

# Der ML-GLR Parser Generator

## Hauptseminar Relationale Grammatiken

Sebastian Bildner

16.11.09

# Überblick

- 1 Einführung
  - Grundlagen
  - Installation
- 2 ML-GLR
  - Aufbau
  - Grammatiken schreiben
  - Erzeugte Dateien
- 3 ML-Lex
  - Aufbau
  - Lexer schreiben
  - Erzeugte Datei
- 4 Anwendung

# Parsergeneratoren

- Parser von Hand zu schreiben ist sehr aufwändig
- Parsergeneratoren erzeugen Parser maschinell aus Grammatikspezifikationen
- Bekannte Parsergeneratoren:
  - Yacc
  - GNU Bison
  - ANTLR
  - ML-Yacc
- Benötigen oftmals separate Lexer (in unserem Fall: ML-Lex)
- ML-GLR basiert auf ML-Yacc, kann aber mit ambigen Grammatiken umgehen

# Parsergeneratoren

Unsere Parser benötigen drei Komponenten:

**ML-Lex** Lexer zur lexikalischen Analyse

- definiert Lexikon
- erzeugt aus Inputstring Tokens  
( $\rightarrow$  Terminalsymbole)

**ML-GLR** Parsergenerator zur syntaktischen Analyse und zur semantischen Auswertung

**SML-Code** vom Benutzer verfasst; zum Einlesen des Inputs und Weiterverwerten des Ergebnisses

# Installation

- `$HOME/.smlnj-pathconfig` mit folgendem Inhalt erstellen (SMLDIR steht für das Installationsverzeichnis von SML/NJ):  
`ml-ghr-lib.cm SMLDIR/ml-ghr/lib/  
mlghr-tool.cm SMLDIR/ml-ghr/tool/`
- ML-GLR mit Subversion herunterladen:  
`cd SMLDIR  
svn checkout URL ml-ghr`
- `build` (Windows: `build.bat`) aufrufen um Image zu erzeugen
- Heap image installieren:  
`SMLDIR/ml-ghr/src/ml-ghr.x86-linux` nach  
`SMLDIR/bin/.heap/ml-ghr.x86-linux` verschieben
- ML-GLR durch symbolischen Link aufrufbar machen:  
`ln -s SMLDIR/bin/ml-ghr SMLDIR/bin/.run-sml`

## Installation – Anmerkungen

- Die letzten beiden Schritte sehen unter Windows wahrscheinlich anders aus
- Keine Pfade mit Nicht-ASCII-Zeichen verwenden!
- Der Name des Image richtet sich nach dem Betriebssystem:

`ml-qlr.x86-linux` unter Linux

`ml-qlr.x86-darwin` unter Mac OS X

`ml-qlr.x86-win32` unter Windows ? (ist geraten)

# Aufteilung der Grammatikspezifikation

ML-GLR Benutzerdeklarationen (StandardML-Code)

%%

ML-GLR Deklarationen (Terminale, Nicht-Terminale, ...)

%%

ML-GLR Grammatikregeln

# Benutzerdeklarationen

Beliebiger SML-Code; normalerweise Definitionen von Datentypen

Beispiel:

```
datatype noun = Noun of string
datatype verb = Verb of string
datatype sentence = Sentence of (noun * verb)
```

# Deklarationen

Folgende Deklarationen müssen immer vorhanden sein:

`%name` Name der Grammatik (im Folgenden `<name>`)

`%pos` Datentyp der Positionsbeschreibung (i.d.R. `int`)

`%term` terminale Symbole

`%nonterm` nichtterminale Symbole

`%start` Startsymbol (muss ein nichtterminales Symbol sein)

`%eop` Stoppsymbol (muss ein terminales Symbol sein)

`%lexheader` siehe nächste Folie

## Hilfsdeklarationen

`%graphview` erzeuge Grafik (Graphviz muss installiert sein)

`%verbose` erzeuge lesbare Beschreibung (`*.desc`)

# Deklarationen – %lexheader

## %lexheader-Deklaration

Header für den <name>LexFun-Funktor;

Wird benötigt, um den Lexer in die Grammatik einzubinden. Nimmt neben der Tokens-Struktur auch eine Interface-Struktur als Argument und muss mit der %header-Deklaration im Lexer identisch sein.

## Beispiel:

```
%lexheader (functor <name>LexFun(  
    structure Tokens:      <name>_TOKENS  
    structure Interface: INTERFACE)  
: LEXER)
```

# Grammatikregeln

Regeln haben die folgende Form:

```
A : B C ... (SML Code)
   | D E ... (SML Code)
```

- Mehrere Produktionsregeln für das gleiche Symbol werden mit dem |-Zeichen aufgelistet
- Der SML-Ausdruck in den Klammern wird verwendet zur semantischen Auswertung.
- Der Ergebniswert dieses Ausdrucks ist dann in der nächsthöheren Regel weiterverwertbar und über den Namen der produzierenden Regel ansprechbar.

# Grammatikregeln – Beispiel

S : NP V (Sentence (NP,V))

NP : N (SimpleNP N)  
| N N (Comp (N1,N2))

N : n (Noun n)

V : v (Verb v)

## Anmerkung

Falls ein Symbol mehrfach verwendet wird, werden die Werte im SML-Code durchnummeriert.

## Erzeugte Dateien

Aus der *\*.grm*-Datei werden folgende Dateien erzeugt:

*\*.grm.sig* enthält Signaturen:

`<name>_VALS` für den Funktor `<name>ValsFun`

`<name>_TOKENS` für Tokens-Struktur in  
`<name>ValsFun`

*\*.grm.lnk.sml* Strukturen zur Verbindung von Lexer und Parser  
(am wichtigsten ist `<name>Parser`, hier werden die  
anderen Strukturen der Datei kombiniert)

*\*.grm.sml* Implementierung des Parsers (siehe nächste Folien)

## Inhalt der \*.grm.sml-Datei

### functor <name>ValsFun

- hat Signatur <name>\_VALS
- nimmt Strukturen Token und Grammar als Argumente (werden vom ChartParser gestellt)
- beinhaltet folgende Strukturen:

`Tokens` hat Signatur <name>\_TOKENS; Funktionen zur Erzeugung von Tokens für Terminalsymbole mit Hilfe der Struktur Token

`ParserData` siehe nächste Folie

# Inhalt der \*.grm.sml-Datei

```
structure ParserData in <name>ValsFun
```

```
structure Header Benutzerdeklarationen
```

```
datatype svalue Terminale und Nichtterminale des Benutzers
```

```
type result Typ des Startsymbols
```

```
    type pos Typ der Positionswerte (meistens int)
```

```
structure Actions Auswertungen der Grammatik-Regeln (durch  
den Code in den Klammern hinter den  
Grammatik-Regeln in der *.grm-Datei)
```

# Aufteilung

ML-Lex Benutzerdeklarationen

%%

ML-Lex Definitionen

%%

ML-Lex Regeln

# Benutzerdeklarationen – Obligatorische, Teil 1

Folgende Deklarationen müssen immer vorhanden sein (und reichen oftmals auch aus):

```
type svalue = Tokens.svalue
```

Datentyp der Grammatiksymbole

```
type pos = Interface.pos
```

Datentyp der Positionsdaten

```
fun eof () : lexresult = (  
  [Tokens.EOF(!Interface.lin, !Interface.col)],  
  "")
```

Diese Funktion erzeugt ein EOF-Token, bei dem der Parser stoppt.

## Benutzerdeklarationen – Obligatorische, Teil 2

```
type ('a,'b) token = ('a,'b) Tokens.token
```

Datentyp der Tokens

```
type lexresult = (svalue,pos) token list * string
```

Ergebnis einer erfolgreichen Regelanwendung: eine Liste mit möglichen Tokens für den gematchten String und der gematchte String.

## Benutzerdeklarationen – Beispiel

```
structure T = Tokens
structure I = Interface

type ('a,'b) token = ('a,'b) T.token
type svalue = T.svalue
type pos = I.pos
type lexresult = (svalue,I.pos) token list * string

open I (* providing pos, lin, col, init, coln, error *)
fun eof() : lexresult = (T.EOF(!lin,!col)::nil,"")
```

# Definitionen

`%full` kein Argument; benutze 8 Bits für Zeichen

`%header` Funktor für Integration mit Parser (siehe nächste Seite)

`%s` Liste der definierten Zustände  
Beispiel: `%s STATEA STATEB;`

`name = RegulärerAusdruck` Werden im Regelteil mit `{name}` benutzt  
Beispiel: `digit=[0-9];`

## Definitionen – Erläuterung zu %header

`%header`

```
(functor <name>LexFun (structure Tokens:<name>_TOKENS
                        structure Interface: INTERFACE)
  : LEXER);
```

### Anmerkung

- Ohne Header-Deklaration erzeugt ML-Lex eine fertige Struktur namens `Mlex`, aber ML-GLR benötigt einen Funktor namens `<name>LexFun`.
- Die `%header`-Deklaration muss mit der `%lexheader`-Deklaration in der Grammatik-Spezifikation übereinstimmen.
- Die LEXER-Signatur ist definiert in `ml-glr/lib/base.sig`

# Regeln

Regulärer Ausdruck => (Code);

Wenn der Input auf den regulären Ausdruck passt, wird der Code ausgeführt. Der Rückgabewert muss den Typ `lexresult` haben und wird als Token weitergereicht, außer im Code wird die `lex()`-Funktion aufgerufen, wodurch der Lexer weiterläuft.

Beispiel (siehe auch übernächste Folie):

```
{firstname} => (  
    [Tokens.person (yytext, !lin, !col)],  
    yytext);
```

## Regeln – Code

Im Code können folgende Werte verwendet werden:

`yytext` Inputstring

`yypos` Anfangsposition des Strings im Input

`yylineno` Zeilennummer (falls `%count` in den Definitionen)

und folgende Funktionen:

`lex()` ruft den Lexer rekursiv auf

`YYBEGIN state` wechselt in Zustand `state`

`REJECT()` Regel zurückweisen und Alternative suchen

## Regeln – Beispiel

```
{firstname} => (  
    [Tokens.person (yytext, !lin, !col)],  
    yytext);
```

### Erläuterungen

`firstname` Name eines regulären Ausdrucks

`person` Terminalsymbol aus der Grammatik; das erste Argument muss der dortigen Typdefinition entsprechen – in diesem Fall:  
`%term person of string`

`!lin, !col` Positionsdaten (siehe Interface)

Das Token stehen in einer Liste, um Alternativen zu ermöglichen

## Regeln – Zustände

```
<STATE ...>Regulärer Ausdruck => (Code);
```

Bei dieser Regelform kann die Regel nur aktiviert werden, wenn der Lexer sich gerade in einem der in den spitzen Klammern genannten Zustände befindet. Im Code kann der Zustand mit YYBEGIN gewechselt werden. INITIAL ist der vordefinierte Anfangszustand.

### Beispiel:

```
<INITIAL>{firstname} => (  
    YYBEGIN WORK;  
    ([Tokens.person(yytext !lin, !col)], yytext)  
);
```

## Erzeugte Datei

Aus der `*.lex`-Datei erzeugt ML-Lex eine `*.lex.sml`-Datei mit dem `<name>LexFun`-Funktoren der Signatur `LEXER`:

```
signature LEXER = sig
  structure UserDeclarations : sig
    type ('a,'b) token
    type pos
    type svalue
    type lexresult =
      (svalue,pos) token list * string
  end
  val makeLexer : (int -> string) -> unit ->
    UserDeclarations.lexresult
end
```

## Vom Benutzer verfasste Dateien

*interface.sml* enthält functor Interface. Nicht notwendig, bietet aber nützliche Funktionen, um z.B. Positionsdaten zu verwalten und Fehlermeldungen auszugeben.

\**interface.sml* enthält structure Interface (erzeugt mit functor Interface).

\**parse.sml* enthält den fertigen Parser in Form der Struktur `<name>` mit der `parse()`-Funktion

\**.cm* Compilation Manager

Die Dateien *\*.parser.sml* und *\*.cm* müssen dem Namen der Grammatik angepasst werden. Die Interface-Dateien können einfach von anderen Grammatiken kopiert werden.

# Compilation Manager

## Library

```
structure MyGrm          (* Benutzer-Struktur *)
structure MyGrmParser    (* Kombi. von Lexer, Parser *)
```

is

```
$/basis.cm              (* Basis-Bibliothek von SML *)
$/ml-glr-lib.cm         (* ML-GLR Parser Generator *)
$/mlglr-tool.cm        (* Aktiviert MLGLR-Befehl *)
```

```
MyGrm.lex               (* Lexer Spezifikation *)
MyGrm.grm:MLGLR        (* Grammatik Spezifikation *)
```

```
interface.sml          (* Interface Funktor *)
MyGrm.interface.sml    (* Interface Struktur *)
MyGrm.parse.sml        (* Impl. der Benutzer-Struktur *)
```

## Beispiel-Sitzung

```
- CM.make "MyGrm.cm";
```

```
  ⋮
```

*Haufenweise Output*

```
  ⋮
```

```
[New bindings added.]
```

```
val it = true : bool      ← false wenn es Fehler gab
```

```
- MyGrm.parse();
```

```
Kanzlerin Merkel spricht.
```

```
val it = [Sentence (Compound (#,#),Verb "spricht")]  
         : MyGrmParser.result list
```