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 Felder & 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("TOKEN_EXPECTED", Token.kindToString(expected));
    }
}

public static void error (String msgKey, Object... msgParams) {
    Errors.error(la.line, la.col, msgKey, msgParams);
    // panic mode
    throw new Error(Messages.getString("PANIC_MODE"));
}
```

Panic Mode

- beim ersten gefundenen Fehler wird Analyse abgebrochen
- Abbruch 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
- Bessere Lösung:
  `throw new Error(``Messages.getString("PANIC_MODE")`**`);`**
  - Error wird von den Testfällen gefangen und ausgewertet

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:
      scan(); break;
    case e: case f:
      // Erkennung von e und f in B!
      B(); check(c); break;
    case d:
      scan(); 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 error
  check(c);
}
```

ODER:

```
if (sym == a) {

  scan();
} else if (sym == e ||
                  sym == f) {
  B();
}
// kein 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) {
    if (sym == a) {
      scan();
    } else {
      B();
    } // kein error
  }
  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*)

- Keine neuen Angabe- und Test-Klassen

- Abgabe
  - siehe Abgabeanleitung auf Homepage!
  - elektronisch bis Mi, 09.11.2005, 20:15
    - alle zum Ausführen benötigten Dateien
  - auf Papier bis Mi, 09.11.2005, 20:15
    - nur Parser.java