F2

Concrete and abstract grammars

Lennart Andersson

Revision 2009-03-17

2009

Concrete grammar for arithmetic expressions - Canonical

$$
\begin{array}{ll}
\mathit{expr} & \rightarrow \mathit{number} \\
\mathit{expr} & \rightarrow \mathit{expr} + \mathit{expr} \\
\mathit{expr} & \rightarrow \mathit{expr} * \mathit{expr} \\
\mathit{number} & \rightarrow \mathit{digit} \\
\mathit{number} & \rightarrow \mathit{digit}\ \mathit{number} \\
\mathit{digit} & \rightarrow 0 \\
\ldots & \rightarrow \\
\mathit{digit} & \rightarrow 9
\end{array}
$$

Concrete grammar for arithmetic expressions - EBNF

$$
\begin{array}{ll}
\mathit{expr} & \rightarrow \mathit{number} \mid \mathit{expr}\ +\ \mathit{expr} \mid \mathit{expr}\ *\ \mathit{expr} \\
\mathit{number} & \rightarrow \mathit{digit}\ \mathit{digit}^{*} \\
\mathit{digit} & \rightarrow 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9
\end{array}
$$

Derivation of 2+3*4

$$
\begin{aligned}
&\mathit{expr} \Rightarrow \mathit{expr} + \mathit{expr} \Rightarrow \mathit{number} + \mathit{expr} \Rightarrow \mathit{digit} + \mathit{expr} \\
&\Rightarrow 2 + \mathit{expr} \Rightarrow 2 + \mathit{expr} * \mathit{expr} \Rightarrow 2 + \mathit{number} * \mathit{expr} \\
&\Rightarrow 2 + \mathit{digit} * \mathit{expr} \Rightarrow 2 + 3 * \mathit{expr} \Rightarrow 2 + 3 * \mathit{number} \\
&\Rightarrow 2 + 3 * \mathit{digit} \Rightarrow 2 + 3 * 4
\end{aligned}
$$

Abstract representation in Java

```
interface Expr {
    int value();
}

class Add implements Expr {
    Expr expr1, expr2;
    Add(Expr expr1, Expr expr2) {
        this.expr1 = expr1;
        this.expr2 = expr2;
    }
    int value() {
        return expr1.value() + expr2.value();
    }
}
```

Java representation – 2

```
class Mul implements Expr {
    Expr expr1, expr2;
    int value() {
        return expr1.value() * expr2.value();
    }
}

class Int implements Expr {
    int value;
    int value() {
        return value;
    }
}
```


Abstract representation in Haskell

```
data Expr = Num Integer | Add Expr Expr | Mul Expr Expr
```

The semantics is defined by

```
value (Num n) = n
value (Add expr1 expr2) = value expr1 + value expr2
value (Mul expr1 expr2) = value expr1 * value expr2
```


Language operations

- $w \in L$
- $L_1 \cup L_2$
- $L_1 \cap L_2$
- $L_1 \cdot L_2$ or $L_1 L_2$
- L^n where $n \geq 0$
- L^*

Concrete grammar for regular expressions

$expr \longrightarrow \emptyset$
$expr \longrightarrow \epsilon$
$expr \longrightarrow \sigma$, where $\sigma \in \Sigma$
$expr \longrightarrow expr \cdot expr$
$expr \longrightarrow expr$ `"|"` $expr$
$expr \longrightarrow expr^*$
$expr \longrightarrow$ `"("` $expr$ `")"`

Σ is an alphabet. Notice that $\cdot$ and $|$ and * are symbols. The $\cdot$ is usually omitted.

Semantics of regular expressions

$$
\begin{aligned}
\mathcal{L}[\![\emptyset]\!] &= \{\ \} \\
\mathcal{L}[\![\epsilon]\!] &= \{\,\epsilon\,\} \\
\mathcal{L}[\![\sigma]\!] &= \{\,\sigma\,\}, \qquad \sigma \in \Sigma \\
\mathcal{L}[\![\alpha\,\beta]\!] &= \mathcal{L}[\![\alpha]\!]\mathcal{L}[\![\beta]\!] \\
\mathcal{L}[\![\alpha \mid \beta]\!] &= \mathcal{L}[\![\alpha]\!] \cup \mathcal{L}[\![\beta]\!] \\
\mathcal{L}[\![\alpha^*]\!] &= (\mathcal{L}[\![\alpha]\!])^*
\end{aligned}
$$

Peano's axioms

1. $0 \in \mathbb{N}$.
2. $n \in \mathbb{N} \Rightarrow n' \in \mathbb{N}$.
3. $n \in \mathbb{N} \Rightarrow n' \neq 0$.
4. If $n, m \in \mathbb{N}$ then $n' = m' \Rightarrow n = m$.
5. If
   - $A \subseteq \mathbb{N}$
   - $0 \in A$
   - $n \in A \Rightarrow n' \in A$

   then $A = \mathbb{N}$.

Java representation

```
abstract class N {
  abstract N add(N m);
  abstract N mul(N m);
}

class Zero extends N {
  N add(N m) {
    return m;
  }
  N mul(N m) {
    // omission
  }
}
```


. . . Java representation

```
class Suc extends N {
  private N n;
  Suc(N n) {
    this.n = n;
  }
  N add(N m) {
    return new Suc(n.add(m));
  }
  N mul(N m) {
    // omission
  }
}
```

Haskell representation

```
data N = Zero | Suc N
  deriving Eq

(Zero == Zero) = True
(Suc n == Suc m) = n==m
(_ == _) = False
```


Induction over $\mathbb{N}$

Theorem. *Let $p \in \mathbb{N} \to \mathbb{B}$ be a property. If*

1. *$p(\mathit{Zero})$ is true and*
2. *$p(k)$ implies $p(\mathit{Suc}\ k)$ for every $k \in \mathbb{N}$*

then $p(n)$ is true for all $n \in \mathbb{N}$. ■

Induction over Expr

Theorem. *Let $p \in \mathit{Expr} \to \mathbb{B}$ be a property. If*

1. *$p(\mathit{Num}\ n)$ is true for all n.*
2. *$p(e1)$ and $p(e2)$ implies $p(\mathit{Add}\ e1\ e2)$ for every $e1,\ e2 \in \mathit{Expr}$*
3. *$p(e1)$ and $p(e2)$ implies $p(\mathit{Mul}\ e1\ e2)$ for every $e1,\ e2 \in \mathit{Expr}$*

then $p(e)$ is true for all $e \in \mathit{Expr}$. ■

Mirror example

```
value :: Expr -> Integer
value (Num n) = n
value (Add expr1 expr2) = value expr1 + value expr2
value (Mul expr1 expr2) = value expr1 * value expr2

mirror :: Expr -> Expr
mirror (Num i) = Num i
mirror (Add e1 e2) = Add (mirror e2) (mirror e1)
mirror (Mul e1 e2) = Mul (mirror e2) (mirror e1)
```

Application of the principle

Theorem. `value` e = (`mirror` e) *for all* e *in* *Expr*. ■

Induction principle

Theorem. *Let* $p \in \mathit{Expr} \to \mathbb{B}$ *be a property. If*

1. $p(\mathtt{Num}\ n)$ *is true for all* n.
2. $p(e1)$ *and* $p(e2)$ *implies* $p(\mathtt{Add}\ e1\ e2)$ *for every* $e1,\ e2 \in \mathit{Expr}$
3. $p(e1)$ *and* $p(e2)$ *implies* $p(\mathtt{Mul}\ e1\ e2)$ *for every* $e1,\ e2 \in \mathit{Expr}$

then $p(e)$ *is true for all* $e \in \mathit{Expr}$. ■