Lokale Deklarationen

SML erlaubt zwei Arten, Deklarationen mit beschränktem Wirkungsbereich zu machen:

1. Deklarationen, deren Bereich ein Ausdruck ist:

   `let` *decl* `in` *expression* `end` ,

2. Deklartionen, deren Bereich eine Deklaration ist:

   `local` *decl* `in` *decl'* `end` .

Unabhängig davon können Deklarationen

1. *sequentiell* erfolgen:

   $$\mathtt{val}\ \mathtt{x} = \mathtt{x} + \mathtt{y} + 3;\ \mathtt{val}\ \mathtt{y} = \mathtt{x} * 2$$

   Das ergibt, wenn `x` und `y` gegenwärtig den Wert `1` haben,

   ```
   val x = 5 : int
   val y = 10 : int
   ```

2. *simultan* erfolgen:

   $$\mathtt{val}\ \mathtt{x} = \mathtt{x} + \mathtt{y} + 3\ \ \mathtt{and}\ \ \mathtt{y} = \mathtt{x} * 2$$

   Das ergibt, wenn `x` und `y` gegenwärtig den Wert `1` haben,

   ```
   val x = 5 : int
   val y = 2 : int
   ```

   Auch Deklarationen mit Mustern, z.B.

   $$\mathtt{val}\ (\mathtt{z}\ \mathtt{as}\ (\mathtt{x}, \mathtt{y})) = (\mathtt{x} + \mathtt{y} + 3, \mathtt{x} * 2)\,,$$

   sind simultane Deklarationen der Mustervariablen.

Lokale Deklarationen in Deklarationen

Diese Deklarationen

`local` *decl* `in` *decl'* `end`

grenzen Hilfsdeklarationen von anderen Deklarationen ab:

$\langle expl \rangle \equiv$

```
local
  fun add q [] = [q]
    | add q (u::U) = if q=u then (u::U)
                     else (u::(add q U))
in
  fun union [] U = U
    | union (q::Q) U = union Q (add q U)
  fun positives [] acc = acc
    | positives (n::ns) acc =
    if n > 0 then
        then positives ns (add n acc)
    else positives ns acc
end
```

Das ist insbesondere nützlich, wenn die Hilfsdeklaration in mehreren Deklarationen gebraucht wird. Außerdem wird klar gemacht, daß man `add` sonst nirgends benutzt. Bei

$\langle expl \rangle + \equiv$

```
fun add q [] = [q] | add q (u::U) = ...
fun union [] U = U | union (q::Q) U = ...
fun positives [] acc = acc
  | positives (n::ns) acc = ...
```

ist das nicht klar, und man verbraucht den Namen `add`.

Lokale Deklarationen in Ausdrücken

Die Deklarationen

`let` *decl* `in` *expression* `end` ,

werden oft verwendet, z.B. um Hilfsfuntionen zu definieren, ohne Argumente aus einer übergeordneten Funktion zu nennen:

$\langle expl \rangle {+}{\equiv}$

```
fun accept (Ma{trans = trans,
               start = start,
               final = final, ...}) w =
  let
    fun nextstates q a =
        let fun nexts [] acc = acc
              | nexts ((r,b,p)::trs) acc =
                if (r,b) = (q,a)
                  then nexts trs (add p acc)
                else nexts trs acc
        in nexts trans [] end
    and reach Q [] = Q
      | reach Q (a::l) =
        let
          val Qs = map (fn q => (nextstates q a)) Q
        in reach (Union Qs) l end
  in
    intersects (reach start w) final
  end
```

Hier hängen die Hilfsfunktionen impliztit von `Ma{...}` ab.

Bemerkung:

Oft werden lokale Definitionen in Wertausdrücken verwendet, wo man –zumindest auf Top-Level– besser eine lokale Definition in einer Deklaration machen sollte, z.B.

⟨*expl*⟩+≡
```
fun pos ns =
  let
    fun positives [] acc = acc
      | positives (n::ns) acc =
        if n > 0 then
            then positives ns (add n acc)
        else positives ns acc
  in
    positives ns []
  end
```

Auf eingebetter Ebene ist das möglich, aber weniger nützlich:

⟨*expl*⟩+≡
```
fun accept (Ma{start = start,...}) w =
  let
    local
        fun nexts [] acc = acc | ...
    in
        fun nextstates q a = nexts trans []
    end
    fun reach Q (a::l) = ... nextstates ... | ...
  in
    intersects (reach start w) final
  end
```

Feldtypen

SML hat Felder (Records) und entsprechende Feldtypen.

⟨*felder*⟩≡

```
    {label1 = wert1, .., labelk = wertk}
    : {label1 : typ1, .., labelk : typk}
```

Der Zugriff auf eines der Felder erfolgt durch

⟨*felder*⟩+≡

```
    #labelj {label1 = wert1, ...} => wertj
```

etwa

⟨*felder*⟩+≡

```
  - val F = {a = 2, b = 3};
  val F = {a=2,b=3} : {a:int, b:int}

  - #b {a = 2, b = 3}
  val it = 3 : int
```

Die Reihenfolge der Felder spielt keine Rolle:

⟨*expl*⟩+≡

```
  - {a = 2, b = 3} = {b = 3, a = 2}
  val it = true
```

In Feldmustern kann man Felder durch ... weglassen:

⟨*expl*⟩+≡

```
  - val {a = x, ...} = {a = 2, b = 3};
  val x = 2 : int
```

Beispiel: deterministische Automaten

$\langle dfa \rangle \equiv$

```
local
  type label = char
  and state = int
in
  type dfa = {alphabet : label list,
              initial : state,
              transition : state * label -> state,
              final : state list}
end

- val A = {alphabet = [#"a",#"b"],
           initial = 0,
           transition = fn (0,#"a") => 1
                         | (0,#"b") => 0
                         | (1,#"a") => 0
                         | (1,#"b") => 1,
           final = [1]}

val A = {alphabet=[#"a",#"b"],final=[1],
         initial=0,transition=fn} : dfa


- val s = #transition A (#initial A, #"b");
val s = 0 : int
```

Beispiel: Syntaktische Kategorien

⟨*cat*⟩≡

```
datatype numerus  = sing | plur
and casus = nom | gen | dat | acc
and person = fst | snd | thrd
and tempus = pres | past | fut

datatype cat =
  NP of {cas : casus, num : numerus, pers : person}
| VP of {pers : person, num : numerus,
         tense : tempus, subcat : cat list}

type lexeme = string * cat


- val cat1 = VP{pers = snd,
                num = plur,
                tense = pres,
                subcat = [NP{cas = dat,
                             num = sing,
                             pers = thrd},
                          NP{cas = acc,
                             num = plur,
                             pers = thrd}]};

val cat1 = VP{num=plur,pers=snd,tense=#,subcat=#} : cat

- val word : lexeme = ("gebt", cat1);
val word = ("gebt",VP {num=plur,pers=snd,..}) : lexeme
```