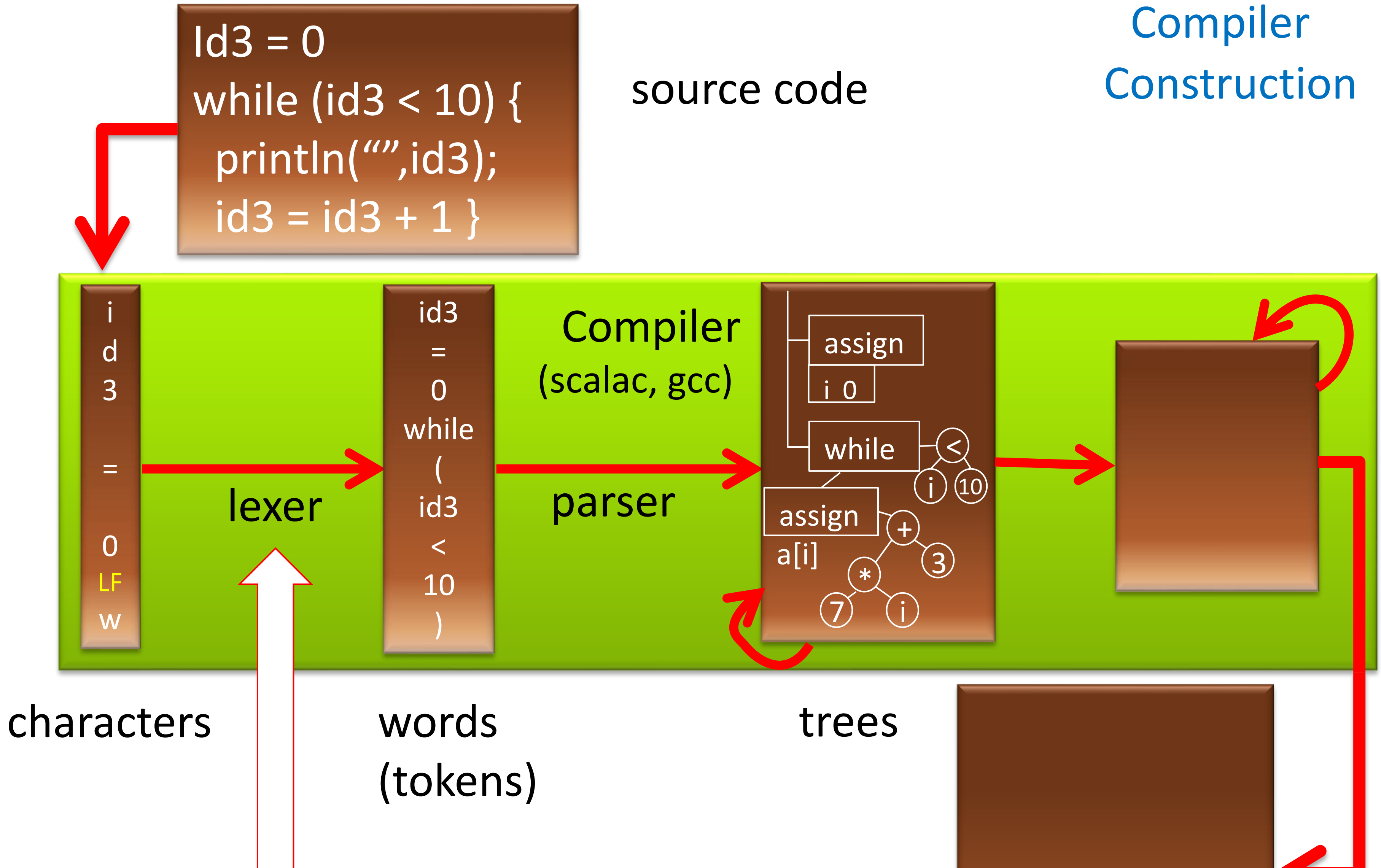
Compiler Construction



Lexical analyzer (lexer) is specified using regular expressions. Groups characters into tokens and classifies them into token classes.

Lexical Analysis Summary

- lexical analyzer maps a stream of characters into a stream of tokens
  - while doing that, it typically needs only bounded memory
- we can specify tokens for a lexical analyzers using regular expressions
- it is not difficult to construct a lexical analyzer manually
  - we give an example
  - for manually constructed analyzers, we often use the first character to decide on token class; a notion first(L) = { a | aw in L }
- we follow the longest match rule: lexical analyzer should eagerly accept the longest token that it can recognize from the current point
- it is possible to automate the construction of lexical analyzers; the starting point is conversion of regular expressions to automata
  - tools that automate this construction are part of compiler-compilers, such as JavaCC described in the Tiger book
  - automated construction of lexical analyzers from regular expressions is an example of compilation for a *domain-specific language*

Formal Languages vs Scala

Formal language theory:

- A – alphabet
- A* - words over A
- $w_1 \cdot w_2$ or $w_1\, w_2$
- ε – empty word
- $c \in A \rightarrow c \in A^*$
- $|w|$ - word length
- $w_{p..q} = w_{(p)}w_{(p+1)} \ldots w_{(q-1)}$
  $w = w_{(0)}w_{(1)} \ldots w_{(|w|-1)}$
- $L \subseteq A^*$ - language

Scala representation:

- A – type
- List[A] (or Seq[A]...)
- w1 ::: w2
- List()
- if c:A then List(c):List[A]
- w.length
- w.slice(p,q)
  w(i)
- L : List[List[A]] (for finite L)

Formal Languages vs Scala

Formal language theory:

$L_1 \subseteq A^*$, $L_2 \subseteq A^*$

$L_1 \cdot L_2 = \{u_1 u_2 \mid u_1 \in L_1 , u_2 \in L_2 \}$

Scala (for finite languages)

```
type Lang[A] = List[List[A]]
def product[A](L1 : Lang[A],
               L2 : Lang[A]) : Lang[A] =
  for (w1 <- L1; w2 <- L2)
   yield (w1 ::: w2)
```

{ Peter, Paul, Mary} · { France, Germany} =
{PeterFrance, PeterGermany,
PaulFrance, PaulGermany,
MaryFrance,MaryGermany}

```
val p = product(List("Peter".toList, "Paul".toList, "Mary".toList),
                List("France".toList, "Germany".toList))
```

L^n

$L^* = \bigcup_{n \geq 0} L^n$

Fact about Indexing Concatenation

Concatenation of w and v has these letter:

$$w_{(0)} \dots w_{(|w|-1)} \; v_{(0)} \dots v_{(|v|-1)}$$

$(wv)_{(i)} = w_{(i)}$, if $i < |w|$

$(wv)_{(i)} = v_{(i-|w|)}$, if $i \geq |w|$

Star of a Language. Exercise with Proof

$L^* = \{ w_1 \dots w_n \mid n \geq 0, w_1 \dots w_n \in L \}$

$= \cup_n L^n$ where $L^{n+1} = L\, L^n$, $L^0 = \{\varepsilon\}$. Obviously also $L^{n+1} = L^n\, L$

Exercise. Show that $\{a,ab\}^* = S$ where

$S = \{w \in \{a,b\}^* \mid \forall 0 \leq i < |w|.$ if $w_{(i)} = b$ then: $i > 0$ and $w_{(i-1)} = a\}$

Proof. We show $\{a,ab\}^* \subseteq S$ and $S \subseteq \{a,ab\}^*$. $\varepsilon \in S$ $|\varepsilon| = 0$

1) $\{a,ab\}^* \subseteq S$: We show that for all n, $\{a,ab\}^n \subseteq S$, by induction on n

- Base case, $n=0$. $\{a,ab\}^0 = \{\varepsilon\}$, so $i < |w|$ is always false and '->' is true.
- Suppose $\{a,ab\}^n \subseteq S$. Showing $\{a,ab\}^{n+1} \subseteq S$. Let $w \in \{a,ab\}^{n+1}$.

Then $w = vw'$ where $w' \in \{a,ab\}^n$, $v \in \{a,ab\}$. Let $i < |w|$ and $w_{(i)} = b$.
$v_{(0)} = a$, so $w_{(0)} = a$ and thus $w_{(0)} \,!= b$. Therefore $i > 0$. Two cases:
1.1) $v = a$. Then $w_{(i)} = w'_{(i-1)}$. By I.H. $i-1 > 0$ and $w'_{(i-2)} = a$. Thus $w_{(i-1)} = a$.
1.2) $v = ab$. If $i = 1$, then $w_{(i-1)} = w_{(0)} = a$, as needed. Else, $i > 1$ so
$w'_{(i-2)} = b$ and by I.H. $w'_{(i-3)} = a$. Thus $w_{(i-1)} = (vw')_{(i-1)} = w'_{(i-3)} = a$.

Proof Continued

$S = \{w \in \{a,b\}^* \mid \forall 0 \le i < |w|.\ \text{if } w_{(i)} = b \text{ then: } i > 0 \text{ and } w_{(i-1)} = a\}$

For the second direction, we first prove:

(*) If $w \in S$ and $w = w'v$ then $w' \in S$.

Proof. Let $i < |w'|$, $w'_{(i)} = b$. Then $w_{(i)} = b$ so $w_{(i-1)} = a$ and thus $w'_{(i-1)} = a$.

2) $S \subseteq \{a,ab\}^*$. We prove, by induction on n, that for all n,

for all w, if $w \in S$ and $n = |w|$ then $w \in \{a,ab\}^*$.

- Base case: n=0. Then w is empty string and thus in $\{a,ab\}^*$.

- Let n>0. Suppose property holds for all k < n. Let $w \in S$, $|w| = n$.

There are two cases, depending on the last letter of w.

2.1) $w = w'a$. Then $w' \in S$ by **(*)**, so by IH $w' \in \{a,ab\}^*$, so $w \in \{a,ab\}^*$.

2.2) $w = vb$. By $w \in S$, $w_{(|w|-2)} = a$, so $w = w'ab$. By **(*)**, $w' \in S$, by IH $w' \in \{a,ab\}^*$, so $w \in \{a,ab\}^*$.

In any case, $w \in \{a,ab\}^*$. We proved the entire equality.

Regular Expressions

- One way to denote (often infinite) languages
- Regular expression is an expression built from:
  - empty language $\varnothing$
  - $\{\varepsilon\}$, denoted just ε
  - $\{a\}$ for a in Σ, denoted simply by a
  - union, denoted | or, sometimes, +
  - concatenation, as multiplication (dot), or omitted
  - Kleene star * (repetition) L^* $L_1 \cdot L_2$
- E.g. identifiers: letter (letter | digit)*

  (letter,digit are shorthands from before)

Kleene (from Wikipedia)

Stephen Cole Kleene
(January 5, 1909, Hartford, Connecticut, United States – January 25, 1994, Madison, Wisconsin) was an American mathematician who helped lay the foundations for theoretical computer science. One of many distinguished students of Alonzo Church, Kleene, along with Alan Turing, Emil Post, and others, is best known as a founder of the branch of mathematical logic known as recursion theory. Kleene's work grounds the study of which functions are computable. A number of mathematical concepts are named after him: Kleene hierarchy, Kleene algebra, the Kleene star (**Kleene closure**), Kleene's recursion theorem and the Kleene fixpoint theorem. He also **invented** **regular expressions**, and was a leading American advocate of mathematical intuitionism.

These RegExp extensions preserve definable languages. Why?

- [a..z] = a|b|...|z (use ASCII ordering)

  (also other shorthands for finite languages)
- e? (optional expression) $L(e?) = L(e) \cup \{\varepsilon\}$ $(e|\varepsilon)$
- e+ (repeat at least once) $e(e|\varepsilon)^*$ ee^*
- complement: !e (do not match) $\Sigma = \{a,b,c\}$!b $(a|c)$
- intersection: e1 & e1 (match both) $(aa)^* \,\&\, (aaa)^*$
- $e^{k..*}$, $e^{p..q} = e^p(e|\varepsilon)^{q-p}$

  $e^k e^*$ $eeee^*$ $e^{3..*}$ $e^{2..5} = \overbrace{ee}^{e^2}(eee|ee|e|\varepsilon)$ aaa,aaa $\boxed{e^2(e|\varepsilon)^3}$
- quantification: can prove previous theorem automatically!

$\{a,ab\}^* = \{w \in \{a,b\}^* \mid \forall i.\ w_{(i)} = b$ --> $i > 0$ & $w_{(i-1)} = a\}$

http://www.brics.dk/mona/

While Language – Example Program

```
num = 13;
while (num > 1) {
  println("num = ", num);
  if (num % 2 == 0) {
    num = num / 2;
  } else {
    num = 3 * num + 1;
  }
}
```

Tokens (Words) of the *While* Language

regular expressions

Ident ::=

letter (letter | digit)*

integerConst ::=

digit digit*

stringConst ::=

“ AnySymbolExceptQuote* “

keywords

if else while println

special symbols

( ) && < == + - * / % ! - { } ; ,

letter ::= a | b | c | … | z | A | B | C | … | Z

digit ::= 0 | 1 | … | 8 | 9

Manually Constructing Lexers by example

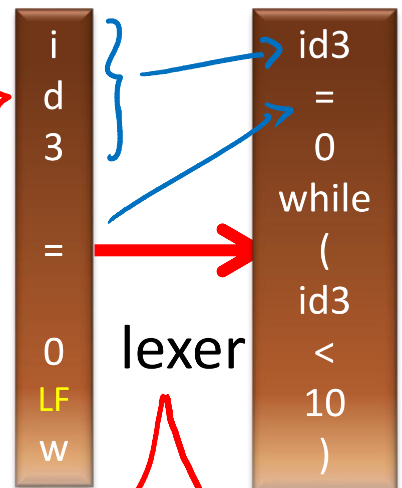
Lexer input and Output

Stream of Char-s
(lazy List[Char])

```
class CharStream(fileName : String){
 val file = new BufferedReader(
    new FileReader(fileName))
  var current : Char = ' '
  var eof : Boolean = false

  def next = {
   if (eof)
    throw EndOfInput("reading" + file)
   val c = file.read()
   eof = (c == -1)
   current = c.asInstanceOf[Char]
  }

  next // init first char
}
```




```
class Lexer(ch : CharStream) {
  var current : Token
  def next : Unit = {
   lexer code goes here
  }
}
```

Stream of Token-s

```
sealed abstract class Token
case class ID(content : String) // "id3"
          extends Token
case class IntConst(value : Int) // 10
          extends Token
case class AssignEQ() '='
          extends Token
case class CompareEQ // '=='
          extends Token
case class MUL() extends Token // '*'
case  class PLUS() extends Token // +
case clas LEQ extends Token // '<='
case class OPAREN extends Token //(
case class CPAREN extends Token //)
...
case class IF extends Token // 'if'
case class WHILE extends Token
case class EOF extends Token
          // End Of File
```

Identifiers and Keywords

'a' <= ch.current <= 'z'

```
if (isLetter) {
  b = new StringBuffer
  while (isLetter || isDigit) {
    b.append(ch.current)
    ch.next
  }
 keywords.lookup(b.toString) {
  case None => token=ID(b.toString)
  case Some(kw) => token=kw
 }
}
```

regular expression for identifiers:

letter (letter|digit)*

Keywords look like identifiers, but are simply indicated as keywords in language definition

A constant Map from strings to keyword tokens

if not in map, then it is ordinary identifier

Integer Constants and Their Value

regular expression for integers:

digit digit*

```
if (isDigit) {
  k = 0
  while (isDigit) {
    k = 10*k + toDigit(ch.current)
    ch.next
  }
  token = IntConst(k)
}
```

Deciding which Token

- How do we know when we are supposed to analyze string, when integer sequence etc?
- Manual construction: use **lookahead** (next symbol in stream) to decide on token class
- compute first(e) - symbols with which e can start
- check in which first(e) current token is
- If L is a language, then

$$\text{first}(L) = \{a \mid \exists v.\ a\,v \in L\}$$

first of a regexp

- Given regular expression e, how to compute first(e)?
  - use automata (we will see this next)
  - rules that directly compute them (also work for grammars, we will see them for parsing)
- Examples of first(e) computation:
  - first(ab*) = a
  - first((ab*)|c) = {a,c}
  - first(a*b*c) = {a,b,c}   abc   bc   c
  - first((cb|a*c*)d*e)) = {a, c, d, e}
- Notion of nullable (r) - whether , that is, whether empty string belongs to the regular language.

first symbols of words in a regexp

$a \in A \leftarrow$ symbols

$$\text{first}(a) = \{a\}, \quad a \in A$$

$\text{first} : \text{RegExp} \to \mathcal{P}(A)$

$\text{first}(r) \subseteq A$

$$\text{first}(r_1 | r_2) = \text{first}(r_1) \cup \text{first}(r_2)$$

$$\text{first}(r^*) = \text{first}(r)$$

$$\text{first}(r_1 r_2) = \begin{cases} \text{first}(r_1), & \varepsilon \notin r_1 \\ \text{first}(r_1) \cup \text{first}(r_2), & \underbrace{\varepsilon \in r_1}_{\text{nullable}(r_1)} \end{cases}$$

$$\varepsilon \in r \longleftrightarrow \text{nullable}(r)$$

Can regexp can derive the empty word

$nullable(r*) = true$

$nullable(r_1 | r_2) = nullable(r_1) \,||\, nullable(r_2)$

$nullable(a) = false$ $\quad a \in A \quad \varepsilon \notin A$

$nullable(\varepsilon) = true$

$nullable(r_1 r_2) = nullable(r_1) \,\&\&\, nullable(r_2)$

Converting Well-Behaved Regular Expression into Programs

Regular Expression

- a
- r1·r2
- (r1 | r2)

  $first(r_1) \cap first(r_2) = \emptyset$

- r*

Code

- **if** (current=a) next **else** error
- (code for r1) **;**
  (code for r2)
- **if** (current in first(r1))
  code for r1
  else
  code for r2
- **while**(current in first(r))
  code for r

Subtleties in General Case

- Sometimes first(e1) and first(e2) overlap for two different token classes:
- Must remember where we were and go back, or work on recognizing multiple tokens at the same time
- Example: comment begins with division sign, so we should not ‘drop’ division token when checking for comment!

Decision Tree to Map Symbols to Tokens

```
ch.current match {
  case '(' => {current = OPAREN; ch.next; return}
  case ')' => {current = CPAREN; ch.next; return}
  case '+' => {current = PLUS; ch.next; return}
  case '/' => {current = DIV; ch.next; return}
  case '*' => {current = MUL; ch.next; return}
  case '=' => { // more tricky because there can be =, ==
    ch.next
    if (ch.current=='=') {ch.next; current = CompareEQ; return}
    else {current = AssignEQ; return}
  }
  case '<' => { // more tricky because there can be <, <=
    ch.next
    if (ch.current=='=') {ch.next; current = LEQ; return}
    else {current = LESS; return}
  }
}
```