Formal Languages

and a “taste” of their algebra

Examples of Languages

$A = \{a,b\}$

$A^* = \{\varepsilon, a, b, aa, ab, ba, bb, aaa, aab, aba, \ldots\}$

Examples of two languages, subsets of Σ^* :

$L_1 = \{a, bb, ab\}$ (finite language, three words)

$L_2 = \{ab, abab, ababab, \ldots\}$

$= \{ (ab)^n \mid n \geq 1 \}$ (infinite language)

Formal Languages vs Scala

Formal language theory:

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

Scala representation:

- A – type
- List[A]
- w1 ::: w2 w1,w2:List[A]
- List()
- if c:A then List(c):List[A]
- w.length
- w.slice(p,q)
  w(i)
- L : Set[List[A]] (finite L)
  L : List[A] => Boolean
  (computable L)

Properties of Words

Concatenation is associative:

$$(w_1 \cdot w_2) \cdot w_3 = w_1 \cdot (w_2 \cdot w_3)$$

Empty word ε is left and right identity:

$$w \cdot \varepsilon = w$$

$$\varepsilon \cdot w = w$$

In the terminology of abstract algebra, the structure

$(A^*, \cdot, \varepsilon)$ is a **monoid**

Cancellation

If $w_1 \; w_3 = w_1 \; w_2$

then $w_3 = w_2$

If $w_3 \, w_1 = w_2 \; w_1$

then $w_3 = w_2$

There are many other properties, many easily provable from definition of operations.

Fact about Indexing Concatenation

Concatenation of w and v has these letters:

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

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

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

Properties of Length

$|\varepsilon| = 0$

$|c| = 1$ if $c \in A$

$|w_1 w_2| = |w_1| + |w_2|$ $w_i \in A^*$

Reverse of a Word

$\varepsilon^{-1} = \varepsilon$

$c^{-1} = c$ if $c \in A$

$(w_1 w_2)^{-1} = w_2^{-1} w_1^{-1}$

Sets of Words are Languages

Formal Languages vs Scala

Formal language theory:

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

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

$L^0 = \{\varepsilon\}$

$L^{n+1} = L \cdot L^n$

Scala (for finite languages)

```scala
type Lang[A] = Set[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}

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

Concatenation of Sets of Words

- Consider an alphabet A and all possible languages $L \subseteq A^*$
- Is this a monoid?
- Does the cancellation law hold?

  if $L_1 L_2 = L_1 L_3$ is it then $L_2 = L_3$?

Examples of Operations

$L = \{a, ab\}$

$L\,L = \{ aa, aab, aba, abab \}$

compute LLL

$L^* = \{\varepsilon, a, ab, aa, aab, aba, abab, aaa, \ldots \}$

Is bb inside L^* ?

Is it the case that

$L^* = \{ w \mid \text{immediately before each b there is a} \}$

If yes, prove it. If no, give a counterexample.

Observation

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

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

Obviously also $L^{n+1} = L^n\, L$

Star of a Language. Exercise with Proof

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

$= \cup_n L^n$ 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 $S \subseteq \{a,ab\}^*$.

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

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

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

Regular Expressions

- One way to denote (often infinite) languages
- Regular expression is an expression built from:
  - empty language $\emptyset$ (empty set of words)
  - {ε}, denoted just ε (set containing the empty word)
  - $\{a\}$ for $a \in A$, denoted simply by a
  - union of sets of words, denoted | (some folks use +)
  - concatenation of sets of words (dot, or not written)
  - Kleene star * (repetition)
- E.g. identifiers: letter (letter | digit)*

  (letter,digit are shorthand sets from before)

Regular Expressions

- Regular expressions are just a notation for some particular operations on languages

  letter (letter | digit)*

- Denotes the set

  letter (letter $\cup$ digit)*

Why is * called Kleene star?

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) ε|e
- e+ (repeat at least once) $e\,e^* = e^+$ $e^* = \varepsilon | e^+$
- $e^{k..*} = e^k\, e^*$ $e^{p..q} = e^p\, (\varepsilon|e)^{q-p}$
- complement: !e (A* - e) – non-obvious, need to use automata
- intersection: e1 & e1 (e1 ∩ e2) = ! (!e1 | !e1)

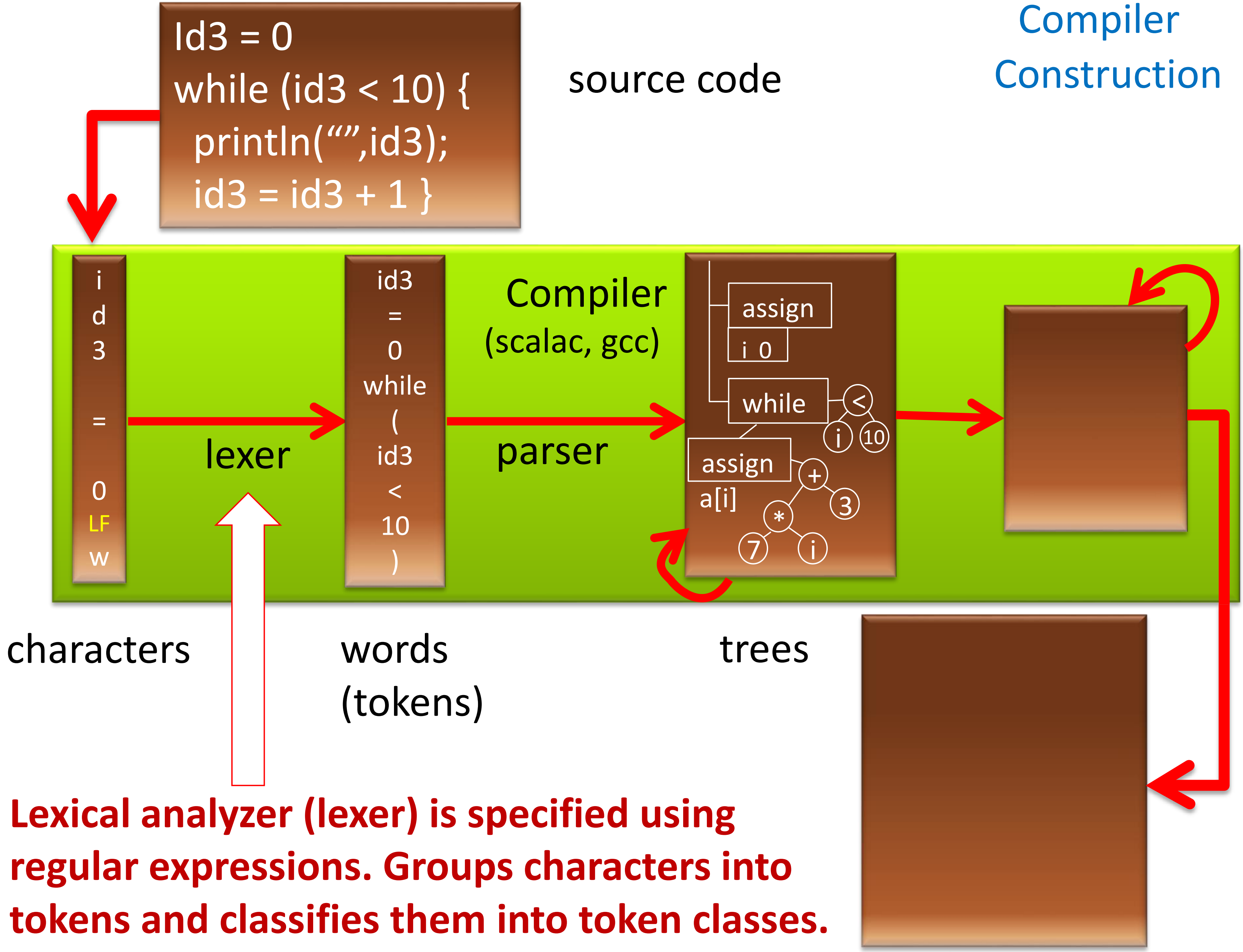
(Advanced) Quantification: we can also allow expressions with ∀

Techniques of Monadic Second-Order Logic of Strings

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

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

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*

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

Ident ::=

letter (letter | digit)*

integerConst ::=

digit digit*

stringConst ::=

“ AnySymbolExceptQuote* “

regular expressions

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

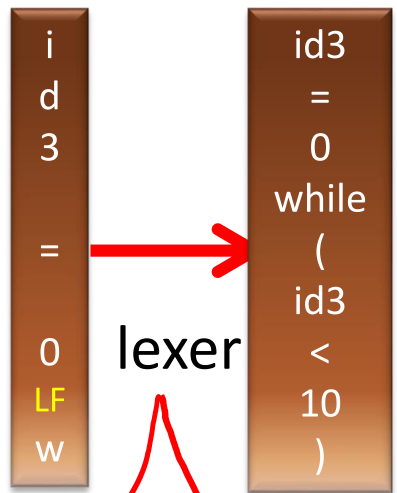
Stream of Char-s

(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
```

Recognizing Identifiers and Keywords

```
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 is Coming

- 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 $L \subseteq A^*$ is a language, then first(L) is set of all alphabet symbols that start some word in L

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

First Symbols of a Set of Words

first({a, bb, ab}) = {a,b}

first({a, ab}) = {a}

first({aaaaaaa})={a}

first({a})={a}

first({})={}

first({ε})={}

first({ε,ba})={b}

first of a regexp

- Given regular expression e, how to compute first(e)?
  - use automata (we will see this later)
  - 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}
  - first((cb|a*c*)d*e)) = { , d, e }
- Notion of nullable(r) - whether empty string belongs to the regular language.

first symbols of words in a regexp

first : RegExp => Set[A]   first(e) $\subseteq$ A

Define recursively:

first($\emptyset$) = $\emptyset$

first(ε) = $\emptyset$

$a \in A$ first(a) = $\{a\}$

first(e_1 | e_2) = first (e_1) $\cup$ first (e_2)

first(e^*) = first (e)

$$\text{first}(e_1\, e_2) = \begin{cases} \text{first}(e_1), & \varepsilon \notin \mathcal{L}(e_1) \\ \text{first}(e_1) \cup \text{first}(e_2), & \boxed{\varepsilon \in \mathcal{L}(e_2)} \end{cases}$$

nullable (e_2)

Can regular expr contain empty word

nullable(L) means $\varepsilon \in L$

nullable : RegExp => {0,1} (Boolean)

Define recursively:

$a \in A$

nullable($\varnothing$) = false

nullable(ε) = true

nullable(a) = false

nullable(e_1 | e_2) = nullable(e_1) $\lor$ nullable(e_2)

nullable(e^*) = true

nullable(e_1 e_2) = nullable(e_1) $\land$ nullable(e_2)

Converting Well-Behaved Regular Expression into Programs

Regular Expression	**Code**
a	**if** (current=a) next **else** error
r1 r2	(code for r1) **;** (code for r2)
(r1 | r2)	**if** (current in first(r1)) code for r1 **else** code for r2
r*	**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}
 }
}
```

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}
  }
}
```

What happens if we omit it?

consider input '<= '

Skipping Comments

```
if (ch.current='/') {
  ch.next
  if (ch.current='/') {
     while (!isEOL && !isEOF) {
       ch.next
     }
  } else {      // what do we set as the current token now?
                current = DIVSign
  }
}
```

Nested comments?   /* foo /* bar */ baz */

Longest Match (Maximal Munch) Rule

- There are multiple ways to break input chars into tokens
- Consider language with identifiers - ID, <=, <, =
- Consider these input characters:

```
interpreters <= compilers
```

- These are some ways to analyze it into tokens:

  ID(interpreters) LEQ ID(compilers)

  ID(inter) ID(preters) LESS AssignEQ ID(com) ID(pilers)

  ID(i) ID(nte) ID(rpre) ID(ter) LESS AssignEQ ID(co) ID(mpi) ID(lers)

- This is resolved by **longest match rule**:

If multiple tokens could follow, take the **longest token** possible

Consequences of Longest Match Rule

- Consider language with three operators:

  <, <=, =>

- For sequence '<=>' , lexer will report an error
  - Why?

- In practice, this is not a problem
  - we can always insert extra spaces

Longest Match Exercise

- Recall the maximal munch (longest match) rule: lexer should eagerly accept the longest token that it can recognize from the current point
- Consider the following specification of tokens, the numbers in parentheses gives the name of the token given by the regular expression

  (1) a(ab)* (2) b*(ac)* (3) cba (4) c+

- Use the maximal munch rule to tokenize the following strings according to the specification
  - c a c c a b a c a c c b a b c

    4 2 2 3 4
  - c c c a a b a b a c c b a b c c b a b a c

    2
- If we do not use the maximal munch rule, is another tokenization possible?

?

- Give an example of a regular expression and an input string, where the regular expression is able to split the input strings into tokens, but it is unable to do so if we use the maximal munch rule.

Token Priority

- What if our token classes intersect?
- Longest match rule does not help
- Example: a keyword is also an identifier
- Solution - **priority**: order all tokens,
  if overlap, take one with higher priority

- Example: if it looks both like keyword and like identifier, then it is a keyword (we say so)