Kapitel 12

Syntax

Die konkrete Syntax einer Programmiersprache definiert eine textuelle Darstellung für die Phrasen der abstrakten Syntax. Sie besteht aus zwei Teilen, die man als lexikalische Syntax und als kontextfreie Syntax bezeichnet. Die lexikalische Syntax regelt, welche Wörter für die Darstellung von Phrasen zur Verfügung stehen, und wie Wörter durch Zeichenreihen darzustellen sind. Die kontextfreie Syntax regelt, wie Phrasen durch Folgen von Wörtern dargestellt werden. Sie wird mithilfe von kontextfreien Grammatiken definiert.

Die konkrete Syntax muss so definiert sein, dass jede Zeichenreihe höchstens eine Phrase beschreibt. Außerdem muss es möglich sein, zu einer vorgelegten Zeichenreihe zu entscheiden, ob es sich um die Darstellung einer Phrase handelt. Falls dies der Fall ist, muss es möglich sein, die dargestellte Phrase zu rekonstruieren.

In diesem Kapitel lernen wir am Beispiel von F, wie man eine lexikalische und eine kontextfreie Syntax definiert und die entsprechenden Analysephasen realisiert. Damit sind wir in der Lage, alle Verarbeitungsphasen eines Interpreters für F zu realisieren.

Die konkrete Syntax von F orientieren wir an der von Standard ML.

12.1 Lexikalische Syntax

Für F benötigen wir drei Klassen von Wörtern:

1. *Schlüsselwörter.* Dabei handelt es sich um die folgenden Wörter:

   ```
   "if"  "then"  "else"  "fn"
   "+"  "-"  "*"  "<="  "("  ")"  ":"  "->"  "=>"
   ```

Statt von Schlüsselwörtern spricht man auch von *reservierten Wörtern*.

2. *Konstanten.* Neben den *Booleschen Konstanten*

   `"false"` `"true"`

   benötigen wir *Zahlkonstanten*. Eine *positive Zahlkonstante* ist eine nichtleere Zeichenreihe, die nur aus Ziffern besteht (zum Beispiel `"377"`). Eine *negative Zahlkonstante* ist eine Zeichenreihe mit mindestens zwei Zeichen, die mit dem Zeichen `"~"` beginnt und ansonsten nur aus Ziffern besteht (zum Beispiel `"~377"`).

3. *Bezeichner.* Ein Bezeichner ist eine Zeichenreihe *s* wie folgt:

   a) *s* besteht nur aus Buchstaben (klein und groß) und Ziffern.

   b) Das erste Zeichen von *s* ist ein Buchstabe.

   c) *s* ist weder ein Schlüsselwort noch eine Konstante.

Als Nächstes müssen wir festlegen, wie eine Folge von Wörtern durch eine Zeichenreihe dargestellt werden kann. Die drei Zeichen für Zwischenräume, Tabulatorsprünge und Zeilenwechsel bezeichnen wir als *Trennzeichen*. Zwischen zwei Wörtern können beliebig viele Trennzeichen stehen. Bestimmte Wortkombinationen können ohne Trennzeichen geschrieben werden. Beispielsweise stellt die Zeichenreihe

```
"(3+4)*7"
```

die Wortfolge

`"("`, `"3"`, `"+"`, `"4"`, `")"`, `"*"`, `"7"`

dar. Die Trennzeichen-freie Kombination von Wörtern wird für Programmiersprachen mithilfe eines Prinzips definiert, das als *longest match rule* bekannt ist. Dafür legt man fest, dass eine Zeichenreihe in Wörter zerlegt wird, indem die Wörter, von links nach rechts fortschreitend, wie folgt abgelesen werden:

1. Führende Trennzeichen werden überlesen.
2. Wenn das erste zu lesende Zeichen kein Trennzeichen ist, liest man ein möglichst langes Wort.

Diese Lesevorschrift sorgt dafür, dass die Zeichenreihe

```
"if x<=2then 1else 2+y"
```

die Wortfolge

`"if"`, `"x"`, `"<="`, `"2"`, `"then"`, `"1"`, `"else"`, `"2"`, `"+"`, `"y"`

darstellt. Dagegen stellt die Zeichenreihe

```
"ifx<=2then1else2+y"
```

die folgende Wortfolge dar:

```
"ifx", "<=", "2", "then1else2", "+", "y"
```

Wir haben jetzt die lexikalische Syntax von F vollständig definiert. Die Definition besteht aus zwei Teilen:

1. Definition der Wörter und Darstellung der Wörter durch Zeichenreihen.
2. Darstellung von Wortfolgen durch Zeichenreihen.

Der erste Teil gilt nur für F. Der zweite Teil ist allgemeiner. Wenn man ihn um das *Überlesen von Kommentaren* erweitert, gilt er auch für andere Programmiersprachen. In Standard ML beginnt ein Komentar mit `(*` und endet mit `*)`. Innerhalb eines Kommentars dürfen alle Zeichen vorkommen, mit der Einschränkung, dass die *Kommentarklammern* `(*` und `*)` nur balanciert auftreten dürfen.

Auch die Klassifikation der Wörter in Schlüsselwörter, Konstanten und Bezeichner gilt allgemein für Programmiersprachen.

Realisierung der lexikalischen Analyse

Wie überlegen uns jetzt, wie wir die lexikalische Syntax von F mit einer Prozedur `lex` realisieren können, die prüft, ob eine Zeichenreihe eine zulässige Darstellung einer Folge von Wörtern ist, und diese im Erfolgsfall liefert. Eine solche Prozedur bezeichnet man als *Lexer*.

Für die Realisierung der kontextfreien Syntax ist es vorteilhaft, wenn die Prozedur `lex` die gelesenen Wörter als Werte des in Abbildung 12.1 deklarierten Konstruktortyps `token` liefert. Diese Darstellung vereinfacht das Erkennen von Zahlkonstanten und Bezeichnern. Die für `token` gewählte Darstellung von Zahlkonstanten durch Werte des Typs `int` hat den Nachteil, dass nur kleine Zahlkonstanten dargestellt werden können.

Abbildung 12.2 zeigt die Realisierung eines Lexers für F. Wenn die vorgelegte Zeichenreihe keine gültige Darstellung einer Wortfolge ist, oder eine zu große Zahlkonstante enthält, wirft die Prozedur `lex` die Ausnahme `Error` oder `Overflow`.

Die Hilfsprozeduren `lex'`, `lexNum` und `lexA` sind *verschränkt rekursiv* definiert. Dabei handelt es sich um verschränkte iterative Rekursion, die wie einfache iterative Rekursion auch bei großen Rekursionstiefen effizient ist.

```
datatype token =
    LEQ                   (* <= *)
  | ARROW                 (* -> *)
  | DARROW                (* => *)
  | ADD                   (* +  *)
  | SUB                   (* -  *)
  | MUL                   (* *  *)
  | LPAR                  (* (  *)
  | RPAR                  (* )  *)
  | COLON                 (* :  *)
  | ICON of int           (* integer constant *)
  | IF                    (* if    *)
  | THEN                  (* then  *)
  | ELSE                  (* else  *)
  | FN                    (* fn    *)
  | FALSE                 (* false *)
  | TRUE                  (* true  *)
  | ID  of string         (* identifier *)
```

Abbildung 12.1: Darstellung der Wörter von F

Das Hauptargument der Prozeduren `lex'`, `lexNum` und `lexA` ist jeweils die Liste `cs` der noch zu lesenden Zeichen und erscheint ganz rechts. Bei den anderen Argumenten handelt es sich um Akkumulatorargumente, die für die iterative Formulierung der Prozeduren benötigt werden. Ganz links steht jeweils die Liste `ts` der bereits gelesenen Wörter in reversierter Ordnung.

Die Prozedur `lexNum` versucht eine Zahlkonstante zu lesen. Das Argument `s` repräsentiert das Vorzeichen der zu lesenden Zahlkonstanten und hat den Wert `~1` oder 1. Das Argument `n` repräsentiert die Zahl, die den bisher gelesenen Ziffern entspricht.

Die Prozedur `lexA` versucht eine Wort zu lesen, das aus Buchstaben und Ziffern besteht und mit einem Buchstaben anfängt. Das Argument `xs` ist die Liste der bisher gelesenen Zeichen in reversierter Ordnung.

Abbildung 12.3 zeigt den *Dependenzgraphen* der Prozeduren des Lexers. Eine Kante von einer Prozedur p zu einer Prozedur q bedeutet, dass q von p aufgerufen werden kann. Zyklen, die durch mehr als einen Knoten laufen, stellen verschränkte Rekursionen dar.

```
val ord0 = ord #"0"

fun lexA' "if"    = IF
  | lexA' "then"  = THEN
  | lexA' "else"  = ELSE
  | lexA' "fn"    = FN
  | lexA' "false" = FALSE
  | lexA' "true"  = TRUE
  | lexA' s       = ID s

fun lex' ts nil                 = rev ts
  | lex' ts (#" " ::cs)         = lex' ts cs
  | lex' ts (#"\n"::cs)         = lex' ts cs
  | lex' ts (#"\t"::cs)         = lex' ts cs
  | lex' ts (#"<" :: #"=" ::cs) = lex' (LEQ::ts) cs
  | lex' ts (#"-" :: #">" ::cs) = lex' (ARROW::ts) cs
  | lex' ts (#"=" :: #">" ::cs) = lex' (DARROW::ts) cs
  | lex' ts (#"+" ::cs)         = lex' (ADD::ts) cs
  | lex' ts (#"-" ::cs)         = lex' (SUB::ts) cs
  | lex' ts (#"*" ::cs)         = lex' (MUL::ts) cs
  | lex' ts (#"(" ::cs)         = lex' (LPAR::ts) cs
  | lex' ts (#")" ::cs)         = lex' (RPAR::ts) cs
  | lex' ts (#":" ::cs)         = lex' (COLON::ts) cs
  | lex' ts (c    ::cs)         =
      if Char.isDigit c
        then lexNum ts 1 0 (c::cs)
      else if c = #"~" andalso not(null cs)
        then lexNum ts ~1 0 cs
      else if Char.isAlpha c
        then lexA ts [c] cs
      else raise Error

and lexNum ts s n cs =
  if not(null cs) andalso Char.isDigit (hd cs)
  then lexNum ts s (10*n + (ord(hd cs) - ord0)) (tl cs)
  else lex'(ICON(s*n)::ts) cs

and lexA ts xs cs =
  if not(null cs) andalso Char.isAlphaNum(hd cs)
  then lexA ts (hd cs::xs) (tl cs)
  else lex'(lexA'(implode(rev xs)) :: ts) cs

fun lex s =lex' nil (explode s)

lex : string -> token list (* Error, Overflow *)
```

Abbildung 12.2: Lexer für F

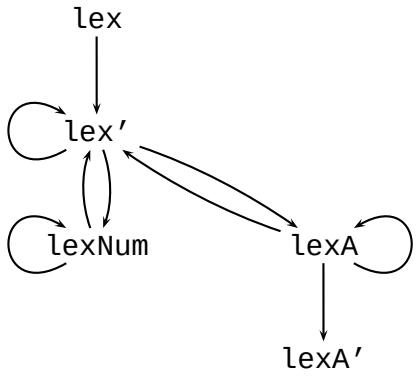

Abbildung 12.3: Dependenzgraph des Lexers

12.2 Kontextfreie Syntax für Typen

Für die Beschreibung der kontextfreien Syntax von F benötigen wir eine ganze Reihe neuer Konzepte. Wir beginnen mit der kontextfreien Syntax für Typen.

Die abstrakte Syntax von Typen haben wir in Abschnitt 11.1 wie folgt definiert:

$$t \in \mathit{Ty} \;=\; \texttt{bool} \mid \texttt{int} \mid t_1 \to t_2$$

Für die kontextfreie Syntax benötigen wir Klammern:

```
(int -> int) -> (int -> int)
```

Wie in Standard ML soll es möglich sein, die Klammern auf der rechten Seite des Pfeils wegzulassen:

```
(int -> int) -> int -> int
```

Außerdem soll es möglich sein, beliebig viele zusätzliche Klammern zu setzen:

```
((int -> int) -> (int -> int))
(((int) -> int) -> (int -> int))
((((int)) -> int) -> (int -> int))
```

Die Möglichkeit zum Weglassen und Hinzufügen von Klammern hat zur Folge, das jeder Typ durch unendlich viele verschiedene Wortfolgen dargestellt werden kann. Andererseits muss sichergestellt sein, dass jede Wortfolge höchstens einen Typ darstellt.

Kontextfreie Syntaxen beschreibt man mit *kontextfreien Grammatiken*. Die kontextfreie Syntax von Typen definieren wir durch die folgende kontextfreie Gram-

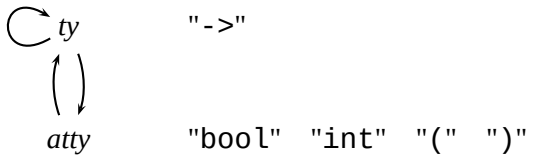

Abbildung 12.4: Dependenzgraph der Grammatik für Typen

matik:[1]

ty	= *atty* | *atty* `"->"` *ty*	Typ
atty	= `"bool"` | `"int"` | `"("` *ty* `")"`	atomarer Typ

Die Grammatik führt zwei *Nonterminale ty* und *atty* ein. Für *ty* werden zwei *Alternativen* festgelegt, für *atty* drei. Jede Alternative ist eine Folge von Nonterminalen und Wörtern. Im Kontext einer Grammatik werden Wörter auch als *Terminale* bezeichnet.

Abbildung 12.4 zeigt den *Dependenzgraphen* der Grammatik. Eine Kante von einem Nonterminal N zu einem Nonterminal N' bedeutet, dass N' in einer der Alternativen von N vorkommt. Für jedes Nonterminal sind die Wörter angegeben, die in seinen Alternativen vorkommen.

Eine Grammatik beschreibt eine Menge von *Syntaxbäumen*. Abbildung 12.5 zeigt einen Syntaxbaum der obigen Grammatik. Syntaxbäume sind geordnete Bäume (siehe Abschnitt 5.5) wie folgt:

1. Jedes Blatt ist mit einem Wort markiert.
2. Die Wurzel und die inneren Knoten sind mit Nonterminalen markiert.
3. Für jeden mit einem Nonterminal markierten Knoten gilt: wenn N die Marke des Knotens ist und $S_1, \ldots, S_n$ die Marken der Nachfolger des Knotens sind (in der Reihenfolge der Nachfolger), dann spezifiziert die Grammatik für das Nonterminal N die Alternative $S_1, \ldots, S_n$.

Jeder Syntaxbaum hat also mindestens zwei Knoten.

Die Grenze eines Syntaxbaums (also die Folge der Marken der Blätter von links nach rechts) bezeichnen wir als *den durch den Baum dargestellten Satz*. Die *Sätze* einer Grammatik sind genau die Wortfolgen, die durch die Syntaxbäume der Grammatik dargestellt werden können.

[1] Für Experten: Wir schreiben kontextfreie Grammatiken in BNF-Notation und verzichten auf ein Startsymbol.

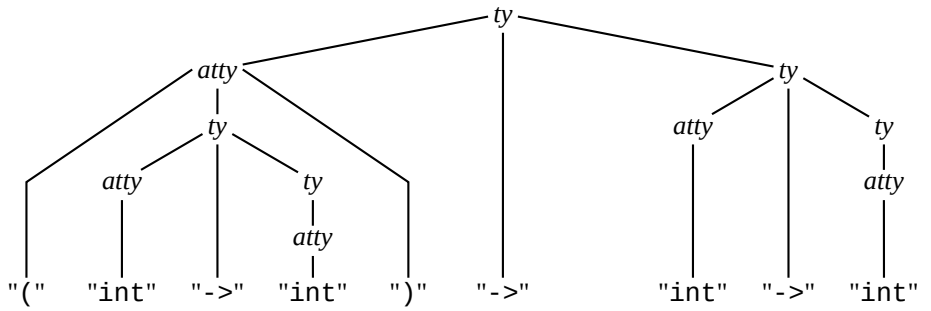

Abbildung 12.5: Ein Syntaxbaum

Die Sätze unserer kontextfreien Grammatik sind genau die Wortfolgen, die wir als Darstellungen von Typen zulassen wollen. Der Syntaxbaum in Abbildung 12.5 stellt den folgenden Satz dar:

```
(int -> int) -> int -> int
```

Eine Grammatik heißt *eindeutig*, wenn es zu einer Wortfolge *ts* und zu einem Nonterminal *N* höchstens einen Syntaxbaum gibt, der *ts* darstellt und dessen Wurzel mit *N* markiert ist. Eine Grammatik ist also genau dann eindeutig, wenn verschiedene Syntaxbäume, deren Wurzel mit demselben Nonterminal markiert ist, immer verschiedene Sätze darstellen. Wir interessieren uns nur für eindeutig Grammatiken. Unsere Grammatik für Typen ist eindeutig.

Jeder Syntaxbaum unserer Beispielsgrammatik stellt genau einen Typ $t \in Ty$ dar. Es gibt also eine Funktion, die jedem Syntaxbaum unserer Beispielsgrammatik einen Typ zuordnet. Da diese Funktion offensichtlich ist, verzichten wir auf eine formale Definition.

Eine Wortfolge *ts* heißt *Darstellung eines Typs t* genau dann, wenn es einen Syntaxbaum gibt, der *ts* und *t* darstellt. Offensichtlich kann jeder Typ durch unendlich viele Sätze der Grammatik dargestellt werden. Andererseits existiert zu einer Wortfolge *ts* höchstens ein Typ *t*, so dass *ts* eine Darstellung von *t* ist.

Abbildung 12.6 stellt den Zusammenhang zwischen kontextfreier und abstrakter Syntax grafisch dar.

Realisierung der syntaktischen Analyse durch rekursiven Abstieg

Für die syntaktische Analyse benötigen wir eine Prozedur, die prüft, ob eine Wortfolge einen Typ darstellt, und diesen im Erfolgsfall liefert. Eine solche Prozedur bezeichnet man als *Parser*.

Für die Realisierung von Parsern gibt es verschiedene algorithmische Techniken.

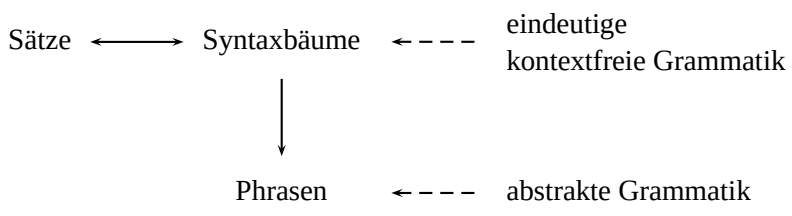

Abbildung 12.6: Zusammenhang zwischen kontextfreier und abstrakter Syntax

Wir zeigen hier eine einfache Technik, die als *rekursiver Abstieg* bekannt ist. In der Praxis benutzt man sogenannte *Parsergeneratoren*, die zu einer Grammatik und einer Beschreibung der Abbildung auf die abstrakte Syntax automatisch einen entsprechenden Parser liefern. Diese Parsergeneratoren überprüfen auch automatisch, ob die vorgelegte Grammatik eindeutig ist.

Wir entwickeln zuerst einen *Tester* für unsere Beispielsgrammatik. Das ist eine Prozedur, die entscheidet, ob eine Wortfolge ein Satz der Grammatik ist. Den Tester werden wir dann zu einem Parser erweitern.

Für jedes der zwei Nonterminale der Grammatik führen wir eine Prozedur ein:

```
ty   : token list -> token list  (* Error *)
atty : token list -> token list  (* Error *)
```

Die Prozedur `ty` bekommt eine Wortliste und prüft, ob die Wortliste mit einen Satz für das Nonterminal *ty* beginnt. Wenn dies der Fall ist, werden die dem Satz nachfolgenden Wörter geliefert. Dabei wird versucht, einen möglichst langen Satz zu lesen. Zum Beispiel:

```
ty [ID"int", ARROW, ID"int", ARROW, ID"int", DARROW]
```

```
[DARROW] : token list
```

Wenn kein Satz gelesen werden kann, wird die Ausnahme `Error` geliefert. Die Prozedur `atty` arbeitet analog.

Die Prozeduren `ty` und `atty` bezeichnen wir als *Testprozeduren*. Abbildung 12.7 zeigt die Testprozeduren sowie eine Prozedur `test`, die testet, ob eine Wortfolge ein Satz der Grammatik füt Typen ist. Die Abbildung zeigt auch die Grammatik, damit Sie sehen können, wie die Testprozeduren aus den Gleichungen der Grammatik abgeleitet sind. Die Testprozeduren sind entsprechend der Definition der Nonterminale *ty* und *atty* verschränkt rekursiv deklariert. Die verschränkte Rekursion terminiert, da jede erfolgreiche Anwendung der Testprozeduren die Wortliste jeweils um mindestens ein Wort verkürzt.

Der Dependenzgraph der Grammatik (siehe Abbildung 12.4) entspricht genau dem Dependenzgraphen der Testprozeduren `ty` und `atty`. Die angegebenen

```
  ty = atty | atty "->" ty
atty = "bool" | "int" | "(" ty ")"

fun ty ts = case atty ts of
              ARROW::tr => ty tr
            |    tr     => tr

and atty (ID"bool"::ts) = ts
  | atty (ID"int" ::ts) = ts
  | atty (LPAR    ::ts) = (case ty ts of
                             RPAR::tr => tr
                           |    _     => raise Error)
  | atty       _        = raise Error

fun test ts = null (ty ts) handle Error => false
val test : token list -> bool
```

Abbildung 12.7: Tester für die Grammatik für Typen

Wörter werden jeweils durch die entsprechende Parsingprozedur gelesen.

Die Prozeduren `ty` und `atty` müssen sich jeweils für eine der Alternativen für die Nonterminale *ty* und *atty* entscheiden. Die Entscheidung erfolgt jeweils mithilfe des ersten, noch nicht gelesenen Wortes. Man spricht von einer *Ein-Wort-Vorrausschau*. Die Prozedur `ty` liest zunächst einen Satz für *atty*, da beide Alternativen entsprechend beginnen müssen. Danach wird die Entscheidung zwischen der ersten und zweiten Alternative mithilfe des ersten, noch nicht gelesenen Worts getroffen.

Wir interessieren uns hier nur für kontextfreie Grammatiken, deren Sätze mithilfe einer Ein-Wort-Vorrausschau erkannt werden können. Bei der Formulierung unserer Grammatik für Typen können wir diese Eigenschaft noch offensichtlicher machen, indem wir die Alternativen für *ty* mit sogenannten *Optionsklammern* beschreiben:

```
  ty = atty [ "->" ty ]
atty = "bool" | "int" | "(" ty ")"
```

Zwischen den Syntaxbäumen der Grammatik und den Rekursionsbäumen der Testprozeduren besteht ein sehr enger Zusammenhang. Betrachten Sie dazu den Syntaxbaum in Abbildung 12.5. Der Rekursionsbaum für den Aufruf der Testprozedur `ty` mit dem durch den Syntaxbaum dargestellten Satz hat offensichtlich genau die Form des Syntaxbaums, wenn wir von den mit Wörtern markierten

```
fun match (a,ts) t = if null ts orelse hd ts <> t
                     then raise Error
                     else (a, tl ts)

fun combine a ts p f = let val (a',tr) = p ts
                       in (f(a,a'), tr)
                       end

fun ty ts = case atty ts of
              (a, ARROW::tr) => combine a tr ty Arrow
            |      ats       => ats

and atty (ID"int" ::ts) = (Int ,ts)
  | atty (ID"bool"::ts) = (Bool,ts)
  | atty (LPAR    ::ts) = match (ty ts) RPAR
  | atty       _        = raise Error

fun parse ts = case ty ts of
                 (a,[]) => a
               |    _   => raise Error

val parse : token list -> ty
```

Abbildung 12.8: Parsingprozeduren für Typen

Blätter absehen. Die Syntaxbäume der Grammatik sind also bis auf Details die Rekursionsbäume der abgeleiteten Testprozeduren.

Mithilfe dieser Beobachtung ist es einfach, die Testprozeduren so zu erweitern, dass sie im Erfolgsfall den dargestellten Typ liefern. Die erweiterten Testprozeduren werden als *Parsingprozeduren* bezeichnet und haben den Typ

```
token list -> ty * token list
```

Abbildung 12.8 zeigt den damit konstruierten Parser. Die Hilfprozeduren `match` und `combine` vereinfachen die Formulierung der Parsingprozeduren.

12.3 Kontextfreie Syntax für arithmetischen Ausdrücke

Als Nächstes entwickeln wir eine kontextfreie Syntax für Ausdrücke, die mit Konstanten, Bezeichnern und den arithmetischen Operationen $+$, $-$, $*$ und $\leq$ gebildet werden können. Bei der Darstellung solcher Ausdrücke können besonders viele Klammern eingespart werden. Beispielsweise soll

```
3*x + x*y*z - y <= x+y
```

denselben Ausdruck darstellen wie

exp = *asexp* ["<=" *asexp*]
asexp = [*asexp* ("+" | "-")] *mulexp*
mulexp = [*mulexp* "*"] *atexp*
atexp = *constant* | *identifier* | "(" *exp* ")"

Abbildung 12.9: Kontextfreie Grammatik für arithmetische Ausdrücke

```
(((3*x) + ((x*y)*z)) - y) <= (x+y)
```

Im Rahmen der kontextfreien Syntax bezeichnet man Symbole für zweistellige Operationen, die wie + und * zwischen ihre Argumente geschrieben werden, als *Infixoperatoren*. Zunächst vereinbaren wir die folgende *Rangfolge* für die beteiligten Infixoperatoren:

```
<=
+  -
*
```

Es gibt also drei Rangstufen, wobei + und - auf derselben Stufe stehen. Wenn Klammern fehlen, müssen Infixoperatoren mit tieferem Rang tiefer geklammert werden als Infixoperatoren mit höherem Rang. Damit können wir den Ausdruck

```
3*x + x*y*z - y <= x+y
```

wie folgt klammern:

```
((3*x) + (x*y*z) - y) <= (x+y)
```

Die Rangfolge für die Infixoperatoren realisiert also die übliche Punkt-vor-Strich-Regel. Statt von einer Rangfolge spricht man auch von einer *Präzedenz*.

Da die Infixoperatoren +, - und * ohne Klammerung mehrfach hintereinander angewendet werden können (zum Beispiel `x*y*z`), benötigen wir zusätzliche Klammerungsregeln. Wir vereinbaren, dass innerhalb einer Rangstufe fehlende Klammern von links her gesetzt werden müssen. Damit können wir die fehlenden Klammern bei

```
((3*x) + (x*y*z) - y) <= (x+y)
```

wie folgt setzen:

```
(((3*x) + ((x*y)*z)) - y) <= (x+y)
```

Man spricht von *linksassoziativer Klammerung*. Bei *rechtsassoziativer Klammerung* würde man `x*y*z` zu `x*(y*z)` klammern.

Die Grammatik in Abbildung 12.9 definiert eine kontextfreie Syntax für arithmetischen Ausdrücke gemäß den gerade besprochenen Regeln. Die Formulierung der

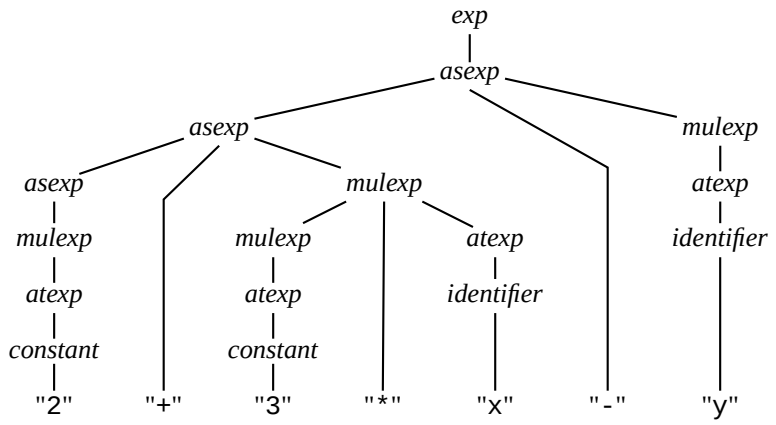

Abbildung 12.10: Ein Syntaxbaum der Grammatik für arithmetische Ausdrücke

Grammatik verwendet neben Optionsklammern eine weitere abkürzende Notation. Ohne abkürzende Notation können wir die Grammatik wie folgt formulieren:

$$
\begin{aligned}
\mathit{exp} &= \mathit{asexp} \mid \mathit{asexp}\ \texttt{"<="}\ \mathit{asexp} \\
\mathit{asexp} &= \mathit{mulexp} \mid \mathit{asexp}\ \texttt{"+"}\ \mathit{mulexp} \mid \mathit{asexp}\ \texttt{"-"}\ \mathit{mulexp} \\
\mathit{mulexp} &= \mathit{atexp} \mid \mathit{mulexp}\ \texttt{"*"}\ \mathit{atexp} \\
\mathit{atexp} &= \mathit{constant} \mid \mathit{identifier} \mid \texttt{"("}\ \mathit{exp}\ \texttt{")"}
\end{aligned}
$$

Die Definitionen für die Nonterminale *constant* und *identifier* sind gemäß den lexikalischen Definitionen in Abschnitt 12.1 zu ergänzen. Das Nonterminal *constant* bekommt also alle Konstanten als Alternativen, und *identifier* alle Bezeichner. Abbildung 12.10 zeigt einen Syntaxbaum der Grammatik. Die Grammatik ist eindeutig.

Abbildung 12.11 zeigt den Dependenzgraphen der Grammatik. Dieser verdeutlicht das Konstruktionsprinzip hinter der Grammatik: ein Nonterminal für jede Rangstufe und zusätzlich ein Nonterminal für atomare Ausdrücke.

Realisierung des Parsers

Damit die Infixoperatoren `+`, `-` und `*` linksassoziativ gruppiert werden, definiert unsere Grammatik die Nonterminale *asexp* und *mulexp linksrekursiv*. Das hat zur Folge, dass die entsprechenden Testprozeduren divergieren, da sie sich sofort selbst aufrufen.

Um dieses Problem zu lösen, gehen wir zu der Grammatik in Abbildung 12.12 über, die die Infixoperatoren `+`, `-` und `*` rechtsassoziativ gruppiert und damit die

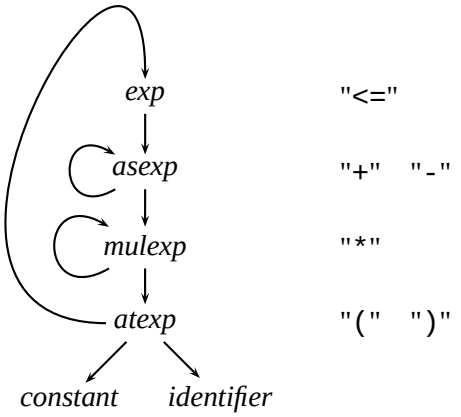

Abbildung 12.11: Dependenzgraph der Grammatik für arithmetische Ausdrücke

störenden Linksrekursionen vermeidet. Da die neue Grammatik dieselben Sätze wie die alte hat, ist ein Tester für die neue Grammatik auch ein Tester für die alte.

Den so erhaltenen Tester erweitern wir wieder zu einem Parser. Die Konstruktion der Ausdrücke ist diesmal jedoch komplizierter, da wir die Fehlgruppierung der Infixoperatoren +, - und * korrigieren müssen. Um die Korrektur einfach realisieren zu können, modifizieren wir die Grammatik erneut, indem wir zwei *Hilfsnonterminale asexp′* und *mulexp′* einführen (siehe Abbildung 12.13). Die modifizierte Grammatik liefert dieselben Sätze wie bisher. Abbildung 12.14 zeigt einen Syntaxbaum der modifizierten Grammatik. Bei den Syntaxbäumen der modifizierten Grammatik erlauben wir Blätter, die mit den Nonterminalen *asexp′* oder *mulexp′* markiert sind. Diese sogenannten *Epsilon-Blätter* repräsentieren die *leeren Alternativen* für die Hilfsnonterminale *asexp′* und *mulexp′* und tragen kein Wort zum dargestellten Satz bei.

Abbildung 12.15 zeigt den aus der modifizierten Grammatik abgeleiteten Parser.

12.4 Kontextfreie Syntax für F

Die Grammatik in Abbildung 12.16 definiert eine kontextfreie Syntax für F. Die Syntax für Typen entspricht der aus Abschnitt 12.2. Die Syntax für Ausdrücke erweitert die für arithmetische Ausdrücke um Konditionale, Abstraktionen und Applikationen. Applikationen werden vor den Infixoperationen geklammert und linksassoziativ gruppiert.

Um eine brauchbare Grammatik für die Ableitung der Parsingprozeduren zu be-

$$
\begin{aligned}
\mathit{exp} &= \mathit{asexp}\ [\ \texttt{"<="}\ \mathit{asexp}\ ] \\
\mathit{asexp} &= \mathit{mulexp}\ [\ (\texttt{"+"}\ |\ \texttt{"-"})\ \mathit{asexp}\ ] \\
\mathit{mulexp} &= \mathit{atexp}\ [\ \texttt{"*"}\ \mathit{mulexp}\ ] \\
\mathit{atexp} &= \mathit{constant}\ |\ \mathit{identifier}\ |\ \texttt{"("}\ \mathit{exp}\ \texttt{")"}
\end{aligned}
$$

Abbildung 12.12: Grammatik nach Elimination der Linksrekursionen

$$
\begin{aligned}
\mathit{exp} &= \mathit{asexp}\ [\ \texttt{"<="}\ \mathit{asexp}\ ] \\
\mathit{asexp} &= \mathit{mulexp}\ \ \mathit{asexp}' \\
\mathit{asexp}' &= [\ (\texttt{"+"}\ |\ \texttt{"-"})\ \ \mathit{mulexp}\ \ \mathit{asexp}'\ ] \\
\mathit{mulexp} &= \mathit{atexp}\ \ \mathit{mulexp}' \\
\mathit{mulexp}' &= [\ \texttt{"*"}\ \ \mathit{atexp}\ \ \mathit{mulexp}'\ ] \\
\mathit{atexp} &= \mathit{constant}\ |\ \mathit{identifier}\ |\ \texttt{"("}\ \mathit{exp}\ \texttt{")"}
\end{aligned}
$$

Abbildung 12.13: Grammatik nach Einführung der Hilfsnonterminale

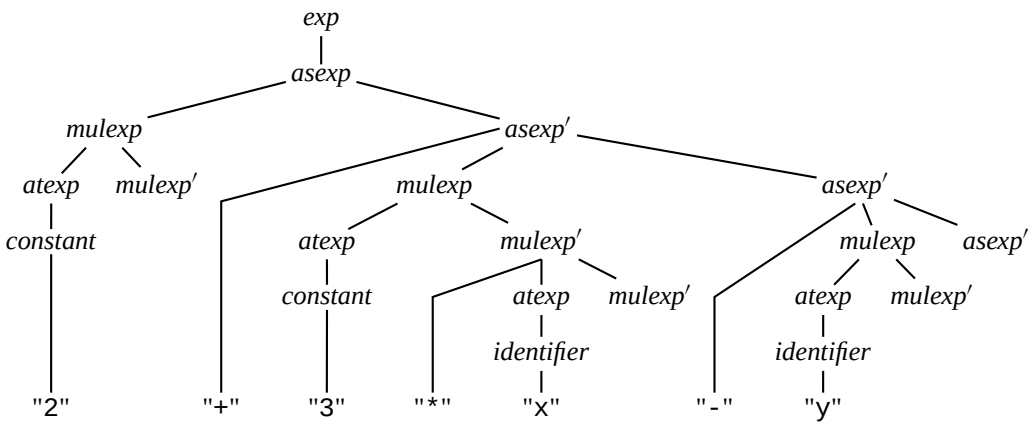

Abbildung 12.14: Ein Syntaxbaum der Grammatik in Abbildung 12.13

```
fun match (a,ts) t = if null ts orelse hd ts <> t
                     then raise Error
                     else (a, tl ts)

fun combine a ts p f = let val (a',tr) = p ts
                       in (f(a,a'), tr)
                       end

fun opa ops (a,a') = Op(a,ops,a')

fun exp ts = case asexp ts of
               (a, LEQ::tr) => combine a tr asexp (opa Leq)
             |     ats      => ats

and asexp ts = asexp' (mulexp ts)

and asexp'(a, ADD::ts) = asexp'(combine a ts mulexp (opa Add))
  | asexp'(a, SUB::ts) = asexp'(combine a ts mulexp (opa Sub))
  | asexp'    ats      = ats

and mulexp ts = mulexp'(atexp ts)

and mulexp'(a, MUL::ts) = mulexp'(combine a ts atexp (opa Mul))
  | mulexp'    ats      = ats

and atexp (FALSE ::ts) = (Con False, ts)
  | atexp (TRUE  ::ts) = (Con True , ts)
  | atexp (ICON n::ts) = (Con(IC n), ts)
  | atexp (ID s  ::ts) = (Id s     , ts)
  | atexp (LPAR  ::ts) = match (exp ts) RPAR
  | atexp      _       = raise Error

fun parse ts = case exp ts of
                 (a, nil) => a
               |    _     => raise Error
val parse : token list -> exp
```

Abbildung 12.15: Parser für arithmetische Ausdrücke

$$
\begin{aligned}
\mathit{atty} &= \texttt{"bool"} \mid \texttt{"int"} \mid \texttt{"("}\ \mathit{ty}\ \texttt{")"} \\
\mathit{ty} &= \mathit{atty}\ [\ \texttt{"->"}\ \mathit{ty}\ ] \\
\mathit{exp} &= \texttt{"if"}\ \mathit{exp}\ \texttt{"then"}\ \mathit{exp}\ \texttt{"else"}\ \mathit{exp} \\
&\quad \mid \texttt{"fn"}\ \mathit{identifier}\ \texttt{":"}\ \mathit{ty}\ \texttt{"=>"}\ \mathit{exp} \\
&\quad \mid \mathit{asexp}\ [\ \texttt{"<="}\ \mathit{asexp}\ ] \\
\mathit{asexp} &= [\ \mathit{asexp}\ (\texttt{"+"} \mid \texttt{"-"})\ ]\ \mathit{mulexp} \\
\mathit{mulexp} &= [\ \mathit{mulexp}\ \texttt{"*"}\ ]\ \mathit{apexp} \\
\mathit{apexp} &= [\ \mathit{apexp}\ ]\ \mathit{atexp} \\
\mathit{atexp} &= \mathit{constant} \mid \mathit{identifier} \mid \texttt{"("}\ \mathit{exp}\ \texttt{")"}
\end{aligned}
$$

Abbildung 12.16: Kontextfreie Grammatik für F

$$
\begin{aligned}
\mathit{atty} &= \texttt{"bool"} \mid \texttt{"int"} \mid \texttt{"("}\ \mathit{ty}\ \texttt{")"} \\
\mathit{ty} &= \mathit{atty}\ [\ \texttt{"->"}\ \mathit{ty}\ ] \\
\mathit{exp} &= \texttt{"if"}\ \mathit{exp}\ \texttt{"then"}\ \mathit{exp}\ \texttt{"else"}\ \mathit{exp} \\
&\quad \mid \texttt{"fn"}\ \mathit{identifier}\ \texttt{":"}\ \mathit{ty}\ \texttt{"=>"}\ \mathit{exp} \\
&\quad \mid \mathit{asexp}\ [\ \texttt{"<="}\ \mathit{asexp}\ ] \\
\mathit{asexp} &= \mathit{mulexp}\ \ \mathit{asexp}' \\
\mathit{asexp}' &= [\ (\texttt{"+"} \mid \texttt{"-"})\ \ \mathit{mulexp}\ \ \mathit{asexp}'\ ] \\
\mathit{mulexp} &= \mathit{apexp}\ \ \mathit{mulexp}' \\
\mathit{mulexp}' &= [\ \texttt{"*"}\ \ \mathit{apexp}\ \ \mathit{mulexp}'\ ] \\
\mathit{apexp} &= \mathit{atexp}\ \ \mathit{apexp}' \\
\mathit{apexp}' &= [\ \mathit{atexp}\ \ \mathit{apexp}'\ ] \\
\mathit{atexp} &= \mathit{constant} \mid \mathit{identifier} \mid \texttt{"("}\ \mathit{exp}\ \texttt{")"}
\end{aligned}
$$

Abbildung 12.17: Grammatik für die Ableitung der Parsingprozeduren

```
fun firstAtexp (TRUE  ::_) = true
  | firstAtexp (FALSE ::_) = true
  | firstAtexp (ICON _::_) = true
  | firstAtexp (ID _  ::_) = true
  | firstAtexp (LPAR  ::_) = true
  | firstAtexp      _      = false

fun exp (IF::ts) = let val (a1,ts1) = match (exp ts) THEN
                       val (a2,ts2) = match (exp ts1) ELSE
                       val (a3,ts3) = exp ts2
                   in  (If(a1,a2,a3), ts3)
                   end

  | exp (FN::ID s::COLON::ts) =
                   let val (a1,ts1) = match (ty ts) DARROW
                       val (a2,ts2) = exp ts1
                   in  (Abs(s,a1,a2), ts2)
                   end

  | exp ts       = case asexp ts of
                     (a, LEQ::tr) => combine a tr asexp (opa Leq)
                   |     ats      => ats

and appexp ts = appexp'(atexp ts)

and appexp'(a,ts) = if firstAtexp ts
                    then appexp'(combine a ts atexp App)
                    else (a,ts)
and  ···
```

Abbildung 12.18: Parsingprozeduren für F

kommen, müssen wir, wie in Abschnitt 12.3 gezeigt, die Linksrekursionen eliminieren und entsprechende Hilfsnonterminale einführen. Damit bekommen wir die Grammatik in Abbildung 12.17. Diese entspricht in vielen Teilen den bereits bekannten Grammatiken für Typen und für arithmetische Ausdrücke. Die Ableitung des Parsers gelingt jetzt problemlos. Abbildung 12.18 zeigt die Parsingprozeduren für das um zusätzliche Alternativen erweiterte Nonterminal *exp* und das neue Nonterminal *apexp* für Applikationen.

12.5 Bemerkungen zur konkreten Syntax von Standard ML

Die Rangfolge und die Gruppierung der Infixoperatoren für Standard ML Ausdrücke ist wie folgt:

`orelse`	rechtsassoziativ
`andalso`	rechtsassoziativ
`before`	linksassoziativ
`o  :=`	linksassoziativ
`=  <>  <  >  <=  >=`	linksassoziativ
`::  @`	rechtsassoziativ
`+  -  ^`	linksassoziativ
`*  /  div  mod`	linksassoziativ

Dabei ist zu beachten, dass `orelse` und `also` sematisch gesehen keine Operationen sind, sondern aus Konditionalen abgeleitete Formen sind.

Wie in F stehen Applikationen auf der niedrigsten Rangstufe und werden linksassoziativ geklammert. Die atomaren Ausdrücke von Standard ML sind:

- Konstanten
- Bezeichner
- geklammerte Ausdrücke
- Tupelkonstruktionen (*exp* `,` ... `,` *exp*)
- Listenkonstruktionen [*exp* `,` ... `,` *exp*]
- Sequentialisierungen (*exp* `;` ... `;` *exp*)
- Let-Ausdrücke
- Projektionen (zum Beispiel `#2`)
- Op-Ausdrücke (zum Beispiel `op::`)

Es gibt Konstanten für ganze Zahlen, Gleitkommazahlen, Zeichen und Zeichenreihen. Außerdem sind

```
true   false   ()   nil   []   ~   ref
```

Konstanten. Die Tatsache, dass die konkrete Syntax dem Negationsoperator den Status einer Konstante gibt, hat einige Konsequenzen. Beispielsweise kann man den folgenden Ausdruck schreiben:

```
map ~ [1, 2, 3]
[~1, ~2, ~3] : int list
```

Andererseits wird der Ausdruck

```
~ ~ 5
```

zu `(~ ~) 5` geklammert und liefert einen Typfehler. Bei der Konstruktion der abstrakten Syntax werden Anwendungen des Negationsoperators in einstellige Operatoranwendungen übersetzt. Die nicht-applikativen Auftreten des Negationsoperators (zum Beispiel das zweite Auftreten in `(~ ~)`) werden in den Ausdruck

```
fn x => ~x
```

übersetzt.

Bezeichner dürfen die Zeichen "`'`" und "_" enthalten. Außerdem gibt es zusammengesetzte Bezeichner wie `S.T.x`. Schließlich gibt es sogenannte *symbolische Bezeichner*. Dabei handelt es sich um nichtleere Zeichenreihen, die beliebig aus den Zeichen

```
!  %  &  $  #  +  -  /  :  <  =  >  ?  @  \  ~  ‘  ^  |  *
```

zusammengesetzt werden können, soweit es sich nicht um ein Schlüsselwort handelt. Das erklärt einige bisher völlig mysteriöse Fehlermeldungen. Beispielsweise wird

```
1+~2
```

in Standard ML in die Wörter "`1`", "`+~`" und "`2`" zerlegt (wegen der longest match rule, "`+~`" ist ein symbolischer Bezeichner).

Schließlich geben wir noch die Schlüsselwörter von Standard ML an, die wie normale Bezeichner aussehen:[2]

```
abstype  and  andalso  as  before  case  datatype  div  do
else  end  eqtype  exception  fn  fun  functor  handle  if
in  include  infix  infixr  let  local  mod  nonfix  o  of
op  open  orelse  raise  rec  sharing  sig  signature  struct
structure  then  type  val  where  with  withtype  while
```

[2] Für Experten: Wir vereinfachen hier etwas und behandeln `ref` und Bezeichner mit vordefiniertem Infixstatus als Schlüsselwörter.

Übungen

Aufgabe 12.1 (Syntaxbäume)

1. Zeichnen Sie einen Syntaxbaum für den Satz

   ```
   (bool -> int -> bool) -> int
   ```

   gemäß der Grammatik für Typen in Abschnitt 12.2. Wie viele solcher Syntaxbäume gibt es?

2. Zeichnen Sie einen Syntaxbaum für den Satz

   ```
   3*(2*y+x)*z
   ```

   gemäß der linksrekursiven Grammatik für arithmetische Ausdrücke in Abbildung 12.9. Wie viele solcher Syntaxbäume gibt es?

3. Zeichnen Sie je einen Syntaxbaum für den Satz `2+3-4+7` gemäß

   a) der linksrekursiven Grammatik für arithmetische Ausdrücke in Abbildung 12.9.

   b) der rechtsrekursiven Grammatik für arithmetische Ausdrücke in Abbildung 12.12.

   c) der modifizierten rechtsrekursiven Grammatik für arithmetische Ausdrücke in Abbildung 12.13.

Aufgabe 12.2 (Mehrdeutige Grammatik) Zeigen Sie mit einem Gegenbeispiel, dass die kontextfreie Grammatik

exp = `"x"` | *exp exp*

nicht eindeutig ist. Geben Sie eine eindeutige Grammatik an, die dieselben Sätze darstellt.

Aufgabe 12.3 (Dependenzgraph) Zeichnen Sie den Dependenzgraphen der Grammatik in Abbildung 12.17.

Aufgabe 12.4 (Darstellung von Typen) Gegeben sei die Typdeklaration

```
datatype ty  = Bool | Int | Arrow of ty * ty
```

Schreiben Sie eine Prozedur

```
ty : ty -> string
```

die Typen durch Zeichenreihen darstellt. Die Darstellung soll gemäß der kontextfreien Grammatik in Abschnitt 12.2 erfolgen und möglichst wenig Klammern und Trennzeichen enthalten. Führen Sie für jedes Nonterminal der Grammatik eine Hilfsprozedur ein.

Aufgabe 12.5 (Darstellung von Ausdrücken) Seien die Typdeklarationen

```
datatype ops = Add | Sub | Mul | Leq

datatype exp = Con of int | Op of exp * ops * exp
```

gegeben. Schreiben Sie eine Prozedur

```
exp : exp -> string
```

die Ausdrücke durch Zeichenreihen darstellt. Die Darstellung soll gemäß der linksrekursiven Grammatik in Abbildung 12.9 erfolgen und möglichst wenig Klammern und Trennzeichen enthalten. Führen Sie für jedes Nonterminal der Grammatik eine Hilfsprozedur ein.

Aufgabe 12.6 (Lexer und Parser für Typen mit Pfeil und Stern) Sie sollen einen Lexer und einen Parser für Typen schreiben, die mit `int`, `->`, `*` und Klammern gebildet werden können. Die konkrete Syntax der Typen soll der von Standard ML entsprechen. Dabei steht `->` auf einer höheren Rangstufe als `*` und wird rechtsassoziativ gruppiert. Beispielsweise soll

```
int * int * int -> int * int
```

denselben Typ darstellen wie

```
(int * int * int) -> (int * int)
```

Die mit `*` dargestellten Produkte können zwei-, drei- oder höherstellig sein. Beispielsweise sollen

```
int * int * int
(int * int) * int
int * (int * int)
```

drei verschiedene Typen darstellen.

1. Schreiben Sie einen Lexer wie folgt:

   ```
   lex : string -> token list  (* Error *)

   datatype token = INT | ARROW | STAR | LPAR | RPAR
   ```

2. Geben Sie eine eindeutige kontextfreie Grammatik für die Typen an. Halten Sie sich dabei an die obigen Vorgaben. Behandeln sie `*` als rechtsassoziativen Infixoperator. Zeichen Sie den Dependenzgraphen der Grammatik.

3. Schreiben Sie einen Parser wie folgt:

```
parse : token list -> ty  (* Error *)

datatype ty = Int
            | Arrow of ty * ty
            | Star of ty list
```

Die mit * dargestellten Produkte sollen so wie in den folgenden Beispielen interpretiert werden:

```
int * int * int       ⟹    Star[Int, Int, Int]
(int * int) * int     ⟹    Star[Star[Int, Int], Int]
int * (int * int)     ⟹    Star[Int, Star[Int, Int]]
```

Geben Sie zuerst eine um ein Hilfsnonterminal erweiterte Grammatik an, aus der die Parsingprozeduren abgeleitet werden können. Die Parsingprozedur für das Hilfsnonterminal soll den Typ

```
token list -> ty list * token list
```

haben. Zeichen Sie den Dependenzgraphen der erweiterten Grammatik.

Aufgabe 12.7 (Kontextfreie Syntax von applikativen Ausdrücken) Sei die folgende abstrakte Syntax für applikative Ausdrücke gegeben:

```
datatype exp = Id  of string          (* Identifier  *)
             | App of exp * exp       (* Application *)
```

1. Geben Sie eine eindeutige kontextfreie Grammatik für diese Ausdrücke an, die Applikationen linksassoziativ gruppiert und bei der Teilausdrücke nach Belieben geklammert werden können. Nehmen Sie dabei an, dass das Nonterminal *identifier* für Bezeichner bereits definiert ist.

2. Schreiben Sie eine Prozedur

```
exp : exp -> string
```

die Ausdrücke gemäß der obigen Grammatik mit möglichst wenigen Klammern darstellt.

3. Geben Sie eine modifizierte Form der obigen Grammatik an, die sich für die Ableitung der Parsingprozeduren eignet.

4. Schreiben Sie eine Prozedur

```
test : token list -> bool
```

die testet, ob eine Liste von Wörtern einen Ausdruck gemäß der obigen Grammatik darstellt. Dabei sollen Wörter wie folgt dargestellt sein:

```
datatype token = ID of string       (* identifier *)
               | LPAR               (* "("        *)
               | RPAR               (* ")"        *)
```

Aufgabe 12.8 (Lexer und Parser für Ausdrücke mit Cons und Append) Sie sollen einen Lexer und einen Parser für Ausdrücke schreiben, die mit Bezeichnern, Applikation, Klammern und den Listenoperationen Cons (`::`) und Append (`@`) gebildet werden können. Die konkrete Syntax der Ausdrücke soll der von Standard ML entsprechen. Dabei stehen die Infixoperatoren `::` und `@` auf derselben Rangstufe und werden rechtsassoziativ gruppiert. Applikation steht auf der tiefsten Rangstufe und wird linksassoziativ gruppiert. Beispielsweise soll

```
x :: y x x :: z @ u
```

denselben Ausdruck darstellen wie

```
x :: (((y x) x) :: (z @ u))
```

Die lexikalische Syntax der Ausdrücke soll aus Bezeichnern und den Schlüsselwörtern "`::`", "`@`", "`(`" und "`)`" bestehen. Ein Bezeichner soll eine nichtleere Zeichenreihe sein, die aus Buchstaben und Ziffern besteht und mit einem Buchstaben anfängt.

1. Schreiben Sie einen Lexer wie folgt:

   ```
   lex : string -> token list  (* Error *)

   datatype token = CONS | APPEND | LPAR | RPAR
                  | ID of string
   ```

2. Geben Sie eine eindeutige kontextfreie Grammatik für die Ausdrücke an. Halten Sie sich dabei an die obigen Vorgaben. Zeichen Sie den Dependenzgraphen der Grammatik.

3. Geben Sie eine Grammatik an, aus der sich die Parsingprozeduren ableiten lassen. Zeichen Sie den Dependenzgraphen der Grammatik.

4. Schreiben Sie einen Parser wie folgt:

   ```
   parse : token list -> exp  (* Error *)

   datatype ops = Cons | Append

   datatype exp = Id of string
                | Op of exp * ops * exp
                | App of exp * exp
   ```

Aufgabe 12.9 (Interpreter für FR) Schreiben Sie einen Interpreter für FR (F mit rekursiven Abstraktionen). Die dafür notwendigen Programme haben Sie im Wesentlichen in diesem und im vorhergehenden Kapitel kennengelernt.

Realisieren Sie den Interpreter mit einem direkt ausführbaren Programm, das einen Dateinamen als Startargument erwartet. In der Datei soll ein Ausdruck stehen. Der Interpreter soll den Ausdruck ausführen und sein Ergebnis und seinen Typ ausgeben. Falls die Eingabe nicht als ein zulässiger Ausdruck interpretiert werden kann, soll ein Fehler gemeldet werden. Der Interpreter soll mitteilen, um welche Art von Fehler es sich handelt: lexikalischer Fehler, syntaktischer Fehler (die kontextfreie Syntax betreffend), Bindungsfehler oder Typfehler. Außerdem sollen Fehler, die bei der Ausführung auftreten, gemeldet werden (Overflow, Out of Memory).

Der Interpreter soll mit fünf Strukturen realisiert werden:

`Lex`	Lexikalische Syntax
`Parse`	Kontextfreie Syntax
`Env`	Umgebungen
`Elab`	Statische Semantik
`Eval`	Dynamische Semantik

Jede Struktur soll mit einem Signaturconstraint deklariert werden, der nur die Prozeduren sichtbar macht, die außerhalb erforderlich sind.