Syntaxanalyse mit Rekursivem Abstieg

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

Satz: a e g c f d b

```
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


```
private Token t;              // last recognized token
private Token la;             // lookahead token
private Token.Kind sym;       // kind of lookahead token
public Scanner scanner;       // reference to Scanner

private void scan () {
   t = la; la = scanner.next(); sym = la.kind;
}

private void check (Token.Kind expected) {
   if (sym == expected) {
      scan();
   } else {
      error(TOKEN_EXPECTED, expected);
   }
}

public void error (Message msg, Object... msgParams) {
   scanner.errors.error(la.line, la.col, msg, msgParams);
   // panic mode
   throw new Errors.PanicMode();
}
```

Panic Mode

- Beim ersten Fehler Analyse abbrechen
- Abbruch mit **throw new Errors.PanicMode();**
  - Error wird von den Testfällen gefangen und ausgewertet
- **Nicht System.exit(0); ,** weil es
  - VM beendet
  - JUnit Testlauf unterbricht

Bsp 1: S = a B c.

SEQUENZ

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

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

ALTERNATIVEN

first(B) = { e, f }

```
private 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 }

```
private 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 }

```
private 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 ITERATION

first(B) = { e, f }

```
private 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 ITERATION

first(B) = { e, f }

```
private void S () {
  while (true) {
    if (sym == a) {
      scan();
    } else if (sym == e || sym == f) {
      B();
    } else {
      break; // Schleife verlassen
    }
  }
  check(c);
}
```

Bsp 6: S = B { a B }. (1)

SEQUENZ mit ITERATION

first(B) = { e, f }

```
private void S () {
   B();
   while (sym == a) {
      scan();
      B();
   }
}
```

Bsp 6: S = B { a B }. (2)

SEQUENZ mit ITERATION

first(B) = { e, f }

```
private void S () {
  while (true) {
    B();
    if (sym == a) {
      scan();
    } else {
      break;
    }
  }
}
```

Bsp 7: S = a { B } C.

first(B) = { e, f }
first(C) = { g, h }

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

UE 3: Syntaxanalyse (*Parser*)

- Neue Test-Klasse: ParserErrorTest

- Abgabe
  - elektronisch bis Mi, 11.11.2009, 18:00
    - alle zum Ausführen benötigten Dateien
  - auf Papier
    - nur Parser.java