Lexical Analysis, Scanners

- **Function**
  1. Read the input stream (sequence of characters), group the characters into primitives (tokens). Returns token as `<type, value>`.
  2. Throw out certain sequences of characters (blanks, comments, etc.).
  3. Build the symbol table, string table, constant table, etc.
  4. Generate error messages.
  5. Convert, for example, string → integer.

- Tokens are described using regular expressions
  Note: See Lecture 3 on Formal Languages to refresh your knowledge of regular expressions, DFAs, NFAs.

Construction of a Scanner

- **Tools**: state automata and transition diagrams.
- Regular expressions enable the automatic construction of scanners.
- **Scanner generator** (e.g., `Lex`):
  In: Regular expressions.
  Out: Scanner.

- **Environment**:
  
  ![Diagram showing the interaction between Scanner and Parser](image)

How is a Scanner Programmed?

- Describe tokens with regular expressions.
- Draw transition diagrams.
- Code the diagram as table/program.

Example Scanner

- **Example**: Write a scanner for the following tokens.
  Several categories of tokens:
  - `keyword = BEGIN | END`
  - `id = letter (letter | digit)*`
  - `integer = digit+`
  - `op = + | - | * | / | // | | = | :=`

- **Simplification**:
  - Assume that there is a blank character after each token.
  - This simplification can easily be removed!
The Scanner Represents Tokens as Tuples

<table>
<thead>
<tr>
<th>Tuple type</th>
<th>&lt;Typecode, value&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>undefined</td>
<td>&lt;0, 0&gt;</td>
</tr>
<tr>
<td>id</td>
<td>&lt;1, table-pointer&gt;</td>
</tr>
<tr>
<td>integer</td>
<td>&lt;2, value&gt;</td>
</tr>
<tr>
<td>BEGIN</td>
<td>&lt;3, 0&gt;</td>
</tr>
<tr>
<td>END</td>
<td>&lt;4, 0&gt;</td>
</tr>
<tr>
<td>+</td>
<td>&lt;5, 0&gt;</td>
</tr>
<tr>
<td>-</td>
<td>&lt;6, 0&gt;</td>
</tr>
<tr>
<td>*</td>
<td>&lt;7, 0&gt;</td>
</tr>
<tr>
<td>/</td>
<td>&lt;8, 0&gt;</td>
</tr>
<tr>
<td>//</td>
<td>&lt;9, 0&gt;</td>
</tr>
<tr>
<td>=</td>
<td>&lt;10, 0&gt;</td>
</tr>
<tr>
<td>^</td>
<td>&lt;11, 0&gt;</td>
</tr>
<tr>
<td>/</td>
<td>&lt;12, 0&gt;</td>
</tr>
<tr>
<td>&lt;</td>
<td>&lt;13, 0&gt;</td>
</tr>
<tr>
<td>&gt;</td>
<td>&lt;14, 0&gt;</td>
</tr>
<tr>
<td>other</td>
<td>&lt;15, 0&gt;</td>
</tr>
<tr>
<td>accept</td>
<td>&lt;16, 0&gt;</td>
</tr>
<tr>
<td>blank</td>
<td>&lt;17, 0&gt;</td>
</tr>
<tr>
<td>digit</td>
<td>&lt;18, 0&gt;</td>
</tr>
<tr>
<td>not blank</td>
<td>false</td>
</tr>
<tr>
<td>true</td>
<td>false</td>
</tr>
<tr>
<td>false</td>
<td>true</td>
</tr>
<tr>
<td>true</td>
<td>true</td>
</tr>
</tbody>
</table>

4.11

4.9

4.7

4.5

4.3

4.1

3. Interpreting the Table

```c
Token t = new_Token();
int state = 0;
while (1) {
    char ch = getc(inputfile);
    if ch == ' ' goto state0;
    if ch >= '0' && ch <= '9' goto state1;
    state0:
    b) Direct Jumps
    goto stateE; /* in other cases */
    switch (ch) {
        case '0': state = 1; break;
        ... case '9': state = 1; break;
        case ':': state = 0; break;
        default: state = E; break;
    } return t;
    state:
    ...  c) using a Switch statement
    switch (state) {
        case 0:
            switch (ch) {
                case '0': state = 1; break;
                ... case '9': state = 1; break;
                case ':': state = 0; break;
                default: state = E; } break;
                case 1: ...
```
5. Direct Coding of Diagrams (not via a table) Data Structures and Functions

Variables:
- `t->tokentype = current symbol class`
- `value = value`
- `ch = current character`
- `chtyp = vector for 1-character tokens`
- `symtab = symbol table`

Initialization:
- Initialize `chtyp` according to the previous description;
- Initialize the symbol table with reserved words;
- `symtab = symbol table`

Functions:
- `getc
- `skip_blanks`
- `symtab_lookup(id)`
- `is_letter(ch)`
- `is_digit(ch)`

Diagram with simplification removed

Removed simplification:
- Space is not necessary as concluding character

Scanner Lookahead Problems

- Lookahead is sometimes needed to determine symbol type.
  - Example: in FORTRAN
    - `DO 10 I = 1.25` is an assignment, but
    - `DO 10 I = 1,25` is a for-statement.
      It is `.` or `=`, which determines whether the scanner returns `DO10I` or `DO`.
  - Another Example: in Pascal.
    Two character lookahead needed
    - 715..816

5. Scanner Fragment with Direct Coding Continued

```c
Token getNextToken( void )
{
    char ch = getc( inputfile );
    char idstr[ ... ];  // lexeme buffer for identifiers
    t = new_Token();
    while(i
    ch = getc ( inputfile );  // eat whitespace
    if(i
    while(i
        append( ch, idstr );
        ch = getc( inputfile );
    }

    return t;
}
```