Fachrichtung 6.2 — Informatik
Universität des Saarlandes
Tutorenteam der Vorlesung Programmierung 1

Programmierung 1 (Wintersemester 2012/13)

Lösungsblatt 12

(Kapitel 13)

Hinweis: Dieses Übungsblatt enthält von den Tutoren für die Übungsgruppe erstellte Aufgaben.

Die Aufgaben und die damit abgedeckten Themenbereiche sind für die Klausur weder relevant, noch irrelevant.

1 Überschrift

Aufgabe 12.1 (*Viele schöne Begriffe*)
Finden Sie kurze und prägnante Beschreibungen für die folgenden Begriffe:

(a) Parser

(b) lexikalische Syntax

(c) syntaktische Kategorie

(d) maximum munch rule

(e) Lexer

(f) Eindeutigkeit

(g) phrasale Syntax

(h) abstrakte Syntax

(i) konkrete Grammatik

(j) Ableitung

(k) Prüfer

(l) RA-tauglich

(m) konkrete Syntax

(n) abstrakte Grammatik

(o) Metavariable

(p) Syntaxbaum

(q) Affinität

Finden Sie Parallelen und stellen Sie Beziehungen zwischen den Begriffen auf.

Aufgabe 12.2
Überzeugen Sie sich, dass Sie mit Hilfe von Hilfskategorien tatsächlich linksklammerende Ausdrücke erhalten.
Sei dazu die folgende Grammatik und der folgende dazugehörige Parser gegeben:

plusexp	::=	[*plusexp* "+"] *unminusexp*
unminusexp	::=	["-"] *pexp*
pexp	::	"1" | "("*plusexp*")"

Der Parser benutzt die folgenden beiden Datentypen:

```
datatype token = OpPlus | OpMinus | RPAR | LPAR | EINS
datatype exp = Plus of exp * exp | Minus of exp | ConEins
```

```
fun plusexp ts = plusexp' (unminusexp ts)

and plusexp' (a, OpPlus::tr) = plusexp' (extend (a, tr) unminusexp Plus)
| plusexp' s = s

and unminusexp (OpMinus::tr) = let val (a, tr') = pexp tr in (Minus a, tr') end
| unminusexp ts = pexp ts

and pexp (EINS::tr) = (ConEins, tr)
| pexp (LPAR::tr) = match (plusexp tr) RPAR
| pexp s = raise Error ''pexp''
```

(a) Die obige Grammatik ist nicht RA-tauglich. Wandeln Sie diese in eine RA-taugliche Grammatik um, die die gleichen Ausdrücke akzeptiert!

(b) Parsen Sie nun gedanklich folgende Liste, indem Sie jeden Aufruf notieren bis Sie auf das Ergebnis kommen: $[EINS, OpPlus, EINS, OpPlus, EINS]$. Ihr Anfang sollte also so aussehen:

$plusexp'(unminusexp[EINS, OpPlus, EINS, OpPlus, EINS])$
$= plusexp'(pexp[EINS, OpPlus, EINS, OpPlus, EINS])$
$= \ldots$

(c) Betrachten Sie erneut die Grammatik und den darauf beruhenden Parser. Was fällt Ihnen auf? Basiert der Parser wirklich exakt auf der obigen Grammatik? Erklären Sie!

Lösung 12.2:
Als Hilfe:
Umgedreht sieht das Ganze so aus:

plusexp	::=	*unminusexp plusexp'*
plusexp'	::=	["+" *uminusexp plusexp'*]
unminusexp	::=	["-"] *pexp*
pexp	::=	"1" | "("*plusexp*")"

Aufgabe 12.3
Stellen Sie eine Grammatik für die wie folgt definierten Ausdrücke auf:

- Es gibt die Operatoren A und B.
- A ist ein unärer Operator. Ist φ ein gültiger Ausdruck, so ist es auch $A\varphi$.
- B ist ein binärer Operator. Sind φ und ψ gültige Ausdrücke, so ist es auch $\varphi B\psi$.
- A klammert stärker als B.
- B klammert implizit rechts.
- 1 ist immer ein gültiger Ausdruck.
- Ausdrücke dürfen Klammern enthalten, d.h. ist φ ein gültiger Ausdruck, so ist es auch (φ)

Schreiben Sie dann einen Parser für Ihre Grammatik mit dem folgenden Datentypen *exp* und den angegebenen Tokens:

```
datatype token = OpA | OpB | Eins
datatype exp = A of exp | B of exp * exp | ConEins
```

Lösung 12.3:
Wie geht man nun vor? Der schwächste Operator steht in der syntaktischen Kategorie am weitesten oben, d.h .in der ersten Kategorie, die man definiert. In unserem Fall ist dies das B.

bexp ::= *aexp* [“B” *bexp*]

Es ist angegeben, dass B implizit rechts klammert. Intuitiv bedeutet das: rechts kann wieder ein Ausdruck, der mit *bexp* (=einer Kategorie, die gleich stark klammert) gebildet wurde, vorkommen, ohne dass wir ihn direkt klammern müssten. Deshalb stehen die Optionalklammern rechts. Links dagegen muss ein stärker geklammerter (=weiter unten vorkommender) Ausdruck stehen.

Unäre Operatoren sind sehr einfach:

bexp ::= *aexp* [“B” *bexp*]
aexp ::= [“A”] *pexp*

Entweder sie kommen vor einem beliebigen Ausdruck oder eben nicht. Deshalb enthalten die Optionalklammern hier nur den Operator selbst. Würde der Operator nicht vor, sondern nach einem Ausdruck stehen, so würde die Optionalklammer rechts stehen. Da aber keine syntaktischen Kategorien in den Optionalklammern vorkommen, man also am ersten Wort eines Satzes entscheiden kann, ob es zur Kategorie gehört oder weiter mit *pexp* verfahren werden soll, spielt dies für die RA-Tauglichkeit keine Rolle.

Die Kategorie *pexp* enthält Konstanten und Bezeichner, so wie eine geklammerte Darstellung der “obersten” Kategorie. Diese sorgt dafür, dass ein schwacher Ausdruck (bei uns *bexp*) in der am stärksten klammerten Kategorie stehen kann.
bexp ::= *aexp* [“B” *bexp*]
aexp ::= [“A”] *pexp*
pexp :: “1” | “(” *bexp* “)”
Einen Parser zu einer Grammatik schreiben ist sehr mechanisch. Pro Kategorie gibt es eine Prozedur und alle Prozeduren werden mit *and* verbunden. D.h. das Grundgerüst zu der Grammatik oben sieht so aus:

```
fun bexp ts = ...

and aexp ts = ...

and pexp ts = ...
```

Jetzt muss man die richtigen Fallunterscheidungen machen: Im Fall von bexp heißt das: Beginnt das, was man von aexp zurückerhält mit einem B, so muss man die restliche Liste nochmals mit *bexp* Parsen und gibt dies zurück, beginnt die Liste dahingegen mit etwas anderen, so ist man fertig. Dies lässt sich am einfachsten durch ein *case* ausdrücken:

```
fun bexp ts = case aexp ts of
    (a1, OpB::tr) => let val (a2,tr') = bexp tr in (B(a1,a2),tr') end
    | s => s

and aexp ts = ...

and pexp ts = ...
```

In aexp muss man unterscheiden, ob die Liste nun mit dem Operator A beginnt oder eben nicht. Beginnt sie damit, so wird vor den restlichen Ausdruck ein A gesetzt - ansonsten nicht.

```
fun bexp ts = case aexp ts of
    (a1, OpB::tr) => let val (a2,tr') = bexp tr in (B(a1,a2),tr') end
    | s => s

and aexp (OpA::tr) = let val (a, tr') = pexp tr in (A a, tr') end
  | aexp ts = pexp ts

and pexp ts = ...
```

Die Kategorie *pexp* ist in fast allen Parsern sehr ähnlich: Sie enthält, falls es Klammern in den Ausdrücken gibt und die Klammern in *pexp* vorkommen, einen Aufruf mit *match* und das Parsen der Konstanten, in unserem Fall also nur die 1. Gleichzeitig wirft Sie einen Fehler, wenn etwas nicht so geparst werden kann. Intuitiv werden alle Fehler nach unten durchgegeben, da in allen Fällen, die in den anderen Prozeduren nicht behandelt werden können, einfach die nächst-tieferliegende aufgerufen wird.

```
fun bexp ts = case aexp ts of
    (a1, OpB::tr) => let val (a2,tr') = bexp tr in (B(a1,a2),tr') end
    | s => s

and aexp (OpA::tr) = let val (a, tr') = pexp tr in (A a, tr') end
  | aexp ts = pexp ts

and pexp (Eins::tr) = (ConEins, tr)
  | pexp (LPAR::tr) = match (bexp tr) RPAR
  | pexp s = raise Error ''pexp''
```

Aufgabe 12.4
Bearbeiten Sie Aufgabe 12.3 erneut, dabei soll jedoch der Operator B implizit links klammern.
Hinweis: Bevor Sie einen Parser schreiben können, müssen Sie Ihre Grammatik in eine RA-taugliche umwandeln.

Lösung 12.4:
Die Kategorien *aexp* und *pexp* sehen aus wie zuvor, da wir nichts am Operator *A* oder an der Klammerung und den Konstanten geändert haben. Der einizige Unterschied findet sich in der syntaktischen Kategorie *bexp*: Die Optionalklammern stehen nicht mehr rechts, sondern links. Damit ist die Grammatik auch nicht mehr RA-tauglich.

bexp	::=	[*bexp* "B"] *aexp*
aexp	::=	["A"] *pexp*
pexp	::	"1" | "(" *bexp* ")"

Wie wandelt man nun eine linksrekursive Grammatik in eine rechtsrekursive um, die die gleichen Ausdrücke erkennt? Den ersten Schritt kann man informell mit "Spiegeln" beschreiben. Man spiegelt die Kategorien mit linksrekursiven Optioinalklammern, die die RA-Tauglichkeit beeinflussen - die Optionalklammern links bei aexp stören dabei nicht - am Operator. Aus der bisherigen Kategorie *bexp* wird also *bexp* ::= *aexp*["B" *bexp*]. Man kann sich auch vorstellen, dass der Operator einfach implizit rechts klammern würde.

Nun haben wir die Ausdrücke in ihrer Struktur verändert - immerhin klammern sie jetzt implizit rechts statt links, aus $1B1B1$ wird statt $(1B1)B1$ jetzt $1B(1B1)$. Dies müssen wir noch ändern. Dazu dienen die Hilfskategorien. Intuitiv verschieben wir einfach nur den Zeitpunkt, zu dem der zweite Teil des Ausdrucks gelesen wird. Ihr Aufstellen ist eigentlich sehr mechanisch:

- Die Optionalklammern werden in eine Hilfskategorie verschoben, also $bexp' ::= [$"B" *bexp*$]$
- in der Hauptkategorie wird die komplette Optionalklammer durch den Namen der Hilfskategorie ersetzt, also $bexp ::= aexp\, bexp'$
- Ersetzen des Namens der Hauptkategorie in der Hilfskategorie durch deren Definition, also hier das Ersetzen von *bexp* durch $aexp\, bexp'$ in der Kategorie $bexp'$

So erhält man schließlich die folgende Grammatik:

bexp	::=	$aexp\, bexp'$
$bexp'$	::=	["B" $aexp\, bexp'$]
aexp	::=	["A"] *pexp*
pexp	::	"1" | "(" *bexp* ")"

Das Schreiben des Parsers funktioniert analog zur vorherigen Aufgabe. Die Kategorien *aexp* und *pexp* können wir direkt übernehmen. Man beachte das Typschema der Prozeduren, die zu Hilfskategorien gehören: Sie bekommen das Ergebnis der anderen Prozeduren als Argument, d.h. einen Tupel aus dem bisher geparstem Ausdruck und der Restliste.

```
fun bexp ts = ...

and bexp' (a,ts) = ...

and aexp (OpA::tr) = let val (a, tr') = pexp tr in (A a, tr') end
| aexp ts = pexp ts

and pexp (Eins::tr) = (ConEins, tr)
| pexp (LPAR::tr) = match (bexp tr) RPAR
| pexp s = raise Error ''pexp''
```

Da es keine Unterscheidung mehr in *bexp* gibt müssen wir nur die den syntaktischen Kategorien entsprechenden Prozeduren aufrufen und zwar in umgekehrter Reihenfolge als notiert:

```
fun bexp ts = bexp' (aexp ts)

and bexp' (a,ts) = ...

and aexp (OpA::tr) = let val (a, tr') = pexp tr in (A a, tr') end
| aexp ts = pexp ts
```

```
and pexp (Eins::tr) = (ConEins, tr)
| pexp (LPAR::tr) = match (bexp tr) RPAR
| pexp s = raise Error ''pexp''
```

Nun gibt es für $bexp'$ zwei Möglichkeiten: entweder die restliche Liste beginnt mit dem Operator B oder eben nicht. Im zweiten Fall geben wir das Argument als Ergebnis aus. Im ersten Fall ist die Lösung komplizierter: Haben wir eine Hilfskategorie der Form ["X" $Yexp\,Xexp'$], so schreiben wir $Xexp'(extend(< bisher\,geparster\,Ausdruck, Restliste\,ohne\,Operator >)Yexp < zu\,X\,gehoeriger\,Expression >)$. Informell gesagt, schauen wir uns die Restliste mit der Kategorie $Yexp$ an und ergänzen ihn als rechten Teil unseres Operators bevor wir schauen, ob man den restlichen Ausdruck nochmals mit $bexp'$ lesen kann. Ohne den Aufruf mit $bexp'$ klammern wir nicht mehr. Dadurch, dass wir danach nochmals mit $bexp'$ parsen, verschieben wir das zusammensetzen der Ausdrücke so nach hinten (siehe 12.reflinksrekursiv), dass die Klammerung wieder passt.

```
fun bexp ts = bexp' (aexp ts)

and bexp' (a, OpB::tr) = bexp' (extend (a, tr) aexp B)
| bexp s = s

and aexp (OpA::tr) = let val (a, tr') = pexp tr in (A a, tr') end
| aexp ts = pexp ts

and pexp (Eins::tr) = (ConEins, tr)
| pexp (LPAR::tr) = match (bexp tr) RPAR
| pexp s = raise Error ''pexp''
```