Kapitel 9: Parsing mit Kombinatoren und Generatoren

Lernziele dieses Kapitels

1. Manuelle Konstruktion eines einfachen Scanners
2. Parsing mit Kombinatoren
   - Rolle der Laziness
   - Prinzip "Replacing failure by a list of successes"
3. Verwendung des Parsergenerators `happy`
   - Grammatikregeln mit semantischen Definitionen
   - Explizite Angabe von Prioritäten
   - Environments während des Parsings
4. Verwendung des Scannergenerators `alex`
5. Implementierung eines Interpreters für eine einfache imperative(!) Sprache

Scanner und Parser

Erste Phasen von Compilern und Interpretern:

1. Scanner: lexikalische Analyse, Gruppierung der Eingabezeichen zu Token
2. Parser: syntaktische (strukturelle) Analyse eines Tokenstroms
   - zur Erzeugung eines Syntaxbaums oder
   - zur direkten Auswertung oder Codeerzeugung

Beispiel:

- Eingabestring: `" (23 - x) *(2)"`
- Tokenstrom: `[Other '(',Number 23,Other '-',Name "x",Other ')',Other '*',Other '(',Number 2,Other ')']`
- abstrakte Syntax: `(CI 23 :-: V "x") :*: CI 2`

Kombinatoren vs. Generatoren

- Kombinator-Parsing:
  Komposition des Parsers für die ganze Grammatik aus Parsern für die einzelnen Nichtterminale
  - Vorteil: Parsen einer kontextsensitiven Sprache möglich
  - Nachteil: Grammatik muss vorher in eine Form ohne Linksrekursion transformiert werden, ggfs. mit Anpassung der Attribute
- Parsergenerator:
  Erzeugung einer Zustandsübergangstabelle aus einer annotierten Grammatik, die einen generischen Parser steuert
  - Vorteil: Linksrekursion ist kein Problem
  - Nachteil: zusätzliches Tool (Parsergenerator) nötig

Prinzip des Kombinator-Parsings

- Zusammensetzung des Parsers für die ganze Struktur aus Parsern für die Teilstrukturen
  - orientiert an der Struktur der Grammatik
  - ein Parser für jedes Nichtterminal
  - möglicherweise rekursiv
- jeder Teilparser erzeugt
  - die erkannte Teilstruktur
  - die Resteingabe

Probleme des Kombinator-Parsings

- keine Struktur erkannt
- mehrere Möglichkeiten, lokal unbekannter Kontext soll entscheiden

Lösung: Liste aller Möglichkeiten (Struktur,Resteingabe)
Prinzip: replace failure by a list of successes [Wadler, 1985]

- bei anderen Programmiersprachen: exponentieller Speicherbedarf!
- Vorteil der Laziness in Haskell:
  es wird jeweils nur ein Pfad des Suchbaums im Speicher gehalten:
  Backtracking ähnlich wie in Prolog!

Ein einfacher Scanner

```
data Token = EOS | Number Integer | Name String | Other Char

readToken :: String -> (Token,String)
readToken [] = (EOS,[])
readToken (c:cs)
 | isDigit c    = let (t,r) = span isDigit (c:cs)
                  in (Number (read t), r)
 | isAlphaNum c = let (t,r) = span isAlphaNum (c:cs)
                  in (Name t, r)
 | isSpace c    = readToken (dropWhile isSpace cs)
 | otherwise    = (Other c, cs)

scanner :: String -> [Token]
scanner s = let (tok,rest) = readToken s
            in if tok == EOS then [] else (tok : scanner rest)
```

Prinzipielle Arbeitsweise des Parsers

```
type Parse a b = [a] -> [(b,[a])]
```

- `a`: Tokentyp, `b`: Typ der syntaktischen Elemente (SE)
- Eingabe: Liste von Token
- Erkennen von SE in allen Präfixen der Tokenliste
- Ausgabe: alle Alternativen von (SE, nicht verbrauchten Token)

Kombination von Parsern zu komplexeren Parsern:

```
infixl 4 <|>     -- Alternative von Parsern
infixr 5 >*>     -- Sequenz von Parsern
infixr 5 >->     -- wie p>*>q, q erbt Attribute von p
```

Parser-Kombinatoren

- Parser, der nichts akzeptiert

```
none :: Parse a b
none inp = []
```

- Parser, der akzeptiert, ohne zu lesen

```
succeed :: b -> Parse a b
succeed val inp = [(val,inp)]
```

- Match eines Tokens, welches eine Bedingung `p` erfüllt

```
spot :: (a->Bool) -> Parse a a
spot p (x:xs) | p x = [(x,xs)]
spot _ _ = []
```

- Match genau eines Tokens

```
token :: Eq a => a -> Parse a a
token t = spot (==t)
```

- Match eines Zeichens

```
match :: Char -> Parse Token Token
match c = token (Other c)
```

- Lesen eines Zeichens

```
other :: Parse Token Char
other (Other c : xs) = [(c,xs)]
other _ = []
```

- Lesen einer Zahl

```
number :: Parse Token Integer
number (Number i : xs) = [(i,xs)]
number _ = []
```

- Lesen eines Namens

```
name :: Parse Token String
name (Name s : xs) = [(s,xs)]
name _ = []
```

- Alternative zweier Parser

```
(<|>) :: Parse a b -> Parse a b -> Parse a b
(p1 <|> p2) inp = p1 inp ++ p2 inp
```

- Sequenz zweier Parser

```
(>*>) :: Parse a b -> Parse a c -> Parse a (b,c)
(p1 >*> p2) inp = [ ((s1,s2),s) | (s1,r) <- p1 inp,
                                  (s2,s) <- p2 r    ]
```

- Sequenz mit ererbtem Attribut

```
(>->) :: Parse a b -> (b -> Parse a c) -> Parse a c
(p1 >-> p2) inp = [ (s2,s) | (s1,r) <- p1 inp,
                             (s2,s) <- p2 s1 r    ]
```

- Verknüpfen mit einer semantischen Aktion

```
build :: Parse a b -> (b->c) -> Parse a c
build p f inp = [ (f x, rem) | (x,rem) <- p inp ]
```

- beliebige Wiederholung eines Parsers

```
list :: Parse a b -> Parse a [b]
list p = (succeed []) <|>
         ((p >*> list p) `build` (uncurry (:)))
```

- Einschränkung auf Lookahead-Zeichen

```
withLookahead :: Parse a b -> (a->Bool) -> Parse a b
withLookahead p pred s
 = [ (x,r) | (x,r) <- p s, not (null (spot pred r)) ]
```

Parsen einer Ausdrucksgrammatik

Grammatik $G = (\{E, T, F\}, \{\texttt{+}, \texttt{-}, \texttt{*}, \texttt{num}, \texttt{id}, \texttt{(}, \texttt{)}\}, P, E)$
wobei `num` natürliche Zahlen und `id` Variablen darstellen.

Die Produktionen P:

$$
\begin{aligned}
E &\rightarrow E\,\texttt{+}\,T \mid E\,\texttt{-}\,T \mid T \\
T &\rightarrow T\,\texttt{*}\,F \mid F \\
F &\rightarrow \texttt{num} \mid \texttt{id} \mid \texttt{-}\,F \mid \texttt{(}\,E\,\texttt{)}
\end{aligned}
$$

Problem: Linksrekursion $\Rightarrow$ Nichttermination

Elimination der Linksrekursion

Grammatik $G = (\{E, E', T, T', F\}, \{\texttt{+}, \texttt{-}, \texttt{*}, \texttt{num}, \texttt{id}, \texttt{(}, \texttt{)}\}, P, E)$
wobei `num` natürliche Zahlen und `id` Variablen darstellen.

Die Produktionen P:

$$
\begin{aligned}
E &\rightarrow T\,E' \\
E' &\rightarrow \texttt{+}\,T\,E' \mid \texttt{-}\,T\,E' \mid \epsilon \\
T &\rightarrow F\,T' \\
T' &\rightarrow \texttt{*}\,F\,T' \mid \epsilon \\
F &\rightarrow \texttt{num} \mid \texttt{id} \mid \texttt{-}\,F \mid \texttt{(}\,E\,\texttt{)}
\end{aligned}
$$

Abstrakte Syntax und eval-Funktion

```
data S = CI Integer
       | V String
       | Neg S
       | S :+: S
       | S :-: S
       | S :*: S
       deriving (Eq,Show)

eval :: S -> Integer
eval (CI i)   = i
eval (V _)    = error "eval (V _) undefined"
eval (Neg a)  = - eval a
eval (a:+:b)  = eval a + eval b
eval (a:-:b)  = eval a - eval b
eval (a:*:b)  = eval a * eval b
```

Parser für $L(G) \longrightarrow \mathtt{S}$

```
buildSyntaxTree :: String -> S
buildSyntaxTree s =
   let tokStream = scanner s
   in head [ x | (x,[]) <- parseE tokStream ]

parseE, parseT, parseF :: Parse Token S
parseE', parseT' :: S -> Parse Token S

parseF =
     (number `build` CI)
 <|> (name   `build` V )
 <|> (((match '-') >*> parseF)
              `build` (\ (_,e) -> Neg e ))
 <|> (((match '(') >*> parseE >*> match ')')
              `build` (\ (_,(e,_)) -> e))
```

```
parseE = parseT >-> parseE'

parseE' exp =
     succeed exp
 <|> ((match '+' >*> parseT) `build` (\ (_,e)-> exp:+:e)
        >-> parseE')
 <|> ((match '-' >*> parseT) `build` (\ (_,e)-> exp:-:e)
        >-> parseE')

parseT = parseF >-> parseT'

parseT' exp =
     succeed exp
 <|> ((match '*' >*> parseF) `build` (\ (_,e)-> exp:*:e)
        >-> parseT')
```

Beispiele

```
Expr> buildSyntaxTree "2- 345*(2-3---1)+x2"
CI 2 :-: (CI 345 :*: (CI 2 :-: (CI 3)
          :-: (Neg (Neg (CI 1))))) :+: (V "x2")

Expr> eval $ buildSyntaxTree
              ("1"++concat (replicate 1000 "+1"))
1001
```

Der Parsergenerator `happy`

`http://www.haskell.org/happy/`

- Vorbild: `yacc` (yet another compiler compiler) von UNIX
- Eingabe: kontexfreie Grammatik mit semantischen Funktionen
- Ausgabe: LALR(1)-Parser als Haskell-Funktion
- Reduktion einer Grammatikregel bewirkt Anwendung der semantischen Funktion, z.B. Erzeugung eines Syntaxbaumknotens

Aufbau einer happy-Quelldatei

1. Haskell-Definitionen am Anfang der Parserdatei, insbesondere Modulkopf

```
{
   module Parser where
}
```

2. Name der Funktion, die das Parsing durchführt (von happy erzeugt)

```
%name parser
```

3. Deklaration von Tokentypen

```
%tokentype { Token }
```

4. Tokendeklarationen (Grammatiksymbol/Haskell-Wert)
5. Produktionen mit semantischen Funktionen
6. Haskell-Funktionen am Ende der Parserdatei

4. Tokendeklarationen (Symbol/Haskell-Wert)

```
$$: Zusatzrückgabewert des Scanners
%token
      let             { TokenLet }
      in              { TokenIn }
      int             { TokenInt $$ }     $$ :: Int
      var             { TokenVar $$ }     $$ :: String
      '='             { TokenEq }
      '+'             { TokenPlus }
      '-'             { TokenMinus }
      '*'             { TokenTimes }
      '/'             { TokenDiv }
      '('             { TokenOB }
      ')'             { TokenCB }
```

5. Produktionen mit semantischen Funktionen

```
%%
 Exp    :: { S }
 Exp    : Exp '+' Term          { $1 :+: $3 }
        | Exp '-' Term          { $1 :-: $3 }
        | Term                  { $1 }

 Term   : Term '*' Factor       { $1 :*: $3 }
        | Term '/' Factor       { $1 :/: $3 }
        | Factor                { $1 }

 Factor : int                   { CI $1 }
        | var                   { Var $1 }
        | '(' Exp ')'           { $2 }
        | let var '=' Exp in Exp  { Let ($2,$4) $6 }
```

6. Haskell-Funktionen am Ende der Parserdatei

- Definition des Tokentyps
- obligatorische `happyError`-Funktion
- Typdefinitionen für semantische Werte
- Scanner für lexikalische Analyse
- Schnittstelle nach außen

Definition des Tokentyps

```
data Token
  = TokenLet
  | TokenIn
  | TokenInt Integer
  | TokenVar String
  | TokenEq
  | TokenPlus
  | TokenMinus
  | TokenTimes
  | TokenDiv
  | TokenOB
  | TokenCB
```

happyError-Funktion

```
happyError :: [Token] -> a
happyError _ = error ("Parse error\ n")
```

Typdefinitionen für semantische Werte

```
data S = CI Integer
       | V String
       | Neg S
       | S :+: S
       | S :-: S
       | S :*: S
       | S :/: S
       | Let (String,S) S
       deriving (Eq,Show)
```

Scanner (auch als Lexer bezeichnet)

```
lexer :: String -> [Token]
lexer [] = []
lexer (c:cs)
      | isSpace c = lexer cs
      | isDigit c = lexNum (c:cs)
      | isAlpha c = lexVar (c:cs)
lexer ('=':cs) = TokenEq   : lexer cs
lexer ('+':cs) = TokenPlus : lexer cs
...
lexNum cs = TokenInt (read num) : lexer rest
    where (num,rest) = span isDigit cs
lexVar cs = case span isAlpha cs of
              ("let",rest) -> TokenLet     : lexer rest
              ("in",rest)  -> TokenIn      : lexer rest
              (var,rest)   -> TokenVar var : lexer rest
```

Schnittstelle nach außen

```
parse :: String -> S
parse = parser . lexer
```

`parser`: diese Funktion erzeugt `happy`

Wdh.: Aufbau einer `happy`-Quelldatei

1. Haskell-Definitionen am Anfang der Parserdatei, insbesondere Modulkopf
2. Name der Funktion, die das Parsing durchführt (von happy erzeugt)
3. Deklaration von Tokentypen
4. Tokendeklarationen (Grammatiksymbol/Haskell-Wert)
5. Produktionen mit semantischen Funktionen
6. Haskell-Funktionen am Ende der Parserdatei

Andere semantische Werte

Beispiel: Berechnung des Ausdrucks während des Parsens:

```
Term   : Term '*' Factor          { $1 * $3 }
       | Term '/' Factor          { $1 `div` $3 }
       | Factor                   { $1 }
```

- Semantischer Wert: `Int`
- Problem: Variablen und `let`-Ausdrücke
- Lösung: Environment, Funktionen als semantische Werte

Funktionen als semantische Werte

```
Exp    :: { Env->Integer }
Exp    : Exp '+' Term          { \ p -> $1 p + $3 p }
       | Exp '-' Term          { \ p -> $1 p - $3 p }
       | Term                  { $1 }
Term   : Term '*' Factor       { \ p -> $1 p * $3 p }
       | Term '/' Factor       { \ p -> $1 p `div` $3 p }
       | Factor                { $1 }
Factor : int          { \ p -> $1 }
       | var          { \ p -> case lookup $1 p of
                                 Nothing -> error ($1 ++ " not found")
                                 Just i  -> i }
       | '(' Exp ')'   { $2 }
       | let var '=' Exp in Exp  { \ p -> $6 (($2,$4 p):p) }
```

Parsen einer Sequenz eines Grammatiksymbols

- Linksrekursion (konstanter Stackbedarf)

```
prods : prod                    { [$1] }
      | prods prod              { $2 : $1 }
```

- Rechtsrekursion (Stackbedarf proportional zur Sequenzlänge)

```
prods : prod                    { [$1] }
      | prod prods              { $1 : $2 }
```

- ϵ-Produktionen

```
prods : {- empty -}             { [] }
      | prods prod              { $2 : $1 }
```

Explizite Angabe von Prioritäten

bisher: Prioritäten durch abgestufte Grammatik (unschön)

gewünscht:

```
Exp   : Exp '+' Exp          { $1 :+: $3 }
      | Exp '*' Exp          { $1 :*: $3 }
      ...
```

explizite Angabe der Prioritäten

- Priorität der Operatoren steigt von Zeile zu Zeile
- jede Zeile beginnt mit Angabe der Assoziativität
- Bsp.:

```
%nonassoc '>' '<'        %nonassoc: Kette von Operatoren unsinnig
%left '+' '-'            %left: implizit linksgeklammert
%left '*' '/'
%right '^'               %right: implizit rechtsgeklammert
```


Kontextabhängige Prioritäten (unäres Minus)

```
%left '+' '-'
%left '*' '/'
%right NEG          NEG: neuer Name für schon vergebenen Operator
%%
Exp   : Exp '+' Exp                 { $1 :+: $3 }
      | Exp '-' Exp                 { $1 :-: $3 }
      | Exp '*' Exp                 { $1 :*: $3 }
      | Exp '/' Exp                 { $1 :/: $3 }
      | '(' Exp ')'                 { $2 }
      | '-' Exp    %prec NEG        { Neg $2 }
      | int                         { Int $1 }
      | var                         { V $1 }
      | let var '=' Exp in Exp      { Let ($2,$4) $6 }
```

Auflösung verbleibender Mehrdeutigkeiten

- reduce/reduce-Konflikt: mehrere Produktionen passen auf Präfix
  Auflösung: Wahl der textuell ersten Produktion in der Spezifikation
- shift/reduce-Konflikt: Produktion passt auf unterschiedliche Präfixe
  Auflösung: Wahl des längsten Präfixes

Bsp. für shift/reduce-Konflikt

- Produktion: `Factor : ... | let var '=' Exp in Exp`
- Eingabestring: `let x = 3 in 1 + let y = x in y`

Klammerungsmöglichkeiten mit Präfix für `let`:

1. `(let x = 3 in 1) + let y = x in y` `x` undefiniert!
2. `let x = 3 in (1 + let y = x in y)`

Der Scannergenerator `alex`

`http://www.haskell.org/alex/`

- Vorbild: `lex` (lexical analyzer generator) von UNIX
- Eingabe: Tokenbeschreibungen durch reguläre Ausdrücke
- Ausgabe: Scanner als Haskell-Funktion
- Effizienz durch endlichen Automaten

Aufbau einer `alex`-Quelldatei

1. Haskell-Definitionen am Anfang der Scannerdatei, insbesondere Modulkopf

   ```
   {
      module Scanner where
   }
   ```

2. Angabe des Wrappers
   - `%wrapper "basic"`: erzeugt `alexScanTokens :: String -> [Token]`
   - `%wrapper "posn"`: Token mit Zeilen und Spalteninformation
   - `%wrapper "monad"`: mit einer Zustandsmonade
3. Makrodefinitionen
4. Regeln: Zuordnung von erkanntem String zu erzeugtem Token
5. Haskell-Definitionen am Ende der Scannerdatei

Einfaches Beispiel

```
{ module ScannerEx where }
%wrapper "basic"
$digit = 0-9                    -- Ziffern
$alpha = [a-zA-Z]               -- Buchstaben
tokens :-
  $white+                                ;  -- Leerzeichen
  "--".*                                 ;  -- Zeilenkommentar
  let                                    { \ s -> Tlet }
  in                                     { \ s -> Tin }
  $digit+                                { \ s -> Tint (read s) }
  [\=\+\-\*\/\(\)]                       { \ s -> Tother (head s) }
  $alpha [$alpha $digit \_ \']*          { \ s -> Tvar s }
{
data Token = Tlet | Tin | Tother Char | Tvar String | Tint Integer
           deriving (Eq,Show) }
```

Auswahlstrategie

Bsp.: Zahlen

- Regeln für Integer- und Gleitpunktzahlen definiert
- Eingabestring: `12.76e3`

Mögliche Tokenfolgen

1. `[Tint 12, Tdot, Tint 76, Tname "e3"]`
2. `[Tdouble 12.76, Tname "e3"]`
3. `[Tdouble 12.76e3]`

Wahl von `alex`:

- Auswahl einer Regel, die den längsten Eingabepräfix matcht
- gibt es mehrere solcher Regeln, matcht die textuell erste in der Spezifikation

Fallbeispiel: Interpreter für Mini-Pascal

- Möglicher Quelltext:

```
program loop;
begin
   read(a);
   s := 0;
   for i:=0 to a do s := s + i*i;
   write(s)
end.
```

- Erzeugte abstrakte Syntax:

```
([],[Read "a",
     "s" ::=: (C 0),
     For ("i",C 0,V "a")
         ["s" ::=: (V "s" :+: (V "i" :*: (V "i")))],
     Write (V "s")])
```

Scannerspezifikation Scanner.x (1)

```
{
module Scanner where
}

%wrapper "basic"

$digit = 0-9
$alpha = [a-zA-Z]
```

Scanner.x (2): Regeln

```
tokens :-
  $white+                          ;
  "--".*                           ;
  program                          { \s -> Tprogram }
  read                             { \s -> Tread }
                  ...
  end                              { \s -> Tend }
  ":="                             { \s -> Tassign }
  ";"                              { \s -> Tsemicolon }
  "+"                              { \s -> Tplus }
                  ...
  "."                              { \s -> Tdot }
  $digit+                          { \s -> Tint (read s) }
  [\=\+\-\*\/\(\)]                 { \s -> Tother (head s) }
  $alpha [$alpha $digit \_ \']*    { \s -> Tvar s }
```

Scanner.x (3): Haskell-Definitionen am Ende

```
{

data Token = Tprogram
           | Tread      | Twrite   | Tfor       | Tassign
           | Tto        | Tdo      | Tbegin     | Tend
           | Tsemicolon | Tplus    | Tminus     | Ttimes
           | Tlparen    | Trparen  | Tdot       | Tother Char
           | Tint Int               | Tvar String
           deriving (Eq,Show)

}
```

Parserspezifikation Parser.y (1)

```
{
module Parser(Declaration,Statement(..),Exp(..),Var,parse) where

import Scanner
import Char

}

%name parser
%tokentype { Token }
```

Parser.y (2): Tokenzuordnungen, Prioritäten

```
%token  program          { Tprogram }
        read             { Tread }
                ...
        for              { Tfor }
        int              { Tint $$ }
        var              { Tvar $$ }
        ":="             { Tassign }
        '+'              { Tplus }
                ...
        ')'              { Trparen }
        '.'              { Tdot }

%left '+' '-'
%left '*'
%%
```

Parser.y (3): Produktionen (a)

```
Prg :: { ([Declaration],[Statement]) }
Prg : program var ';' Stmt '.'  { ([],$4) }

Block :: { [Statement] }
Block : {- empty -}             { [] }
      | Stmt                    { $1 }
      | Block ';' Stmt          { $1 ++ $3 }

Stmt :: { [Statement] }
     : read '(' var ')'                      { [Read $3] }
     | write '(' Exp ')'                     { [Write $3] }
     | var ":=" Exp                          { [$1 ::=: $3] }
     | for var ":=" Exp to Exp do Stmt       { [For ($2,$4,$6) $8] }
     | begin Block end                       { $2 }
     | begin Block ';' end                   { $2 }
```

Parser.y (3): Produktionen (b)

```
Exp :: { Exp }
    : int             { C $1 }
    | var             { V $1 }
    | Exp '+' Exp     { $1 :+: $3 }
    | Exp '-' Exp     { $1 :-: $3 }
    | Exp '*' Exp     { $1 :*: $3 }
    | '(' Exp ')'     { $2 }
```

Parser.y (4): Haskell-Definitionen am Ende (a)

```
{

parse :: String -> ([Declaration],[Statement])
parse = parser . alexScanTokens

happyError :: [Token] -> a
happyError _ = error ("Parse error\ n")

type Var = String
type Declaration = ()
```

Parser.y (4): Haskell-Definitionen am Ende (b)

```
data Statement = Read Var
               | Write Exp
               | Var ::=: Exp
               | For (Var,Exp,Exp) [Statement]
               deriving Show

data Exp = C Int
         | V Var
         | Exp :+: Exp
         | Exp :-: Exp
         | Exp :*: Exp
         deriving Show

}
```

Interpreter.hs (1)

```
module Interpreter where

import IO
import Monad
import Parser

type Env = [(Var,Int)]
```

Interpreter.hs (2)

```
interpret :: Env -> Statement -> IO Env
interpret env x =
 case x of
  Read v -> do putStr (v++": ")
               val <- getLine
               let r = read val :: Int
               return ((v,r):env)
  Write e -> do let v = eval env e
                putStr (show (v::Int) ++"\n")
                return env
  v ::=: e -> do let r = eval env e
                 return ((v,r):env)
  For (v,l,h) stmts
   -> foldM (\ env' i -> foldM interpret ((v,i):env') stmts)
            env [eval env l .. eval env h]
```

Interpreter.hs (3)

```
eval :: Env -> Exp -> Int
eval env (C i) = i
eval env (V v) = case lookup v env of
                   Nothing -> error (show v ++ " undefined")
                   Just i  -> i
eval env (a :+: b) = eval env a + eval env b
eval env (a :-: b) = eval env a - eval env b
eval env (a :*: b) = eval env a * eval env b
```

Interpreter.hs (4)

```
main :: IO ()
main = do hSetBuffering stdout NoBuffering
          putStr "Eingabedatei: "
          s <- getLine
          str <- readFile s
          let (declarations,statements) = parse str
          env <- foldM interpret [] statements
          putStr $ "\ nEnvironment: "++ show env ++ "\ n"
          return ()
```

Bsp.: loop.pas

```
program loop;
begin
   read(a);
   s := 0;
   for i:=0 to a do  s := s + i*i;
   write(s)
end.
```

```
Interpreter> main
Eingabedatei: loop.pas
a: 4
30

Environment: [("s",30),("i",4),("s",14),("i",3),("s",5),("i",2),
              ("s",1),("i",1), ("s",0),("i",0),("s",0),("a",4)]
```

Bsp.: fibs.pas (1)

```
program fibs;
begin
   read(n);
   a := 0;
   b := 1;
   for i:=1 to n do
   begin
      tmp := a;
      a := a+b;
      b := tmp;
      write(a)
   end;
end.
```

Bsp.: fibs.pas (2)

```
Eingabedatei: fibs.pas
n: 6
1
1
2
3
5
8
Environment: [("b",5),("a",8),("tmp",5),("i",6),
              ("b",3),("a",5),("tmp",3),("i",5),
              ("b",2),("a",3),("tmp",2),("i",4),
              ("b",1),("a",2),("tmp",1),("i",3),
              ("b",1),("a",1),("tmp",1),("i",2),
              ("b",0),("a",1),("tmp",0),("i",1),
              ("b",1),("a",0),("n",6)]
```