Lexer: Handcodierte Implementierung

Carsten Gips (HSBI)

Lexer: Erzeugen eines Token-Stroms aus einem Zeichenstrom

```
/* demo */
a= [5  , 6]     ;
```

```
<ID, "a"> <ASSIGN> <LBRACK> <NUM, 5> <COMMA> <NUM, 6> <RBRACK> <SEMICOL>
```

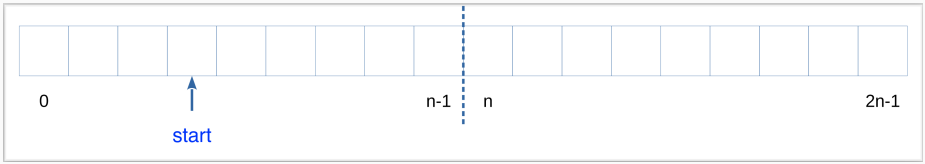
Manuelle Implementierung: Rekursiver Abstieg

```
def nextToken():
    while (peek != EOF):  # globale Variable, über consume()
        switch (peek):
            case ' ': case '\t': case '\n': WS(); continue
            case '[': consume(); return Token(LBRACK, '[')
            ...
            default:
                if isLetter(peek): return NAME()
                raise Error("invalid character: "+peek)
    return Token(EOF_Type, "<EOF>")

def WS():
    while (peek == ' ' || peek == '\t' || ...): consume()

def NAME():
    buf = StringBuilder()
    do { buf.append(peek); consume(); } while (isLetter(peek))
    return Token(NAME, buf.toString())
```

Read-Ahead: Unterscheiden von “<” und “<=”

```
def nextToken():
    while (peek != EOF):  # globale Variable
        switch (peek):
            case '<':
                if match('='): consume(); return Token(LE, "<=")
                else: consume(); return Token(LESS, '<')
            ...
    return Token(EOF_Type, "<EOF>")

def match(c):   # Lookahead: Ein Zeichen
    consume()
    if (peek == c): return True
    else: rollBack(); return False
```

Puffern des Input-Stroms: Double Buffering




```
start = 0; end = 0; fill(buffer[0:n])

def consume():
    peek = buffer[start]
    start = (start+1) mod 2n
    if (start mod n == 0):
        fill(buffer[start:start+n-1])
        end = (start+n) mod 2n

def rollBack():
    if (start == end): raise Error("roll back error")
    start = (start-1) mod 2n
```

Typische Muster für Erstellung von Token

1. Schlüsselwörter
   - Ein eigenes Token (RE/DFA) für jedes Schlüsselwort, oder
   - Erkennung als Name und Vergleich mit Wörterbuch
2. Operatoren
   - Ein eigenes Token für jeden Operator, oder
   - Gemeinsames Token für jede Operatoren-Klasse
3. Bezeichner: Ein gemeinsames Token für alle Namen
4. Zahlen: Ein gemeinsames Token für alle numerischen Konstante
5. String-Literale: Ein gemeinsames Token
6. Komma, Semikolon, Klammern, ...: Je ein eigenes Token
7. Regeln für White-Space und Kommentare etc. ...

Fehler bei der Lexikalischen Analyse

```
fi (a==42) { ... }
```

=> Was tun, wenn keines der Pattern auf den Anfang des Eingabestroms passt?

Fehler bei der Lexikalischen Analyse

```
fi (a==42) { ... }
```

=> Was tun, wenn keines der Pattern auf den Anfang des Eingabestroms passt?

- Aufgeben ...
- "Panic Mode": Entferne so lange Zeichen, bis ein Pattern passt.
- Ein-Schritt-Transformationen:
  - Füge fehlendes Zeichen in Eingabestrom ein.
  - Entferne ein Zeichen aus Eingabestrom.
  - Vertausche ein Zeichen:
    - Ersetze ein Zeichen durch ein anderes.
    - Vertausche zwei benachbarte Zeichen.
- Fehler-Regeln: Matche typische Typos

Wrap-Up

- Zusammenhang DFA, RE und Lexer
- Implementierungsansatz: Manuell codiert (rekursiver Abstieg)
- Read-Ahead
- Puffern mit Doppel-Puffer-Strategie
- Typische Fehler beim Scannen

LICENSE