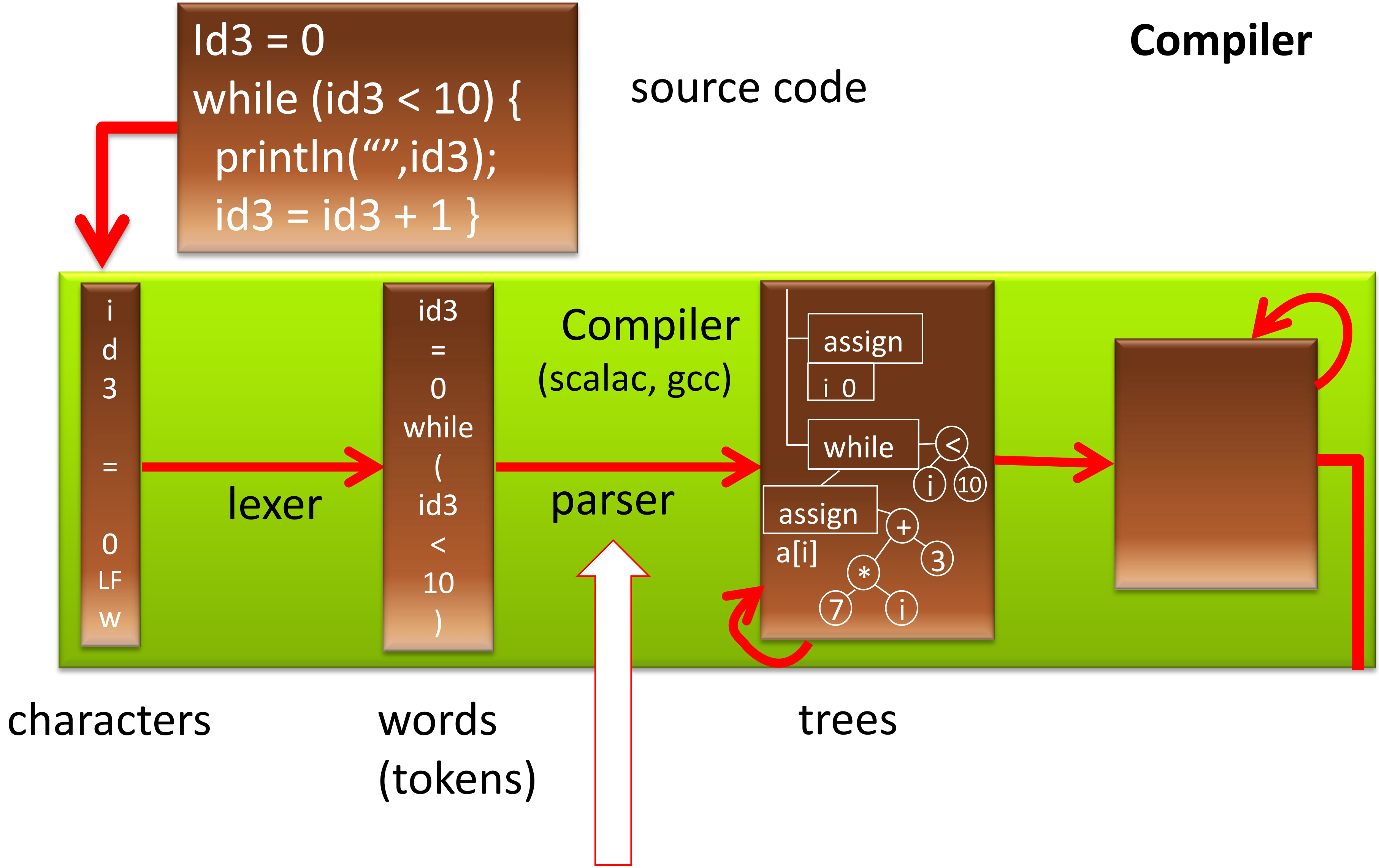
Compiler



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

```
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 }

L(statmt) – all statements of while language

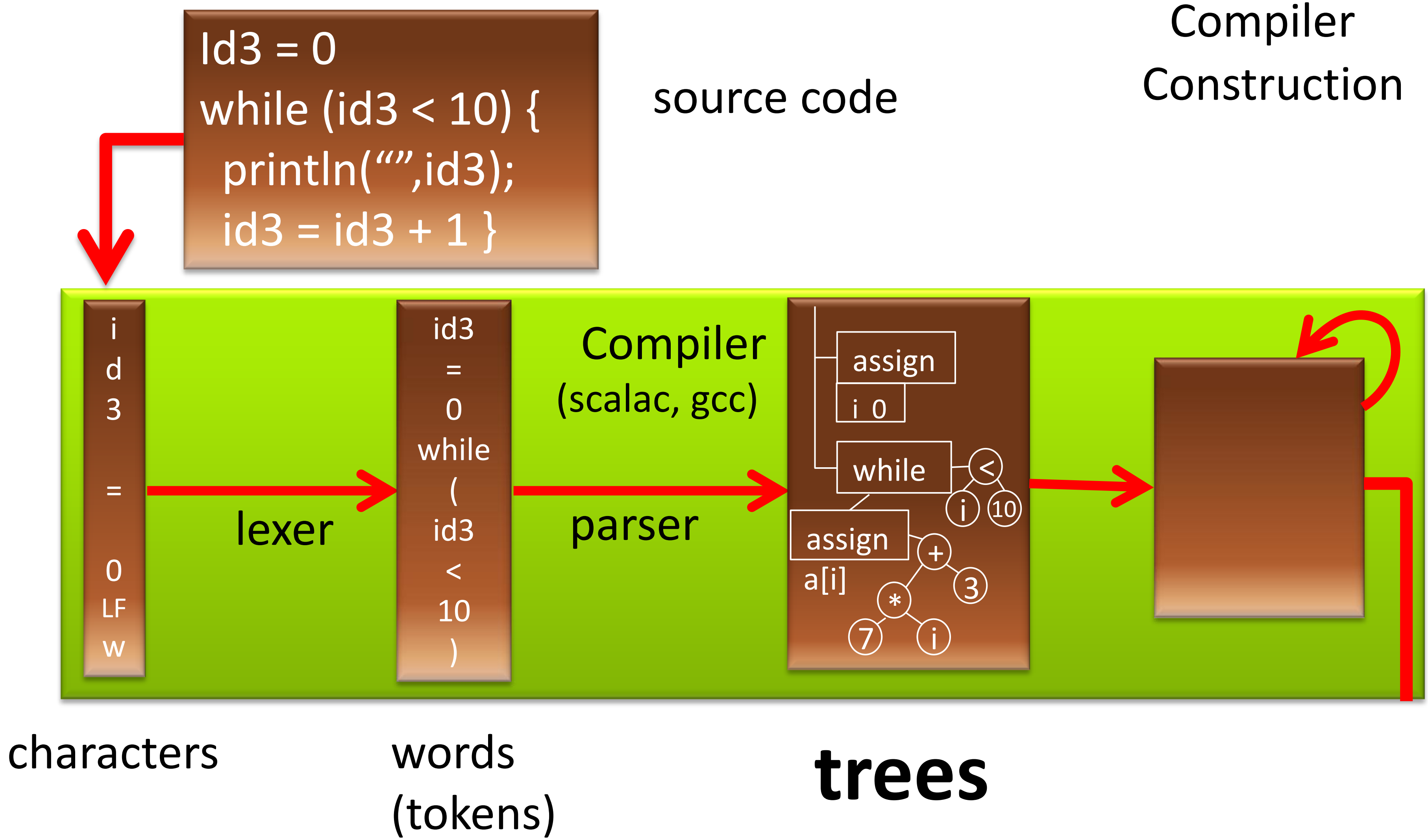
first(N) = { a | aw in $L_G(N)$, a – terminal,

w – string of terminals}

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

(we will see how to compute first in general)

Compiler Construction



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 = {
  …
// | while ( expr ) 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)
```

Constructing Tree for ‘while’

```
def expr : Expr = { … }
// statmt ::=
def statmt : Statmt = {
  …
// if ( expr ) statmt (else statmt)?
// case class While(cond : Expr, body : Expr) extends Statmt
```

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
```

Parsing Statements Worked Nicely

Let's look at expressions

Simplified Expressions in While

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

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

Trees for Expressions

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

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
```

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 use the recursive case?!

Parse Tree vs Abstract Syntax Tree

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

foo + 42 / bar + arg

Node in parse tree is given by one grammar alternative.

Ambiguous grammar: if some string has multiple parse trees (then it is has multiple corresponding abstract trees)

AST is tree as Scala value

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

Layer the expression grammar to express priorities

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

```
def expr : Expr = {
  var e = term
  while (lexer.token == PlusToken) {
    lexer.next
    e = Plus(e, term)
  }
  e
}
```

Decompose first by the least-priority operator (+)

Using recursion instead of *

expr ::= term (+ term)* ⟹ expr ::= term (+ expr)?

```
def expr : Expr = {
  val e = term
  if (lexer.token == PlusToken) {
    lexer.next
    Plus(e, expr)
  } else e
}
def term : Expr = {
  val e = simpleExpr
  if (lexer.token == DivideToken) {
    lexer.next
    Divide(e, term)
  } else e
}
```

What we have seen so far

- Automatically constructing lexers
- Manual recursive descent parser
  - parser for the while language
- Constructing syntax trees
- Refactoring grammars
- Now we can build simple parsers
  - When does it work – LL(1) grammars
  - How can we automate it?