Compiler Phases – Lexical analysis

- The lexer reads the source file and divides the text into lexical units (tokens), such as:
  - Reserved words BEGIN, IF, ...
  - Identifiers x, StringTokenizer, ...
  - Special characters +, *, -, ^, ...
  - Numbers 30, 3.14, ...
  - Comments (* text *),
  - Strings "text".
- Lexical errors (such as 'illegal character', 'unwelled character string', 'comment without end') are reported.

Lexical Analysis of English

- The sentence
  "The boy's cowbell won't play."
  would be translated to the list of tokens
    the, boy's, possessive, cowbell, will, not, play

Lexical Analysis of Java

- The sentence
  "x = 3.14 * (9.0+y);"
  would be translated to the list of tokens
    <ID,x>, EQ, <FLOAT,3.14>, STAR, LPAREN,
    <FLOAT,9.0>, PLUS, <ID,y>, RPAREN, SEMICOLON

Diagram:
- Compiler Phases
  - Source to Lexer to Parser
  - IR, Intermediary Code, Compiler Code, Semantic Analysis
  - AST, Abstract Syntax Tree
### White space

In most modern languages white space (blanks and tabs) are significant. FORTRAN and Algol-68 are different: whitespace may be added anywhere to improve readability.

The FORTRAN statement

```
DO 5 I = 1.25
```

is an assignment statement, meaning the same as:

```
D05 I = 1.25
```

This statement, on the other hand, is a loop statement:

```
DO 5 I = 1,25
    ...  
5 CONTINUE
```

### Buffering

- If done incorrectly, lexical analysis can be an expensive phase of the compiler. It is the only phase which actually considers each and every character of the program.

- It is, for example, crucial not to read one character at a time from the input file. Rather, a large block of the input text file must be read and kept in a buffer. This buffer is then used to provide the lex and character.

- Sometimes the lex may also need to look ahead at characters to come before deciding on what token appears next in the text. The buffer is useful in such circumstances also.

### Example - Lexical Analysis

- Break up the source code (a text file) and into tokens.

<table>
<thead>
<tr>
<th>Source Code</th>
<th>Stream of Tokens</th>
</tr>
</thead>
<tbody>
<tr>
<td>PROCEDURE Foo ();</td>
<td>PROCEDURE, &lt;id,Foo&gt;, LPAR, RPAR, SC,</td>
</tr>
<tr>
<td>VAR i : INTEGER;</td>
<td>VAR, &lt;id,i&gt;, COLON, &lt;id,INTEGER&gt;, SC,</td>
</tr>
<tr>
<td>BEGIN</td>
<td>BEGIN, &lt;id,i&gt;, CEQ, &lt;int,i&gt;, SC,</td>
</tr>
<tr>
<td>i := 1;</td>
<td>WHILE, &lt;id,i&gt;, LT, &lt;int,20&gt;, DO,</td>
</tr>
<tr>
<td>WHILE i &lt; 20 DO</td>
<td>PRINT, &lt;id,i&gt;, MUL, &lt;int,2&gt;, SC,</td>
</tr>
<tr>
<td>PRINT i * 2;</td>
<td>&lt;id,i&gt;, CEQ, &lt;id,i&gt;, MUL, &lt;int,2&gt;, PLUS,</td>
</tr>
<tr>
<td>i := i * 2 + 1;</td>
<td>&lt;int,i&gt;, SC, ENDO, SC, END, &lt;id,Foo&gt;, SC</td>
</tr>
<tr>
<td>ENDO;</td>
<td></td>
</tr>
<tr>
<td>END Foo;</td>
<td></td>
</tr>
</tbody>
</table>

### Free vs. Fixed Format

- Most languages are free format, i.e. it does not matter where on a line of text a certain token occurs.

- FORTRAN (at least early versions) uses a fixed format where the first 6 characters on the input line is a label, and the last characters (columns 72-80) a comment. A "C" in the first column furthermore signifies a continuation line.

- Python, Occam, and some functional languages use indentation to indicate nesting.
**Communication**

- The Lexer may communicate with the parser in many different ways.
- Lexical analysis might, for example, run as a special pass writing the tokens on a temporary file which is read by the parser.
- Or-and this is probably the most common situation-the parser makes a procedure call to the lexer whenever a token is needed.
- The Lexer and the Parser could also run as two concurrent processes communicating over a pipe.

---

**Keywords**

- Most languages have reserved keywords, which means that these words may not be redefined by the user.
- PL/I does not reserve keywords which makes it difficult for the lexer to distinguish between user-defined identifiers and keywords.

**Error handling**

- What do we do when an error is encountered during lexical analysis?
  - Panic Replace characters until a well-formed token is found.
  - Replace Insert an incorrect character.
  - Delete Insert a missing character.
  - Transpose Switch two characters.

---

**Transition Diagrams**

![Transition Diagram](attachment:image.png)
TYPE TokenType = (Assign, End, ...);
VAR s : (State0, State1, ...);
c : CHAR;
PROCEDURE GetToken () : TokenType;
BEGIN
CASE s OF
  State0 :
c := NextChar();
  CASE c OF
    "": s := State1;
    "E": s := State4;
    "0": s := State9;
    ELSE s := State7
  END;
  State1 :
c := NextChar();
  IF c = "=" THEN s := State2
  ELSE s := State3
  END;
  State2 :
RETURN Assign;
  State3 :
PutChar(c); RETURN Colon;
  State4 :
c := NextChar();
  IF c = "N" THEN s := State5
  ELSE s := State7
  END;
  State5 :
c := NextChar();
  IF c = "D" THEN s := State6
  ELSE s := State7
  END;
  State6 :
c := NextChar();
  IF IsLetterOrDigit(c)
  THEN s := State8
  ELSE s := State7
  END;
  State7 :
PutChar(c); RETURN End;
END GetToken;

State8 :
c := NextChar();
  IF NOT IsLetterOrDigit(c)
  THEN s := State9
  END;
State9 :
PutChar(c); RETURN Ident;
State10 :
c := NextChar();
  IF NOT IsDigit(c)
  THEN s := State11
  END;
State11 :
PutChar(c); RETURN Int;
END GetToken;

Regular Grammars

- A grammar is regular if all rules are of the form $A \rightarrow aB$ and $A \rightarrow a$.
- By convention, the symbols $A, B, C, \ldots$ are non-terminals, $a, b, c, \ldots$ are terminals, and $\alpha, \beta, \gamma, \ldots$ are strings of symbols.
- Regular grammars are used to describe the lexical structure of programs, i.e., what tokens look like.
Regular Grammars...

- The following grammar describes C identifiers:

\[
\begin{align*}
\text{id} & \rightarrow \text{letter} | \text{letter } S \\
S & \rightarrow \text{letter} | \text{letter } S \\
\text{letter} & \rightarrow A | \cdots | Z | a | \cdots | z
\end{align*}
\]

- Here's a derivation of the identifier `cow5`:

\[
\text{id} \Rightarrow \text{letter } S \Rightarrow c S \Rightarrow c \text{ letter } S \Rightarrow c o S \Rightarrow c o \text{ letter } S \Rightarrow c o w S \Rightarrow c o w \text{ digit} \Rightarrow \text{cow5}
\]

Regular Expressions

Regular expressions (REs) have the same expressive power as regular grammars. An RE for FP numbers:

\[
(\text{"\|" } | \text{"\-\"})?\text{digit } * \text{.digit } + (E(\text{"\+" } | \text{"\-\"})?\text{digit } +)
\]

<table>
<thead>
<tr>
<th>Character</th>
<th>Matches</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>.</code></td>
<td>The character.</td>
</tr>
<tr>
<td>`e1</td>
<td>e2`</td>
</tr>
<tr>
<td><code>e1 e2</code></td>
<td><code>S_1 S_2</code>, if <code>e1</code> matches <code>S_1</code> and <code>e2</code> matches <code>S_2</code>.</td>
</tr>
<tr>
<td><code>e*</code></td>
<td>One or more <code>S</code> if <code>S</code> is matched by <code>e</code>.</td>
</tr>
<tr>
<td><code>e?</code></td>
<td>Zero or more <code>S</code> if <code>S</code> is matched by <code>e</code>.</td>
</tr>
<tr>
<td><code>(e)</code></td>
<td><code>S</code>, if <code>S</code> is matched by <code>e</code>.</td>
</tr>
<tr>
<td><code>\text{\textbackslash e}</code></td>
<td><code>S</code>, if <code>S</code> is matched by <code>e</code>.</td>
</tr>
</tbody>
</table>

Regular Grammars...

- This is a grammar for floating point numbers. As written, it is not quite regular: We treat `\text{digit}` as a terminal.

\[
\begin{align*}
\text{float} & \rightarrow \text{\textpm float1 } | \text{\textpm float1 } | \text{\textpm float1 } \\
\text{float1} & \rightarrow \text{\textpm float1 } | \text{\textpm float1 } | \text{\textpm float1 } \\
\text{float2} & \rightarrow \text{\textpm float3 } \\
\text{float3} & \rightarrow \text{\textpm float4 } | \text{\textpm float4 } \\
\text{float4} & \rightarrow \text{\textpm float4 } | \text{\textpm float4 } \\
\text{float5} & \rightarrow \text{\textpm float6 } \\
\text{float6} & \rightarrow \text{\textpm float7 } | \text{\textpm float7 } | \text{\textpm float7 } \\
\text{float7} & \rightarrow \text{\textpm float7 } | \text{\textpm float7 } | \text{\textpm float7 }
\end{align*}
\]
The Chomsky Hierarchy

<table>
<thead>
<tr>
<th>TYPE</th>
<th>GRAMMAR</th>
<th>PSR</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Unrestricted</td>
<td>$\alpha \rightarrow \beta$</td>
</tr>
<tr>
<td>1</td>
<td>Context Sensitive</td>
<td>$\alpha \rightarrow \beta,</td>
</tr>
<tr>
<td>2</td>
<td>Context Free</td>
<td>$A \rightarrow \beta$</td>
</tr>
<tr>
<td>3</td>
<td>Regular</td>
<td>$A \rightarrow a\beta$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$A \rightarrow a$</td>
</tr>
</tbody>
</table>

Regular Languages are less powerful than context free languages.

- Regular languages are organized in the Chomsky Hierarchy according to their generative power.
- Type 3 languages are more restrictive (can describe simpler languages than) type 2 languages.
- Type 3 languages can be parsed in linear time, type 2 languages in cubic time.
- Programming languages are in between type 2 and 3.
- Two natural languages (Swiss German and Bambara) are known not to be context free.

Finite Automata

- Here's a transition diagram describing Pascal identifiers:
- Circles represent states. They represent how much of the input string we have processed.
- Arrows represent transitions from one state to the next, when the character labeling the arrow is matched.
- State 1 is the start state.
- Accepting states are represented by double circles.
**DFA Error States**

- The transition function \( T : S \times \Sigma \to S \) is a function. Hence \( T(s, c) \) must be defined for every state \( s \) and character \( c \).
- But we have ignored any erroneous input. We should have said:

\[
\begin{array}{c}
\text{letter} \\
\text{other} \\
\text{error}
\end{array}
\xrightarrow{\text{letter}}
\begin{array}{c}
\text{letter} \\
\text{other} \\
\text{digit}
\end{array}
\xrightarrow{\text{digit}}
\]

but this would be tedious. Instead, we normally assume that these error transitions always exist.

**Examples**

- Strings with exactly one \( b \):

\[
\begin{array}{c}
1 \\
2
\end{array}
\xrightarrow{\text{not } b}
\begin{array}{c}
1 \\
2
\end{array}
\xrightarrow{\text{not } b}
\]

- Strings with at most one (i.e. 0 or 1) \( b \):

\[
\begin{array}{c}
1 \\
2
\end{array}
\xrightarrow{\text{not } b}
\begin{array}{c}
1 \\
2
\end{array}
\xrightarrow{\text{not } b}
\]

**Finite Automata...**

- Parsing a string of characters using this transition diagram can be indicated by listing the states and transitions used:

\[
\rightarrow 1 \xrightarrow{\text{letter}} 2 \xrightarrow{\text{digit}} 2 \xrightarrow{\text{digit}} 2
\]

- This shows that the string of characters "tmp8" form a legal Pascal identifier.

**DFA**

- A Deterministic Finite Automaton \( M \) consists of
  - An alphabet \( \Sigma \),
  - A set of states \( S \),
  - A transition function \( T : S \times \Sigma \to S \),
  - A start state \( s_0 \in S \),
  - A set of accepting states \( A \subset S \).
- \( T \) records the transitions between states, depending on input:

\[
\begin{array}{c}
x \\
y
\end{array}
\xrightarrow{c}
\begin{array}{c}
y
\end{array}
\]

\( T(x, c) = y \)
C Comments

- C comments are of the form
  ```
  /* ... (no */s) ... */
  ```
- Here's the corresponding transition diagram:

  ![Transition Diagram for C Comments]

Lookahead

- The end of an identifier is reached when the next character is not a letter or digit.
- The string "tmp8*hi;" has two identifiers, terminated by "*" and ";", respectively.
- Here's the corresponding transition diagram:

  ![Transition Diagram for Lookahead]

- [other] means that we're expecting some other character (not letter or digit) as lookahead.

Floating Point Literals

- Transition diagram for natural numbers:

  ![Transition Diagram for Natural Numbers]

- Transition diagram for signed natural numbers:

  ![Transition Diagram for Signed Natural Numbers]

- Transition diagram for signed real numbers:

  ![Transition Diagram for Signed Real Numbers]

- Transition diagram for FP numbers:

  ![Transition Diagram for FP Numbers]
- What if two tokens start with the same character? Note that this is not a DFA since there are three transitions on the same character:

  ![Diagram of NFA for different comparison operators]

- We can break out the offending character:

  ![Expanded NFA diagram]

Towards an NFA

- Here are transition diagrams for recognizing ;=, <=, and =:

  ![Transition diagrams for ;=, <=, and =]

Towards an NFA...

- But, we'd like just one start state, since, at any time during parsing, any token could occur:

  ![Simplified NFA diagram]
NFA

- A Nondeterministic Finite Automaton \( M \) consists of
  - An alphabet \( \Sigma \),
  - A set of states \( S \),
  - A transition function \( T : S \times (\Sigma \cup \{\epsilon\}) \to \mathcal{P}(S) \),
  - A start state \( s_0 \in S \),
  - A set of accepting states \( A \subset S \).
- \( \mathcal{P}(S) \) is the power-set of \( S \), the set of all subsets of \( S \).
- On any transition, we can go to a set of states:
  \[
  T(x, c) = \{y, z\}
  \]

Towards an NFA...

- But, this factoring of states becomes tedious. Instead we can construct a Nondeterministic Finite Automaton (NFA), by adding \( \epsilon \)-transitions:

\[
\begin{align*}
\epsilon & \quad \rightarrow \quad = \quad = \quad = \quad \text{return} :=
\end{align*}
\]

NFA Example

- Consider the following NFA transition diagram:

\[
\begin{align*}
1 & \quad \xrightarrow{a} \quad 2 \quad \xrightarrow{b} \quad \epsilon \quad \xrightarrow{\epsilon} \quad 4 \quad \xrightarrow{\epsilon} \quad 3 \quad \xrightarrow{a} \quad 1
\end{align*}
\]

- \( abb \) is accepted by these moves: \( 1 \xrightarrow{a} 2 \xrightarrow{b} 4 \xrightarrow{\epsilon} 2 \xrightarrow{b} 4 \)
- or by these moves: \( 1 \xrightarrow{a} 3 \xrightarrow{\epsilon} 4 \xrightarrow{\epsilon} 2 \xrightarrow{b} 4 \xrightarrow{\epsilon} 2 \xrightarrow{b} 4 \)
- The NFA accepts \( ab^+|ab^*|b^* \), or, simpler, \((a|\epsilon)b^*\).

\( \epsilon \)-Transitions

- An \( \epsilon \)-transition occurs without consulting the input and without consuming any characters:
Readings and References

- Read Louden, pp. 31–80.
- Or, read the Dragon book, pp. 83–140.