Kapitel 12

Syntax

In diesem Kapitel geht es um die konkrete Syntax von Programmiersprachen. Die konkrete Syntax regelt, wie die durch die abstrakte Syntax gegebenen Phrasen durch Zeichenfolgen dargestellt werden. Man unterscheidet zwischen der Zeichen-, Wort- und Baumdarstellung von Phrasen (siehe Kapitel 1) (siehe Kapitel 1). Die Baumdarstellung von Phrasen ist durch die abstrakte Syntax gegeben. Die Wortdarstellung vermittelt zwischen Zeichen- und Baumdarstellung. Entsprechend besteht die konkrete Sytax aus zwei Teilen, die als *lexikalische* und als *phrasale Syntax* bezeichnet werden. Die lexikalische Syntax regelt, welche Wörter für die Darstellung von Phrasen zur Verfügung stehen, und wie Wörter durch Zeichenreihen darzustellen sind. Die phrasale Syntax vermittelt zwischen der lexikalischen und abstrakten Syntax und regelt, wie Phrasen durch Wortfolgen dargestellt werden.

Eine konkrete Syntax muss so definiert sein, dass jede Zeichenreihe höchstens eine Phrase beschreibt (Eindeutigkeit der konkreten Syntax). Außerdem muss eine konkrete Syntax durch einen so genannten *Parser* realisierbar sein, der zu einer vorgelegten Zeichenreihe entscheidet, ob es sich um die Darstellung einer Phrase handelt, und gegebenfalls die Baumdarstellung der Phrase liefert.

Die Entwicklung eines Parsers ist eine anspruchsvolle Programmieraufgabe, bei der eine Vielzahl von Details berücksichtigt werden müssen. Als Grundlage benötigt man eine hinreichend formale Beschreibung der zu realisierenden konkreten Syntax. Besonders komplex ist die phrasale Syntax, die man mit so genannten *kontextfreier Grammatiken* beschreibt.

Als Beispiel betrachten wir die im letzten Kapitel eingeführte Minisprache F, deren konkrete Syntax wir an die von Standard ML anlehnen. Wir werden zeigen, wie man einen Parser für F schreibt. Dabei verwenden wir die Methode des *rekursiven Abstiegs*. Mit dem Parser und den semantischen Prozeduren aus dem letzten Kapitel sind wir dann in der Lage, einen vollständigen Interpreter für F zu realisieren.

12.1 Lexikalische Syntax

Die lexikalische Syntax regelt, welche Wörter für die Darstellung von Phrasen zur Verfügung stehen, und wie Wörter durch Zeichenreihen darzustellen sind. Als Beispiel betrachten wir die lexikalische Syntax von F. Die Wörter von F können wir in 4 Klassen einteilen:

- **Operatoren**. Dabei handelt es sich um die folgenden Wörter:

  `"+"` `"-"` `"*"` `"<="`

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

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

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

- **Konstanten**. Neben den **Booleschen Konstanten** `"false"` und `"true"` gibt es in F **Zahlkonstanten**. Eine **einfache 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"`).

- **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.

Diese Klassifizierung der Wörter von F ist semantisch orientiert. Aus Sicht der lexikalischen Syntax ist die folgende Klassifizierung sinnvoller:

- **Symbolische Wörter**. Dabei handelt es sich um die folgenden Wörter:

  `"+"` `"-"` `"*"` `"<="` `"("` `")"` `":"` `"->"` `"=>"`

- **Zahlkonstanten**. Wie oben.

- **Alpha-numerische Wörter**. Alle Wörter, die nur aus Buchstaben und Ziffern gebildet sind, und die mit einem Buchstaben beginnen.

Neben den Wörtern benötigt man so genannte **Leerzeichen**, die zwischen die Wörter geschrieben werden können. Für F verwenden wir die Leerzeichen Zwischenraum `" "`, Tabulator `"\t"` und Zeilenwechsel `"\n"`. Zwischen zwei Wörter dürfen beliebig viele Leerzeichen geschrieben werden.

Bestimmte Wortkombinationen können in Programmiersprachen ohne trennende Leerzeichen geschrieben werden. Beispielsweise stellt die Zeichenreihe

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

die Wortfolge

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

dar. Die dabei zugrundeliegende **Trennungsregel** besagt, dass zwei Wörter ohne trennendes Leerzeichen direkt hintereinander geschrieben werden dürfen, wenn es kein Wort gibt, das mit den Zeichen des ersten Worts und dem ersten Zeichen des zweiten Worts beginnt. Beispielsweise dürfen die Wörter "`1`" und "`else`" ohne Leerzeichen hintereinander geschrieben werden, da es in F kein Wort gibt, das mit "`1e`" beginnt. Dagegen dürfen die Wörter "`if`" und "`x`" nicht ohne Leerzeichen hintereinander geschrieben werden, da "`ifx`" ein Wort ist (ein Bezeichner).

Damit ist die lexikalische Syntax von F vollständig definiert. Wir gehen jetzt noch auf einige Aspekte der lexikalischen Syntax von Standard ML ein.

Die lexikalische Syntax der meisten Programmiersprachen erlaubt dem Programmierer, die Zeichendarstellung von Programmen mit so genannten **Kommentaren** zu versehen. Kommentare werden wie Leerzeichen behandelt und sind in der Wortdarstellung des Programms nicht mehr sichtbar. In Standard ML beginnt ein Komentar mit "`(*`" und endet mit "`*)`". Beispielsweise liefert die Zechendarstellung

```
"if x then 1 (* der Dann-Fall *) else 0 (*der Sonst-Fall *)"
```

die Wortdarstellung

```
"if", "x", "then", "1", "else", "0"
```

Innerhalb eines Kommentars dürfen alle Zeichen vorkommen, mit der Einschränkung, dass die **Kommentarklammern** "`(*`" und "`*)`" nur balanciert auftreten dürfen.

In Standard ML besteht die Klasse der symbolischen Wörter aus allen nichtleeren Zeichenreihen, die mit den Zeichen

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

gebildet werden können. Symbolische Wörter können in Standard ML als Schlüsselwörter, Operatoren und Bezeichner fungieren. Das erklärt einige bisher völlig mysteriöse Fehlermeldungen. Beispielsweise liefert die Zeichendarstellung

```
"1+~2"
```

in Standard ML die Wortdarstellung

```
"1", "+~", "1"
```

und dem Wort "`+~`" wird die Rolle eines Bezeichners zugewiesen.

Auch die Klasse der alpha-numerischen Wörter ist in Standard ML größer als in F. Insbesondere dürfen alpha-numerischen Wörter die Zeichen "'", "_" und "." enthalten. Der Punkt ist notwendig, um zusammengesetzte Bezeichner wie `S.T.x` bilden zu können.

Schließlich geben wir noch die Schlüsselwörter von Standard ML an, die wie normale Bezeichner aussehen:[1]

```
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
```

12.2 Ein Lexer

Eine lexikalische Syntax vermittelt zwischen Zeichen- und Wortfolgen. Sie muss so definiert sein, dass jede Zeichenfolge höchstens eine Wortfolge darstellt. Eine Zeichenfolge heißt **lexikalisch zulässig**, wenn sie eine Wortfolge darstellt (gemäß einer gegebenen lexikalisch Syntax). Ein **Lexer** ist eine Prozedur, die zu einer vorgelegten Zeichenfolge entscheidet, ob sie lexikalisch zulässig ist, und gegebenenfalls die dargestellte Wortfolge liefert.

Die Realisierung eines Lexers für F ist keine ganz leichte Aufgabe. Daher zeigen wir zunächst, wie man einen **lexikalischen Tester**

```
lex : char list → bool
```

für F schreibt, der testet, ob eine Zeichenfolge lexikalisch zulässig ist. Danach erweitern wir den Tester so, dass er die dargestellten Wörter liefert.

Abbildung 12.1 zeigt einen lexikalischen Tester für F. Er liest die vorgelegte Zeichenfolge von links nach rechts. Die leere Zeichenfolge ist zulässig. Leerzeichen werden überlesen. Wenn das erste Zeichen kein Leerzeichen ist, muss ein Wort überlesen werden. Zunächst wird geprüft, ob ein symbolisches Wort überlesen werden kann. Das ist einfach, da es nur endlich viele symbolische Wörter gibt. Es muss nur darauf geachtet werden, dass ein möglichst langes symbolisches Wort überlesen wird (durch entsprechende Anordnung der Regeln, kritisch sind "->" und "-"). Wenn kein symbolisches Wort überlesen werden kann, verbleiben für eine zulässige Zeichenfolge nur 3 Möglichkeiten:

1. Die Zeichenfolge beginnt mit "~". In diesem Fall muss eine möglichst lange negative Zahlkonstante überlesen werden.

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

```
fun lex nil                  = true
  | lex (#" " ::cr)          = lex cr                (* Leerzeichen *)
  | lex (#"\n"::cr)          = lex cr
  | lex (#"\t"::cr)          = lex cr
  | lex (#"<" :: #"=" ::cr) = lex cr         (* symbolische Wörter *)
  | lex (#"-" :: #">" ::cr) = lex cr
  | lex (#"=" :: #">" ::cr) = lex cr
  | lex (#"+" ::cr)          = lex cr
  | lex (#"-" ::cr)          = lex cr
  | lex (#"*" ::cr)          = lex cr
  | lex (#":" ::cr)          = lex cr
  | lex (#"(" ::cr)          = lex cr
  | lex (#")" ::cr)          = lex cr
  | lex (#"~" ::c::cr) =                (* negative Zahlkonstanten *)
      if Char.isDigit c then lexN cr else false
  | lex (c::cr) =
      if Char.isDigit c then lexN cr    (* einfache Zahlkonstanten *)
      else if Char.isAlpha c then lexA cr   (* alpha-numer. Wörter *)
      else false

and lexN cs = if not(null cs) andalso Char.isDigit(hd cs)
              then lexN(tl cs) else lex cs

and lexA cs = if not(null cs) andalso Char.isAlphaNum(hd cs)
              then lexA(tl cs) else lex cs
```

Abbildung 12.1: Lexikalischer Tester für F

2. Die Zeichenfolge beginnt mit einer Ziffer. In diesem Fall muss eine möglichst lange einfache Zahlkonstante überlesen werden.

3. Die Zeichenfolge beginnt mit einem Buchstaben. In diesem Fall muss ein möglichst langes alpha-numerisches Wort überlesen werden.

Die Hilfsprozedur `lexN` überliest zunächst möglichst viele Ziffern und liest dann mit `lex` weiter. Die Hilfsprozedur `lexA` überliest zunächst möglichst viele Buchstaben und Ziffern und liest dann mit `lex` weiter. Die Prozeduren `lex`, `lexN` und `lexA` sind verschränkt rekursiv definiert. Dabei handelt es sich jeweils um iterative Rekursion, die auch bei großen Rekursionstiefen effizient ist.

Die Hilfsprozeduren `lexN` und `lexA` sind für die Erweiterung des Testers zu einem Lexer erforderlich. Für den Tester selbst sind sie unnötigt, da `lex` sich an den entsprechenden Stellen auch einfach selbst aufrufen könnte.

Unser Lexer wird die gelesenen Wörter nicht als Zeichenreihen, sondern als Werte des in Abbildung 12.2 deklarierten Konstruktortyps `token` liefern. Diese Darstellung unterscheidet zwischen Zahlkonstanten, Bezeichnern und den verblei-

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

Abbildung 12.2: Darstellung der Wörter von F

benden Wörtern und versieht Zahlkonstanten mit der dargestellten Zahl (statt den darstellenden Zeichen).

Als Beispiel betrachten wir die Zeichenfolge

```
"if~17<=xy then bool else 1+ "
```

Für diese soll unser Lexer die Wortfolge

```
[IF, ICON ~17, LEQ, ID "xy", THEN, ID "bool", ELSE, ICON 1, ADD]
```

liefern.

Die Wörter "`bool`" und "`int`" können in F und Standard ML sowohl als Typkonstanten und Bezeichner fungieren. Die Entscheidung, welche Rolle ein Auftreten dieser Wörter in der Darstellung einer Phrase spielt, kann erst auf der Ebene der phrasalen Syntax getroffen werden. Daher wird unser Lexer "`bool`" und "`int`" immer als Bezeichner klassifizieren.

Wir werden den Lexer für F mit einer Prozedur

```
lex : token list → char list → token list  (* Error, Overflow *)
```

realisieren. Das erste Argument ist ein Akkumulatorargument, das die Liste der bereits gelesenen Wörter übergibt (in reversierter Ordnung). Das zweite Argument übergibt die noch zu lesende Zeichenfolge. Wenn diese lexikalisch zulässig ist, wird die Liste der insgesamt gelesenen Wörter geliefert. Ansonsten bekommt man die Ausnahme `Error` oder `Overflow`. Die Ausnahme `Overflow` signalisiert,

dass eine Zahlkonstante gelesen wurde, die eine Zahl darstellt, die zu groß ist, um als Wert von `int` darstellbar zu sein.

Abbildung 12.3 zeigt die Realisierung des Lexers. Wie beim Tester sind die Prozeduren `lex`, `lexN` und `lexA` verschränkt endrekursiv. Deswegen benötigen `lexN` und `lexA` mehrere Akkumulatorargumente. Die Prozedur

```
lexA : token list → char list → char list → token list
```

liest alpha-numerische Wörter. Das erste Argument übergibt die Liste der zuvor gelesenen Wörter. Das zweite Argument übergibt die bereits gelesenen Zeichen des alpha-numerischen Wortes, das gerade gelesen wird. Das dritte Argument übergibt die noch zu lesende Zeichenfolge. Die Prozedur

```
lexN : token list → int → int → char list → token list
```

liest Zahlkonstanten. Das erste Argument übergibt die Liste der zuvor gelesenen Wörter. Das zweite Argument übergibt das bereits gelesene Vorzeichen (-1 oder 1). Das dritte Argument übergibt die Zahl, die durch die bereits gelesenen Ziffern dargestellt wird. Das vierte Argument übergibt die noch zu lesende Zeichenfolge.

Abbildung 12.4 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. Bis auf die Aufrufe von `lexA` sind alle Aufrufe iterativ.

```
val ord0 = ord #"0"

fun lex ts nil                 = rev ts
  | lex ts (#" " ::cr)         = lex ts cr
  | lex ts (#"\n"::cr)         = lex ts cr
  | lex ts (#"\t"::cr)         = lex ts cr
  | lex ts (#"<" :: #"=" ::cr) = lex (LEQ::ts) cr
  | lex ts (#"-" :: #">" ::cr) = lex (ARROW::ts) cr
  | lex ts (#"=" :: #">" ::cr) = lex (DARROW::ts) cr
  | lex ts (#"+" ::cr)         = lex (ADD::ts) cr
  | lex ts (#"-" ::cr)         = lex (SUB::ts) cr
  | lex ts (#"*" ::cr)         = lex (MUL::ts) cr
  | lex ts (#":" ::cr)         = lex (COLON::ts) cr
  | lex ts (#"(" ::cr)         = lex (LPAR::ts) cr
  | lex ts (#")" ::cr)         = lex (RPAR::ts) cr
  | lex ts (#"~" ::c::cr) =
      if Char.isDigit c then lexN ts ~1 0 (c::cr) else raise Error
  | lex ts (c::cr) =
      if Char.isDigit c then lexN ts 1 0 (c::cr)
      else if Char.isAlpha c then lexA ts [c] cr
      else raise Error

and lexN ts s n cs =
    if not(null cs) andalso Char.isDigit(hd cs)
    then lexN 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

and lexA' "if"    = IF
  | lexA' "then"  = THEN
  | lexA' "else"  = ELSE
  | lexA' "fn"    = FN
  | lexA' "false" = FALSE
  | lexA' "true"  = TRUE
  | lexA' x       = ID x
```

Abbildung 12.3: Lexer für F

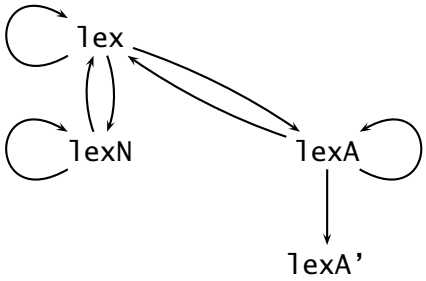

Abbildung 12.4: Dependenzgraph des Lexers