

2. Lexical Analysis

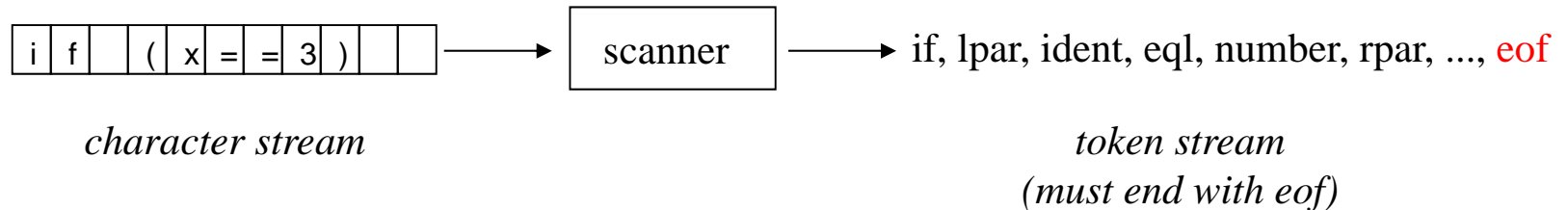
2.1 Tasks of a Scanner

2.2 Regular Grammars and Finite Automata

2.3 Scanner Implementation

Tasks of a Scanner

1. Recognizes tokens



2. Skips meaningless characters

- blanks
- tabulator characters
- end-of-line characters (CR, LF)
- comments

Why is Scanning not Part of Parsing?

Tokens have a syntactical structure, e.g.

```
ident = letter {letter | digit | '_'}.  
number = digit {digit}.  
if =      "{" "if".  
eq =      "=" "=".  
...
```

Why is scanning not part of parsing?

E.g., why is *ident* considered to be a terminal symbol and not a nonterminal symbol?

Why is Scanning not Part of Parsing?

It would make parsing more complicated

(e.g. difficult distinction between keywords and names)

```
Statement = ident "=" Expr ";"
           | "if" "(" Expr ")" ... .
```

One would have to write this as follows:

```
Statement = "i" ( "f" "(" Expr ")" ...
                 | notF {letter | digit} "=" Expr ";"
                 )
           | notI {letter | digit} "=" Expr ";".
```

The scanner must eliminate blanks, tabs, end-of-line characters and comments

(these characters can occur anywhere => would lead to very complex grammars)

```
Statement = "if" {Blank} "(" {Blank} Expr {Blank} ")" {Blank} ... .
Blank = " " | "\r" | "\n" | "\t" | Comment.
```

Tokens can be described with regular grammars

(simpler and more efficient than context-free grammars)

2. Lexical Analysis

2.1 Tasks of a Scanner

2.2 Regular Grammars and Finite Automata

2.3 Scanner Implementation

Regular Grammars



Definition

A grammar is called regular if it can be described by productions of the form:

$$\begin{array}{ll} X = a. & a, b \in TS \\ X = b Y. & X, Y \in NTS \end{array}$$

Example Grammar for identifiers

Ident = letter.
Ident = letter Rest.
Rest = letter.
Rest = digit.
Rest = '_'.
Rest = letter Rest.
Rest = digit Rest.

e.g., derivation of the name xy3

Ident \Rightarrow letter Rest \Rightarrow letter letter Rest \Rightarrow letter letter digit

Alternative definition

A grammar is called regular if it can be described by a single non-recursive EBNF production.

Example Grammar for identifiers

Ident = letter {letter | digit | '_'}.

Examples



Can we transform the following grammar into a regular grammar?

$E = T \{ "+" T \}.$
 $T = F \{ "*" F \}.$
 $F = \text{id}.$

A large, empty rectangular box with a black border, intended for the user to write their answer to the question.

Can we transform the following grammar into a regular grammar?

$E = F \{ "*" F \}.$
 $F = \text{id} \mid "(" E ")".$

A large, empty rectangular box with a black border, intended for the user to write their answer to the question.

Limitations of Regular Grammars



Regular grammars cannot deal with *nested structures*

because they cannot handle *central recursion*!

But central recursion is important in most programming languages

- nested expressions $\text{Expr} \Rightarrow * \dots "(" \text{Expr} ")" \dots$
- nested statements $\text{Statement} \Rightarrow \text{"do" Statement "while" "(" Expr ")"}$
- nested classes $\text{Class} \Rightarrow \text{"class" "{"} \dots \text{Class} \dots \text{"}"}$

For productions like these we need context-free grammars

But most lexical structures are regular

identifiers	letter {letter digit}
numbers	digit {digit}
strings	"\" {noQuote} "\"
keywords	letter {letter}
operators	">" "="

Exception: nested comments

`/* /* ... */ */`

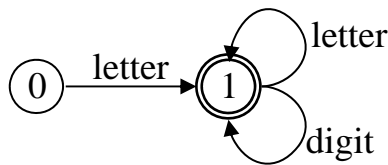
The scanner must treat them in a special way

Deterministic Finite Automaton (DFA)



Can be used to analyze regular languages

Example



○ final state
start state is always
state 0 by convention

State transition function as a table

δ	letter	digit
s0	s1	error
s1	s1	s1

"finite", because δ
can be written down
explicitly

Definition

A deterministic finite automaton is a 5 tuple (S, I, δ, s_0, F)

- S set of states
- I set of input symbols
- $\delta: S \times I \rightarrow S$ state transition function
- s_0 start state
- F set of final states

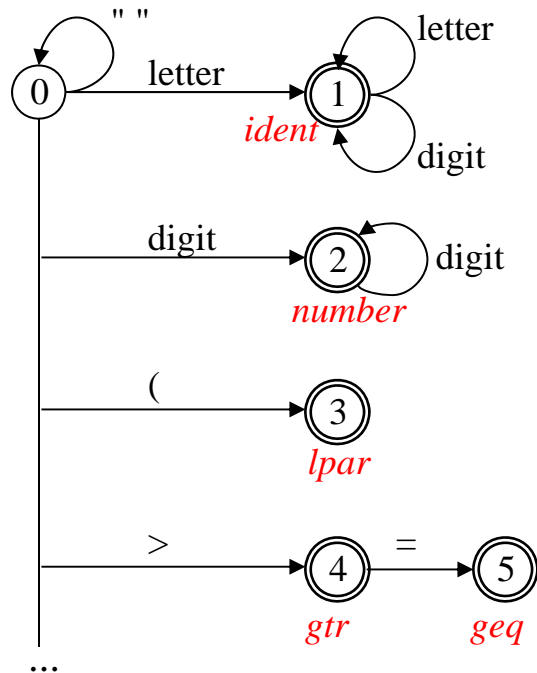
The **language** recognized by a DFA is
the set of all symbol sequences that lead
from the start state into one of the
final states

A DFA has recognized a sentence

- if it is in a final state
- and if the input is totally consumed or there is no possible transition with the next input symbol

The Scanner as a DFA

The scanner can be viewed as a big DFA



Example input: max >= 30

$s_0 \xrightarrow{m} s_1 \xrightarrow{a} s_1 \xrightarrow{x} s_1$

- no transition with " " in s_1
- *ident* recognized

$s_0 \xrightarrow{" "} s_0 \xrightarrow{>} s_4 \xrightarrow{=} s_5$

- skips blanks at the beginning
- does not stop in s_4
- no transition with " " in s_5
- *geq* recognized

$s_0 \xrightarrow{" "} s_0 \xrightarrow{3} s_2 \xrightarrow{0} s_2$

- skips blanks at the beginning
- no transition with " " in s_2
- *number* recognized

After every recognized token the scanner starts in s_0 again

2. Lexical Analysis

2.1 Tasks of a Scanner

2.2 Regular Grammars and Finite Automata

2.3 Scanner Implementation

Scanner Interface



```
class Scanner {  
    static void    init (Reader r) {...}  
    static Token  next () {...}  
}
```

For efficiency reasons methods are static
(there is just one scanner per compiler)

Initializing the scanner

```
InputStream s = new FileInputStream("myfile.mj");  
Reader r = new InputStreamReader(s);  
Scanner.init(r);
```

Reading the token stream

```
for (;;) {  
    Token t = Scanner.next();  
    ...  
}
```

Tokens

```
class Token {
    int kind;        // token code
    int line;        // token line (for error messages)
    int col;         // token column (for error messages)
    int val;         // token value (for number and charCon)
    String string;   // token string
}
```

Token codes for MicroJava

<u>error token</u>	<u>token classes</u>	<u>operators and special characters</u>		<u>keywords</u>	<u>end of file</u>
static final int					
none = 0,	ident = 1, number = 2, charCon = 3,	plus = 4, /* + */ minus = 5, /* - */ times = 6, /* * */ slash = 7, /* / */ rem = 8, /* % */ eq = 9, /* == */ neq = 10, /* != */ lss = 11, /* < */ leq = 12, /* <= */ gtr = 13, /* > */ geq = 14, /* >= */	assign = 15, /* = */ semicolon = 16, /* ; */ comma = 17, /* , */ period = 18, /* . */ lpar = 19, /* (*/ rpar = 20, /*) */ lbrack = 21, /* [*/ rbrack = 22, /*] */ lbrace = 23, /* { */ rbrace = 24, /* } */	class_ = 25, else_ = 26, final_ = 27, if_ = 28, new_ = 29, print_ = 30, program_ = 31, read_ = 32, return_ = 33, void_ = 34, while_ = 35,	eof = 36;

Scanner Implementation

Static variables in class *Scanner*

```
static Reader in;           // input stream
static char ch;             // next input character (still unprocessed)
static int line, col;       // line and column number of the character ch
static final int eofCh = '\u0080'; // character that is returned at the end of the file
```

init()

```
public static void init (Reader r) {
    in = r;
    line = 1; col = 0;
    nextCh(); // reads the first character into ch and increments col to 1
}
```

nextCh()

```
private static void nextCh() {
    try {
        ch = (char) in.read(); col++;
        if (ch == '\n') { line++; col = 0; }
        else if (ch == '\uffff') ch = eofCh;
    } catch (IOException e) { ch = eofCh; }
}
```

- *ch* = next input character
- returns *eofCh* at the end of the file
- increments *line* and *col*

next()

```

public static Token next() {
    while (ch <= ' ') nextCh(); // skip blanks, tabs, eols
    Token t = new Token(); t.line = line; t.col = col;
    switch (ch) {
        case 'a': case 'b': ... case 'z': case 'A': case 'B': ... case 'Z':
            readName(t); break;
        case '0': case '1': ... case '9':
            readNumber(t); break;
        case ';': nextCh(); t.kind = semicolon; break;
        case '.': nextCh(); t.kind = period; break;
        case eofCh: t.kind = eof; break; // no nextCh() any more
        ...
        case '=': nextCh();
            if (ch == '=') { nextCh(); t.kind = eql; } else t.kind = assign;
            break;
        ...
        case '/': nextCh();
            if (ch == '/') {
                do nextCh(); while (ch != '\n' && ch != eofCh);
                t = next(); // call scanner recursively
            } else t.kind = slash;
            break;
        default: nextCh(); t.kind = none; break;
    }
    return t;
} // ch holds the next character that is still unprocessed

```

} names, keywords
 } numbers
 } simple tokens
 } compound tokens
 } comments
 } invalid character

Further Methods

private static void readName(Token t)

- At the beginning *ch* holds the first letter of the name
- Reads further letters and digits and stores them in *t.string*
- Looks up the name in a keyword table (using hashing or binary search)
if found: *t.kind = token number of the keyword;*
otherwise: *t.kind = ident;*
- At the end *ch* holds the first character after the name

private static void readNumber(Token t)

- At the beginning *ch* holds the first digit of the number
- Reads further digits, converts them into a number and stores the number value to *t.val*.
if overflow: report an error
- *t.kind = number;*
- At the end *ch* holds the first character after the number

Further Methods

private static void readCharCon(Token t)

- At the beginning *ch* holds a single quote
- Reads further characters and stores them in *t.string*
- At the end *ch* holds the first character after the closing quote
- Sets the following token fields:
 - t.kind* = *charCon*;
 - t.val* = *numeric char value*;

valid char constants

'x'
'\r'
'\n'
'\t'

invalid char constants

'xy'
"
'x
↑

Scanner reports an error,
but returns a *charCon*