Formal Languages and Regular Expressions

Language

Definition

A *language* over alphabet A is a set $L \subseteq A^*$. Example for $A = \{0,1\}$:

- a finite language like $L = \{1, 10, 1001\}$ or the empty language $\emptyset$
- infinite but very difficult to describe (there are random languages: there exist more languages as subsets of A^* than there are finite descriptions)
- infinite but having some nice structure, where words follow a certain "pattern" that we can describe precisely and check efficiently $\leftarrow$ these are our focus

$L_2 = \{01, 0101, 010101, \ldots\} =$ those non-empty words that are of the form $01 \ldots 01$ where the block 01 is repeated some finite positive number of times. Using notation $(01)^n$ for a word consisting of block 01 repeated n times, we can write
$L_2 = \{(01)^n \mid n \geq 1\}$.
Languages are sets, so we can take their union ($\cup$), intersection ($\cap$), and apply other set operations on languages.
Languages $\emptyset$ and $\{\varepsilon\}$ are very different: $\emptyset$ is a set that contains no words, whereas $\{\varepsilon\}$ contains precisely one word, the word of length zero.

Concatenating Languages

In addition to operations such as intersection and union that apply to sets in general, languages support additional operations, which we can define because their elements are words. The first one translates concatenation of words to sets of words, as follows.

Definition (Language concatenation)

Given $L_1 \subseteq A^*$ and $L_2 \subseteq A^*$, define $L_1 \cdot L_2 = \{w_1 w_2 \mid w_1 \in L_1, w_2 \in L_2\}$

Example: $\{\varepsilon, a, aa\} \cdot \{b, bb\} = \{b, bb, ab, abb, aab, aabb\}$

The definition above states that $w \in L_1 L_2$ if and only if there is one or more ways to split w into words w_1 and w_2, so that $w = w_1 w_2$ and such that $w_1 \in L_1$ and $w_2 \in L_2$.

Definition (Language exponentiation)

Given $L \subseteq A^*$, define

$$L^0 = \{\varepsilon\}$$
$$L^{n+1} = L \cdot L^n$$

Theorem

Given $L \subseteq A^*$, $L^n = \{w_1 \ldots w_n \mid w_1, \ldots, w_n \in L\}$

Expanding the Definition

If L is an arbitrary language, compute each of the following:

- $L\emptyset$
- $\emptyset L$
- $L\{\varepsilon\}$
- $\{\varepsilon\}L$
- $\emptyset\{\varepsilon\}$
- LL
- $\{\varepsilon\}^n$
- $\{w_1\}\{w_2\}$

Expanding the Definition

If L is an arbitrary language, compute each of the following:

- $L\emptyset$
- $\emptyset L$
- $L\{\varepsilon\}$
- $\{\varepsilon\}L$
- $\emptyset\{\varepsilon\}$
- LL
- $\{\varepsilon\}^n$
- $\{w_1\}\{w_2\}$

Note the difference in results between concatenation with:

- the empty language $\emptyset$, which contains no words
- the language $\{\varepsilon\}$, which contains exactly one word, ε

Expanding the Definition

If L is an arbitrary language, compute each of the following:

- $L\emptyset$
- $\emptyset L$
- $L\{\varepsilon\}$
- $\{\varepsilon\}L$
- $\emptyset\{\varepsilon\}$
- LL
- $\{\varepsilon\}^n$
- $\{w_1\}\{w_2\}$

Note the difference in results between concatenation with:

- the empty language $\emptyset$, which contains no words
- the language $\{\varepsilon\}$, which contains exactly one word, ε

Is it the case that always $L_1L_2 = L_2L_1$? Prove or give counterexample.

Concatenation of Languages

Let A be alphabet. Consider the set of all languages $L \subseteq A^*$

Is this a monoid?

Concatenation of Languages

Let A be alphabet. Consider the set of all languages $L \subseteq A^*$

Is this a monoid?

- Is there a neutral element?

Concatenation of Languages

Let A be alphabet. Consider the set of all languages $L \subseteq A^*$

Is this a monoid?

- Is there a neutral element?
- Which law needs to hold? Does it hold?

Concatenation of Languages

Let A be alphabet. Consider the set of all languages $L \subseteq A^*$

Is this a monoid?

- Is there a neutral element?
- Which law needs to hold? Does it hold?

Does the cancelation law hold?

Representing Languages in Programs

In general not possible: formal languages need not be recursively enumerable sets.
A reasonably powerful representation: computable characteristic function.
As for any subset of a set, a language $L \subseteq A^*$ is given by its *characteristic function* $f_L : A^* \to \{0, 1\}$ defined by: $f_L(w) \ = \ (\textbf{if}\ w \in L\ \textbf{then}\ 1\ \textbf{else}\ 0)$.
Here we use the `contains` field as the characteristic function and build the language $L_2 = \{(01)^n \mid n \geq 1\}$.

```
case class Lang[A](contains: List[A] -> Boolean)
def f(w: List[Int]): Boolean = w match {
 case Cons(0, Cons(1, Nil())) ⇒ true
 case Cons(0, Cons(1, wRest)) ⇒ f(wRest)
 case _ ⇒ false
}
val L2 = Lang(f)
val test = L2.contains(0::1::0::1::Nil()) // true
```

Representing Language Concatenation

We can use code to express concatenation of computable languages.

```
def concat(L1: Lang[A], L2: Lang[A]): Lang[A]= {
  def f(w: List[A]) = {
    val n = w.length
    def checkFrom(i: BigInt) = {
      require(0 <= i && i <= n)
      (L1.contains(w.slice(0, i)) && L2.contains(w.slice(i, n))) ||
      (i < n && checkFrom(i + 1))
    }
    checkFrom(0, w.length)
  }
  Lang(f) // return the language whose characteristic function is f
}
```

Repetition of a Language: Kleene Star

Definition (Kleene star)

Given $L \subseteq A^*$, define

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

Theorem

For $L \subseteq A^*$, for every $w \in A^*$ we have $w \in L^*$ if and only if

$$\exists n \geq 0. \exists w_1, \ldots, w_n \in L.\ w = w_1 \ldots w_n$$

$\{a\}^* = \{\varepsilon, a, aa, aaa, \ldots\}$
$\{a, bb\}^* = \{\varepsilon, a, bb, abb, bba, aa, bbbb, aabb, \ldots\}$ (describe this language)

Repetition of a Language: Kleene Star

Definition (Kleene star)

Given $L \subseteq A^*$, define

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

Theorem

For $L \subseteq A^*$, for every $w \in A^*$ we have $w \in L^*$ if and only if

$$\exists n \geq 0. \exists w_1, \ldots, w_n \in L.\ w = w_1 \ldots w_n$$

$\{a\}^* = \{\varepsilon, a, aa, aaa, \ldots\}$
$\{a, bb\}^* = \{\varepsilon, a, bb, abb, bba, aa, bbbb, aabb, \ldots\}$ (describe this language)

- words whose all contiguous blocks of b-s have even length

Can L^* be finite for some L? If so, describe all such L

Repetition of a Language: Kleene Star

Definition (Kleene star)

Given $L \subseteq A^*$, define

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

Theorem

For $L \subseteq A^*$, for every $w \in A^*$ we have $w \in L^*$ if and only if

$$\exists n \geq 0. \exists w_1, \ldots, w_n \in L.\ w = w_1 \ldots w_n$$

$\{a\}^* = \{\varepsilon, a, aa, aaa, \ldots\}$
$\{a, bb\}^* = \{\varepsilon, a, bb, abb, bba, aa, bbbb, aabb, \ldots\}$ (describe this language)

- words whose all contiguous blocks of b-s have even length

Can L^* be finite for some L? If so, describe all such L

- $\{\varepsilon\}^* = \{\varepsilon\}$, $\emptyset^* = \{\varepsilon\}$, for all others L has a word of length ≥ 1, so L^* is infinite

Star and the Empty Word

Concatenating with an empty word has no effect, so we have the following:

$$L^* = (L \setminus \{\varepsilon\})^* = \{\varepsilon\} \cup \bigcup_{n \geq 1} (L \setminus \{\varepsilon\})^n$$

Moreover, $w \in L^*$ if and only if either $w = \varepsilon$ or, for some n where $1 \leq n \leq |w|$,

$$w = w_1 \ldots w_n$$

where $w_i \in L$ and $|w_i| \geq 1$ for all i where $1 \leq i \leq n$.

Star and the Empty Word

Concatenating with an empty word has no effect, so we have the following:

$$L^* = (L \setminus \{\varepsilon\})^* = \{\varepsilon\} \cup \bigcup_{n \geq 1} (L \setminus \{\varepsilon\})^n$$

Moreover, $w \in L^*$ if and only if either $w = \varepsilon$ or, for some n where $1 \leq n \leq |w|$,

$$w = w_1 \dots w_n$$

where $w_i \in L$ and $|w_i| \geq 1$ for all i where $1 \leq i \leq n$.

- we omit ε because it leaves concatenation the same
- we can assume $n \leq |w|$ because all blocks have length at least one

Star and the Empty Word

Concatenating with an empty word has no effect, so we have the following:

$$L^* = (L \setminus \{\varepsilon\})^* = \{\varepsilon\} \cup \bigcup_{n \geq 1} (L \setminus \{\varepsilon\})^n$$

Moreover, $w \in L^*$ if and only if either $w = \varepsilon$ or, for some n where $1 \leq n \leq |w|$,

$$w = w_1 \dots w_n$$

where $w_i \in L$ and $|w_i| \geq 1$ for all i where $1 \leq i \leq n$.

- we omit ε because it leaves concatenation the same
- we can assume $n \leq |w|$ because all blocks have length at least one

If L is computable (has a computable characterstic function), is L^* also computable?

Star and the Empty Word

Concatenating with an empty word has no effect, so we have the following:

$$L^* = (L \setminus \{\varepsilon\})^* = \{\varepsilon\} \cup \bigcup_{n \geq 1} (L \setminus \{\varepsilon\})^n$$

Moreover, $w \in L^*$ if and only if either $w = \varepsilon$ or, for some n where $1 \leq n \leq |w|$,

$$w = w_1 \dots w_n$$

where $w_i \in L$ and $|w_i| \geq 1$ for all i where $1 \leq i \leq n$.

- we omit ε because it leaves concatenation the same
- we can assume $n \leq |w|$ because all blocks have length at least one

If L is computable (has a computable characterstic function), is L^* also computable?

- try all possible ways of splitting w
- if $k = |w|$, for each point between the letters of w you can decide to split there or not, so there are 2^{k-1} ways to split: $w = \square \underbrace{|\square| \dots |\square|}_{k-1} \square$

Star and the Empty Word

Concatenating with an empty word has no effect, so we have the following:

$$L^* = (L \setminus \{\varepsilon\})^* = \{\varepsilon\} \cup \bigcup_{n \geq 1} (L \setminus \{\varepsilon\})^n$$

Moreover, $w \in L^*$ if and only if either $w = \varepsilon$ or, for some n where $1 \leq n \leq |w|$,

$$w = w_1 \dots w_n$$

where $w_i \in L$ and $|w_i| \geq 1$ for all i where $1 \leq i \leq n$.

- we omit ε because it leaves concatenation the same
- we can assume $n \leq |w|$ because all blocks have length at least one

If L is computable (has a computable characterstic function), is L^* also computable?

- try all possible ways of splitting w
- if $k = |w|$, for each point between the letters of w you can decide to split there or not, so there are 2^{k-1} ways to split: $w = \Box \underbrace{|\Box| \dots |\Box|}_{k-1} \Box$
- Exercise: find a way to check $w \in L^*$ with polynomially many invocations of $w \in L$

Starring: $\{a, ab\}$

Let $A = \{a, b\}$ and $L = \{a, ab\}$.
Come up with a property $P(w)$ that describes the language L^*, such that:

$$L^* = \{w \in A^* \mid P(w)\}$$

Prove that the property and L^* denote the same language.

Starring: $\{a, ab\}$

Let $A = \{a, b\}$ and $L = \{a, ab\}$.
Come up with a property $P(w)$ that describes the language L^*, such that:

$$L^* = \{w \in A^* \mid P(w)\}$$

Prove that the property and L^* denote the same language.

Example properties:

- does not begin with b
- does not contain bb

Conjectured property $P(w)$: there is an "a" immediately before every "b" inside w.

Proving the Property

$$L^* = \{w \in A^* \mid P(w)\}$$

where $P(w)$ is: there is an "a" immediately before every "b" occurrence inside w.
How to prove that this $P(w)$ is correct? Show two directions of set equality:

- $\{a, ab\}^* \subseteq \{w \mid P(w)\}$, that is: if w is a concatenation $w_1 ... w_n$ where each w_i is either a or ab, then, inside w, there is an "a" immediately before every "b".
- $\{w \mid P(w)\} \subseteq \{a, ab\}^*$, that is: if we have a string such that every occurrence of b has an a immediately left to it, then we can split w into some number of blocks $w_1 \dots w_n$ such that each w_i is either a or ab.

Regular Expressions

Regular Expressions

Mathematical expressions used to denote finite and infinite languages. Definition: a regular expression over language A is build inductively as follows:

- $\emptyset$, denoting the empty set of strings
- ε, denoting the language $\{\varepsilon\}$ containing only empty word
- a for $a \in A$, denoting the language with one word of length one, $\{a\}$
- $r_1 \mid r_2$ denoting the union of languages
- $r_1 r_2$ denoting concatenation of languages of r_1 and r_2
- $r*$ denoting the Kleene star of the language of r (a high priority operator)

Examples:

- $(a|ab)^*$ denoting the language $\{a, ab\}^*$
- $(a|b|c)\ (a|b|c|0|1)*$ denotes $\{a, b, c\}\{a, b, c, 0, 1\}^*$, the identifiers that start with one of the three letters a, b, c followed by a sequence of the letters or digits $0, 1$.

Example Use of Regular Expressions: grep

`grep` is a widely used command-line (terminal) tool that filters those lines that match a given pattern. Pattern can be a fixed string,

```
$ cd /etc/dictionaries-common
$ tail -n 5 words
zwieback
zwieback's
zygote
zygote's
zygotes
$ grep 'ncompat' words
incompatibilities
incompatibility
incompatibility's
incompatible
incompatible's
incompatibles
incompatibly
```

grep for clp using a regular expression

Find words that start with c, contain l and end with p:

```
$ grep '^c.*l.*p$' words
cantaloup
clamp
clap
claptrap
clasp
cleanup
clip
clomp
clop
clump
cowslip
```

Some notation specific to `grep`:

- `.` means any character, so `.*` means any string
- `^` means start of the line (otherwise it adds `.*` in front)
- `$` means end of the line (otherwise it adds `.*` at the end)

Another grep Example

Use ‘-E‘ so you don’t have to escape union | and parentheses (,)

```
$ grep -E '^(b|c)(a|i|o)*t$' words
bait
bat
bit
boat
boot
bot
cat
coat
coot
cot
ct
```

One can also use regular expressions for syntax highlighting

Computing 'nullable' for regular expressions

If e is regular expression (its syntax tree), then $L(e)$ is the language denoted by it.
For $L \subseteq A^*$ we define $nullable(L)$ as $\varepsilon \in L$
If e is a regular expression, we can compute $nullable(e)$ to be equal to $nullable(L(e))$, as follows:

$$\begin{array}{rcl}
nullable(\emptyset) & = & false \\
nullable(\varepsilon) & = & true \\
nullable(a) & = & false \\
nullable(e_1|e_2) & = & nullable(e_1) \vee nullable(e_2) \\
nullable(e^*) & = & true \\
nullable(e_1 e_2) & = & nullable(e_1) \wedge nullable(e_2)
\end{array}$$

Computing 'first' for regular expressions

For $L \subseteq A^*$ we define: $first(L) = \{a \in A \mid \exists v \in A^*.\ av \in L\}$.
If e is a regular expression, we can compute $first(e)$ to be equal to $first(L(e))$, as follows:

$$\begin{array}{rcl}
first(\emptyset) & = & \emptyset \\
first(\varepsilon) & = & \emptyset \\
first(a) & = & \{a\},\ \text{for } a \in A \\
first(e_1|e_2) & = & first(e_1) \cup first(e_2) \\
first(e*) & = & first(e) \\
first(e_1 e_2) & = & \text{if } nullable(e_1) \ \text{ then } first(e_1) \cup first(e_2) \\
 & & \text{else } first(e_1)
\end{array}$$

Clarification for first of concatenation

Let e be $\mathbf{a}^*\mathbf{b}$. Then $L(e) = \{b, ab, aab, aaab, \ldots\}$
$first(L(e)) = \{a, b\}$

$e = e_1e_2$ where $e_1 = a^*$ and $e_2 = b$. Thus, $nullable(e_1)$.

$$first(e_1e_2) = first(e_1) \cup first(e_2) = \{a\} \cup \{b\} = \{a, b\}$$

It is *not correct* to use $first(e) =^? \; first(e_1) = \{a\}$.
Nor is it correct to use $first(e) =^? \; first(e_2) = \{b\}$.
We must use their union.

Converting Simple Regular Expresssions into a Lexer Manually

$regular\ expression$	$lexercode$
a (where $a \in A$)	**if** $current == a$ **then** $next$ **else** ...
$r_1 r_2$	$code(r_1);\ code(r_2)$

$r_1 \| r_2$	**if** $current \in first(r_1)$ **then**
	$code(r_1)$
	else $code(r_2)$

r^*	**while** $current \in first(r)$ **do** $code(r)$

More complex cases

In other cases, a few upcoming characters ("lookahead") are not sufficient to determine which token is coming up.

Examples:
A language might have separate numeric literal tokens to simplify type checking:

- integer constants: $digit\ digit^*$
- floating point constants: $digit\ digit^*\ .\ digit\ digit^*$

Floating point constants must contain a period (e.g., Modula-2).

Division sign begins with same character as // comments.
Equality can begin several different tokens.

In such cases, we process characters and store them until we have enough information to make the decision on the current token.

Example of a part of a manually written lexical analyzer

```
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}
  }
}
```

Example of a part of a manually written lexical analyzer

```
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 if we omit ch.next?

Example of a part of a manually written lexical analyzer

```
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 if we omit ch.next?

Lexer could generate a non-existing equality token!

White spaces and comments

Whitespace can be defined as a token, using space character, tabs, and various end of line characters. Similarly for comments.

In most languages (Java, ML, C) white spaces and comments can occur between any two other tokens have no meaning, so parser does not want to see them.

Convention: the lexical analyzer removes the whitespace tokens from its output. Instead, it always finds the next non-whitespace non-comment token.

Other conventions and interpretations of new line became popular to make code more concise (sensitivity to end of line or indentation). Not our problem in this course! Tools that do formatting of source also must remember comments. (We ignore those in this course.)

Skipping simple comments

```
if (ch.current='/') {
  ch.next
  if (ch.current='/') {
     while (!isEOL && !isEOF) {
       ch.next
     }
  } else {
```

Skipping simple comments

```
if (ch.current='/') {
  ch.next
  if (ch.current='/') {
     while (!isEOL && !isEOF) {
       ch.next
     }
  } else {
    $ch.current = DIV$
  }
}
```

Skipping simple comments

```
if (ch.current='/') {
  ch.next
  if (ch.current='/') {
    while (!isEOL && !isEOF) {
      ch.next
    }
  } else {
    $ch.current = DIV$
  }
}
```

Nested comments: this is a single comment:
/* foo /* bar */ baz */
Solution:

Skipping simple comments

```
if (ch.current='/') {
  ch.next
  if (ch.current='/') {
     while (!isEOL && !isEOF) {
       ch.next
     }
  } else {
    $ch.current = DIV$
  }
}
```

Nested comments: this is a single comment:
/* foo /* bar */ baz */
Solution: use a counter for nesting depth

Longest match (maximal munch) rule

Lexical analyzer is required to be greedy: always get the longest possible token at this time. Otherwise, there would be too many ways to split input into tokens!

Longest match (maximal munch) rule

Lexical analyzer is required to be greedy: always get the longest possible token at this time. Otherwise, there would be too many ways to split input into tokens!

Consider language with the following tokens:

ID:	letter(digit | letter)*
LE:	$<=$
LT:	$<$
EQ:	$=$

How can we split this input into subsequences, each of which in a token:

$$interpreters <= compilers$$

Longest match (maximal munch) rule

Lexical analyzer is required to be greedy: always get the longest possible token at this time. Otherwise, there would be too many ways to split input into tokens!

Consider language with the following tokens:

ID:	letter(digit | letter)*
LE:	$<=$
LT:	$<$
EQ:	$=$

How can we split this input into subsequences, each of which in a token:

$$interpreters <= compilers$$

Some candidate solutions:

ID(interpreters) LE ID(compilers) - OK, longest match rule
ID(inter) ID(preters) LE ID(compilers)
ID(interpreters) LT EQ ID(compilers)

Longest match (maximal munch) rule

Lexical analyzer is required to be greedy: always get the longest possible token at this time. Otherwise, there would be too many ways to split input into tokens!

Consider language with the following tokens:

ID:	letter(digit | letter)*
LE:	$<=$
LT:	$<$
EQ:	$=$

How can we split this input into subsequences, each of which in a token:

$$interpreters <= compilers$$

Some candidate solutions:

ID(interpreters) LE ID(compilers)	- OK, longest match rule
ID(inter) ID(preters) LE ID(compilers)	- not longest match: ID(inter)
ID(interpreters) LT EQ ID(compilers)	

Longest match (maximal munch) rule

Lexical analyzer is required to be greedy: always get the longest possible token at this time. Otherwise, there would be too many ways to split input into tokens!

Consider language with the following tokens:

ID:	letter(digit | letter)*
LE:	$<=$
LT:	$<$
EQ:	$=$

How can we split this input into subsequences, each of which in a token:

$$interpreters <= compilers$$

Some candidate solutions:

ID(interpreters) LE ID(compilers)	- OK, longest match rule
ID(inter) ID(preters) LE ID(compilers)	- not longest match: ID(inter)
ID(interpreters) LT EQ ID(compilers)	- not longest match: LT

Longest match rule is greedy, but that's OK

Consider language with ONLY these three operators:

LT:	$<$
LE:	$<=$
IMP:	$=>$

Given sequence: $<=>$

lexer will split it as $<=, >$, return LE as token, then report unknown token error on $>$.

This is the behavior that we expect.

Longest match rule is greedy, but that's OK

Consider language with ONLY these three operators:

LT:	$<$
LE:	$<=$
IMP:	$=>$

Given sequence: $<=>$
lexer will split it as $<=, >$, return LE as token, then report unknown token error on $>$.
This is the behavior that we expect.

This is despite the fact that one could in principle split the input into $<$ and $=>$, which correspond to sequence LT IMP. But a split into $<$ and $=>$ would not satisfy longest match rule, so we do *not* want it. Reporting error is the right thing to do here.

This behavior is not a restriction in practice: programmers we can insert extra spaces to stop longest match rule from taking too many characters.

Token priority

What if our token classes intersect?
Longest match rule does not help, because the same string belongs to two regular expressions
Examples:

- a keyword is also an identifier
- a constant that can be integer or floating point

Solution is priority: order all tokens and in case of overlap take one earlier in the list (higher priority). This avoids having to *subtract* language of one token from another.
Examples:

- if it matches regular expression for both a keyword and an identifier, then we define that it is a keyword.
- if it matches both integer constant and floating point constant regular expression, then we define it to be (for example) integer constant.

Token priorities for overlapping tokens must be specified in language definition.

Automating the Construction of Lexers

Lexical Analyzer Generators

Help us avoid error-prone conversion from regular expressions to lexical analyzers.
How they work:

- Specify tokens using regular expressions
- Use the generator to obtain a lexer

Different solutions exist:

- `lex` tool approach: construct optimized lexer once and for all, generate source code to compile with the rest of the interpreter or compiler. Can be more efficient but is less flexible (we must know regular expressions ahead of time) and complicates the build (automatically generated source code).
- library approach: provide regular expressions to a library function, which returns gives a lexical analyzer. This is what we use in ZipLex (Lab 2)

Derivatives of a Language with Respect to a Letter

A useful concept in automating lexers is *derivative* of a language.

Let $L \subseteq A^*$ be a language and let $x \in A$ be a letter. Then

$$\delta_x L = \{w \in A^* \mid xw \in L\}$$

Example: if $L = \{\varepsilon, b, aa, abb, bbba\}$ then

$$\delta_a L = \{a, bb\}$$

$$\delta_b L = \{\varepsilon, bba\}$$

Observe that δ_x ignores all words that do not begin by x and collects the suffixes of those that begin with x.
If $A = \{a, b\}$ and $L \subseteq A^*$, we can decompose L into disjoint sets:

$$L = (a\,\delta_a L) \cup (b\,\delta_b L) \cup \{\varepsilon \mid nullable(L)\}$$

Derivatives ignore ε if it is in L; the last part puts it back iff L was nullable.