Lecture 3
Building Lexical Analyzers

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

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

Clarification for first of concatenation

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

Clearly $e = e_1 e_2$ where $e_1 = a^*$ and $e_2 = b$. Thus, $\mathit{nullable}(e_1)$.

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

It is *not correct* to use $\mathit{first}(e) \stackrel{?}{=} \mathit{first}(e_1)$ nor to use $\mathit{first}(e) \stackrel{?}{=} \mathit{first}(e_2)$, we must use their union.

Converting Simple Regular Expressions to Source Code

regular expression	*code*
$a \quad (a \in A)$	$\mathit{if}\ (\mathit{current} = a)\ \mathit{next}\ \mathit{else}\ \mathit{error}$
$r_1 r_2$	$\mathit{code}(r_1);\ \mathit{code}(r_2)$
$r_1 \mid r_2$	$\mathit{if}\ (\mathit{current} \in \mathit{first}(r_1))$ $\quad \mathit{code}(r_1)$ $\mathit{else}\ \mathit{code}(r_2)$
$r*$	$\mathit{while}\ (\mathit{current} \in \mathit{first}(r))$ $\quad \mathit{code}(r)$

More complex cases

In other cases, one or even more upcoming characters (“lookahead”) are not sufficient to determine which token is coming up.

Example: a language might have separate 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 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 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 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 those “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 and white space. We ignore those as well here.

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!

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:

$$\textit{interpreters} <= \textit{compilers}$$

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 $\mid$ letter)*
LE:	$<=$
LT:	$<$
EQ:	=

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

$$\textit{interpreters} <= \textit{compilers}$$

Some solutions:

ID(interpreters) LE ID(compilers)

ID(inter) ID(preters) LT EQ ID(com) ID(pilers)

ID(interpreters) LT EQ ID(com) ID(pilers)

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 $\mid$ letter)*
LE:	$<=$
LT:	$<$
EQ:	=

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

$$\textit{interpreters} <= \textit{compilers}$$

Some solutions:

ID(interpreters) LE ID(compilers) - OK, longest match rule

ID(inter) ID(preters) LT EQ ID(com) ID(pilers)

ID(interpreters) LT EQ ID(com) ID(pilers)

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:

$$\textit{interpreters} <= \textit{compilers}$$

Some solutions:

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

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:

$$\textit{interpreters} <= \textit{compilers}$$

Some solutions:

ID(interpreters) LE ID(compilers) - OK, longest match rule

ID(inter) ID(preters) LT EQ ID(com) ID(pilers)

- not longest match: ID(inter)

ID(interpreters) LT EQ ID(com) ID(pilers)

- not longest match: LT

Longest match rule is greedy, but that's OK

Consider language with ONLY these three operators:

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

$$<=>$$

lexer will first return LE as token, and then report unknown token $>$'.
This error is what we expect and that is fine.

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

This is not a problem: programmer we can insert extra spaces to stop maximal munch 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 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).
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) an integer

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