Exercise: Build Lexical Analyzer Part

For these two tokens, using longest match,

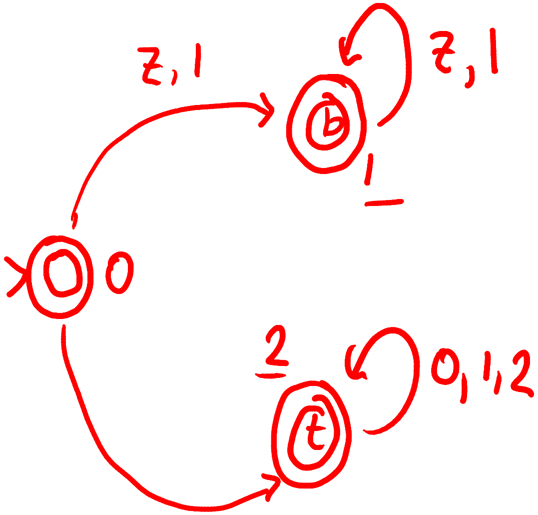
where first has the priority:

binaryToken ::= (**z**|1)*

ternaryToken ::= (0|1|2)*

$(z|1)^* \mid (0|1|2)^*$

1111z1021z1 →



$$\{0\} \xrightarrow{1} \{1,2\} \xrightarrow{1} \{1,2\} \ldots \xrightarrow{1} \{1,2\} \xrightarrow{z} \{1\} \xrightarrow[]{\overset{0,1,2}{1}} \{1\} \xrightarrow{0} \emptyset$$

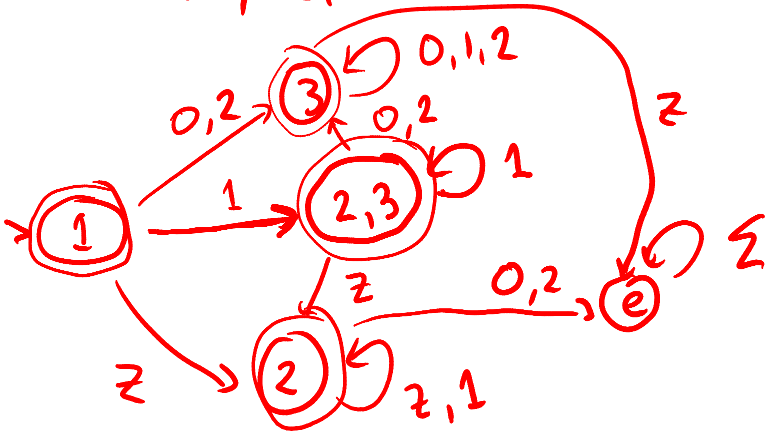
Lexical Analyzer

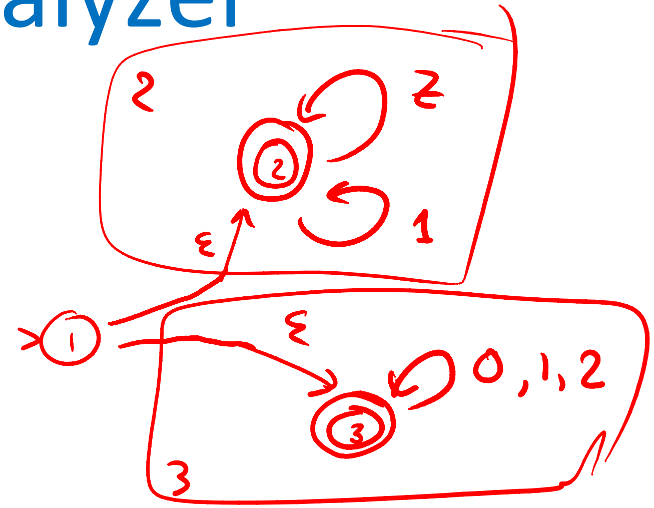
1) binaryToken ::= $(\mathbf{z}|1)^*$

2) ternaryToken ::= $(0|1|2)^*$

$\underbrace{1111z1}_{\text{binary}}\underbrace{021}_{\text{tern}}\underbrace{z1}_{\text{binary}} \rightarrow$

$\underbrace{1\;1\;1\;1\;1\;5}_{\text{binary (priority)}}$





(binaryToken | ternaryToken)*

Σ = {0,1,2,z}

Exercise: Realistic Integer Literals

- Integer literals are in three forms in Scala: decimal, hexadecimal and octal. The compiler discriminates different classes from their beginning.
  - Decimal integers are started with a non-zero digit.
  - Hexadecimal numbers begin with 0x or 0X and may contain the digits from 0 through 9 as well as upper or lowercase digits A to F afterwards.
  - If the integer number starts with zero, it is in octal representation so it can contain only digits 0 through 7.
  - l or L at the end of the literal shows the number is Long.
- Draw a single DFA that accepts all the allowable integer literals.
- Write the corresponding regular expression.

Exercise

- Let L be the language of strings A = {<, =} defined by regexp (<|=| <====*), that is, L contains <,=, and words $<=^n$ for n>2.
- Construct a DFA that accepts L
- Describe how the lexical analyzer will tokenize the following inputs.
  1) <=====
  2) ==<==<==<==<==
  3) <=====<

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

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

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: $\{\ a^nb^n \mid n >= 0\ \}$

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:

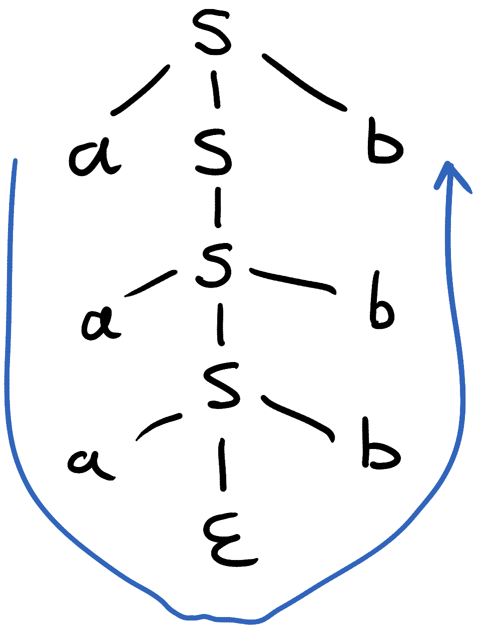
Context Free Grammars

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

Example of a derivation

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

Corresponding derivation tree: leaves give result



aaabbb

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.

Balanced Parentheses Grammar

- Sequence of balanced parentheses

  ((()) ()) - balanced

  ( )) (( ) - not balanced

Exercise: give the grammar and example derivation

Balanced Parantheses Grammar

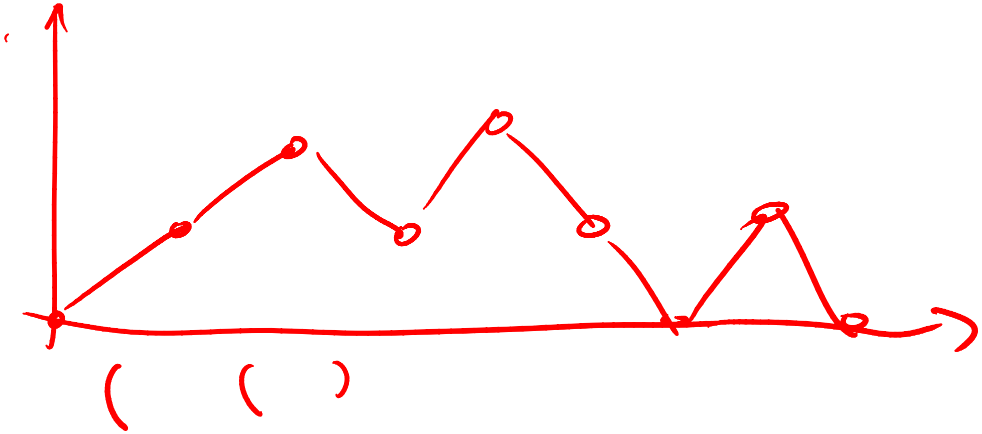
$S ::=$

$\quad | \quad \varepsilon$

$\quad | \quad (S)S$

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

$\rightarrow ()()$



$(()())()$

Remember While Syntax

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

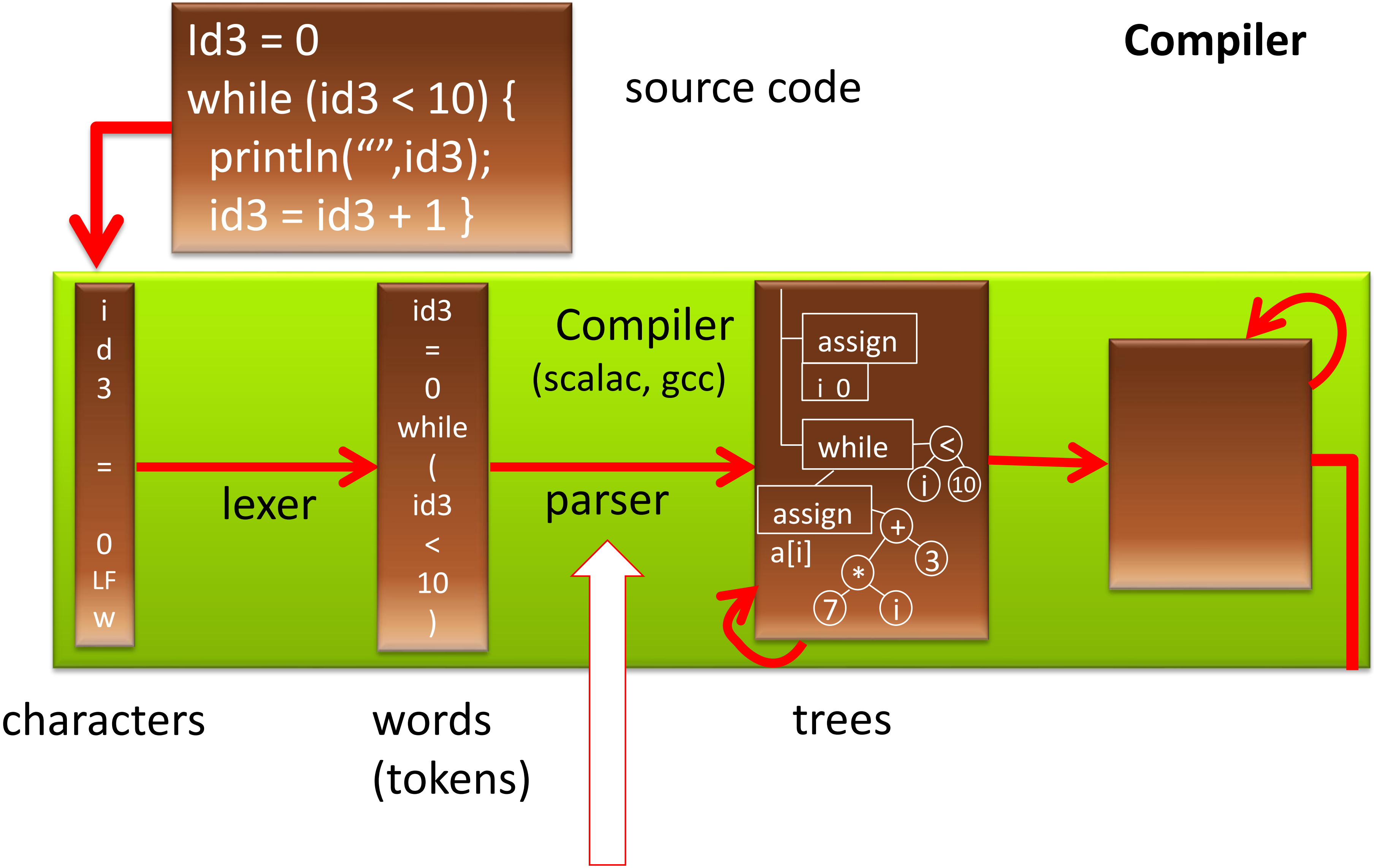
- Kleene star within grammars

  { statmt* }

- Optional parts

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

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

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 }

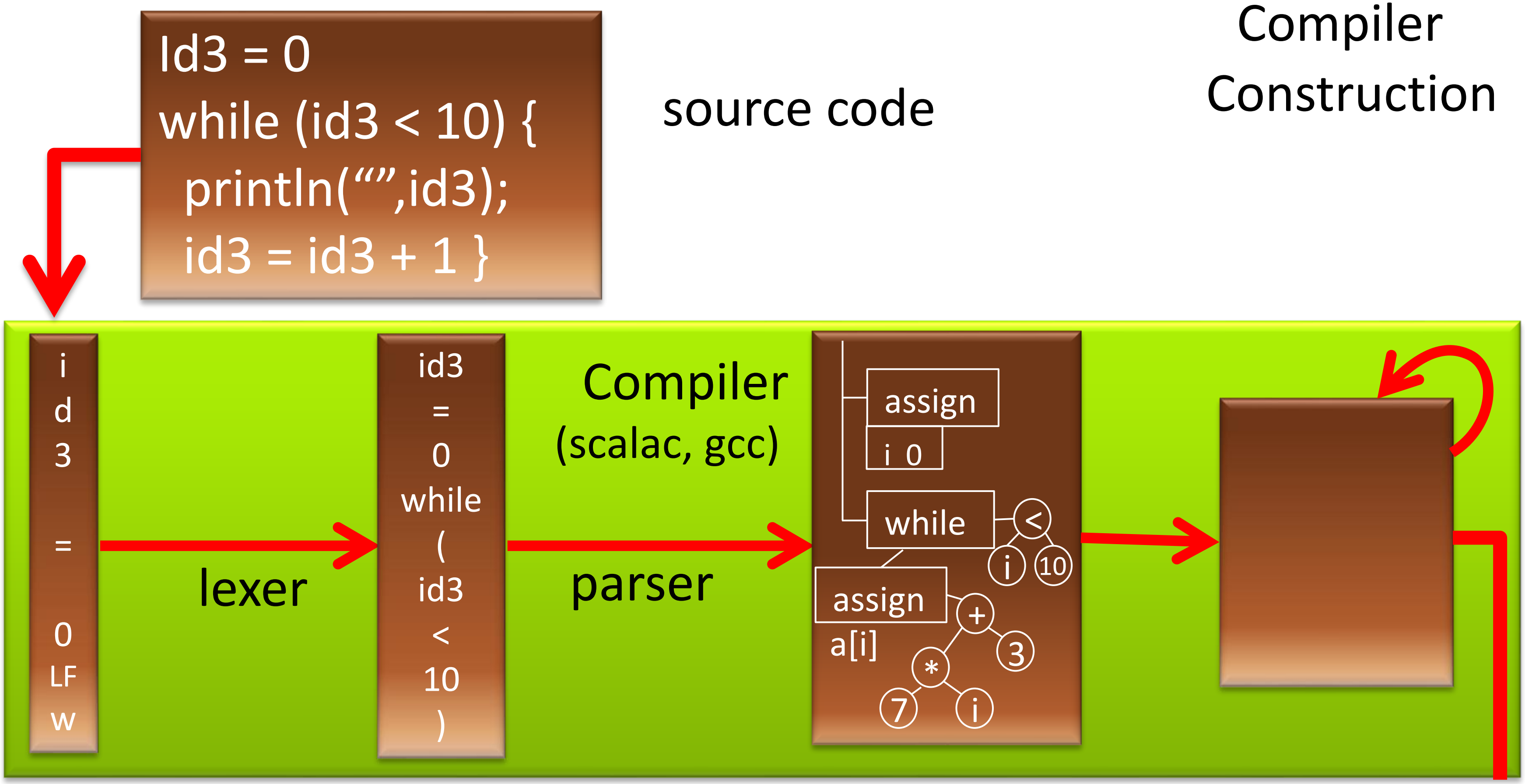
e.g. L(statmt) – all statements of while language

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



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

source code

Compiler
(scalac, gcc)

characters

words
(tokens)

trees

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
```

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 Nothing
    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
```