Programmierung und Modellierung

Methodik der Programmierung: Syntaxerkennung und -Behandlung

Martin Wirsing

in Zusammenarbeit mit

Moritz Hammer

SS 2009

Inhalt

[nach Smolka: Programmierung, Kap. 13, Oldenburg, 2008]

11.1 Abstrakte und konkrete Syntax

- **Abstrakte Syntax**
  - Beschreibt die Sätze einer Sprache in Baumform.
  - Die abstrakte Syntax bildet die Grundlage für die Formulierung der Semantik einer Programmiersprache.
- **Konkrete Syntax**
  - Aufbauend auf der abstrakten Syntax regelt die konkrete Syntax, wie die gültigen Sätze einer Sprache durch Wörter und Zeichen dargestellt werden.
  - Man unterscheidet
    - **Lexikalische Syntax**

      Darstellung von Wortfolgen durch Zeichenfolgen
    - **Phrasale Syntax**

      Darstellung von Bäumen durch Wortfolgen

Abstrakte und konkrete Syntax: Beispiel Arithmetik

- **Abstrakte Syntax**
  - `exp ::= num | id | exp+exp | exp∗exp` (in Baumform, eindeutig geklammert)
  - **In SML:**
```
datatype exp = Con of int | Id of string
             | Sum of exp*exp | Pro of exp*exp;
```
- **Phrasale Syntax**

  realisiert Klammersparnisregeln “Punkt-vor-Strich” und “Klammere-links”:
  - `exp ::= [exp "+"] mexp`
  - `mexp ::= [mexp “*"] pexp`
  - `pexp ::= num | id | "(“ exp ")“`
- **Lexikalische Syntax** (ohne Berücksichtigung von Leerzeichen)
  - `word ::= "+“ | “*“ | "(“ | ")“ | num | id`
  - `num ::= ["~"] pnum`
  - `pnum ::= digit [pnum]`
  - `digit ::= "0“ | … | "9"`
  - `id ::= letter [id]`
  - `letter ::= "a“ | … | "z“ | "A“ | … | "Z"`

Abstrakte und konkrete Syntax: Beispiel Typausdrücke

- **Abstrakte Syntax**
  - `typ  ::= bool | int | typ -> typ`
  - **In SML:** `datatype typ = Int | Bool | Arrow of typ*typ;`
- **Phrasale Syntax**

  realisiert “Klammere-rechts”:
  - `typ ::= ptyp | ptyp -> typ`
  - `ptyp ::= bool | int | „(“ typ „)“`
- **Lexikalische Syntax** (ohne Berücksichtigung von Leerzeichen)
  - `token  ::= “BOOL“ | “INT“ | "(“ | ")“ | ARROW`

12.2 Lexikalische Analyse

- Bei der **Lexikalischen Analyse** wird
  - geprüft, ob eine Folge von Zeichen lexikalisch zulässig ist und,
  - wenn ja, die Folge von Zeichen in eine Folge von Symbolen (token) der Programmiersprache übersetzt.
- Ein **Lexer** (auch Symbolentschlüssler, Scanner oder Tokenizer) ist eine Operation, die die lexikalische Analyse durchführt.
  - Auf die Eingabe einer Folge von Zeichen wird als Ergebnis eine (maximale) Liste von lexikalisch korrekten Wörtern und ein (möglicherweise) unbearbeiteter Reststring berechnet.
  - Die erkannten Wörter werden dabei als Token (Werte des Typs `token`) dargestellt.

12.2.1 Lexikalische Analyse: (einfache) Arithmetik

- Zeichendarstellung
  - Betrachte die Zeichen +, *, (,) und der Einfachheit halber nur zwei Identifikatoren xx, yy
  - Die Zeichen dürfen entweder direkt aufeinander folgen oder durch Leerzeichen getrennt sein.
  - Leerzeichen seien
    - Zwischenraum “ “, Tabulator `"\t“` und Zeilenwechsel `"\n“`.
- Beispiele
  - `“(xx +yy )”`
  - `“xx * yy \t +yy”`

Lexikalische Analyse: (einfache) Arithmetik

- Lexikalische Syntax
  - `word ::= "+" | "*" | "(" | ")" | id`
  - `id ::= "xx" | "yy"`
- Darstellung als Token
  - `word ::= ADD | MUL | LPAR | RPAR | id`
  - `id ::= XX | YY`
- Bei der Lexikalischen Analyse werden die Leerzeichen gelöscht und Wortsymbole in Token umgewandelt.
  - Beispiele
    - `"(xx+yy )"` wird zu `"(" "xx" "+" "yy" ")"`
      bzw. `LPAR XX ADD YY RPAR`
    - `"xx* yy \t+xx"` wird zu `"xx" "*" "yy" "+" "xx"`
      bzw. `XX MUL YY ADD XX`

Lexikalische Analyse: (einfache) Arithmetik

- Lexer

```
datatype token = ADD | MUL | LPAR | RPAR |  XX | YY;
- fun lex nil= nil
   |   lex (#“ “ :: cr)  = lex cr
   |   lex (#"\t“ :: cr) = lex cr
   |   lex (#"\n“ :: cr) = lex cr
   |   lex (#“x“ :: #“x“ :: cr)= XX :: lex cr
   |   lex (#“y“ :: #“y“ :: cr)= YY :: lex cr
   |   lex (#“+“ :: cr)= ADD :: lex cr
   |   lex (#“*“ :: cr)= MUL :: lex cr
   |   lex (#"(“ :: cr)  = LPAR :: lex cr
   |   lex (#")“ :: cr)  = RPAR :: lex cr
   |   lex _ = raise Error "lex“;
val lex : char list -> token list
```

- Beispiele

```
- lex (explode “(xx+yy )”);
   val it = [LPAR, XX, ADD, YY, RPAR] : token list
```

12.2.2 Lexikalische Analyse: Arithmetik

- **Lexikalische Syntax** (ohne Berücksichtigung von Leerzeichen)
  - `word  ::= "+“ | “*“ | "(“ | ")“ | num | id`
  - `num   ::= ["~"] pnum`
  - `pnum  ::= digit [pnum]`
  - `digit ::= "0“ | … | "9"`
  - `id    ::= letter [id]`
  - `letter ::= "a“ | … | "z“ | "A“ | … | "Z"`
- **Lexer**
  - Liest möglichst lange Zahlkonstanten und Identifikatoren (“maximal munch”-Regel); Leerzeichen nur erforderlich zwischen zwei Identifikatoren oder zwei Zahlen.
  - Beispiele

```
- lex(explode "x1");
val it = [ID "x", ICON 1] : token list
- lex(explode"one two");
 val it = [ID "one", ID "two"] : token list
- lex(explode"onetwo");
 val it = [ID "onetwo"] : token list
```

Lexikalische Analyse: Arithmetik

- Lexer

  Besteht aus drei verschränkt rekursiven Funktionen
  - `lex: char list  -> token list`
  - `lexId: char list -> char list  -> token list`
  - `lexInt: int -> int -> char list -> token list`

- In SML

```
- datatype token = ADD | MUL | LPAR | RPAR
                 | ICON of int | ID of string;
fun lex nil = nil
   |   lex (#“ “ :: cr)  = lex cr
   |   lex (#"\t“ :: cr) = lex cr
   |   lex (#"\n“ :: cr) = lex cr
   |   lex (#“+“ :: cr)  = ADD :: lex cr
   |   lex (#“*“ :: cr)  = MUL :: lex cr
   |   lex (#"(“ :: cr)  = LPAR :: lex cr
   |   lex (#")“ :: cr)  = RPAR :: lex cr
```

Lexikalische Analyse: Arithmetik

- Fortsetzung Lexer

```
    | lex (#"~"::c::cr) =
        if Char.isDigit c then lexInt ~1 0 (c::cr)
        else raise Error "~"
    | 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 (*zwei Hilfsfunktionen *)
```

Lexikalische Analyse: Arithmetik

- `lexId cs cs'`:
  - Das erste Argument cs ist die revertierte Liste der bereits gelesenen Buchstaben des Identifikators. Das zweite Argument cs ist der noch nicht gelesene String.
  - Falls cs ein neues Token beginnt, ist die revertierte Liste der gerade gelesenen Buchstaben des Identifikators ein neu erkanntes Token und lex wird mit cs' rekursiv aufgerufen; andernfalls wird zu cs das erste Element von cs' hinzugefügt und lexId rekursiv mit dem Rest von cs' aufgerufen.
- In SML:

```
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')
and
```

- **Beispiele**

```
- lexId [] (explode "Aufgabe5");
val it = [ID "Aufgabe",ICON 5] : tokenlist
- lexId [#"f", #"u", #"A"] (explode "gabe5");
val it = [ID "Aufgabe", ICON 5]: tokenlist
```

Lexikalische Analyse: Arithmetik

- `lexInt s x cs`:
  - Das erste Argument s ist 1 oder ~1, wobei ~1 anzeigt, dass ein negatives Vorzeichen gelesen wurde. Das zweite Argument v ist die Zahl, die sich aus den bisher gelesenen Ziffern ergibt. Das dritte Argument cs ist der noch nicht gelesene String
  - Falls cs ein neues Token beginnt, ist die gerade berechnete Zahl ein neu erkanntes Token und lex wird mit cs rekursiv aufgerufen; andernfalls wird aus v und dem ersten Element von cs ein neuer Wert für v berechnet und lexInt rekursiv mit dem Rest von cs aufgerufen.
- In SML:

```
lexInt s v cs =
    if null cs orelse not(Char.isDigit(hd cs))
      then ICON (s*v) :: lex cs
      else lexInt s (10*v +(ord(hd cs)-ord #"0"))(tl cs)
```

- Beispiele:

```
- lex(explode "~053Bilder");
val it = [ICON ~53,ID “Bilder"] : token list
- lexInt ~1 05 (explode “3Bilder");
val it = [ICON ~53,ID “Bilder"] : token list
```

11.3 Syntaktische Analyse: Parsing durch rekursiven Abstieg

- Bei der **Syntaktischen Analyse** wird
  - geprüft, ob eine Folge von Token (Symbolen) einen syntaktisch korrekten Satz (Wort, Ausdruck) einer Sprache bildet und,
  - wenn ja, die Folge von Token in einen abstrakten Syntaxbaum übersetzt.
- Sei eine Grammatik vom syntaktischer Kategorie K gegeben.

  Parsing nennt man den Prozess der Durchführung der syntaktischen Analyse.
  - Ein **(Syntax-) Prüfer** für K ist eine Operation, die für eine Folge von Token entscheidet, ob es sich um einen Satz gemäß K handelt, wobei ein (möglicherweise) unbearbeiteter Reststring übrig bleiben kann.
  - Ein **Parser** für K ist ein Prüfer, der für den Satz gemäß K eine Baumdarstellung liefert.

- Es gibt mehrere übliche Parsingmethoden:
  - **Rekursiver Abstieg („recursive descent“)**
  - LL-Parsing (Top-Down Parsing)
  - LR-Parsing (Bottom-up Parsing)

11.3.1 Syntaktische Analyse einfacher Zahlenfolgen

- (Abstrakte) Grammatik der Zahlenfolgen

  ```
  seq ::= "0“ | "1“ seq| "2“ seq seq
  ```

- Beispiele
  - Wörter der Grammatik seq

    ```
    0, 10, 110, 200, 2010 20110
    ```

  - Nicht korrekt sind:

    ```
    00, 1, 100, 210, 2001
    ```

Syntaktische Analyse: Ein Prüfer für seq

- `test : int list -> int list`
  - prüft, ob die gegebene Liste mit einem Satz gemäß der syntaktischen Kategorie `seq` beginnt.
  - Dabei werden die Zahlen der Liste eine nach der anderen „gelesen“, bis entweder ein vollständiger Satz (für `seq`) vorliegt oder man ausschließen kann, dass die Liste mit einem Satz von `seq` beginnt.
  - Im positiven Fall ist das Ergebnis die Liste der nicht gelesenen Zahlen, im negativen Fall eine Ausnahme.
- In SML

```
fun test(0::tr) = tr
|   test(1::tr) = test tr
|   test(2::tr) = test (test tr)
|   test _ = raise Error "test";
```

Syntaktische Analyse: Ein Prüfer für seq

- **Beispiele**

```
- test[2,0,1,0];
val it = []:int list
- test[2,0,1,0,5];
val it =[5] : int list
- test[2,0,1];
!Uncaught exception: Error "test"
```

Syntaktische Analyse: Tauglichkeit für Rek. Abstieg

- **Terminierung und Eindeutigkeit des rek. Abstiegs**
  - `test ts` terminiert, da jeder rek Aufruf auf ein echt kürzeres Argument als `ts` angewandt wird.
  - `test ts` ist eindeutig, da die Fallunterscheidung anhand des ersten Elements von `ts` eindeutig bestimmt ist.

- **RA-Tauglichkeit**

  Eine konkrete Grammatik heißt **RA-tauglich**, wenn die algorithmische Interpretation ihrer Gleichungen Folgendes erfüllt

  1. Falls es zu einer Rekursion kommt, wird die Argumentliste um mindestens ein Wort verkürzt.
  2. Die Wahl zwischen verschiedenen Alternativen kann anhand des ersten Wortes der Argumentliste entschieden werden.

Syntaktische Analyse: Ein Parser für seq

- Erweiterung des Prüfers in einen Parser, der Sätze gemäß `seq` in Bäume des Typs

  ```
  datatype tree = A | B of tree | C of tree * tree
  ```

  übersetzt.
- Zusammenhang Baumdarstellung und Sätze von `seq`

```
- fun rep A          = [0]
|   rep (B t)      = 1 :: rep t
|   rep (C(t,t')) = 2 :: rep t @ rep t';
rep: tree -> int list
- rep (C(B A,C(A,A)));
val it = [2,1,0,2,0,0] : int list
```

Syntaktische Analyse: Ein Parser für seq

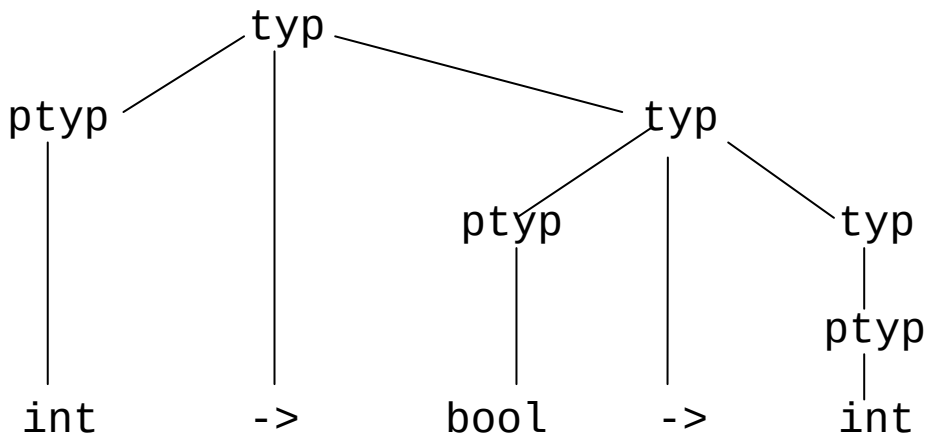
```
fun parse (0::tr) = (A, tr)
|   parse (1::tr) =
          let val (s, ts) = parse tr
          in (B s, ts) end
|   parse (2::tr) =
          let val (s, ts)   = parse tr
              val (s‘, ts‘) = parse ts
          in (C(s,s‘), ts‘) end
|   parse _ = raise Error „parse";

- parse [2,1,0,2,0,0,5];
val it = (C(B A,C(A,A))), [5]) : tree * int list
- parse (rep (C(B A,C(A,A))));
val it = (C(B A,C(A,A))), []) : tree * int list
```

11.3.2 Syntaktische Analyse für Typausdrücke

- **Abstrakte Grammatik**
  - `typ ::= bool | int | typ -> typ`
  - **In SML**

    ```
    datatype typ = Int | Bool | Arrow of typ*typ;
    ```
- **Konkrete Phrasale Grammatik (für Rechtsklammerung)**
  - `typ ::= ptyp | ptyp -> typ`
  - `ptyp ::= bool | int | „(“ typ „)“`
- **Beispiel:** `int -> bool -> int`



Syntaktische Analyse für Typausdrücke

- **Konkrete Phrasale Grammatik ist nicht RA-tauglich:**
  1. `typ ::= ptyp | ptyp -> typ`
  2. `ptyp ::= bool | int | „(“ typ „)“`

     da in 1. das erste Wort nicht ausreicht, um die Alternative eindeutig zu entscheiden.
- **Äquivalente Grammatik:**
  1. `typ ::= ptyp [ -> typ]`
  2. `ptyp ::= bool | int | „(“ typ „)“`

     ist RA-tauglich (aber bei 1. wird Fallunterscheidung benötigt).

Syntaktische Analyse: Ein Prüfer für Typausdrücke

- **Syntaxprüfer**

  Verschränkt rekursive Definition von Prüfern für `typ` und `ptyp`

  - `datatype tok = BOOL | INT | ARROW | LPAR | RPAR;`
  - `ty0  : token list -> token list`
  - `pty0 : token list -> token list`

- **In SML:**

```
fun ty0 ts = case pty0 ts of
                  ARROW::tr => ty0 tr
                  | tr => tr
and pty0 (BOOL::tr) = tr
|   pty0 (INT::tr)  = tr
|   pty0 (LPAR::tr) = (case ty0 tr of
                         RPAR::tr1 => tr1
                         | _ => raise Error „RPAR“
|    pty0 _ = raise Error „pty“;
```

Syntaktische Analyse: Ein Prüfer für Typausdrücke

- **Beispiele**

```
ty0 [INT, ARROW, BOOL, RPAR];
ty0 [INT, ARROW, BOOL, ARROW];
ty0 [LPAR, INT];
```

Syntaktische Analyse: Ein Parser für Typausdrücke

- **Abstrakte Syntax**
  - `datatype typ = Int | Bool | Arrow of typ*typ;`
- **Parser**

  Verschränkt rekursive Definition von Parsern für `typ` und `ptyp`
  - `ty  : token list -> typ * token list`
  - `pty : token list -> typ * token list`
- **In SML:**

```
fun ty ts = case pty ts of
              (t,ARROW::tr) => let val (t1,tr1) = ty tr
                               in (Arrow(t,t1), tr1) end
            | s => s
and pty (BOOL::tr) = (Bool, tr)
|   pty (INT::tr)  = (Int, tr)
|   pty (LPAR::tr) = (case ty tr of
                        (t, RPAR::tr1) => (t, tr1)
                      | _ => raise Error „RPAR“
|    pty _ = raise Error „pty“;
```

Syntaktische Analyse: Ein Parser für Typausdrücke

- **Beispiele**

```
ty [INT, ARROW, BOOL, RPAR];
ty [INT, ARROW, BOOL, ARROW];
ty [LPAR, INT];
```

11.3.3 Syntaktische Analyse: Allgemeine Hilfsfunktionen

- Die erweiterte Untersuchung des zweiten Wortes

```
... case pty ts of
        (t,ARROW::tr) => let val (t1,tr1) = ty tr
                         in (Arrow(t,t1), tr1) end
...
```

  kann allgemeiner so definiert werden

-
```
fun extend (t, tr) p f =
          let val (t1, tr1) = p tr
          in (f(t, t1), tr1) end;
```

Syntaktische Analyse: Allgemeine Hilfsfunktionen

- Matching

```
... (LPAR::tr) = (case ty tr of
                          (t, RPAR::tr1) => (t, tr1)
                          | _ => raise Error „RPAR“...
```

kann allgemeiner so definiert werden

  - ```
    fun match (t, tr) r =
            if null tr orelse hd tr <> r
                  then raise Error „match“
                  else (t, tl tr);
    ```

- **parse**
  - prüft, ob alle Wörter gelesen sind und konvertiert Parser des Typs `token list -> ‘a * token list` in Typ `token list -> ‘a`
  - ```
    fun parse p tr = case (p tr) of
                          (t, nil) => t
                          | _ => raise Error „parse“;
    ```

Syntaktische Analyse: Ein Parser mit Hilfsfunktionen

- **Parser mit match und extend:**

```
fun ty ts = case pty ts of
              (t,ARROW::tr) => extend (t, tr) ty Arrow
              | s => s
and pty (BOOL::tr) = (Bool, tr)
|   pty (INT::tr)  = (Int, tr)
|   pty (LPAR::tr) = match (ty tr) RPAR
|    pty _ = raise Error „pty“;
```

Syntaktische Analyse: Ein Parser mit Hilfsfunktionen

- **Beispiele**

```
- ty [INT, ARROW, BOOL, RPAR];

- parse ty [INT, ARROW, BOOL, RPAR];

- parse ty [INT, ARROW, BOOL, ARROW, INT];
```

11.3.4 Syntaktische Analyse für Arithmetische Ausdrücke

- **Phrasale Syntax**

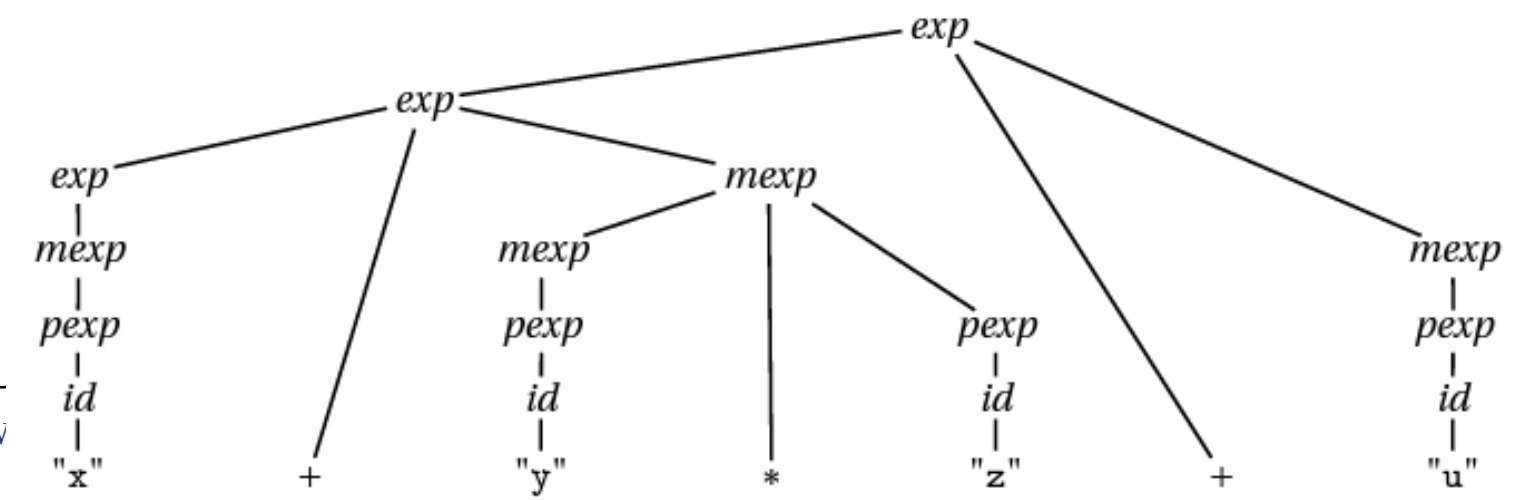
  realisiert Klammersparnisregeln “Punkt-vor-Strich” und “Klammere-links”:

  - `exp  ::= [exp "+"] mexp`
  - `mexp  ::= [mexp “*"] pexp`
  - `pexp  ::= num | id | "(“ exp ")“`

- **Beispiel**

  1. `x + y + z` entspricht `(x + y) + z`
  2. `x + y * z + u` entspricht `(x + (y * z)) + u`

- **Ziel des Parsing von 2.:**



Syntaktische Analyse für Arithmetische Ausdrücke

- **Problem: Die Grammatik**

```
1. exp   ::= [exp "+"] mexp
2. mexp  ::= [mexp “*"] pexp
3. pexp  ::= num | id | "(“ exp ")“
```

ist **nicht** RA-tauglich (wegen der Linksrekursion in 1.,2.)

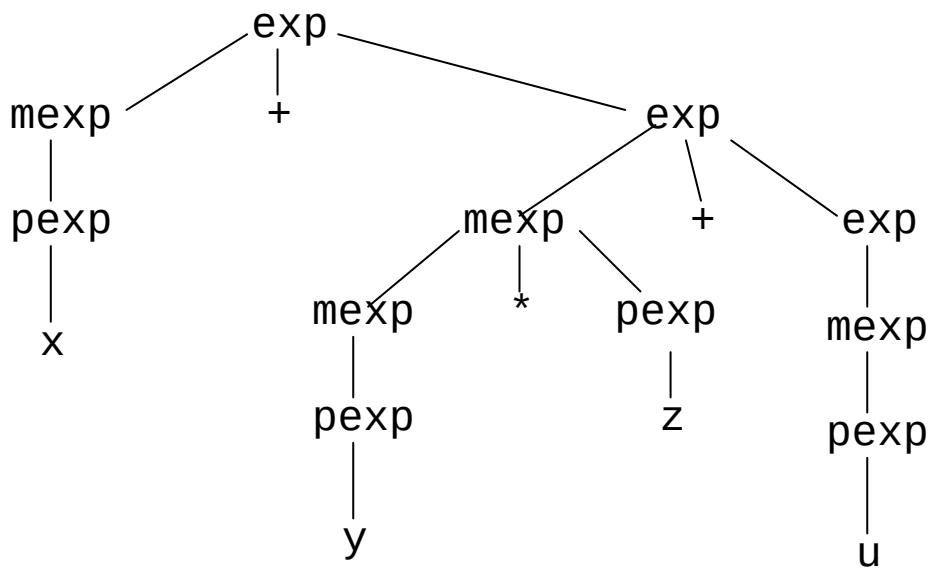
Syntaktische Analyse für Arithmetische Ausdrücke

- **Äquivalente rechtsrekursive Grammatik**

```
1. exp   ::= mexp [“+" exp]
2. mexp  ::= pexp [“*" mexp]
```

ist RA-tauglich, aber unterstützt Rechtsklammerung

- **Beispiel**
  - `x + y * z + u` wird geparst zu

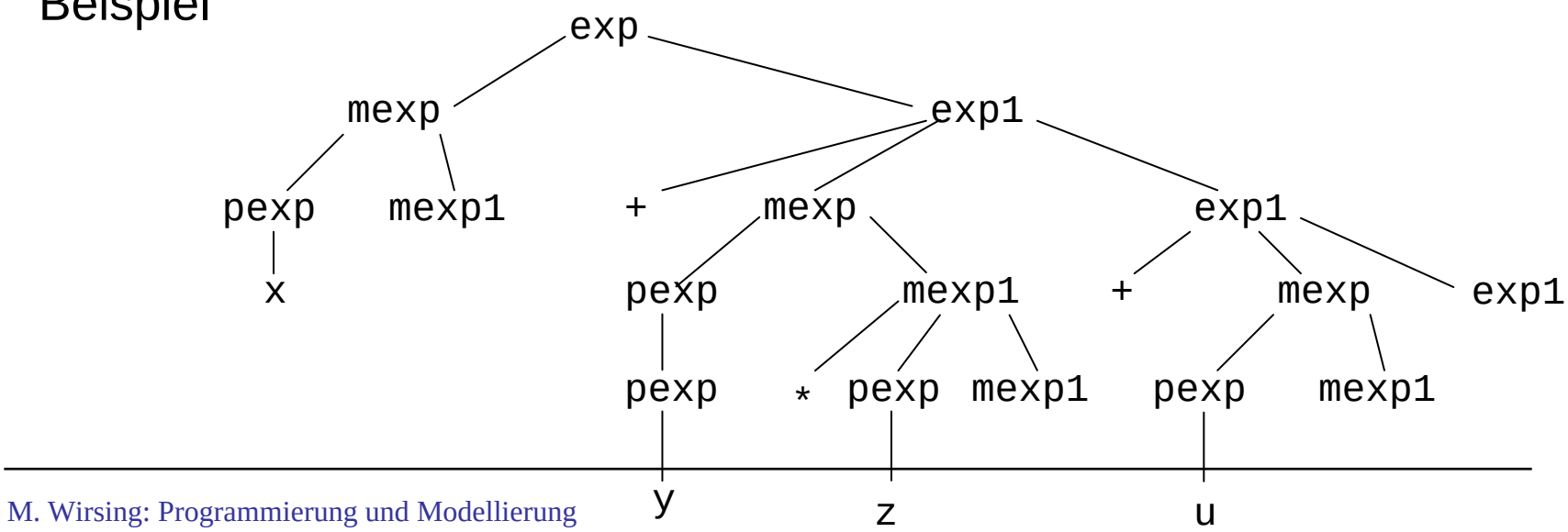


Syntaktische Analyse für Arithmetische Ausdrücke

- **Ziel**
  - **Einfache Umklammerung beim Bilden des abstrakten Syntaxbaums**
  - **Idee: Bilde Grammatik mit mehrfacher Addition und Multiplikation**
- **Neue äquivalente rechtsrekursive Grammatik**

```
1. exp   ::= mexp exp1
2. exp1  ::= [“+" mexp exp1] (für mexp + mexp + … + mexp)
3. mexp  ::= pexp mexp1
4. mexp1 ::= [“*" pexp mexp1](für pexp * pexp * … * pexp)
```

- Beispiel



Syntaktische Analyse: Parser für arithmetische Ausdrücke

- **Abstrakte Syntax**
  - `exp ::= num | id | e+e | e*e`
  - **In SML**

```
datatype exp = Con of int | Id of string
             | Sum of exp*exp | Pro of exp*exp;
```

- **Parser**

```
fun exp ts = exp1 (mexp ts)
and exp1 (e, ADD :: tr) = exp1 (extend (e, tr) mexp Sum)
|   exp1  s = s
and mexp ts = mexp1 (pexp ts)
and mexp1 (e, MUL :: tr) = mexp1 (extend (e, tr) pexp Pro)
|   mexp1  s = s
and pexp (ICON z :: tr) = (Con z, tr)
|   pexp (ID x :: tr) = (Id x, tr)
|   pexp (LPAR :: tr) = match (exp tr) RPAR
|   pexp _ = raise Error “pexp”;
```

Zusammenfassung I

- Die **Abstrakte Syntax** beschreibt die Sätze einer Sprache in Baumform und bildet die Grundlage für die Formulierung der Semantik einer Programmiersprache.
- Aufbauend auf der abstrakten Syntax regelt die **konkrete Syntax**, wie die gültigen Sätze einer Sprache durch Wörter und Zeichen dargestellt werden. Man unterscheidet
  - die **lexikalische Syntax**, d.h. die Darstellung von Wortfolgen durch Zeichenfolgen, und
  - die **phrasale Syntax**, d.h. die Darstellung von Bäumen durch Wortfolgen.

Zusammenfassung II

- Bei der **Lexikalischen Analyse** wird geprüft, ob eine Folge von Zeichen lexikalisch zulässig ist und, wenn ja, wird die Folge von Zeichen in eine Folge von Symbolen der Programmiersprache übersetzt.
  - Ein **Lexer** (auch Symbolentschlüssler, Scanner oder Tokenizer) ist eine Operation, die die lexikalische Analyse durchführt.
- Bei der **Syntaktischen Analyse** wird geprüft, ob eine Folge von Token (Symbolen) einen syntaktisch korrekten Satz (Wort, Ausdruck) einer Sprache bildet und, wenn ja, wird die Folge von Token in einen abstrakten Syntaxbaum übersetzt.
  - **Parsing** nennt man den Prozess der Durchführung der syntaktischen Analyse.
  - Übliche Parsingmethoden sind **Rekursiver Abstieg („recursive descent"), LL-Parsing** (Top-Down Parsing) und **LR-Parsing** (Bottom-up Parsing)