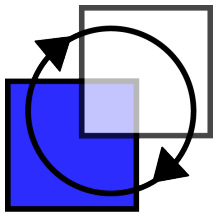
CISPA HELMHOLTZ CENTER FOR INFORMATION SECURITY

Prof. Bernd Finkbeiner, Ph.D.
Jana Hofmann, M.Sc.
Reactive Systems Group



Programmierung 1 (WS 2020/21)
Zusatztutorium 7 (Lösungsvorschläge)
Konkrete Syntax

Hinweis: Diese Aufgaben wurden von den Tutoren für das Zusatztutorium erstellt. Sie sind für die Klausur weder relevant noch irrelevant. 🤔 markiert potentiell schwerere Aufgaben.

1 Lexer

Aufgabe Z7.1 *(Lexer)*
Schreiben Sie eine Prozedur `lex : string → token list`, die ganze Zahlen und Bezeichner einliest. In vielen Aspekten verhält sich die Sprache wie SML.

- Negative Zahlen werden mit ∼ notiert.
- Bezeichner müssen mit einem Buchstaben anfangen, danach können Buchstaben oder Ziffern folgen, jedoch keine anderen Zeichen.
- Bei einem Zeichen, das nicht mehr zu einer Zahl bzw. einem Bezeichner gehört, soll ein neues Token begonnen werden.
- Als Trennzeichen wird ausschließlich das Leerzeichen betrachtet.

Gegeben ist folgender Datentyp:

```
datatype token = ICON of int | ID of string
```

Hinweis: Sie können `Char.isDigit` und `Char.isAlpha` verwenden. Es bietet sich an, die eigentliche Funktionalität in drei verschränkt rekursive Prozeduren `lex'`, `lexInt` und `lexId` auszugliedern.

Lösungsvorschlag Z7.1

```
exception Error of string

fun lex' [] = []
  | lex' (#"␣" :: cr) = lex' cr
  | lex' (#"~" :: c :: cr) = if Char.isDigit c then lexInt ~1 0 (c :: cr)
                             else raise Error "lex"
  | lex' (c :: cr) = if Char.isDigit c then lexInt 1 0 (c :: cr)
                     else if Char.isAlpha c then lexId [] (c :: cr)
                     else raise Error "lex"
and lexInt s a [] = [ICON (s * a)]
  | lexInt s a (c :: cr) = if Char.isDigit c
                           then lexInt s (a * 10 + ord c - ord #"0") cr
                           else ICON (s * a) :: lex' (c :: cr)
and lexId a [] = [ID (implode (rev a))]
  | lexId a (c :: cr) = if Char.isAlpha c orelse Char.isDigit c
                        then lexId (c :: a) cr
                        else ID (implode (rev a)) :: lex' (c :: cr)

fun lex s = lex' (explode s)
```

Aufgabe Z7.2 *(Strings in Strings)*
Wir möchten einen Lexer für Strings und Bezeichner schreiben, wie sie in SML vorkommen können. Zusätzlich soll eine Konkatenation für Strings existieren. Bedenken Sie, dass auch Zitate in einem String autauchen können. Wir wollen uns dabei nach folgender Grammatik richten:

$$exp ::= pexp\ [\ \text{“}\hat{}\text{”}\ exp]$$
$$pexp ::= id \mid \text{“}\texttt{"}\text{”}\ str\ \text{“}\texttt{"}\text{”} \mid \text{“(”}exp\text{“)”}$$

(a) Erstellen Sie den passenden Konstruktortypen `token` und eine Ausnahme `Invalid`

(b) Schreiben Sie einen Lexer für Strings

Lösungsvorschlag Z7.2

(a)
```
datatype token = ID of string | CONCAT | LPAR | RPAR | LITERAL of string
exception Invalid
```

(b)
```
fun lex nil = nil
  | lex (#"␣"  :: xr) = lex xr
  | lex (#"\t" :: xr) = lex xr
  | lex (#"\n" :: xr) = lex xr
  | lex (#"(" :: xr)  = LPAR :: lex xr
  | lex (#")" :: xr)  = RPAR :: lex xr
  | lex (#"^" :: xr)  = CONCAT :: lex xr
  | lex (#"\"" :: xr) = lexStr nil xr
  | lex (c :: xr)     = if Char.isAlpha c then lexId [c] xr
                        else raise Invalid

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

and lexStr c nil                    = raise Invalid
  | lexStr c (#"\\" :: #"\"" :: xr) = lexStr (#"\"" :: c) xr
  | lexStr c (#"\""::xr)            = LITERAL (implode (rev c)) :: lex xr
  | lexStr c (d::xr)                = lexStr (d::c) xr
```

2 Parser & Grammatiken

Aufgabe Z7.3 *(Lext du noch oder parst du schon?)*
Wir betrachten die Sprache $\alpha\beta\gamma$. Zusätzlich zu Zahlen und expliziten Klammern seien folgende Operatoren gegeben: α, β und γ.

- α und β sind binäre Infix-Operatoren und γ ein unärer Operator.
- α klammert rechts und β klammert links.
- γ klammert stärker als β und α klammert am stärksten.

(a) Erstellen Sie zunächst eine Grammatik, welche die obigen Bedingungen erfüllt.

(b) Geben Sie einen Ausdruck nach dieser Grammatik an, in der jeder Operator einmal vorkommt.

(c) Verändern Sie die Grammatik, sodass sie RA-tauglich wird.

(d) Erstellen Sie einen Konstruktortyp `token` und einen Konstruktortyp `exp` für die Baumdarstellung der Ausdrücke.

(e) Schreiben Sie nun einen Parser. Bei einer ungültigen Eingabe soll die Ausnahme `Parse` geworfen werden.

Lösungsvorschlag Z7.3

(a)

$$\begin{aligned} \mathit{beta} &::= \mathit{gamma} \mid \mathit{beta}\ \text{“}\beta\text{”}\ \mathit{gamma} \\ \mathit{gamma} &::= \mathit{alpha} \mid \text{“}\gamma\text{”}\ \mathit{gamma} \\ \mathit{alpha} &::= \mathit{prim} \mid \mathit{prim}\ \text{“}\alpha\text{”}\ \mathit{alpha} \\ \mathit{prim} &::= \mathit{num} \mid \text{“(”}\mathit{beta}\text{“)”} \end{aligned} \qquad \begin{aligned} \mathit{beta} &::= [\mathit{beta}\ \text{“}\beta\text{”}]\ \mathit{gamma} \\ \mathit{gamma} &::= \text{“}\gamma\text{”}\ \mathit{gamma} \mid \mathit{alpha} \\ \mathit{alpha} &::= \mathit{prim}\ [\text{“}\alpha\text{”}\ \mathit{alpha}] \\ \mathit{prim} &::= \mathit{num} \mid \text{“(”}\mathit{beta}\text{“)”} \end{aligned}$$

(b) $1\ \beta\ \gamma\ 2\ \alpha\ 3$

(c) RA-taugliche Grammatik mit Hilfskategorien:

$$\begin{aligned} \mathit{beta} &::= \mathit{gamma}\ \mathit{beta}' \\ \mathit{beta}' &::= [\text{“}\beta\text{”}\ \mathit{gamma}\ \mathit{beta}'] \\ \mathit{gamma} &::= \text{“}\gamma\text{”}\ \mathit{gamma} \mid \mathit{alpha} \\ \mathit{alpha} &::= \mathit{prim}\ [\text{“}\alpha\text{”}\ \mathit{alpha}] \\ \mathit{prim} &::= \mathit{num} \mid \text{“(”}\mathit{beta}\text{“)”} \end{aligned}$$

(d)

```
datatype token = ALPHA | BETA | GAMMA | ICON of int | LPAR | RPAR

datatype exp = Alpha of exp * exp | Beta of exp * exp | Gamma of exp | Icon of int
```

(e)

```
exception Parse

fun beta ts = beta' (gamma ts)
and beta' (e, BETA :: tr) = let val (e', tr') = gamma tr in beta' (Beta (e, e'), tr') end
  | beta' s = s

and gamma (GAMMA :: tr) = let val (e', tr') = gamma tr in (Gamma e', tr') end
  | gamma s = alpha s

and alpha ts = (case prim ts of (e, ALPHA :: tr) ⇒ let val (e', tr') = alpha tr
                                                    in (Alpha (e, e'), tr') end
                              | s ⇒ s)

and prim (ICON x :: tr) = (Icon x, tr)
  | prim (LPAR :: tr) = (case beta tr of (e', RPAR :: tr') ⇒ (e', tr')
                                       | _ ⇒ raise Parse)
  | prim _ = raise Parse

fun parse ts = case beta ts of (e, nil) ⇒ e
                             | _ ⇒ raise Parse
```

Aufgabe Z7.4 *(XY)*
Wir betrachten die Sprache XY, die aus den beiden binären Präfix-Operatoren X und Y und Zahlen besteht. Die zugehörige abstrakte Grammatik ist folgendermaßen definiert:
$e \in Exp = \text{“}X\text{”}\ exp\ exp \mid \text{“}Y\text{”}\ exp\ exp\ exp \mid num$

(a) Machen Sie sich klar, dass XY keine Klammern benötigt.

(b) Stellen Sie eine konkrete Grammatik für XY auf.

(c) Schreiben Sie einen Parser `parse: token list → exp` für XY.

```
datatype token = BIGX | BIGY | ICON of int
datatype exp = X of exp * exp | Y of exp * exp * exp | Icon of int
```

Lösungsvorschlag Z7.4

(a) Wenn man in der Eingabe jeden Operator durch seine Stelligkeit ersetzt, also X durch 2, Y durch 3 und jedes *num*-Token durch 0, dann handelt es sich um die Prälinearisierung des Syntaxbaums. Da wir aus der Prälinearisierung einen Baum eindeutig rekonstruieren können, können wir das auch bei einem Ausdruck aus der Sprache XY.

(b) $exp ::= \text{“}X\text{”}\ exp\ exp \mid \text{“}Y\text{”}\ exp\ exp\ exp \mid num$

(c)
```
exception Parse
datatype token = BIGX | BIGY | ICON of int
datatype exp = X of exp * exp | Y of exp * exp * exp | Icon of int

fun exp (BIGX :: tr) = let val (e1, tr') = exp tr
                           val (e2, tr'') = exp tr' in (X (e1, e2), tr'') end
  | exp (BIGY :: tr) = let val (e1, tr') = exp tr
                           val (e2, tr'') = exp tr'
                           val (e3, tr''') = exp tr'' in (Y (e1, e2, e3), tr''') end
  | exp ((ICON i) :: tr) = (Icon i, tr)
  | exp _ = raise Parse

fun parse ts = case exp ts of (e, nil) ⇒ e | _ ⇒ raise Parse
```

Aufgabe Z7.5 *(JSON)*
JSON (kurz für *JavaScript Object Notation*, `https://json.org`) ist, wie der Name schon vermuten lässt, ein kleiner Teil der Sprache *JavaScript*, mit dem sich Objekte repräsentieren lassen. Wegen der relativ einfachen Syntax wird *JSON* insbesondere bei Webanwendungen gerne als Format verwendet, um kleine strukturierte Datenmengen auszutauschen.

In dieser Aufgabe schreiben wir einen Parser für *JSON*. Die ganze *JSON*-Datei repräsentiert genau einen Wert. Ein Wert kann `null`, `true`, `false`, eine **Zahl**, ein **String**, ein **Array** oder ein **Objekt** sein. Ein Array besteht aus beliebig vielen Werten, die durch je ein Komma getrennt werden, und wird mit eckigen Klammern notiert, also zum Beispiel `[1, "abc", 1.2, true]` oder `[]`. Ein Objekt ist im Prinzip eine Liste von Schlüssel-Wert-Paaren, wobei ein Schlüssel immer ein String ist und mit einem Doppelpunkt vom Wert getrennt wird. Objekte werden mit geschweiften Klammern notiert, also zum Beispiel `{"abc": 123, "xyz": null}` oder `{}`.

Folgende Datentypen sind bereits gegeben:

```
datatype token = LBRACE (* { *) | RBRACE (* } *) | LBRACK (* [ *) | RBRACK (* ] *)
               | COMMA (* , *) | COLON (* : *)
               | STR of string | INT of int | REAL of real | TRUE | FALSE | NULL
datatype value = Null | True | False | Int of int | Real of real | Str of string
               | Array of value list | Object of (string * value) list
```

(a) Stellen Sie eine RA-taugliche Grammatik für *JSON* auf. Lassen Sie der Einfachheit halber zunächst das leere Array (`[]`) und das leere Objekt (`{}`) aus.

(b) Implementieren Sie den zugehörigen Parser.

(c) Es ist nicht schwer, den Parser auch `[]` und `{}` erkennen zu lassen. Erweitern Sie Ihren Parser entsprechend. Wenn Sie die Grammatik analog anpassen, ist diese nicht mehr RA-tauglich. Warum? Ist es möglich, dennoch RA-Tauglichkeit zu erlangen? 🤔

Lösungsvorschlag Z7.5

(a)

$$\begin{aligned} \mathit{value} &::= \mathit{null} \mid \mathit{true} \mid \mathit{false} \mid \mathit{int} \mid \mathit{real} \mid \mathit{string} \mid \text{“[”}\ \mathit{list}\ \text{“]”} \mid \text{“\{”}\ \mathit{kvlist}\ \text{“\}”} \\ \mathit{list} &::= \mathit{value}\,[\text{“,”}\ \mathit{list}] \\ \mathit{kvlist} &::= \mathit{string}\ \text{“ : ”}\ \mathit{value}\,[\text{“,”}\ \mathit{kvlist}] \end{aligned}$$

(b)

```
1  exception Parse
2
3  fun value (NULL :: tr) = (Null, tr)
4    | value (TRUE :: tr) = (True, tr)
5    | value (FALSE :: tr) = (False, tr)
6    | value ((INT x) :: tr) = (Int x, tr)
7    | value ((REAL x) :: tr) = (Real x, tr)
8    | value ((STR x) :: tr) = (Str x, tr)
9    | value (LBRACK :: tr) = (case list tr of (vs, RBRACK :: tr') ⇒ (Array vs, tr')
10                                           | _ ⇒ raise Parse)
11   | value (LBRACE :: tr) = (case kvlist tr of (kvs, RBRACE :: tr') ⇒ (Object kvs, tr')
12                                             | _ ⇒ raise Parse)
13   | value _ = raise Parse
14 and list ts = (case value ts
15                of (v, COMMA :: tr') ⇒ (case list tr' of (vs, tr'') ⇒ (v :: vs, tr''))
16                 | (v, tr') ⇒ ([v], tr'))
17 and kvlist ((STR k) :: COLON :: tr) = (case value tr
18                                        of (v, COMMA :: tr')
19                                           ⇒ (case kvlist tr'
20                                                of (kvs, tr'') ⇒ ((k, v) :: kvs, tr''))
21                                         | (v, tr') ⇒ ([(k, v)], tr'))
22   | kvlist _ = raise Parse
23
24 fun parse ts = case value ts of (v, []) ⇒ v | _ ⇒ raise Parse
```

(c) Der Parser wird folgendermaßen erweitert:

```
1  fun value ...
2    | value (LBRACK :: RBRACK :: tr) = (Array [], tr)
3    | value (LBRACK :: tr) = (case list tr of (vs, RBRACK :: tr') ⇒ (Array vs, tr')
4                                            | _ ⇒ raise Parse)
5    | value (LBRACE :: RBRACE :: tr) = (Object [], tr)
6    | value (LBRACE :: tr) = (case kvlist tr of (kvs, RBRACE :: tr') ⇒ (Object kvs, tr')
7                                              | _ ⇒ raise Parse)
8    ...
```

Die zugehörige Grammatik ist folgende:

$$\begin{aligned} \mathit{value} &::= \mathit{null} \mid \mathit{true} \mid \mathit{false} \mid \mathit{int} \mid \mathit{real} \mid \mathit{string} \mid \text{“[”“]”} \mid \text{“[”}\ \mathit{list}\ \text{“]”} \mid \text{“\{”“\}”} \mid \text{“\{”}\ \mathit{kvlist}\ \text{“\}”} \\ \mathit{list} &::= \mathit{value}\,[\text{“,”}\ \mathit{list}] \\ \mathit{kvlist} &::= \mathit{string}\ \text{“ : ”}\ \mathit{value}\,[\text{“,”}\ \mathit{kvlist}] \end{aligned}$$

Diese Grammatik ist nicht RA-tauglich, da in der Kategorie *value* mindestens zwei Token benötigt werden, um die Fälle voneinander zu unterscheiden. Nach unserer Definition dürfen wir jedoch nur maximal ein Token verwenden. Ein Lösungsansatz ist nun, für “[” und “{” je eine Hilfskategorie einzuführen und dort die Fallunterscheidung vorzunehmen:

$$
\begin{aligned}
\mathit{value} &::= \mathit{null} \mid \mathit{true} \mid \mathit{false} \mid \mathit{int} \mid \mathit{real} \mid \mathit{string} \mid \text{“[”}\ \mathit{array}' \mid \text{“\{”}\ \mathit{object}' \\
\mathit{array}' &::= \text{“]”} \mid \mathit{list}\ \text{“]”} \\
\mathit{object}' &::= \text{“\}”} \mid \mathit{kvlist}\ \text{“\}”} \\
\mathit{list} &::= \mathit{value}\ [\ \text{“,”}\ \mathit{list}\ ] \\
\mathit{kvlist} &::= \mathit{string}\ \text{“ : ”}\ \mathit{value}\ [\ \text{“,”}\ \mathit{kvlist}\ ]
\end{aligned}
$$

Doch auch diese Grammatik ist noch nicht RA-tauglich. Die Kategorien *list* und *kvlist* sind so zu behandeln, als könnten sie für etwas beliebiges stehen, also rein theoretisch auch ein “]” bzw. “}”. Dementsprechend kann die Entscheidnung noch immer nicht eindeutig getroffen werden.

Das Problem lässt sich lösen, indem man einmal *list* bzw. *kvlist* bis zu einem Terminal ersetzt, was allerdings für *list* sehr unschön ist:

$$
\begin{aligned}
\mathit{value} &::= \mathit{null} \mid \mathit{true} \mid \mathit{false} \mid \mathit{int} \mid \mathit{real} \mid \mathit{string} \mid \text{“[”}\ \mathit{array}' \mid \text{“\{”}\ \mathit{object}' \\
\mathit{array}' &::= \text{“]”} \mid \mathit{null}\ [\ \text{“,”}\ \mathit{list}\ ]\ \text{“]”} \mid \mathit{true}\ [\ \text{“,”}\ \mathit{list}\ ]\ \text{“]”} \mid \mathit{false}\ [\ \text{“,”}\ \mathit{list}\ ]\ \text{“]”} \mid \mathit{int}\ [\ \text{“,”}\ \mathit{list}\ ]\ \text{“]”} \\
&\quad \mid \mathit{real}\ [\ \text{“,”}\ \mathit{list}\ ]\ \text{“]”} \mid \mathit{string}\ [\ \text{“,”}\ \mathit{list}\ ]\ \text{“]”} \mid \text{“[”}\ \mathit{array}'\ [\ \text{“,”}\ \mathit{list}\ ]\ \text{“]”} \mid \text{“\{”}\ \mathit{object}'\ [\ \text{“,”}\ \mathit{list}\ ]\ \text{“]”} \\
\mathit{object}' &::= \text{“\}”} \mid \mathit{string}\ \text{“ : ”}\ \mathit{value}\ [\ \text{“,”}\ \mathit{kvlist}\ ]\ \text{“\}”} \\
\mathit{list} &::= \mathit{value}\ [\ \text{“,”}\ \mathit{list}\ ] \\
\mathit{kvlist} &::= \mathit{string}\ \text{“ : ”}\ \mathit{value}\ [\ \text{“,”}\ \mathit{kvlist}\ ]
\end{aligned}
$$

In *einzelnen* Fällen ist es also unter Verwendung gewisser Zusatzinformationen (in unserem Fall, dass eine schließende Klammer direkt nach einer öffnenden immer das leere Array bzw. das leere Objekt ist) einfacher, einen Parser zu einer nicht-RA-tauglichen Grammatik zu implementieren, als die Grammatik erst RA-tauglich zu machen.