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

Programmierung 1 (Wintersemester 2012/13)

Lösungsblatt 14

(Kapitel 13,15,16)

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.*

Aufgabe 14.1 (*Aufgabe aus dem Nachklausurtutorium zum WS 11/12*)
Bei unserem klassischen Ansatz, Schlüsselwörter direkt in *lex* zu lexen, ist es nicht (leicht) möglich, Bezeichner zu erlauben, die so beginnen wie Schlüsselwörter. In SML dagegen sind z.B. *iff* oder *lettie* gültige Bezeichner. Wie würden Sie einen Lexer aufbauen, der "if" als Schlüsselwort *IF*, "iff" aber als Bezeichner *ID "iff"* erkennen soll?
Schreiben Sie einen Lexer, der folgende Tokens aus einer Char list ausliest:

```
datatype token = TRUE | FALSE | ID of string | AND | OR
```

Lösung 14.1:

```
fun testNext [] = true
| testNext (x::xr) = if Char.isAlpha x then false else true

exception Error of string

datatype bexp = True | False | And of bexp * bexp | Or of bexp * bexp

datatype token = TRUE | FALSE | ID of string| AND | OR

fun lex nil = nil
| lex (#" " :: cr) = lex cr
| lex (#" \t" :: cr) = lex cr
| lex (#" \n" :: cr) = lex cr
| lex (#"t" :: #"r" :: #"u" :: #"e" :: cr) = if testNext cr then TRUE :: lex cr else lexId
[#"e",#"u", #"r",#"t"] cr
| lex (#"f" :: #"a" :: #"l" :: #"s" :: #"e" :: cr) = if testNext cr then FALSE :: lex cr
else lexId [#"e",#"s",#"l",#"a",#"f"] cr
| lex (#"a" :: #"n" :: #"d" :: cr) = if testNext cr then AND :: lex cr else lexId [#"d",#"n",#"a"]
cr
| lex (#"o" :: #"r" :: cr) = if testNext cr then OR :: lex cr else lexId [#"r",#"o"] cr
| lex (c::cr) = if Char.isAlpha c then lexId [c] cr else raise Error "lex"

and lexId cs cs' = if null cs' orelse not(Char.isAlpha (hd cs'))
then ID(implode(rev cs)) :: lex cs'
else lexId (hd cs' ::cs) (tl cs')
```

Aufgabe 14.2 (*Aufgabe aus dem Nachklausurtutorium zum WS 11/12*)
Zeigen Sie mit einem Beispiel, dass die Grammatik $exp = "x" \mid exp\ exp$ nicht eindeutig ist. Geben Sie eine eindeutige Grammatik an, die dieselben Wortfolgen darstellt.

Lösung 14.2:

(a) $exp = "x" \mid "x"\ exp$

Aufgabe 14.3 (*Aufgabe aus dem Nachklausurtutorium zum WS 11/12*)
Zeigen Sie mit einem Beispiel, dass die Grammatik $exp = "true"\ exp \mid exp "true" \mid "true"$ nicht eindeutig ist. Geben Sie eine eindeutige Grammatik an, die dieselben Wortfolgen darstellt.

Aufgabe 14.4 (*Aufgabe aus dem Nachklausurtutorium zum WS 11/12*)
Wir betrachten additive Ausdrücke. Schreiben Sie eine eindeutige Grammatik, die additive Ausdrücke

- vollständig klammert $((1 + 2) + 3)$
- minimal klammert $1 + 2 + 3$
- in Präfixschreibweise realisiert $+ + 1\,2\,3$

Ausdrücke, die nur aus einer Konstante bestehen sind auch gültig.

Lösung 14.4:
Vollständig:

$$add = num \mid "("\ add\ "+"\ add\ ")"$$

Minimal:

$$\begin{aligned} exp &= add \mid num \\ add &= add\ "+"\ pexp \\ pexp &= num \mid "("\ add\ ")" \end{aligned}$$

Präfix:

$$e = "+"\ e\ e \mid num$$

Aufgabe 14.5 (*Aufgabe aus dem Nachklausurtutorium zum WS 11/12*)
Gegeben sei folgende Grammatik:

$$exp = pexp\text{"@"}\ exp \,|\, pexp$$
$$pexp = Idofstring \,|\, \text{"("}\ exp\ \text{")"}$$

Zeichnen Sie Ableitungsbäume gemäß dieser Grammatik für die Ausdrücke:

- *Id "x" @ Id "g"*
- *(Id "b" @ Id "f")@ Id "k"*
- *Id "b" @ Id "f"@ Id "k"*

Aufgabe 14.6 (*Aufgabe aus dem Nachklausurtutorium zum WS 11/12*)
Wir betrachten Ausdrücke mit Fakultäten. Alle Ausdrücke beginnen mit dem Fakultätszeichen. Die Fakultät klammert am stärksten, Ausdrücke wie *!!5* sind also ungültig, *!(!5)* hingegen ist gültig. Gegeben ist folgender Datentyp :

```
datatype exp = F of exp | Icon of int
```

a) Schreiben Sie die phrasale Syntax für die oben beschriebene Grammatik

b) Schreiben Sie einen Parser für ihre Syntax

Lösung 14.6:

$$exp := \text{"!"}\ pexp$$
$$pexp := num \,|\, \text{"("}\ exp\ \text{")"}$$

```
fun exp (FAK::tr) = let val (e,ts) = pexp tr in (F(e), ts) end
| exp s = raise Error "Match"

and pexp ((ICON i)::tr) = (Icon i, tr)
| pexp (LPAR::tr) = (case exp tr of (e,RPAR::ts) => (e,ts) | s => raise Error "Match")
| pexp s = raise Error "pexp"
```

Aufgabe 14.7 (*Aufgabe aus dem Nachklausurtutorium zum WS 11/12*)
Wir betrachten Ausdr¨ucke, die mit Bezeichnern und den Operatoren *::* und @ gebildet werden. Die phrasale Syntax sei durch die Grammatik

$$exp \ ::= \ pexp\ [\,(\text{"::"} \ \mid \ \text{"@"})\ exp\,]$$
$$pexp \ ::= \ id \ \mid \ \text{"("}\ exp\ \text{")"}$$

gegeben. Die Operatoren *::* und @ klammern also so wie in SML gleichberechtigt rechts:
$x :: y \,@\, z :: u \,@\, v \ \rightsquigarrow \ x :: (y \,@\, (z :: (u \,@\, v)))$. W¨orter und Baumdarstellungen seien wie folgt dargestellt:

```
datatype token = ID of string | CONS | APPEND | LPAR | RPAR
datatype exp = Id of string | Cons of exp * exp | Append of exp * exp
```

(a) Schreiben Sie einen Lexer *lex* : *char list* → *token list*.

(b) Schreiben Sie einen Parser f¨ur *exp*.

(c) Schreiben Sie eine Prozedur *exp* : *exp* → *string*, die Ausdr¨ucke gem¨aß der obigen Grammatik mit minimaler Klammerung darstellt.

Lösung 14.7:

(a)
```
fun lex nil = nil | lex (#" ":: cr) = lex cr | lex (#"\t":: cr) = lex cr | lex (#"\n":: cr) = lex cr | lex (#":":: #":":: cr) = CONS:: lex cr | lex (#"@":: cr) = APPEND:: lex cr | lex (#"(":: cr) = LPAR:: lex cr | lex (#")":: cr) = RPAR:: lex cr | lex (c::cr) = if Char.isAlpha c then lexId [c] cr else raise Error "lex" and lexId cs cs' = if null cs' orelse not(Char.isAlpha (hd cs')) then ID(implode(rev cs)) :: lex cs' else lexId (hd cs' ::cs) (tl cs')
```

(b)
```
fun match (a,ts) t = if null ts orelse hd ts <> t then raise Error "match" else (a, tl ts)
fun exp ts = exp' (pexp ts)
and exp' (e,CONS::tr) = extend (e,tr) exp Cons | exp' (e,APPEND::tr) = extend (e,tr) exp Append | exp' s = s
and pexp (ID x ::tr) = (Id x,tr) | pexp (LPAR ::tr) = match (exp tr) RPAR | pexp s = raise Error "pexp"
```

(c)
```
fun exp (Cons(e,e')) = pexp e ^``::" êxp e' | exp (Append(e,e')) = pexp e ^``@" êxp e' | exp e = pexp e
and pexp (Id x) = x | pexp e = "(" êxp e ^``)"
```

Aufgabe 14.8 (*Aufgabe aus dem Nachklausurtutorium zum WS 11/12*)
Wir betrachten Ausd¨ucke aus Zahlen sowie den Operatoren Υ (sprich I) und Δ (sprich Delta). Υ klammert stärker als Δ, beide Operatoren klammern rechts. Außerdem sind folgende Datentypen gegeben:

```
datatype token = ICON of int | YPSILON | DELTA | LPAR | RPAR
datatype exp = Con of int | I of exp * exp | D of exp * exp
```

a) Geben Sie eine eindeutige Grammatik an, die eine phrasale Syntax für diese Ausdrücke beschreibt.

b) Schreiben Sie einen Parser für diese Grammatik.

Lösung 14.8:

a) Wichtig! Zuerst macht man sich die gegebenen Informationen klar. Δ klammert schwächer, also muss die Regel hierzu weiter oben stehen. Die Operatoren klammern beide rechts, also ist die Grammatik direkt RA-tauglich.

$$\begin{aligned} exp &::= dexp\,[\Delta\, exp] \\ dexp &::= pexp\,[\Upsilon\, dexp] \\ pexp &::= num \mid \text{"("}\, exp \,\text{")"} \end{aligned}$$

b)
```
fun exp ts = (case pexp ts

of(e, DELTA::tr)=>
let
val(e', t) = exp tr
in
(C(e,e'), t)
end
| s => s)


and pexp ts = (case mexp ts of (e, YPSILON::tr) =>
let val (e', t) = pexp tr in (R(e,e')t) end
| s => s)


and mexp ((ICON z)::tr) = (Con z ,tr)
| mexp(LPAR::tr)= (case exp tr of (e, RPAR::tr) => (e,tr)
| s => raise Error "RPAR")
| mexp s = raise Error "parse"
```

Aufgabe 14.9 (*Aufgabe aus dem Nachklausurtutorium zum WS 11/12*)
Wir betrachten in dieser Aufgabe nun boolesche Ausdrücke, welche aus zwei Konstanten *true*, *false*, sowie zwei booleschen Operationen $and : bool \times bool \to bool$ und $or : bool \times bool \to bool$ bestehen. Ziel dieser Aufgabe ist es, einen Parser und Lexer für die konkrete Syntax von booleschen Ausdrücken anzugeben. Wir lassen Ihnen hierbei jegliche Freiheit bezüglich der Wahl der Zeichen, die Sie verwenden. Wir stellen lediglich 2 Anforderungen:

(a) *and* klammert stärker als *or* : $true \wedge false \vee false \rightsquigarrow (true \wedge false) \vee false$

(b) *and* und *or* klammern jeweils links: $true \wedge true \wedge false \rightsquigarrow (true \wedge true) \wedge false$ und $true \vee true \vee false \rightsquigarrow (true \vee true) \vee false$

Bearbeiten Sie nun folgende Aufgaben:

(a) Geben Sie ihre abstrakte, sowie phrasale Syntax für boolesche Ausdrücke an.

(b) Schreiben Sie einen Lexer für boolesche Ausdrücke.

(c) Schreiben Sie einen Parser für boolesche Ausdrücke.

Hinweis: Beachten Sie, dass sowohl *true* als auch *false* alleine gültige boolesche Ausdrücke sind.

Lösung 14.9:

[a)]**Abstrakte Syntax**
$p \in \{true, false\}$
$b \in bexp = p \,|\, b \wedge b \,|\, b \vee b$

Phrasale Syntax
$orexp \;::=\; [\,orexp \;\; "or"\,] \;\; andexp$
$andexp \;::=\; [\,andexp \;\; "and"\,] \;\; pbool$
$pbool \;::=\; "true" \;\; | \;\; "false" \;\; | \;\; "("\, orexp\, ")"$

RA-taugliche Form der Grammatik
$orexp \;::=\; andexp \;\; orexp'$
$orexp' \;::=\; [\,"or" \;\; andexp \;\; orexp'\,]$
$andexp \;::=\; pbool \;\; andexp'$
$andexp' \;::=\; [\,"and" \;\; pbool \;\; andexp'\,]$
$pbool \;::=\; "true" \;\; | \;\; "false" \;\; | \;\; "("\, orexp\, ")"$
Der Lexer ist durch folgendes Programm gegeben:

(b)
```
exception Error of string datatype bexp = True | False | And of bexp * bexp | Or of bexp
* bexp datatype token = TRUE | FALSE | AND | OR
fun lex nil = nil | lex (#" " :: cr) = lex cr
| lex (#"\t" :: cr) = lex cr
| lex (#"\n" :: cr) = lex cr
| lex (#"t" :: #"r" :: #"u" :: #"e" :: cr) = TRUE :: lex cr
| lex (#"f" :: #"a" :: #"l" :: #"s" :: #"e" :: cr) = FALSE :: lex cr
| lex (#"a" :: #"n" :: #"d" :: cr) = AND :: lex cr
| lex (#"o" :: #"r" :: cr) = OR :: lex cr
| lex s = raise Error "lex"
```

(c) Der Parser ist durch folgendes Programm gegeben:

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

fun orexp ts = orexp' (andexp ts)

and orexp'(e, OR::ts) = orexp'( extend (e,ts) andexp Or)
| orexp' s = s

and andexp ts = andexp' (pbool ts)

and andexp' (e, AND::ts) = andexp' ( extend (e,ts) pbool And)
| andexp' s = s

and pbool(TRUE::ts) = (True, ts)
| pbool(FALSE::ts) = (False,ts)
| pbool (LPAR::ts) = (case orexp ts of (e, RPAR::tr) => (e, tr) | s => raise Error "Match")

| pbool s = raise Error "parse"

fun parse p ts = case p ts of
(a,nil) => a
| s => raise Error "parse"
```

Aufgabe 14.10 (*Aufgabe aus dem Nachklausurtutorium zum WS 11/12*)
Wir betrachten Ausrücke, die mit Bezeichnern und den Operatoren ℵ und ℘, genannt *Hui* und *Buh* gebildet werden. ℘ klammert rechts und ℵ links. Außerdem klammert ℘ st¨arker als ℵ. Die folgenden Datentypen sind gegeben:

```
datatype token = ID of string | HUI | BUH | LPAR | RPAR
datatype exp = Id of string | Hui of exp * exp | Buh of exp * exp
```

a) Geben Sie eine eindeutige Grammatik an, die eine phrasale Syntax für diese Ausdrücke beschreibt.

b) Schreiben Sie einen Parser für diese Grammatik.

Lösung 14.10:

a)

$$\begin{aligned} exp &::= [exp\,ℵ]\,wexp \\ wexp &::= pexp\,[℘\,wexp] \\ pexp &::= id \,|\, "("\ exp\ ")" \end{aligned}$$

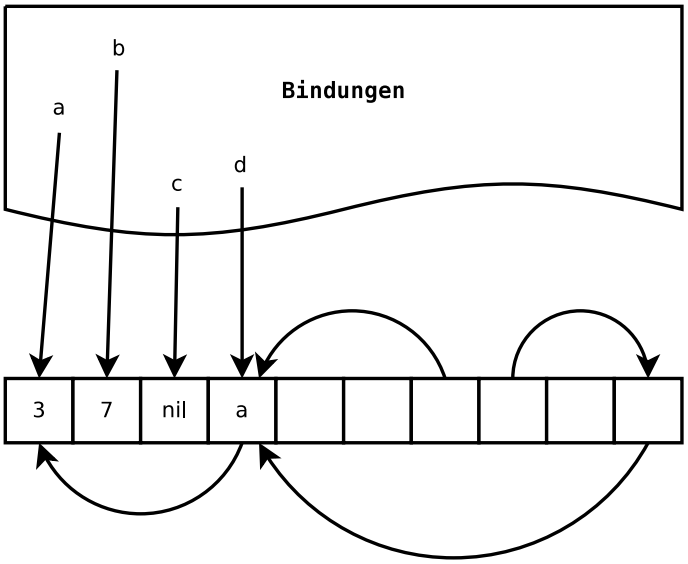
Rechtsrekursive Version:

$$\begin{aligned} exp &::= wexp\,exp' \\ exp' &::= [ℵ\,wexp\,exp'] \\ wexp &::= pexp\,[℘\,wexp] \\ pexp &::= id \,|\, "("\ exp\ ")" \end{aligned}$$

b)
```
fun exp tl = exp' (wexp tl)

and exp' (e, HUI::tl) = let val (e',tr) = wexp tl in exp'(Hui(e,e'),tr)
| exp' s = s

and wexp tl = (case pexp tl of (e,BUH::tr) => let val (e', ts) = wexp tr in (Buh(e,e'),
ts) end | s => s)

and pexp ((ID s)::tl) = (Id s, tl)
| pexp (LPAR::tl) = (case exp tl of (e,RPAR::tr) => (e,tr) | s => raise Error "match")

| pexp s = raise Error "parse"
```

Aufgabe 14.11 (*Aufgabe aus dem Wintersemester 11/12*)
Vervollständigen Sie die Bindungen (links) und das Schaubild (rechts). Im Schaubild sind sowohl die gewählten Bezeichner als auch der Speicher dargestellt. Pfeile deuten eine Beziehung zu einer Speicherzelle an.

```
val a = ref 3
val b = ref 7
val c = ref nil
val d = ref a
val
val
val
val
val
val
```



Aufgabe 14.12 (*Schriftliche Aufgabe aus dem Wintersemester 11/12*)
Überlegen Sie sich, welche der bereits bekannten Datenstrukturen und Algorithmen sich effizienter und/oder eleganter mit Hilfe des von Ihnen im Kapitel 15 neu erworbenen Wissens implementieren lassen. Implementieren Sie die von Ihnen ausgewählten Datenstrukturen und Alogorithmen. Vergessen Sie nicht, Ihre Überlegungen auch schriftlich darzulegen.