

# *Algebraic Compilers*

## *and their implementation in Haskell*

Peter Padawitz

TU Dortmund

January 25, 2008

## Basics

- D. Knuth, *Semantics of Context-Free Languages*, Mathematical Systems Theory 2 (1968)
- J.A. Goguen, J.W. Thatcher, E.G. Wagner, J.B. Wright, *Initial Algebra Semantics and Continuous Algebras*, Journal of the ACM 24 (1977)
- J.W. Thatcher, E.G. Wagner, J.B. Wright, *More on Advice on Structuring Compilers and Proving Them Correct*, Theoretical Computer Science 15 (1981)
- AND MORE

## Projects

- M.G.J. van den Brand, J. Heering, P. Klint, P.A. Olivier, *Compiling Rewrite Systems: The ASF+SDF Compiler*, ACM TOPLAS 24 (2002)
- E. Visser, *Program Transformation with Stratego/XT: Rules, Strategies, Tools, and Systems*, in: C. Lengauer et al., eds., Domain-Specific Program Generation, Springer LNCS 3016 (2004)
- J. Meseguer, G. Rosu, *The Rewriting Logic Semantics Project*, Theoretical Computer Science 373 (2007)
- AND MORE

✿ **Extended CF grammar**

- ✿ Language(s) generated by an ECFG
- ✿ Proof of  $L_1 = L(G)_S$
- ✿ CFGs and ECFGs are equivalent

✿ Sorted sets and functions

✿ Signature  $\Sigma$

✿ Abstract syntax

✿  **$\Sigma$ -algebras**

✿  $T_\Sigma$  is a  $\Sigma$ -algebra

✿  $\Sigma$ -terms as hierarchical lists form a  $\Sigma$ -algebra

✿ The state model of *JavaGra* is a *JavaSig*-algebra

- ✿  $T_\Sigma$  is the initial  $\Sigma$ -algebra
- ✿ Parsers for regular expressions
- ✿ Parser into  $T_{\Sigma(G)}$
- ✿ **Generic parser into any  $\Sigma(G)$ -algebra**
- ✿ *JavaGra*-Parser into any *JavaSig*-algebra

### ✿ **Monadic parsers**

- ✿ Monadic parsers for a rule  $A \rightarrow e$
- ✿ Monadic parsers for regular expressions
- ✿ Monadic *JavaGra*-parser into any *JavaSig*-algebra
- ✿ Attributed  $\Sigma$ -algebras
- ✿ **Multi-pass compilers**

✿ Conclusion

## \* Extended CF grammar (ECFG)

An **extended CF grammar**  $G = (N, T, P, S)$  consists of

- a finite set  $N$  of **nonterminals**,
- a finite set  $T$  of **terminals**,
- a finite set  $P$  of **productions** or **rules** of the form  $A \rightarrow e$  with  $A \in N$  and  $e \in \text{Reg}(N \cup T)$  such that  $e$  is **in disjunctive normal form** and for each  $A \in N$ ,  $P$  contains exactly one rule  $A \rightarrow e$ ,
- a **start symbol**  $S \in N$ .

*Why  $e$  in DNF?*

Because each sum expression  $e_1 | \dots | e_n$  defines a datatype and thus must be named (by a nonterminal).

## ✿ Language(s) generated by an ECFG

Let  $N = \{A_1, \dots, A_n\}$ ,  $P = \{A_1 \rightarrow e_1, \dots, A_n \rightarrow e_n\}$  und  $S = A_1$ .

The **language derived by  $G$**

$$L(G) = \{L(G)_{A_1}, \dots, L(G)_{A_n}\}$$

is the **least solution of the system of equations**

$$A_1 = e_1, \dots, A_n = e_n, \tag{1}$$

i.e. the least tuple  $(L_1, \dots, L_n) \in \wp(T^*)^n$  such that the equations

$$L_1 = L(e_1)[L_1/A_1, \dots, L_n/A_n], \dots, L_n = L(e_n)[L_1/A_1, \dots, L_n/A_n]$$

hold true.  $[L_1/A_1, \dots, L_n/A_n]$  denotes the substitution of  $A_i$  by  $L_i$ .

For some nonterminals  $A$ , there are no rules  $A \rightarrow e$ , but the language  $L(G)_A$  is predefined, e.g.  $L(G)_{Int} =_{def} \mathbb{Z}$ . (1) is then extended by

$$A = L(G)_A.$$

## Example JavaGra

$$\begin{aligned} \textit{Block} &\rightarrow \{ \textit{Command}^* \} \\ \textit{Command} &\rightarrow ; \mid \textit{String} = \textit{IntE}; \mid \textit{if} (\textit{BoolE}) \textit{Block} \mid \\ &\quad \textit{if} (\textit{BoolE}) \textit{Block} \textit{else} \textit{Block} \mid \\ &\quad \textit{while} (\textit{BoolE}) \textit{Block} \\ \textit{IntE} &\rightarrow \textit{Int} \mid \textit{String} \mid (\textit{IntE}) \mid \textit{IntE} - \textit{IntE} \mid \\ &\quad \textit{IntE} (+\textit{IntE})^+ \mid \textit{IntE} (*\textit{IntE})^+ \\ \textit{BoolE} &\rightarrow \textit{Bool} \mid \textit{IntE} > \textit{IntE} \mid ! \textit{BoolE} \end{aligned}$$

The languages for *Int*, *String* und *Bool* are predefined, e.g., as the synonymous standard Haskell types.

An element of  $L(\textit{JavaGra})$ :

```
{fact = 1; while (x > 0) {fact = fact*x; x = x-1;}}
```

✿ **Proof of  $L_1 = L(G)_S$**

Let  $N = \{A_1, \dots, A_n\}$ ,  $P = \{A_1 \rightarrow e_1, \dots, A_n \rightarrow e_n\}$  and  $S = A_1$ .  $L_1$  is given.

1. (*Generalization*) Find languages  $L_2, \dots, L_n \subseteq T^*$  that should satisfy  $L_2 = L(G)_{A_2}, \dots, L_n = L(G)_{A_n}$ .
2. (*Soundness*) Show that  $(L_1, \dots, L_n)$  solves

$$A_1 = e_1, \dots, A_n = e_n.$$

This implies  $L(G)_{A_1} \subseteq L_1, \dots, L(G)_{A_n} \subseteq L_n$ .

3. (*Completeness*) Show the inverse  $L_1 \subseteq L(G)_{A_1}, \dots, L_n \subseteq L(G)_{A_n}$ .



## ✿ CFGs und ECFGs are equivalent

Each ECFG  $G = (N, T, P, S)$  can be turned into an equivalent CFG:

- For each rule  $A \rightarrow e$  of  $P$  with  $e \notin (N \cup T)^*$  add a new nonterminal  $A_e$  together with all rules of a regular grammar  $G_e = (N_e, N \cup T, P_e, A_e)$  with  $L(G_e) = L(e)$ .
- Replace each rule  $A \rightarrow e$  of  $P$  by  $A \rightarrow A_e$ .

## ✿ Sorted sets and functions

Let  $S$  be a set. A family  $A = \{A_s \mid s \in S\}$  of sets is called an  **$S$ -sorted set**.

Let  $A$  and  $B$  be  $S$ -sorted sets. A family  $f = \{f_s : A_s \rightarrow B_s \mid s \in S\}$  of functions is called an  **$S$ -sorted function**.

$A$  bzw.  $f$  are extended to  $Reg(S)$ -sorted sets resp. functions as follows: Let  $s \in S$  and  $e, e' \in Reg(S)$ .

$$\begin{array}{lll}
 A_\varepsilon & = & \{\square\}, & f_\varepsilon(\square) & = & \square, \\
 A_{ee'} & = & A_e \times A_{e'}, & f_{ee'}(a, b) & = & (f_e(a), f_{e'}(b)), \\
 A_{e|e'} & = & A_e \cup A_{e'}, & f_{e|e'}(a) & = & \begin{cases} f_e(a) & \text{if } a \in A_e, \\ f_{e'}(a) & \text{otherwise,} \end{cases} \\
 A_{e^+} & = & \{[a_1, \dots, a_n] \mid \\ & & a_1, \dots, a_n \in A_e, n > 0\}, & f_{e^+}([a_1, \dots, a_n]) & = & [f_e(a_1), \dots, f_e(a_n)], \\
 A_{e^*} & = & A_{e^+|\varepsilon}, & f_{e^*} & = & f_{e^+|\varepsilon}, \\
 A_{e?} & = & A_{e|\varepsilon}, & f_{e?} & = & f_{e|\varepsilon}.
 \end{array}$$

### Beispiel

The language generated by an ECFG  $(N, T, P, S)$  is an  $N$ -sorted set.

## ✿ Signature $\Sigma$

A **signature**  $\Sigma = (S, C)$  consists of a set  $S$  of **sorts** and a  $\text{Reg}(S) \times S$ -sorted set  $C$  of **constructors**.

The  $S$ -sorted set  $T_\Sigma$  of (variable-free)  $\Sigma$ -**terms** is defined inductively as follows:

- For all  $c: \varepsilon \rightarrow s \in \Sigma$ ,  $c \in T_{\Sigma, s}$ .
- For all  $c: e \rightarrow s \in \Sigma$  with  $e \neq \varepsilon$  and  $t \in T_{\Sigma, e}$ ,  $c(t) \in T_{\Sigma, s}$ .

## ✿ Abstract syntax

Let  $G = (N, T, P, S)$  be an ECFG and

$$C = \{c_{A,i} : \text{abs}(e_i) \rightarrow A \mid (A \rightarrow e_1 \mid \dots \mid e_k) \in P, 1 \leq i \leq k\},$$

where the function  $\text{abs} : \text{Reg}(N \cup T) \rightarrow \text{Reg}(N)$  removes all terminals and some elements of  $C$  may be composed of other constructors and the identity  $\text{id} : A \rightarrow A$ .

The signature  $\Sigma(G) = (N, C)$  is called **abstract syntax** of  $G$ .

$\Sigma(G)$ -terms are called **syntax trees** of  $G$ .

Each nonterminal  $A$  corresponds to a sum of regular expressions

Each sum  $e_1 \mid \dots \mid e_k$  is implemented by a (constructor-based) datatype:

$$\text{data } A = C_1 \text{ abs}(e_1) \mid \dots \mid C_n \text{ abs}(e_k)$$

Conversely, the language of an ECFG without a proper sum on the right-hand side of any rule is regular!

## Beispiel JavaSig = (N, C)

$$\begin{aligned} N &= \{ \textit{Block}, \textit{Command}, \textit{IntE}, \textit{BoolE} \} \\ C &= \{ \textit{block} : \textit{Command}^* \rightarrow \textit{Block}, \\ &\quad \textit{skip} : \varepsilon \rightarrow \textit{Command}, \\ &\quad \textit{assign} : \textit{String IntE} \rightarrow \textit{Command}, \\ &\quad \textit{cond} : \textit{BoolE Block Block} \rightarrow \textit{Command}, \\ &\quad \textit{cond}(\_, \_, \textit{block}[\textit{skip}]) : \textit{BoolE Block} \rightarrow \textit{Command}, \\ &\quad \textit{loop} : \textit{BoolE Block} \rightarrow \textit{Command}, \\ &\quad \textit{intE} : \textit{Int} \rightarrow \textit{IntE}, \\ &\quad \textit{var} : \textit{String} \rightarrow \textit{IntE}, \\ &\quad \textit{id} : \textit{IntE} \rightarrow \textit{IntE}, \\ &\quad \textit{sub} : \textit{IntE IntE} \rightarrow \textit{IntE}, \\ &\quad \textit{sum} : \textit{IntE}^+ \rightarrow \textit{IntE}, \\ &\quad \textit{prod} : \textit{IntE}^+ \rightarrow \textit{IntE}, \\ &\quad \textit{boolE} : \textit{Bool} \rightarrow \textit{BoolE}, \\ &\quad \textit{greater} : \textit{IntE IntE} \rightarrow \textit{BoolE}, \\ &\quad \textit{not} : \textit{BoolE} \rightarrow \textit{BoolE} \} \end{aligned}$$

The identity  $\textit{id} : \textit{IntE} \rightarrow \textit{IntE}$  stems from the subexpression  $(\textit{IntE})$  of the *JavaGra* rule for  $\textit{IntE}$ .

## Implementation of *JavaSig* by datatypes

```
type Block    = [Command]
data Command  = Skip | Assign String IntE | Cond BoolE Block Block |
              Loop BoolE Block
data IntE     = IntE Int | Var String | Sub IntE IntE | Sum [IntE] |
              Prod [IntE]
data BoolE    = BoolE Bool | Greater IntE IntE | Not BoolE
```

Let  $\Sigma = (S, C)$  be a signature.

A  $\Sigma$ -Algebra  $(A, OP)$  consists of an  $S$ -sorted set  $A$  and for each  $c : e \rightarrow s \in C$ , a function  $c^A : A_e \rightarrow A_s \in OP$ .

## Implementation of $\Sigma$ -algebras

Let

```
data S1 = C11 e11 | ... | C1n1 e1n1
...
data Sk = Ck1 ek1 | ... | Cknk eknk
```

be an implementation of  $T_\Sigma$  by datatypes. Each instance of the following datatype represents a  $\Sigma$ -algebra:

```
data SigAlg s1...sk = SigAlg {c11 :: e11 -> s1, ... c1n1 :: e1n1 -> s1,
...
ck1 :: ek1 -> sk, ... cknk :: eknk -> sk}
```

## Example A datatype for *JavaSig*-algebras

```
data JavaAlg block command intE boolE =
  JavaAlg {block_ :: [command] -> block,
           skip  :: command,
           assign :: String -> intE -> command,
           cond  :: boolE -> block -> block -> command,
           loop  :: boolE -> block -> command,
           intE_ :: Int -> intE,
           var   :: String -> intE,
           sub   :: intE -> intE -> intE,
           sum_  :: [intE] -> intE,
           prod  :: [intE] -> intE,
           boolE_ :: Bool -> boolE,
           greater :: intE -> intE -> boolE,
           not_  :: boolE -> boolE}
```



## ✿ $T_\Sigma$ is a $\Sigma$ -algebra

- Für alle  $c:\varepsilon \rightarrow s \in \Sigma$  ist  $c^{T_\Sigma} =_{def} c$ .
- Für alle  $c:e \rightarrow s \in \Sigma$  mit  $e \neq \varepsilon$  und  $t \in T_{\Sigma,e}$  ist  $c^{T_\Sigma}(t) =_{def} c(t)$ .

## Implementation of the $\Sigma$ -term algebra

```
termAlg :: SigAlg S1...Sk
```

```
termAlg = SigAlg C11 ... C1n1 ... Ck1 ... Cknk
```

## Implementation of the *JavaSig*-term algebra

```
termAlg :: JavaAlg Block Command IntE BoolE
```

```
termAlg = JavaAlg id Skip Assign Cond Loop IntE Var Sub Sum Prod  
                BoolE Greater Not
```

## \* $\Sigma$ -terms as hierarchical lists

```
listAlg :: JavaAlg (Int -> Bool -> String) (Int -> Bool -> String)
          (Int -> Bool -> String) (Int -> Bool -> String)
```

```
listAlg = JavaAlg
  {block_ = \cs n -> let f []      = "[]"
                        f [c]     = '[':c (n+1) True++]"
                        f (c:cs) = mkList c cs "[" "]" (n+1)
                    in maybeBlanks (f cs) n,
  skip = maybeBlanks "Skip",
  assign = \x e n -> let str = "Assign "++show x++
                        ' ':e (n+10+length x) True
                    in maybeBlanks str n,
  cond = \be b b' n -> let str = "Cond "++g True be++
                              g False b++g False b
                              g b f = f (n+5) b
                    in maybeBlanks str n,
  loop = \be b n -> let str = "Loop "++g True be++g False b
                              g b f = f (n+5) b
```

```

        in maybeBlanks str n,
intE_ = \i -> maybeBlanks ("(IntE "++show i++)"),
var = \x -> maybeBlanks ("(Var "++show x++)"),
sub = \e e' n -> let str = "(Sub "++ g True e++g False e'++)"
                g b e = e (n+5) b
        in maybeBlanks str n,
sum_ = \(e:es) n -> let str = mkList e es "(Sum[" "])" (n+5)
                in maybeBlanks str n,
prod = \(e:es) n -> let str = mkList e es "(Prod[" "])" (n+6)
                in maybeBlanks str n,
boolE_ = \b -> maybeBlanks ("(BoolE "++show b++)"),
greater = \e e' n -> let str = "(Greater "++ g True e++g False e'++)"
                g b e = e (n+9) b
                in maybeBlanks str n,
not_ = \be n -> maybeBlanks ("(Not "++be (n+5) True++)") n}

```

```
maybeBlanks :: String -> Int -> Bool -> String
maybeBlanks str _ True = str
maybeBlanks str n _     = '\n':replicate n ' ' ++ str
```

```
mkList f fs open close n = open ++ f n True ++ concatMap g fs ++ close
                          where g f = ',':f n False
```

## Ein Element von listAlg

```
[Assign "fact" (IntE 1),
 Loop (Greater (Var "x")
             (IntE 0))
 [Assign "fact" (Prod[(Var "fact"),
                     (Var "x")]),
 Assign "x" (Sub (Var "x")
                (IntE 1))]]
```

✿ The state model of *JavaGra* is a *JavaSig*-algebra

```
stateAlg :: JavaAlg (State -> State) (State -> State)
           (State -> Int) (State -> Bool)
```

```
stateAlg = JavaAlg (foldl (flip (.)) id)
  id
  (\x e st -> update st x (e st))
  (\be b b' st -> if be st then b st else b' st)
  realLoop
  const (\x st -> st x)
  (\e e' st -> e st - e' st)
  (\es st -> sum (map ($ st) es))
  (\es st -> product (map ($ st) es))
  const (\e e' st -> e st > e' st)
  (not .)
```

```
where realLoop be b st = if be st then realLoop be b (b st)
                        else st
```

## ✿ $T_\Sigma$ is the initial $\Sigma$ -algebra

For all  $\Sigma$ -algebras  $A$  there is a unique  $\Sigma$ -homomorphism  $eval^A : T_\Sigma \rightarrow A$ .

Since each compile function  $comp : T_\Sigma \rightarrow Z$  should be  $\Sigma$ -homomorphic, the uniqueness implies that  $comp$  is determined by the extension of the target language  $Z$  to a  $\Sigma$ -algebra!

$eval^A$  is the (bottom-up-) **evaluation of  $\Sigma$ -terms in  $A$** :

- For all  $c : \varepsilon \rightarrow s \in \Sigma$ ,  $eval_s^A(c) = c^A$ .
- For all  $c : e \rightarrow s \in \Sigma$  with  $e \neq \varepsilon$  and  $t \in T_{\Sigma, e}$ ,  $eval_s^A(c(t)) = c^A(eval_e^A(t))$ .

## Implementation of $eval$ = generic interpreter

Let  $1 \leq i \leq k$ .

```
eval_si :: SigAlg s1...sk -> Si -> si
eval_si alg (Ci1 ei1) = ci1 (eval_ei1 alg e_i1)
...
eval_si alg (Cini eini) = c1n1 (eval_eini alg eini)
```

For all  $s \notin \{s_1, \dots, s_k\}$ ,  $eval_s$  is an identity.

## Beispiel Generic evaluation of *JavaSig*-terms

```
evBlock :: JavaAlg block command intE boolE -> Block -> block
```

```
evBlock alg = block_ alg . map (evCommand alg)
```

```
evCommand :: JavaAlg block command intE boolE -> Command -> command
```

```
evCommand alg Skip          = skip alg
```

```
evCommand alg (Assign x e)  = assign alg x (evIntE alg e)
```

```
evCommand alg (Cond be cs cs') = cond alg (evBoolE alg be)
                                     (evBlock alg cs)
                                     (evBlock alg cs')
```

```
evCommand alg (Loop be cs)   = loop alg (evBoolE alg be)
                                     (evBlock alg cs)
```

`evIntE :: JavaAlg block command intE boolE -> IntE -> intE`

`evIntE alg (IntE i) = intE_ alg i`

`evIntE alg (Var x) = var alg x`

`evIntE alg (Sub e e') = sub alg (evIntE alg e) (evIntE alg e')`

`evIntE alg (Sum es) = sum_ alg (map (evIntE alg) es)`

`evIntE alg (Prod es) = prod alg (map (evIntE alg) es)`

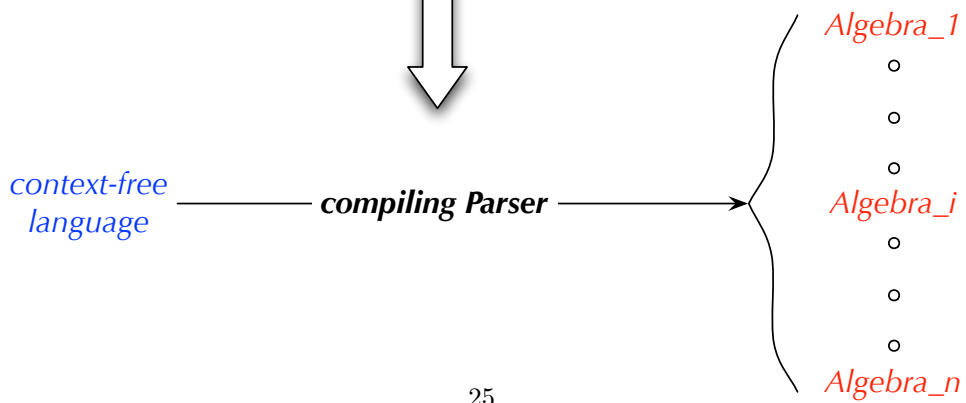
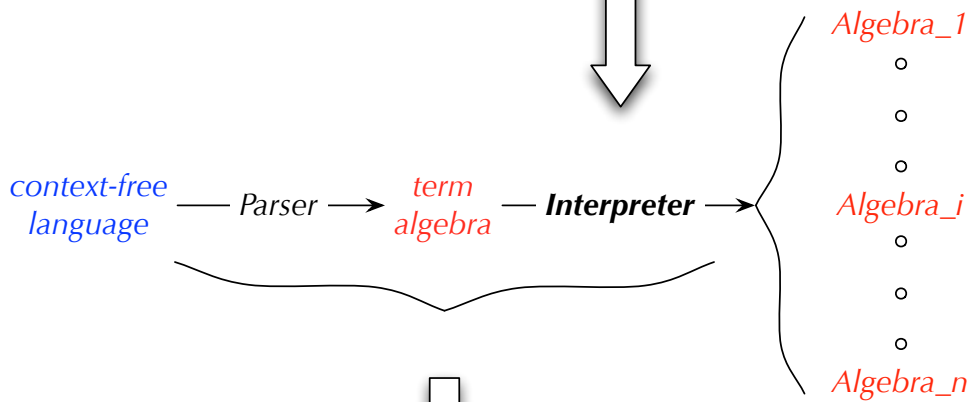
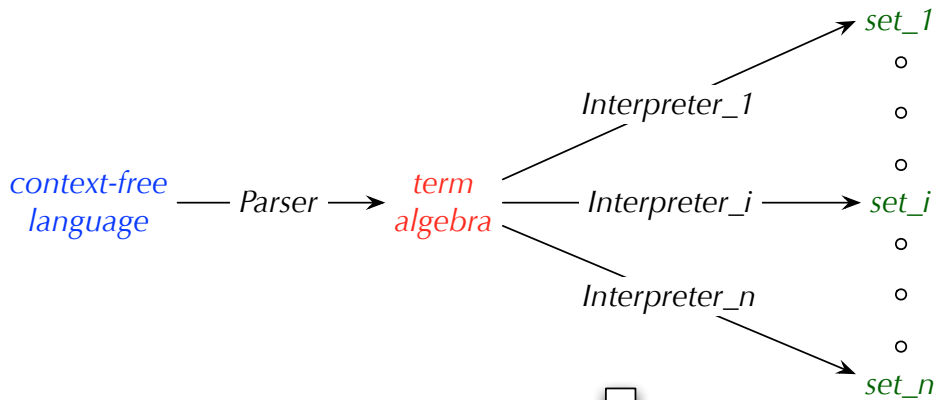
`evBoolE :: JavaAlg block command intE boolE -> BoolE -> boolE`

`evBoolE alg (BoolE b) = boolE_ alg b`

`evBoolE alg (Greater e e') = greater alg (evIntE alg e) (evIntE alg e')`

`evBoolE alg (Not be) = not_ alg (evBoolE alg be)`





## \* Parsers for regular expressions

### Type for deterministic parsers

```
type Parser sym a = [sym] -> Result sym a
data Result sym a = Result a [sym] | Error String
```

*Parser accepting sym*

```
symbol :: sym -> Parser sym ()
symbol sym (sym':syms) | sym == sym' = Result () syms
symbol sym syms                    = Error ("missing "++show sym)
```

*Parser accepting  $RR'$  ( $p$  and  $q$  are parser for  $R$  resp.  $R'$ )*

```
conc :: Parser sym a -> Parser sym b -> Parser sym (a,b)
conc p q syms = case p syms of
  Result a syms
    -> case q syms of
      Result b syms -> Result (a,b) syms
      Error str    -> Error str
  Error str -> Error str
```

*Parser accepting  $R/R'$  ( $p$  and  $q$  are parser for  $R$  resp.  $R'$ )  $\rightsquigarrow$  backtracking*

`par :: Parser sym a -> Parser sym b -> Parser sym (Either a b)`

```
par p q syms = case p syms of
  Result a str -> Result (Left a) str
  _ -> case q syms of
    Result b str -> Result (Right b) str
    Error str -> Error str
```

*Parser accepting  $R^+$  ( $p$  is a parser for  $R$ )*

`plus :: Parser sym a -> Parser sym [a]`

```
plus p syms = case p syms of
  Result a syms
    -> case star p syms of
      Result as syms -> Result (a:as) syms
  Error str -> Error str
```

*Parser accepting  $R^*$  ( $p$  is a parser for  $R$ )*

`star :: Parser sym a -> Parser sym [a]`

```
star p = par (plus p) (Result [])
```

## ✿ Parser into $T_{\Sigma(G)}$

**Schema 1:** Parser for a rule of the form  $A \rightarrow xByCz$  with  $A, B, C \in N$  and  $x, y, z \in T$   
 $\rightsquigarrow$  data **A** = ... | **F B C** | ...

`parseA :: Parser sym A`

```
parseA (x:syms) = case parseB syms of
  Result t (y:syms)
    -> case parseC syms of
      Result u (z:syms) -> Result (F t u) syms
      Error str -> Error str
      _ -> Error "missing z"
  Error str -> Error str
  _ -> Error "missing y"
parseA _ = Error "missing x"
```

## ✿ Generic parser into any $\Sigma(G)$ -algebra

**Schema 1:** Parser for a rule of the form  $A \rightarrow xByCz$  with  $A, B, C \in N$  and  $x, y, z \in T$   
 $\rightsquigarrow \Sigma(G)$  contains sorts  $a, b, c$  and a function  $f : b \rightarrow c \rightarrow a$ .

```
parseA :: SigAlg ... -> Parser sym a
parseA alg (x:syms) = case parseB alg syms of
  Result b (y:syms)
    -> case parseC alg syms of
      Result c (z:syms) -> Result (f alg b c) syms
      Error str -> Error str
      _ -> Error "missing z"
  Error str -> Error str
  _ -> Error "missing y"
parseA _ _ = Error "missing x"
```

**Schema 2:** Parser for a rule of the form  $A \rightarrow B|CD|CE$  with  $B, C, D, E \in N$  and  $C \neq A$ .

$\rightsquigarrow \Sigma(G)$  contains sorts  $a, b, c, d, e$  and functions  $f : b \rightarrow a$ ,  $g : c \rightarrow d \rightarrow a$  and  $h : c \rightarrow e \rightarrow a$ .

```
parseA :: SigAlg ... -> Parser sym a
parseA alg syms = case parseB alg syms of
  Result b syms -> Result (f alg b) syms
  _ -> case parseC alg syms of
    Result c syms -> parseArest alg c syms
    Error str -> Error str

parseArest :: SigAlg ... -> c -> Parser sym a
parseArest alg c syms = case parseD alg syms of
  Result d syms -> Result (g alg c d) syms
  _ -> case parseE alg syms of
    Result e syms -> Result (h alg c e)
    Error str -> Error str
```

**Schema 3:** Parser for a rule of the form  $A \rightarrow B|AD|AE$  with  $B, D, E \in N$ .

$\rightsquigarrow \Sigma(G)$  contains sorts  $a, b, d, e$  and functions  $f : b \rightarrow a$ ,  $g : a \rightarrow d \rightarrow a$  and  $h : a \rightarrow e \rightarrow a$ .

```
parseA :: SigAlg ... -> Parser sym a
parseA alg syms = case parseB alg syms of
  Result b syms -> parseArest alg (f alg b) syms
  _ -> case parseA alg syms of
    Result a syms -> parseArest alg a syms
    Error str -> Error str

parseArest :: SigAlg ... -> a -> Parser sym a
parseArest alg a syms = case parseD alg syms of
  Result d syms -> Result (g alg a d) syms
  _ -> case parseE alg syms of
    Result e syms -> Result (h alg a e)
    _ -> Result a syms
```

## \* *JavaGra-Parser into any JavaSig-Algebra*

`paBlock :: JavaAlg block a b c -> Parser Symbol block`

```
paBlock alg (Lcur:syms) = case star (paCommand alg) syms of
    Result cs (Rcur:syms)
        -> Result (block_ alg cs) syms
    Error str -> Error str
    _ -> Error "missing }"
paBlock _ _ = Error "no block"
```

`paCommand :: JavaAlg a command b c -> Parser Symbol command`

```
paCommand alg (Semi:syms) = Result (skip alg) syms
paCommand alg (Ide x:Upd:syms) = case paintE alg syms of
    Result e (Semi:syms)
        -> Result (assign alg x e)
    Error str -> Error str
    _ -> Error "missing ;"
paCommand alg (Ide x:_) = Error "missing ="
```



```

paCommand alg (If:Lpar:syms)
    = case paBoolE alg syms of
      Result be (Rpar:syms)
        -> case paBlock alg syms of
          Result b (Else:syms)
            -> case paBlock alg syms of
              Result b' syms
                -> Result (cond alg be b b') syms
                  Error str -> Error str
                  Result b syms
                    -> Result (cond alg be b
                                (block_ alg [])) syms
                      Error str -> Error str
                      Error str -> Error str
                _ -> Error "missing )"
paCommand alg (If:_) = Error "missing ("

```

```

paCommand alg (While:Lpar:syms) = case paBoolE alg syms of
    Result be (Rpar:syms)
        -> case paBlock alg syms of
            Result b syms
                -> Result (loop alg be b) syms
            Error str -> Error str
        Error str -> Error str
    _ -> Error "missing )"
paCommand alg (While:_) = Error "missing ("
paCommand _ _ = Error "no command"

```

```

paIntE :: JavaAlg a b intE c -> Parser Symbol intE

```

```

paIntE alg (Num i:syms) = paIntErest alg (intE_ alg i) syms
paIntE alg (Ide x:syms) = paIntErest alg (var alg x) syms
paIntE alg (Lpar:syms) = case paIntE alg syms of
    Result e (Rpar:syms) -> paIntErest alg e syms
    err@(Error _) -> err
    _ -> Error "missing )"
paIntE alg syms = Error "no integer expression"

```

```
paIntErest :: JavaAlg a b intE c -> intE -> Parser Symbol intE
```

```
paIntErest alg e (Minus:syms) = case paIntE alg syms of  
    Result e' syms  
        -> Result (sub alg e e') syms  
    _ -> Result e syms
```

```
paIntErest alg e syms = case plus (conc (symbol Plus) p) syms of  
    Result es syms -> Result (sum_ alg (e:map snd es))  
    _ -> case plus (conc (symbol Times) p) syms of  
        Result es syms  
            -> Result (prod alg (e:map snd es)) syms  
        _ -> Result e syms  
where p = paIntE alg
```

```

paBoolE :: JavaAlg a b c boolE -> Parser Symbol boolE

paBoolE alg (True_:syms) = Result (boolE_ alg True) syms
paBoolE alg (False_:syms) = Result (boolE_ alg False) syms
paBoolE alg (Neg:syms) = case paBoolE alg syms of
    Result be syms -> Result (not_ alg be) syms
    err@(Error _) -> err

paBoolE alg syms = case paIntE alg syms of
    Result e (GR:syms)
        -> case paIntE alg syms of
            Result e' syms
                -> Result (greater alg e e') syms
            Error str -> Error str
    Error str -> Error str
    _ -> Error "no Boolean expression"

```

## ✿ Monadic parsers

```
class Monad m where (>>=)  :: m a -> (a -> m b) -> m b
                    return :: a -> m a
                    fail   :: String -> m a
                    (>>)   :: m a -> m b -> m b
                    p >> q = p >>= const q
```

```
newtype MParser sym a = P {apply :: Parser sym a}
```

```
instance Monad (MParser sym)
  where p >>= f = P {apply = \syms -> case apply p syms of
                                     Result a syms -> apply (f a) syms
                                     Error str  -> Error str}

  return = P . Result
  fail   = P . const . Error
```

## do-Notation

$$m_0 \gg= (\backslash x_1 \rightarrow m_1 \gg= (\backslash x_2 \rightarrow \dots m_{(n-1)} \gg= (\backslash x_n \rightarrow m_n) \dots ))$$

is reduced to:

$$\text{do } x_1 \leftarrow m_0; x_2 \leftarrow m_1; \dots x_n \leftarrow m_{(n-1)}; m_n$$

## ✿ Monadic parsers for regular expressions

*Parser accepting any symbol*

```
item :: MParser sym sym
```

```
item = P {apply = \syms -> case syms of sym:syms -> Result sym syms  
_ -> Error "no symbols"}
```

*Parser accepting elements of R that satisfy f (p is a parser for R)*

```
sat :: MParser sym a -> (a -> Bool) -> String -> MParser sym a
```

```
sat p f err = do a <- p; if f a then return a else fail err
```

*Parser accepting sym*

```
symbolM :: (Eq sym, Show sym) => sym -> MParser sym sym
```

```
symbolM sym = do sat item (== sym) ("no "++show sym)
```

*Parser accepting RR' (p and q are parser for R resp. R')*

```
concM :: MParser sym a -> MParser sym b -> MParser sym (a,b)
```

```
concM p q = do a <- p; b <- q; return (a,b)
```

*Parser accepting  $R/R'$  ( $p$  and  $q$  are parser for  $R$  resp.  $R'$ )*

```
parM :: MParser sym a -> MParser sym a -> MParser sym a
p 'parM' q = {apply = \syms -> case apply p syms of
              res@(Result _ _) -> res
              _ -> apply q syms}
```

*Parser accepting  $R1/\dots/Rn$*

```
parL :: [MParser sym a] -> MParser sym a
parL = foldr1 parM
```

*Parser accepting  $R+$  ( $p$  is a parser for  $R$ )*

```
plusM :: MParser sym a -> MParser sym [a]
plusM p = do a <- p; as <- starM p; return (a:as)
```

*Parser accepting  $R^*$  ( $p$  is a parser for  $R$ )*

```
starM :: MParser sym a -> MParser sym [a]
starM p = plusM p 'parM' return []
```



## ✿ Monadic Parsers for a rule $A \rightarrow e$

**Schema 1:**  $A \rightarrow e$  has the form  $A \rightarrow xByCz$  with  $B, C \in N$  and  $x, y, z \in T$ .

$\rightsquigarrow \Sigma(G)$  contains sorts  $a, b, c$  and a function  $f : b \rightarrow c \rightarrow a$ .

```
parseA :: SigAlg ... -> MParser sym a
```

```
parseA alg = do x <- item; b <- parseB alg; y <- item; c <- parseC alg  
             z <- item; return (f alg b c)
```

**Schema 2:**  $A \rightarrow e$  has the form  $A \rightarrow B|CD|CE$  with  $B, C, D, E \in N$  and  $C \neq A$ .

$\rightsquigarrow \Sigma(G)$  contains sorts  $a, b, c, d, e$  and functions  $f : b \rightarrow a$ ,  $g : c \rightarrow d \rightarrow a$  and  $h : c \rightarrow e \rightarrow a$ .

```
parseA :: SigAlg ... -> MParser sym a
```

```
parseA alg = parL [do b <- parseB alg; return (f alg b),  
                  do c <- parseC alg; parseArest alg c]
```

```
parseArest :: SigAlg ... -> c -> MParser sym a
```

```
parseArest alg c = parL [do d <- parseD alg; return (g alg c d),  
                          do e <- parseE alg; return (h alg c e)]
```

**Schema 3:**  $A \rightarrow e$  has the form  $A \rightarrow B|AD|AE$  with  $B, D, E \in N$ .

$\rightsquigarrow \Sigma(G)$  contains sorts  $a, b, d, e$  and functions  $f : b \rightarrow a$ ,  $g : a \rightarrow d \rightarrow a$  and  $h : a \rightarrow e \rightarrow a$ .

```
parseA :: SigAlg ... -> Parser sym a
```

```
parseA alg = parL [do b <- parseB alg; parseArest alg (f alg b),  
                  do a <- parseA alg; parseArest alg a]
```

```
parseArest :: SigAlg ... -> a -> Parser sym a
```

```
parseArest alg a = parL [do d <- parseD alg; return (g alg a d),  
                          do e <- parseE alg; return (h alg a e),  
                          return e]
```

## ✿ Monadic *JavaGra*-parser into any *JavaSig*-algebra

```
mBlock :: JavaAlg block a b c -> MParser Symbol block
```

```
mBlock alg = do symbolM Lcur; cs <- starM (mCommand alg)
              symbolM Rcur; return (block_ alg cs)
```

```
mCommand :: JavaAlg a command b c -> MParser Symbol command
```

```
mCommand alg = parL [do Semi <- item; return (skip alg),
                    do x <- ident; Upd <- item; e <- mIntE alg
                      Semi <- item; return (assign alg x e),
                    do If <- item; Lpar <- item; be <- p; Rpar <- item
                      b <- q
                      parL [do Else <- item; b' <- q
                            return (cond alg be b b'),
                          return (cond alg be b (block_ alg []))],
                    do While <- item; Lpar <- item; be <- p; Rpar <- item
                      b <- q; return (loop alg be b),
                    fail "no command"]
  where p = mBoolE alg; q = mBlock alg
```

```
mIntE :: JavaAlg a b intE c -> MParser Symbol intE
```

```
mIntE alg = parL [do i <- number; p (intE_ alg i),  
                  do x <- ident; p (var alg x),  
                  do Lpar <- item; e <- mIntE alg; Rpar <- item; p e,  
                  fail "no integer expression"]  
  where p = mIntErest alg
```

```
mIntErest :: JavaAlg a b intE c -> intE -> MParser Symbol intE
```

```
mIntErest alg e = parL [do Minus <- item; e' <- p; return (sub alg e e'),  
                        do es <- plusM (conclM (symbolM Plus) p)  
                          return (sum_ alg (e:map snd es)),  
                        do es <- plusM (conclM (symbolM Times) p)  
                          return (prod alg (e:map snd es)),  
                        return e]  
  where p = mIntE alg
```

```
mBoolE :: JavaAlg a b c boolE -> MParser Symbol boolE
```

```
mBoolE alg = parL [do True_ <- item; return (boolE_ alg True),  
                  do False_ <- item; return (boolE_ alg False),  
                  do Neg <- item; be <- mBoolE alg; return (not_ alg be),  
                  do e <- p; GR <- item; e' <- p  
                    return (greater alg e e'),  
                  fail "no Boolean expression"]  
  where p = mIntE alg
```

```
number :: MParser Symbol Int
```

```
number = do sym <- sat item f "no number"; return (g sym)  
  where f (Num _) = True  
        f _       = False  
        g (Num i) = i
```

```
ident :: MParser Symbol String
```

```
ident = do sym <- sat item f "no identifier"; return (g sym)  
  where f (Ide _) = True  
        f _       = False  
        g (Ide x) = x
```

## ✿ Attributed $\Sigma$ -algebras

**Types for  $n$  attributes**  $At = \{At_1, \dots, At_n\}$

newtype  $At\_1 = At\_1$  typ\_1; ... newtype  $At\_n = At\_n$  typ\_n

A  $\Sigma$ -algebra  $A$  is **At-attributed** if for all  $s \in N$  and  $c : e \rightarrow s \in C$  there are

$$Inh_{s,1}, \dots, Inh_{s,m_s}, Der_{s,1}, \dots, Der_{s,n_s} \in At$$

such that

$$A_s = Inh_{s,1} \times \dots \times Inh_{s,m_s} \rightarrow Der_{s,1} \times \dots \times Der_{s,n_s}, \quad (4.1)$$

and the interpretation of  $c$  in  $A$  is given by a (Haskell) definition of the following form:

For all  $1 \leq i \leq n$  let  $f_i \in A_{s_i}$ . The red variables are called **local variables**.

$$\begin{aligned} c^A(f_1, \dots, f_n)(Inh_{s,1}(x_{s,1}), \dots, Inh_{s,m_s}(x_{s,m_s})) &= (Der_{s,1}(e_{s,1}), \dots, Der_{s,n_s}(e_{s,n_s})) \\ \text{where } (Der_{s_1,1}(x_{s_1,1}), \dots, Der_{s_1,n_{s_1}}(x_{s_1,n_{s_1}})) &= \\ &f_1(Inh_{s_1,1}(e_{s_1,1}), \dots, Inh_{s_1,m_{s_1}}(e_{s_1,m_{s_1}})) \\ &\vdots \\ (Der_{s_n,1}(x_{s_n,1}), \dots, Der_{s_n,n_{s_n}}(x_{s_n,n_{s_n}})) &= \\ &f_n(Inh_{s_n,1}(e_{s_n,1}), \dots, Inh_{s_n,m_{s_n}}(e_{s_n,m_{s_n}})) \end{aligned} \quad (2)$$

## ✿ Multi-pass compilers

Given an  $At$ -attributed  $\Sigma$ -algebra  $A$ , the above definition of  $eval^A : T_\Sigma \rightarrow A$  is a **one-pass compiler** if for all  $1 \leq i \leq n$  and  $1 \leq k \leq n_{s_i}$  the local variable  $x_{s_i,k}$  occurs in the expression  $e_{s_j,l}$  only if  $i < j$ .

Otherwise the well-known **LAG-algorithm** may be applied to (2). It computes the least partition  $\{At^1, \dots, At^r\}$  of  $At = \{At_1, \dots, At_n\}$  – if there is any – such that the sequential composition of  $r$   $N$ -sorted compile functions yields an executable definition of  $eval^A$ , which is then called an **r-pass compiler**. These functions generate resp. transform an  $At$ -annotated  $\Sigma$ -terms:

An  **$At$ -annotated  $\Sigma$ -term** of sort  $s \in N$  is a  $\Sigma$ -term each of whose nodes is labelled not only with a constructor  $c : e \rightarrow s$ , but also with a subtuple of an element of

$$Der_{s,1} \times \dots \times Der_{s,n_s}.$$

$T_\Sigma^{At}$  denotes the  $S$ -sorted set of  $At$ -annotated  $\Sigma$ -terms.

Let  $1 \leq i \leq r$ ,  $1 \leq i_1, \dots, i_m \leq n$ ,  $At' = At_{i_1} \times \dots \times At_{i_m}$ ,  $\{j_1, \dots, j_n\} = \{k \in \{i_1, \dots, i_m\} \mid At_k \in At^i\}$  and  $a = (a_{i_1}, \dots, a_{i_m}) \in At'$ . Then

$$\begin{aligned} \pi^i(a) &=_{def} (a_{j_1}, \dots, a_{j_n}), \\ \pi^i(At') &=_{def} \{\pi^i(a) \mid a \in At'\}. \end{aligned}$$

On the basis of a short version of (2):

$$\begin{aligned}
c^A(f_1, \dots, f_n)(x) &= e \text{ where } x_1 = f_1(e_1) \\
&\vdots \\
&x_n = f_n(e_n),
\end{aligned}$$

the resulting definition of  $eval^A$  looks as follows: Let  $s \in N$ ,  $t \in T_{\Sigma, s}$ ,

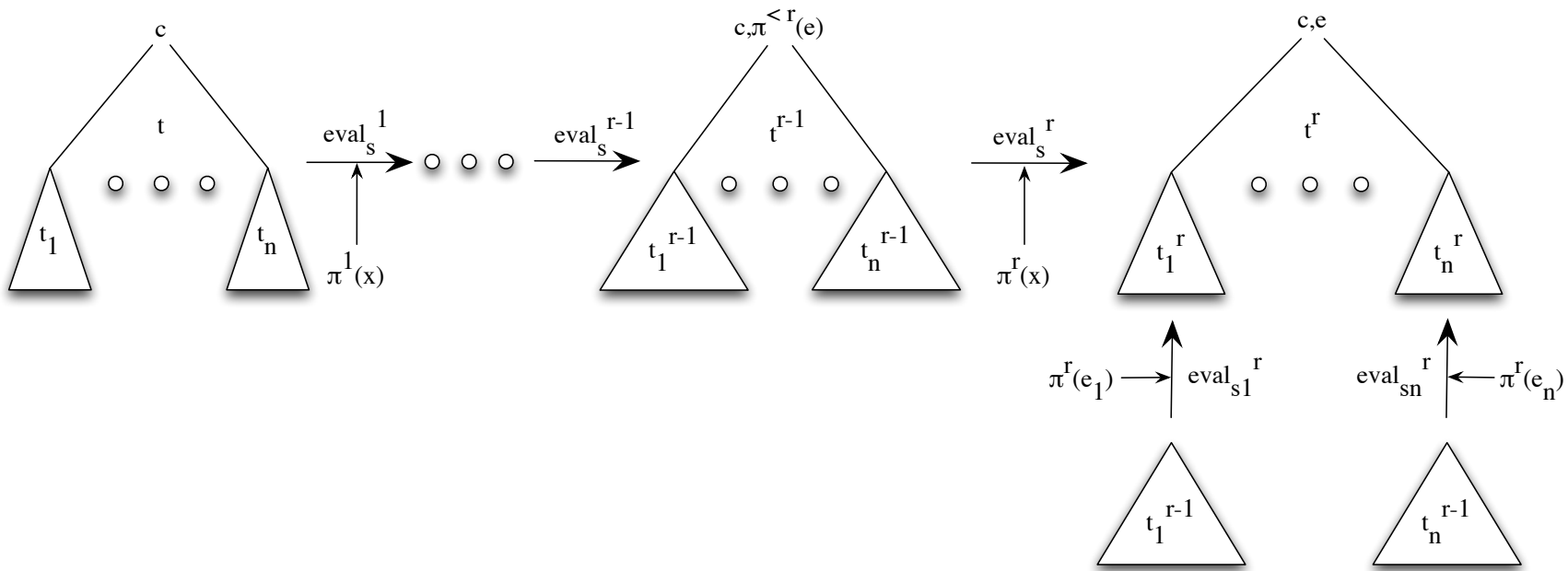
$$[c, a](t_1, \dots, t_n) \in T_{\Sigma, s}^{At}$$

and  $1 \leq i \leq r$ .

$$\begin{aligned}
eval_s^A : T_{\Sigma, s} &\rightarrow (Inh_{s,1} \times \dots \times Inh_{s,m_s}) \rightarrow (Der_{s,1} \times \dots \times Der_{s,n_s}) \\
eval_s^A(t)(x) &= attrs(root(t^r)) \text{ where } t^1 = eval_s^1(t)(\pi^1(x)) \\
&\vdots \\
&t^r = eval_s^r(t^{r-1})(\pi^r(x))
\end{aligned}$$

$$\begin{aligned}
eval_s^i : T_{\Sigma, s}^{At} &\rightarrow \pi^i(Inh_{s,1} \times \dots \times Inh_{s,m_s}) \rightarrow T_{\Sigma, s}^{At} \\
eval_s^i([c, a](t_1, \dots, t_n))(\pi^i(x)) &= [c, a, \pi^i(e)](u_1, \dots, u_n) \\
&\text{where } u_1 = eval_{s_1}^i(t_1)(\pi^i(e_1)) \\
&\vdots \\
&u_n = eval_{s_n}^i(t_n)(\pi^i(e_n))
\end{aligned}$$





*Stepwise annotation of a syntax tree*

## ✿ Conclusion

- sums  $\iff$  nonterminals  $\iff$  datatypes  
 $\rightsquigarrow$  new definition of an ECFG  $G$
- target languages extended to  $\Sigma(G)$ -algebras  
 $\rightsquigarrow$  generic interpreter  $\rightsquigarrow$  generic (monadic) parser/compiler
- attributed  $\Sigma(G)$ -algebra  $\rightsquigarrow$  multi-pass compiler
- Future work:  
web documents with links and attributes modelled as coalgebras