**Hand-Written Lexical Analyzer for While**

**Lexical Analyzer**

Also called: **lexer**, **scanner**, or **tokenizer**

Transforms **sequence of characters**

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **w** | **h** | **i** | **l** | **e** |  | **(** | **c** | **o** | **u** | **n** | **t** |  | **<** |  | **1** | **0** | **)** |  | **{** |  | **LF** |  |  |  | **c** | **o** | **u** | **n** | **t** | **=** |

into **sequence of** [**tokens**](http://lara.epfl.ch/w/cc09:tokens_words_of_while_language)

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **while** | **(** | **count** | **<** | **10** | **)** | **{** | **count** | **=** |

**Interfaces for Lexer**

In practice, we read characters and generate tokens **on demand**

Work with **streams** instead of sequences, with procedures like

* **current** - return current element in stream
* **next** - advance the current element

Lexer uses input stream of characters

* example character stream class: [CharStream.scala](http://lara.epfl.ch/w/cc09:charstream.scala)

Lexer provides:

* types defining tokens, like these [Tokens.scala](http://lara.epfl.ch/w/cc09:tokens.scala)
* lexer interface, like this one: [LexerInterface.scala](http://lara.epfl.ch/w/cc09:lexerinterface.scala)

**Identifiers and Keywords**

Regular expression for identifiers: letter (letter|digit)\*

if (isLetter) {

b = new StringBuffer

while (isLetter || isDigit) {

b.append(ch.current)

ch.next

}

}

Keywords look like identifiers, but are designated as keywords.

One implementation: a constant Map from strings to keyword token numbers

* if not in map, then it is ordinary identifier

**Integer Constants**

Regular expression for numbers: digit digit\*

if (isDigit) {

k = 0

while (isDigit) {

k = 10 \* k + digitToNumber(ch.current)

ch.next

}

}

For digitToNumber etc see [ASCII](http://en.wikipedia.org/wiki/ASCII)

**Skipping Comments**

if (ch.current='/') {

ch.next

if (ch.current='/') {

while (!isEOL && !isEOF) {

ch.next

}

}

}

How about nested comments?

* need a counter

**Longest Match Rule**

There are multiple ways to break character stream into tokens.

Consider language with identifiers, ⇐, <, =

Consider this character stream:

interpreters <= compilers

These are some ways to analyze it into tokens:

1. ID(interpreters) LEQ ID(compilers)
2. ID(inter) ID(preters) LESS AssignEQ ID(com) ID(pilers)
3. 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

Note: consider language with only three operators:

1. ‘<’
2. '<='
3. '=>'

Then for the sequence '<=>' the lexer will report an error:

* first token is '<=
* second token is not recognized

In practice, this is not a problem, we can always use space.

**Token Priority**

What if our token classes intersect?

Longest match rule does not help

Example: a keyword is usually also an identifier

Priority: order all tokens, if in doubt take one with higher priority

* if it looks both like keyword and like an identifier, then it is a keyword

**Combining Lexers for All Tokens**

How do we know when we are supposed to analyze string, when integer sequence etc?

For 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

Given regular expression e, how to compute first(e)?

* use automata (we will see this next)
* rules that directly computate 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) ) =

Notion of nullable ( r ) - whether $\varepsilon \in r$, that is, whether empty string belongs to the regular language.

Sometimes first(e1) and first(e2) overlap for two different token classes:

* we 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!

A partial lexer: [Lexer.scala](http://lara.epfl.ch/w/cc09:lexer.scala) and [LexerTest.scala](http://lara.epfl.ch/w/cc09:lexertest.scala)