Syntaxanalyse mit Rekursivem Abstieg

Satz: a e g c f d b

```
G(S):  S = aAb | cAd.
       A = eB | f.
       B = gS.
```

```
next()-->a   erkenne S
                 erkenne a oder c (a erkannt, wähle erste Alternative)
next()-->e       erkenne A
                     erkenne e oder f (e erkannt, wähle 1. Alt.)
next()-->g           erkenne B
                         erkenne g (g erkannt)
next()-->c               erkenne S
                             erkenne a oder c (c erkannt, wähle 2. Alt.)
next()-->f                   erkenne A
                                 erkenne e oder f (f erkannt, wähle 2. Alt.)
next()-->d                   (A erkannt)
                             erkenne d (d erkannt)
next()-->b               (S erkannt)
                     (B erkannt)
                 (A erkannt)
                 erkenne b (b erkannt)
             (S erkannt)
```

Parser: wichtige Vars & Methoden


```
static Token t;    // last recognized token
static Token la;   // look ahead token
static int sym;    // kind of look ahead token

static void scan () {
  t = la; la = Scanner.next(); sym = la.kind;
}

static void check (int expected) {
  if (sym == expected) scan();
  else error(Messages.getString(expected)+" expected",
                                null, "");
}

static void error (String prefix, String msgKey,
                   String postfix) {
    StringBuffer msg = new StringBuffer();
    msg.append("-- line "); ...
    out.println(msg.toString());
    // Panic mode: Abbruch beim ersten Fehler
    // (s. nächste Folie)
}
```

Panic Mode

- beim ersten gefundenen Fehler wird Analyse abgebrochen
- Abbruch (in den UEs) **nicht** mit ~~`System.exit(0);`~~ , weil
  - dadurch die VM beendet wird
  - das beim Testen mit JUnit zum sofortigen Abbruch führt, d.h.
    - es werden keine weiteren Test ausgeführt
    - es wird kein Ergebnis angezeigt bzw. das GUI wird sofort beendet
- besser:

```
throw new Error(Messages.getString("PANIC_MODE"));
```

(oder `throw new junit.AssertionFailedError(...);`)

- wird von JUnit (als *Error* bzw. *Failure*) abgefangen
  - Tests, GUI & Compiler laufen weiter und können geordnet terminieren
- kann auch in Testfällen abgefangen werden:

```
try { Parser.parse(output); }
catch (Error e) {
  assertEquals(Messages.getString("PANIC_MODE"),
               e.getMessage());
}
```

Zuordnung: Tokencode ⇔ Namen

In *messages.properties:*

```
#----- names for token
Token.0       = none
Token.1       = identifier
Token.2       = number
...
Token.42      = end of file
```

In *Messages.java:*

```
public static String getString (int tokenKind) {
  try {
    return RESOURCE_BUNDLE.getString("Token." +
                                     tokenKind);
  } catch { ... }
}
```

Zugriff auf Tokennamen:

```
Messages.getString(Token.ident)  →  "identifier"
```

Bsp 1: S = a B c.

SEQUENZ

```
static void S () {
  check(a);
  B();
  check(c);
}
```

Bsp 2: S = a | B c | d.

ALTERNATIVEN

first(B) = { e, f }

```
static void S () {
  switch (sym) {
    case a: case d: scan(); break;
    case e: case f:
      // Erkennung von e und f in B!
      B(); check(c); break;
    default: error(...);
  }
}
```

Bsp 3: S = (a | B) c.

SEQUENZ mit ALTERNATIVE

first(B) = { e, f }

```
static void S () {
  switch (sym) {
    case a:
      scan(); break;
    case e: case f:

      B(); break;
    default:
      error(...);
  }
  check(c);
}
```

ODER:

```
if (sym == a)

  scan();
else if (sym == e ||
                sym == f)
  B();
else
  error(...);

check(c);
```

Bsp 4: **S = [a | B] c.**

SEQUENZ mit OPTIONALER ALTERNATIVE

first(B) = { e, f }

```
static void S () {
  switch (sym) {
    case a:
      scan(); break;
    case e: case f:

      B(); break;
  } // KEIN Fehler!
  check(c);
}
```

ODER:

```
if (sym == a)

scan();
else if (sym == e ||
                sym == f)
  B();
// kein else error...!
check(c);
```

Bsp 5: S = { a | B } c. (1)

SEQUENZ mit OPTIONALER ITERATION

first(B) = { e, f }

```
static void S () {
  while (sym == a || sym == e || sym == f) {
    switch (sym) {
      case a: scan(); break;
      case e: case f: B(); break;
    } // kein default nötig!
  }
  check(c);
}
```

Bsp 5: S = { a | B } c. (2)

SEQUENZ mit OPTIONALER ITERATION

first(B) = { e, f }

```
static void S () {
  while (sym != c) {
    switch (sym) {
      case a: scan(); break;
      case e: case f: B(); break;
      default:  // default Zweig hier nötig!
        error(...);
    }
  }
  scan();
}
```

Bsp 6: S = a { B } C.

first(B) = { e, f }
first(C) = ?

```
static void S () {
  check(a);
  while (sym == e || sym == f) B();
  C();
}
```

UE 3: Syntaxanalyse (Parser)

- Testen
  - siehe Testanleitung auf Homepage!
  - in *UB-UE3-Angabe.zip*:
    - messages.properties & Messages.java (neu! mit Tokennamen und Zugriffmethode für Token.kind), Token.java (neu! ohne names-Array), Gerüst für **Parser.java**, ...
    - Testklassen (AllTests.java (neu), **ParserTest.java**, ...)
  - In *MJ-Programs.zip*:
    - Beispiel MicroJava-Programme (AllProds.mj, ScriptExample.mj, ...)
- Abgabe
  - siehe Abgabeanleitung auf Homepage!