Inhalt Kapitel 11: Formale Syntax und Semantik

1. Abstrakte und konkrete Syntax
2. Lexikalische Analyse
3. Formale Sprachen, Grammatiken, BNF
4. Syntaxanalyse konkret

Abstrakte Syntax

- Beschreibt die Syntax als Syntaxbäume
- Beispiel: Ein arithmetischer Ausdruck ist entweder ein atomarer Ausdruck oder eine Summe / Differenz / Produkt von arithmetischen Ausdrücken
- Klammerregeln, Details der Notation werden in der abstrakten Syntax nicht fixiert;
- Beispiel: Ein Satz besteht aus Subjekt, Prädikat, Objekt
- Satzzeichen, Groß- und Kleinschreibung, etc. werden nicht fixiert;

Konkrete Syntax

- Gibt an, wie die syntaktischen Konstrukte konkreten Zeichenketten entsprechen.
- In der Praxis nochmal aufgeteilt in
  - *Lexikalische Analyse*: Zerlegung von echten Zeichenketten, einschl. Leerzeichen etc. in abstrakte Zeichen, sog. *Tokens*. Das sind Zahlen, Bezeichner, Schlüsselworte, zusammengesetzte Symbole (=>).
  - *Phrasale Syntax*: Beschreibt die konkrete Syntax auf der Ebene der Tokens (Klammerung, Operatornotationen (Prä-, Post-, In-, Mixfix), Vorrangregeln)

Beispiel: arithmetische Ausdrücke

- Abstrakte Syntax von Ausdrücken:
  - Als Grammatik: `exp ::= num | id | exp+exp | exp*exp`
  - Als SML Datentyp:

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

- Phrasale Syntax:

```
exp     ::= summand [ADD exp]
summand ::= faktor [MUL summand]
faktor  ::= NUM | ID | LPAR exp RPAR
```

- Lexikalische Syntax für Ausdrücke:

```
ADD ::= "+" | MUL ::= "*" | LPAR ::= "(" | RPAR ::= ")"
NUM    ::= ["~"] PNUM
PNUM ::= DIGIT {PNUM}
DIGIT ::= "0" | ... | "9"
ID     ::= letter {id}
LETTER ::= "a" | ... | "z" | "A" | ... | "Z"
```

Lexikalische Analyse

- Einlesen einer Datei oder eines Strings
- Gemäß einer Spezifikation in eine Liste oder einen Strom von Tokens zerlegen.
  - Strom: endliche oder unendliche Liste. Bei Aufruf einer Funktion wird das jeweils nächste Element “on demand” berechnet.
- Programm, welches die lexikalische Analyse durchführt, heißt *Lexer*.
- In der Praxis werden Lexer aus einer Spezifikation automatisch erzeugt, z.B. mit `lex`, `flex`, `mllex`.

Lexer für vereinfachte arithmetische Ausdrücke

- Spezifikation:
  ```
  ADD ::= "+" | MUL ::= "*" | LPAR ::= "(" | RPAR ::= ")" |
  XX ::= "xx" | YY ::= "yy"
  ```
- Implementation in SML
  ```
  datatype token = ADD | MUL | LPAR | RPAR | XX | YY
  exception error of string;
  - 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
  ```

Beispiel

```
 - lex (explode "(xx+yy )");
 val it = [LPAR, XX, ADD, YY, RPAR] : token list
 - val str = "xx + (\n\t yy * \n \t\t xx)\n";
- TextIO.print str;
xx + (
 yy *
   xx)
val it = () : unit
- lex (explode str);
val it = [XX,ADD,LPAR,YY,MUL,XX,RPAR] : token list
- lex (explode "xx+xx");
val it = [XX,ADD,XX] : token list
- lex (explode "xx+plip");

uncaught exception error
  raised at: stdIn:25.22-25.33
-
```

(Formale) Sprache

- Ein Alphabet $A = \{a_1, \ldots, a_n\}$ ist eine endliche Menge von Symbolen. Die Elemente $a_1, \ldots, a_n$ von A nennt man auch Terminalsymbole.
- Ein Wort (über dem Alphabet A) ist eine endliche Folge von Elementen von A, also von Terminalsymbolen.
- Eine *(formale) Sprache L* (über dem Alphabet A) ist eine endliche oder unendliche Menge von Wörtern über A.

Grammatik in Backus-Naur-Form

- Eine (kontextfreie) Grammatik in Backus-Naur Form (BNF) über einem Alphabet A definiert eine Sprache über A.
- sie besteht aus einer Menge N von Nichtterminalsymbolen (disjunkt von A) und einer Menge von Regeln, sowie einem Startsymbol $S \in N$.
- Regeln haben die Form $X ::=$ Ausdruck, wobei X ein Nichtterminalsymbol ist und Ausdrücke gebildet werden aus:
  - Terminal- und Nichtterminalsymbolen
  - Sequentielle Komposition: $E_1\ E_2$ (Ausdruck E_1 gefolgt von E_2)
  - Auswahl $E_1 \mid E_2$ (Ausdruck E_1 oder Ausdruck E_2)
  - Option $[E]$ (Ausdruck E optional)
  - Wiederholung $\{E\}$ (Ausdruck E eine beliebige Zahl von Malen.
- Die definierte Sprache umfasst alle Wörter, die sich aus dem Startsymbol durch sukzessive Anwendung der Regeln erzeugen lassen. Man schreibt $L(G)$ für die durch die Grammatik G definierte Sprache.

Beispiel

- Grammatik für vereinfachte arithmetische Ausdrücke:

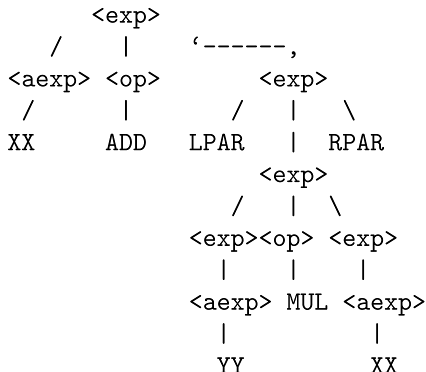
```
<exp>  ::= <aexp> | LPAR <exp> RPAR | <exp> <op> <exp>
<op>   ::= ADD | MUL
<aexp> ::= XX | YY
```

Alphabet $A = \{\texttt{XX}, \texttt{YY}, \texttt{LPAR}, \texttt{RPAR}, \texttt{ADD}, \texttt{MUL}\}$.
Nichtterminalsymbole $N = \{\texttt{<exp>}, \texttt{<aexp>}, \texttt{<op>}\}$.
Startsymbol: `<exp>`
Beispielableitung:

```
<exp> --> <exp> <op> <exp> -->
      --> <exp> <op> LPAR <exp> RPAR
      --> <exp> <op> LPAR <exp> <op> <exp> RPAR
      --> <aexp> <op> LPAR <aexp> <op> <aexp> RPAR
      --> XX ADD LPAR YY MUL XX RPAR
```

Ableitungsbäume, Syntaxanalyse (Parsing)

- Ableitungsbäume repräsentieren Ableitungen:



- *Syntaxanalyse (Parsing)*: Berechnen eines Ableitungsbaumes aus einem Wort (falls moglich).
- Wenn nicht eindeutig möglich: Grammatik umschreiben, Klammer / Priorisierungsregeln (phrasale Syntax)

Grammatik für vereinfachte arithmetische Ausdrücke

```
<exp>  ::= <summand> [ ADD <exp>]
<summand> ::= <faktor> [ MUL <summand> ]
<faktor> ::= XX | YY | LPAR <exp> RPAR
```

Jeder Satz eine eindeutige Ableitung und die Grammatik ist *nicht linksrekursiv*, d.h. es gibt keine Produktionen der Form $X ::= X \ldots$.

Man kann daher einen einfachen Parser nach der Methode des *rekursiven Abstiegs* verwenden.

In der Praxis verwendet man oft auch die allgemeineren und effizienteren LR-Parser.

Diese können mit bestimmten Tools (yacc, bison, mlyacc, AntLR, ...) aus einer Grammatik automatisch erzeugt werden.

Rekursiver Abstieg allgemein

Für jedes Nichtterminalsymbol `X` sieht man eine Funktion

```
parseX : token list -> exp * token list
```

vor, wobei im allgemeinen `exp` der Typ der Syntaxbäume ist und `token list` die Menge der Wörter repräsentiert.

Ist w ein Wort, so soll `parseX` ein Anfangsstück u von w finden, sodass `S --> u` und dann den entsprechenden Ableitungsbaum, sowie den ungeparsten Rest von w, also v, wobei $w = uv$, zurückliefern.

Die Funktionen `parseX` rufen sich gemäß der Produktionen gegenseitig rekursiv auf.

Ein Parser für Ausdrücke

Grammatik:

```
<exp>   ::= <summand> [ ADD <exp>]
<summand> ::= <faktor> [ MUL <summand> ]
<faktor> ::= XX | YY | LPAR <exp> RPAR
```

Programm:

```
datatype exp = Id of string | Sum of exp * exp
                            | Pro of exp * exp
```

```
fun parseexp text = let val (tree,rest) = parsesummand text in
                case rest of
                     ADD :: rest1 => let val (tree2,rest2) =
                                          parseexp rest1 in
                                        (Sum(tree,tree2), rest2) end
                    | _ => (tree,rest)  end
```

Fortsetzung

```
and parsesummand text  = let val (tree,rest) = parsefaktor text
                  case rest of
                    MUL :: rest1 => let val (tree2,rest2) =
                                      parsesummand rest1 in
                                    (Pro(tree,tree2), rest2) end
                   | _ => (tree,rest) end
and parsefaktor text = case text of
                 XX::rest => (Id "xx",rest)
               | YY::rest => (Id "yy",rest)
               | LPAR :: rest => let val (tree,rest1) =
                                   parseexp rest in
                                  case rest1 of
                                     RPAR :: rest2 => (tree,rest2)
                                     | _ => raise (error "parse")
                                           end
               | _ => raise (error "parse")
```

Anwendung

```
- Control.Print.printDepth := 100;
- parseexp (lex (explode "xx+yy*xx*(yy+ xx + xx * yy)*(xx *
                        (yy+xx))xxyyxxyy"));
val it =
  (Sum (Id "xx", Pro
          (Id "yy", Pro (Id "xx", Pro
               (Sum (Id "yy",Sum (Id "xx",Pro (Id "xx",Id "yy"))),
                Pro (Id "xx",Sum (Id "yy",Id "xx")))))),
   [XX,YY,XX,YY])
  : exp * token list
```

Zusammenfassung

- Man unterscheidet abstrakte und konkrete Syntax.
- Bei der *Lexikalischen Analyse* wird eine Eingabe (Datei, String) in eine Folge von Token (Symbolen) zerlegt.
- Bei der *Syntaxanalyse* wird geprüft, ob eine Folge von Token (Symbolen) einen syntaktisch korrekten Satz (Wort, Ausdruck) einer Sprache bildet
- Wenn ja, wird die Folge von Tokens in einen abstrakten Syntaxbaum übersetzt.
- *Parsing* bezeichnet den Prozess der Durchführung der Syntaxanalyse. Übliche Parsingmethoden sind *Rekursiver Abstieg* ("recursive descent", LL-Parsing) und LR-Parsing. *Parsergeneratoren* erzeugen Parser automatisch aus einer Grammatik.