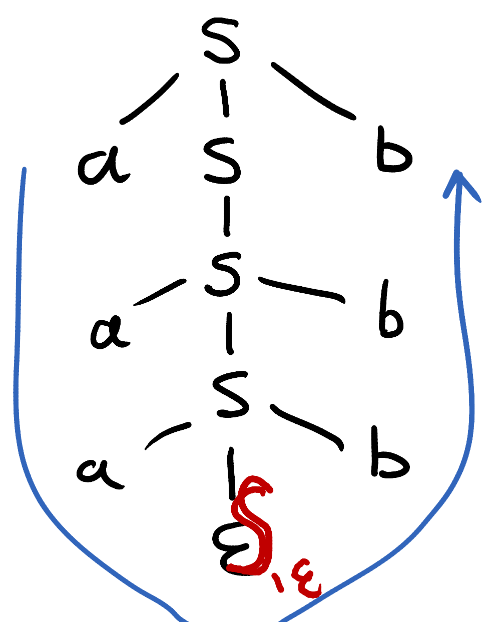
Context Free Grammars

- S ::= "" | a S b (for a^nb^n)

Example of a derivation

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

Corresponding derivation tree: leaves give result



aaabbb

S → a
S → Pa
P → bQ
Q → bSa

S → Pa → bQa
→ bbSaa
→ bbaaa

Grammars for Natural Language

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

→ can also be used to automatically generate essays

Exercise: draw the derivation tree for:

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

Balanced Parentheses Grammar

- Sequence of balanced parentheses

→ ((()) ()) - balanced

( )) (( ) - not balanced

(((( ))))

(((( ))( )))

S → ε | S (S) S

S → ....

S → .P.

P → ...

S → S (S) S → (S) → (S(S)S) → ((S) S)

Exercise: give the grammar and example derivation

→ ((S(S)S) S(S)S) → ((( )) ( ))

Balanced Parantheses Grammar

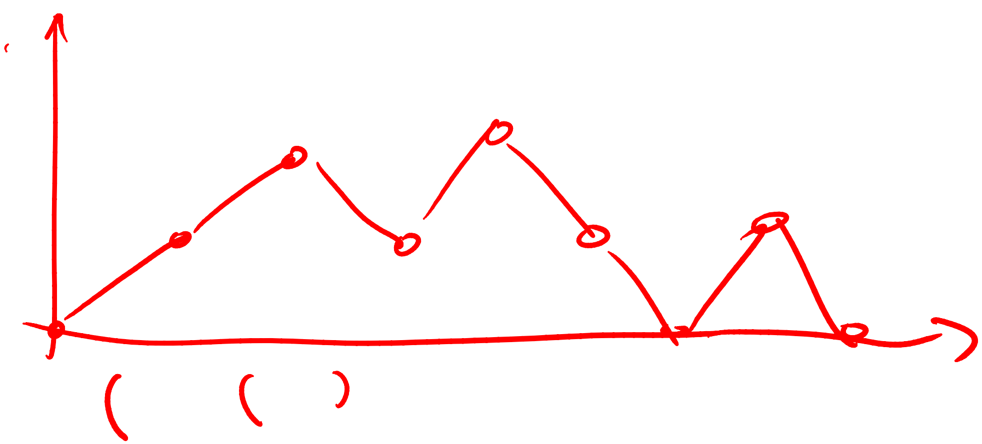
$S ::=$

$\quad \mid \varepsilon$

$\quad \mid (S)S$

$S \rightarrow (S)S \rightarrow (\varepsilon)S \rightarrow ()S \rightarrow ()(S)S$

$\rightarrow ()()$



(()())()

Proving Grammar Defines a Language

Grammar G: S ::= "" | **(**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 "", and "" $\in L_b$, 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 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="" 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.

Remember While Syntax

$\underbrace{A^*}_{A_1}$

$A_1 \to \varepsilon \mid A A_1$

program ::= statmt*

statmt ::= println(stringConst , ident)

| ident = expr

| **if** (expr) statmt (else statmt)?

| **while** (expr) statmt

| { statmt* }

expr ::= intLiteral | ident

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

| ! expr | - expr

Eliminating Additional Notation

- Grouping alternatives

  s ::= P | Q   instead of   s ::= P

  s ::= Q

- Parenthesis notation

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

  A

- Kleene star within grammars

  { statmt* } A_1

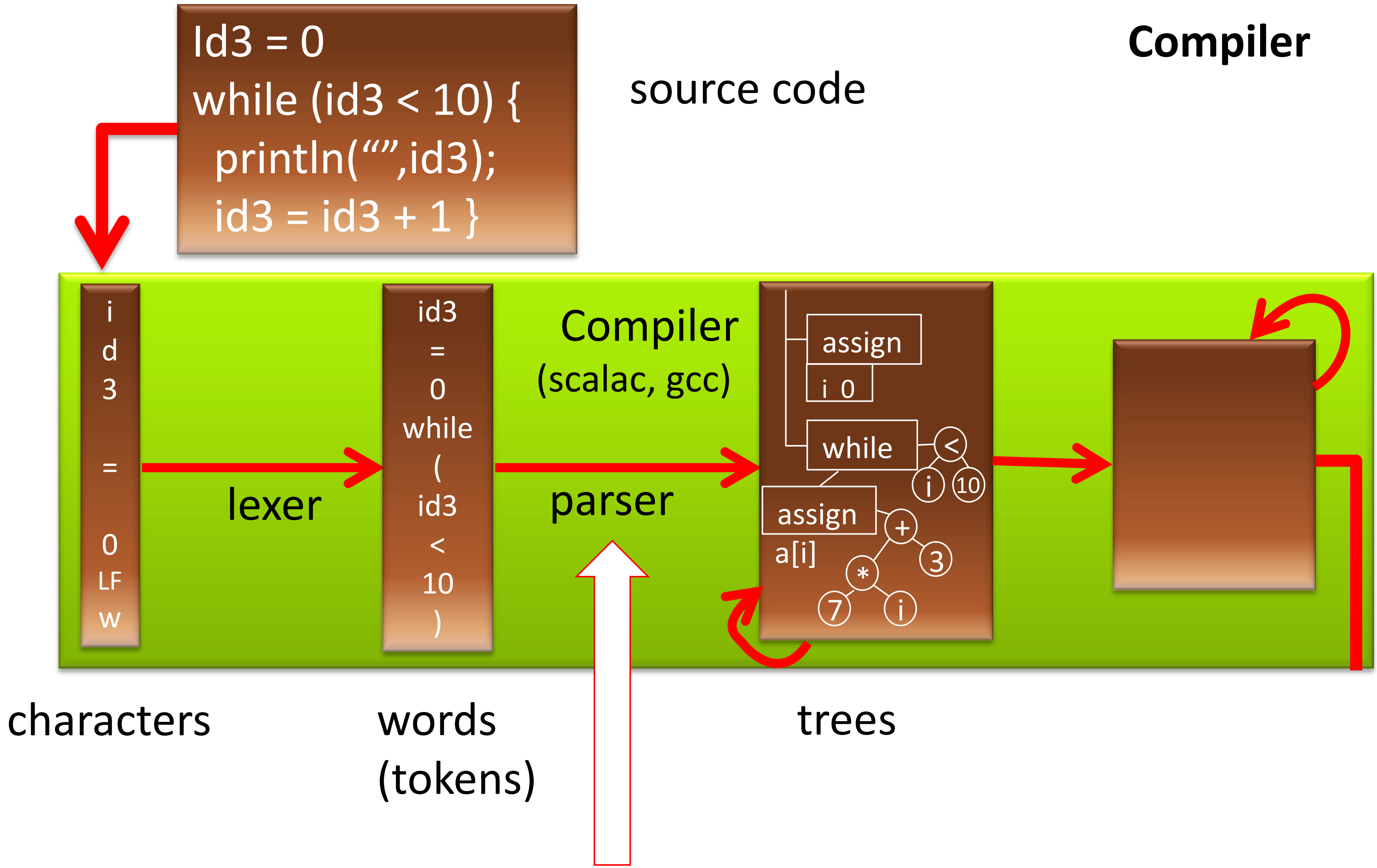
  $A_1 \to \varepsilon \mid statmt\ A_1$

- Optional parts

  if (expr) statmt (else statmt)?

  B

  $B \to \varepsilon \mid$ else statmt


Compiler
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

Recursive Descent Parsing

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

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)  }
```

First Symbols for Non-terminals

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

- Consider a grammar G and non-terminal N

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

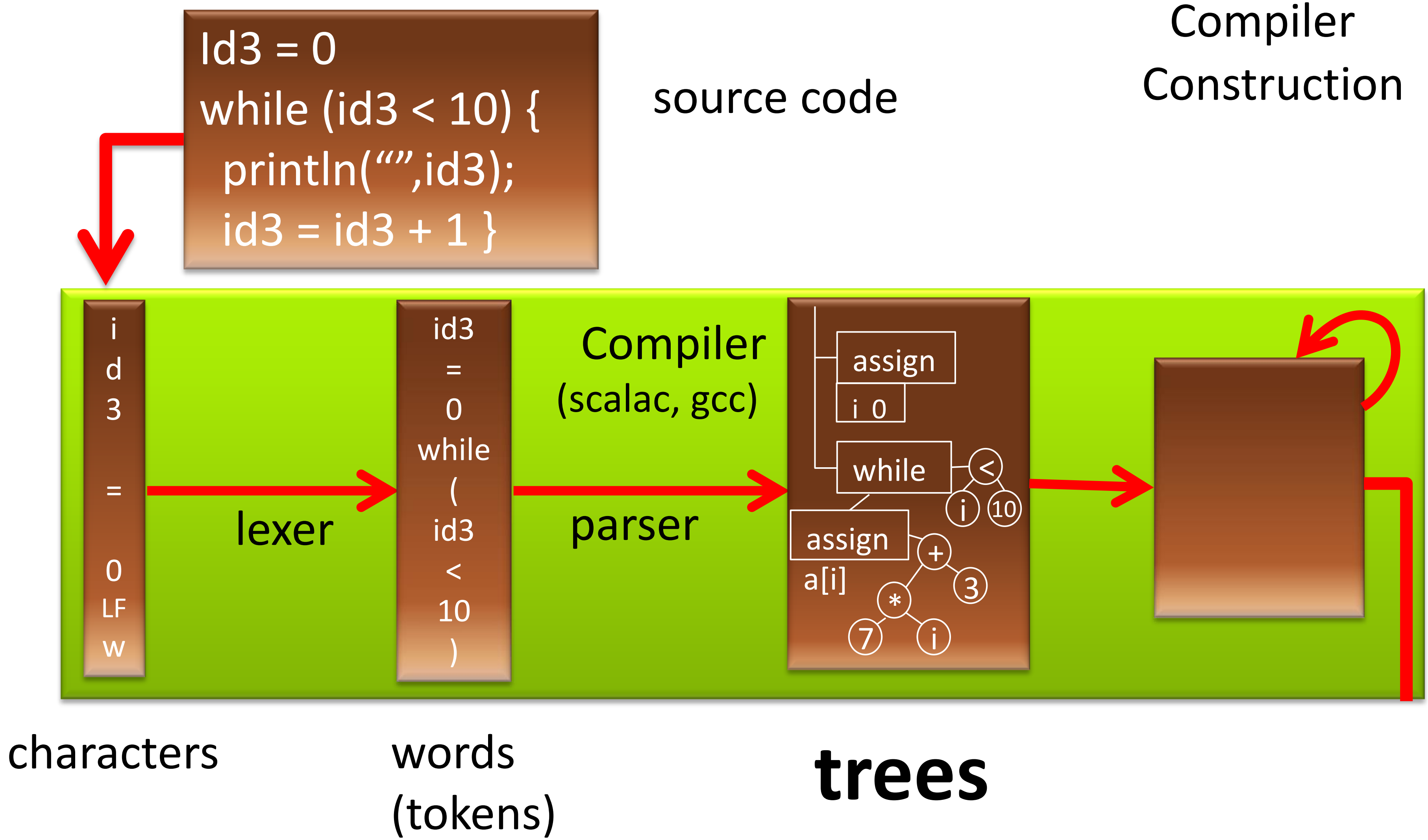
e.g. L(statmt) – all statements of while language

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

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

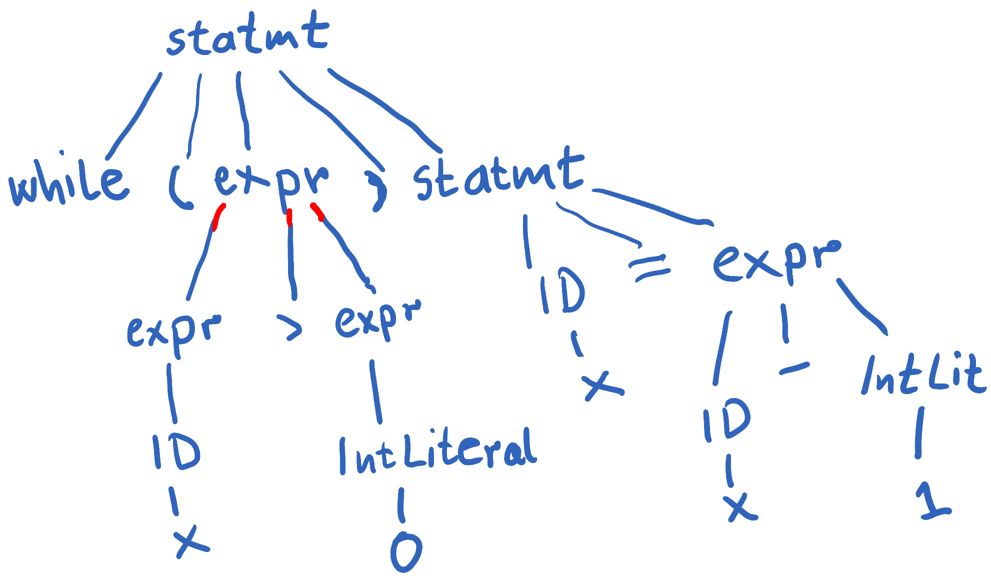
(we will see how to compute first in general)

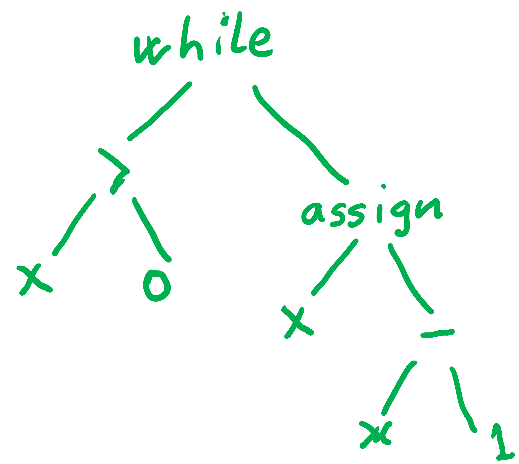
Compiler Construction



Parse Tree vs Abstract Syntax Tree (AST)

while (x > 0) x = x - 1





Pretty printer: takes abstract syntax tree (AST) and outputs the leaves of one possible (concrete) parse tree.

$$\text{parse}(\text{prettyPrint}(\text{ast})) \approx \text{ast}$$

Parse Tree vs Abstract Syntax Tree (AST)

- Each node in parse tree has children corresponding precisely to right-hand side of grammar rules
- Nodes in abstract syntax tree contain only useful information and usually omit e.g. the punctuation signs

Abstract Syntax Trees for Statements

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

```
abstract class Statmt
case class PrintlnS(msg : String, var : Identifier) extends Statmt
case class Assignment(left : Identifier, right : Expr) extends Statmt
case class If(cond : Expr, trueBr : Statmt,
                        falseBr : Option[Statmt]) extends Statmt
case class While(cond : Expr, body : Expr) extends Statmt
case class Block(sts : List[Statmt]) extends Statmt
```

Abstract Syntax Trees for Statements

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

```
abstract class Statmt
case class PrintlnS(msg : String, var : Identifier) extends Statmt
case class Assignment(left : Identifier, right : Expr) extends Statmt
case class If(cond : Expr, trueBr : Statmt,
              falseBr : Option[Statmt]) extends Statmt
case class While(cond : Expr, body : Statmt) extends Statmt
case class Block(sts : List[Statmt]) extends Statmt
```

Our Parser Produced Nothing ☹

```
def skip(t : Token) : unit = if (lexer.token == t) lexer.next
  else error("Expected"+ t)
// statmt ::=
def statmt : unit = {
  // 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
```

Parser Returning a Tree ☺

```
def expect(t : Token) : Token = if (lexer.token == t) { lexer.next;t}
  else error("Expected"+ t)
// statmt ::=
def statmt : Statmt = {
  // println ( stringConst , ident )
  if (lexer.token == Println) { lexer.next;
    skip(openParen); val s = getString(expect(stringConst));
    skip(comma);
    val id = getIdent(expect(identifier)); skip(closedParen)
    PrintlnS(s, id)
 // | ident = expr
  } else if (lexer.token.class == Ident) { val lhs = getIdent(lexer.token)
    lexer.next;
    skip(equality); val e = expr
    Assignment(lhs, e)
```

Constructing Tree for 'if'

```
def expr : Expr = { ... }
// statmt ::=
def statmt : Statmt = {
  ...
// if ( expr ) statmt (else statmt)?
// case class If(cond : Expr, trueBr: Statmt, falseBr: Option[Statmt])
  } else if (lexer.token == ifKeyword) { lexer.next;
    skip(openParen); val c = expr; skip(closedParen);
    val trueBr = statmt
    val elseBr = if (lexer.token == elseKeyword) {
      lexer.next; Some(statmt) } else None
    If(c, trueBr, elseBr)   // made a tree node ☺
  }
```

Task: Constructing Tree for 'while'

```
def expr : Expr = { … }
// statmt ::=
def statmt : Statmt = {
  …
// while ( expr ) statmt
// case class While(cond : Expr, body : Expr) extends Statmt
} else if (lexer.token == WhileKeyword) {




} else
```

Here each alternative started with different token

statmt ::=

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

What if this is not the case?

Left Factoring Example: Function Calls

statmt ::=

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

foo = 42 + x
foo (u , v)

code to parse the grammar:

```
} else if (lexer.token.class == Ident) {
   ???
}
```

Left Factoring Example: Function Calls

statmt ::=

println (stringConst , ident)

| ident assignmentOrCall

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

| while (expr) statmt

| { statmt* }

assignmentOrCall ::= “=“ expr | (expr (, expr)*)

code to parse the grammar:

```
} else if (lexer.token.class == Ident) {
    val id = getIdentifier(lexer.token); lexer.next
    assignmentOrCall(id)
}            // Factoring pulls common parts from alternatives
```

Beyond Statements: Parsing Expressions

While Language with Simple Expressions

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

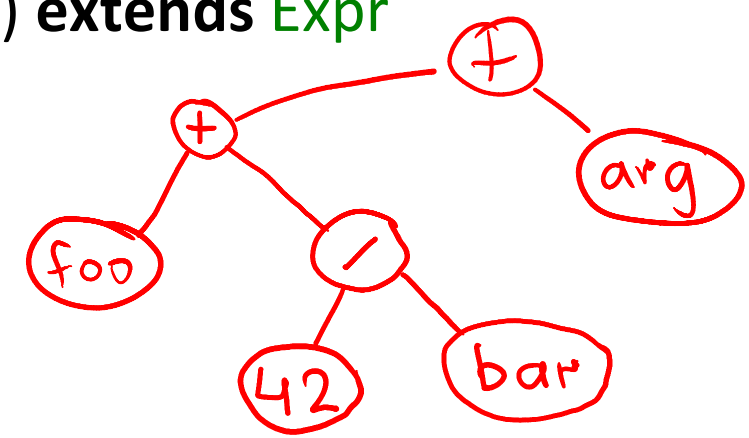
```
expr ::= intLiteral | ident
       | expr ( + | / ) expr
```

Abstract Syntax Trees for Expressions

```
expr ::= intLiteral | ident
       | expr + expr | expr / expr
```

```scala
abstract class Expr
case class IntLiteral(x : Int) extends Expr
case class Variable(id : Identifier) extends Expr
case class Plus(e1 : Expr, e2 : Expr) extends Expr
case class Divide(e1 : Expr, e2 : Expr) extends Expr
```

foo + 42 / bar + arg



Parser That Follows the Grammar?

```
expr ::= intLiteral | ident
       | expr + expr | expr / expr
```

input:

(foo + 42 / bar + arg)

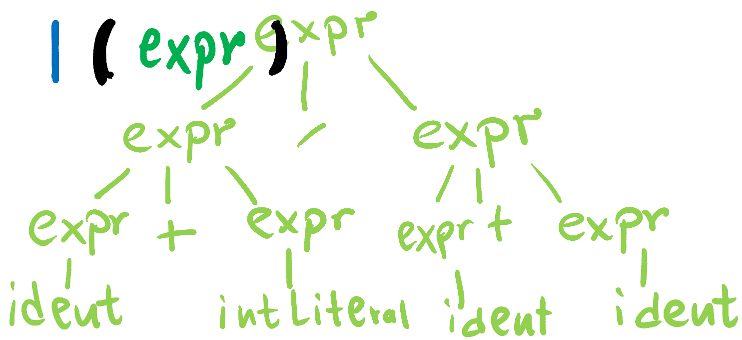
```
def expr : Expr = {
  if (??) IntLiteral(getInt(lexer.token))
  else if (??) Variable(getIdent(lexer.token))
  else if (??) {
    val e1 = expr; val op = lexer.token; val e2 = expr
    op match Plus {
     case PlusToken => Plus(e1, e2)
     case DividesToken => Divides(e1, e2)
    } }
```

When should parser enter the recursive case?!

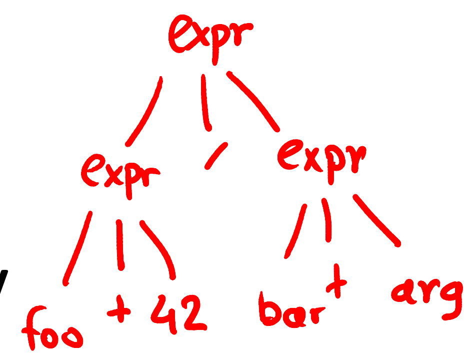
Ambiguous Grammars

expr ::= intLiteral | ident
| expr + expr | expr / expr

| (expr)

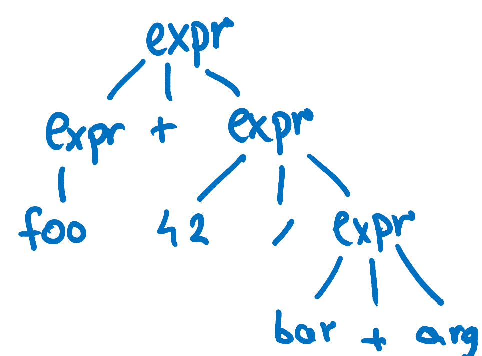


foo + 42 / bar + arg



Each node in parse tree is given by one grammar alternative.

Ambiguous grammar: if some token sequence has multiple parse trees (then it is has multiple abstract trees).

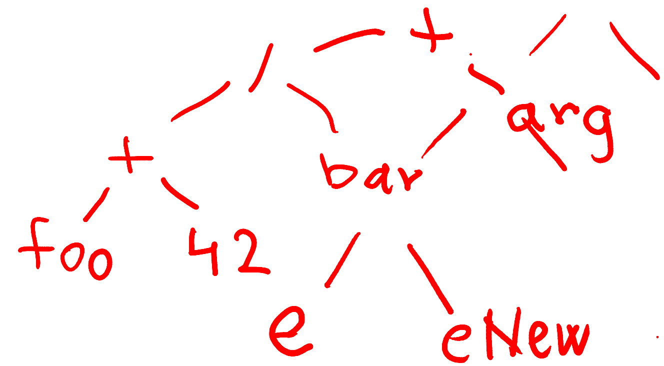


An attempt to rewrite the grammar

```
expr ::= simpleExpr (( + | / ) simpleExpr)*
simpleExpr ::= intLiteral | ident
```

```
def simpleExpr : Expr = { ... }
def expr : Expr = {
  var e = simpleExpr
  while (lexer.token == PlusToken ||
         lexer.token == DividesToken)) {
    val op = lexer.token
    val eNew = simpleExpr
    op match {
      case TokenPlus => { e = Plus(e, eNew) }
      case TokenDiv => { e = Divide(e, eNew) }
    }
  }
  e }
```

foo + 42 / bar + arg



Not ambiguous, but gives wrong tree.