The ACCENT Compiler Compiler

A compiler compiler for the entire class of context-free languages

Valentin Ziegler

mailto:ziegler@informatik.hu-berlin.de

3. Juli 2003

Accent

- “Compiler Compiler”
  für beliebige kontextfreie Grammatiken
- entwickelt von der Fraunhofer FIRST
- veröffentlicht als freie Software (GPL v2)

Accent

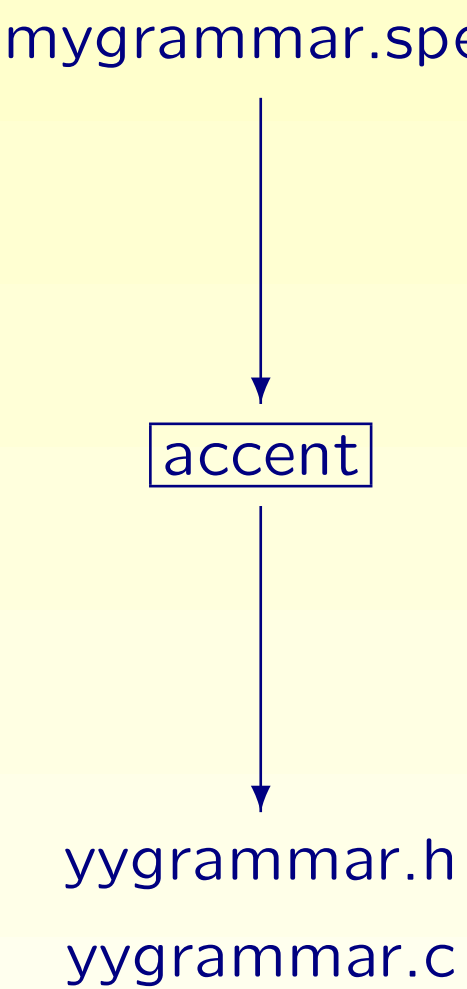
- "Compiler Compiler" Wirklich ?
  für beliebige kontextfreie Grammatiken
- entwickelt von der Fraunhofer FIRST
- veröffentlicht als freie Software (GPL v2)
- reiner Parsergenerator

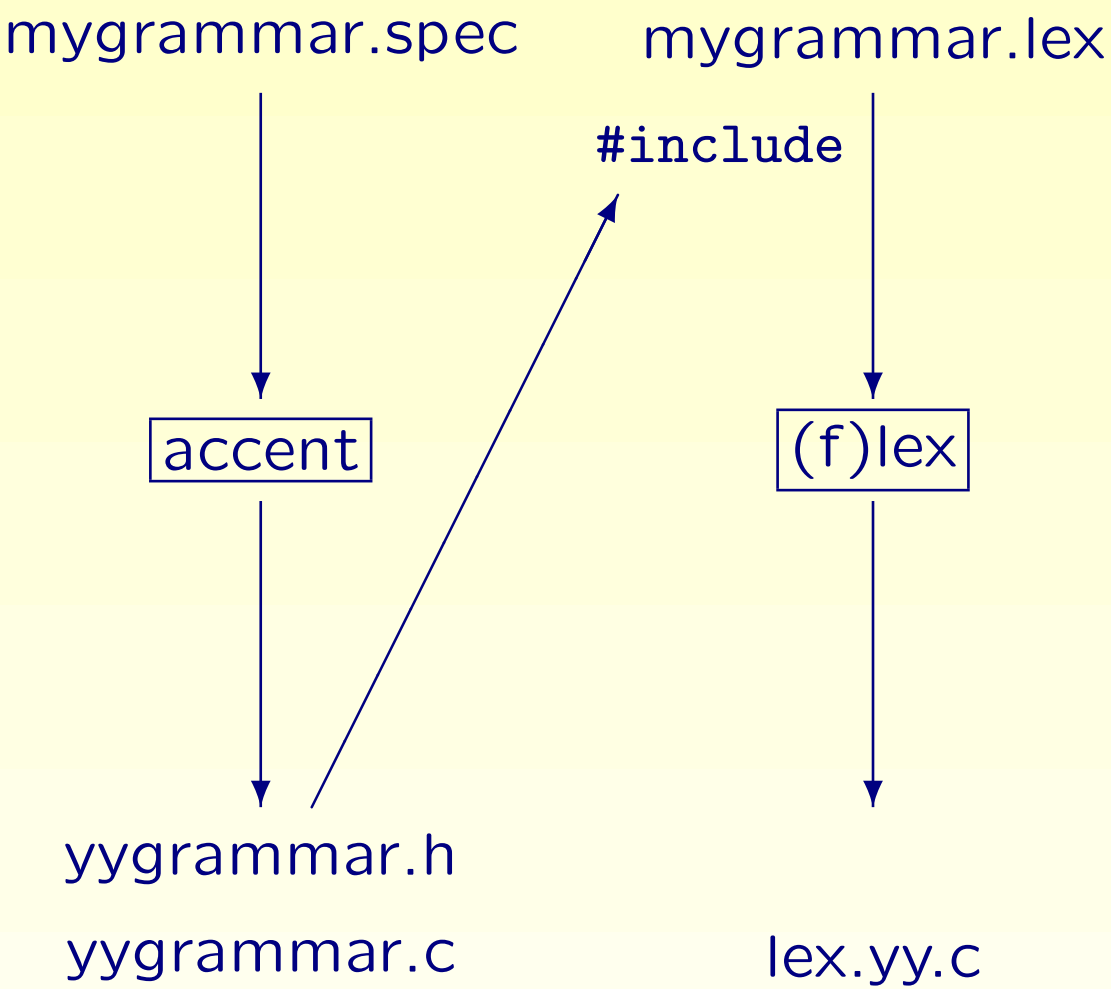
Gliederung

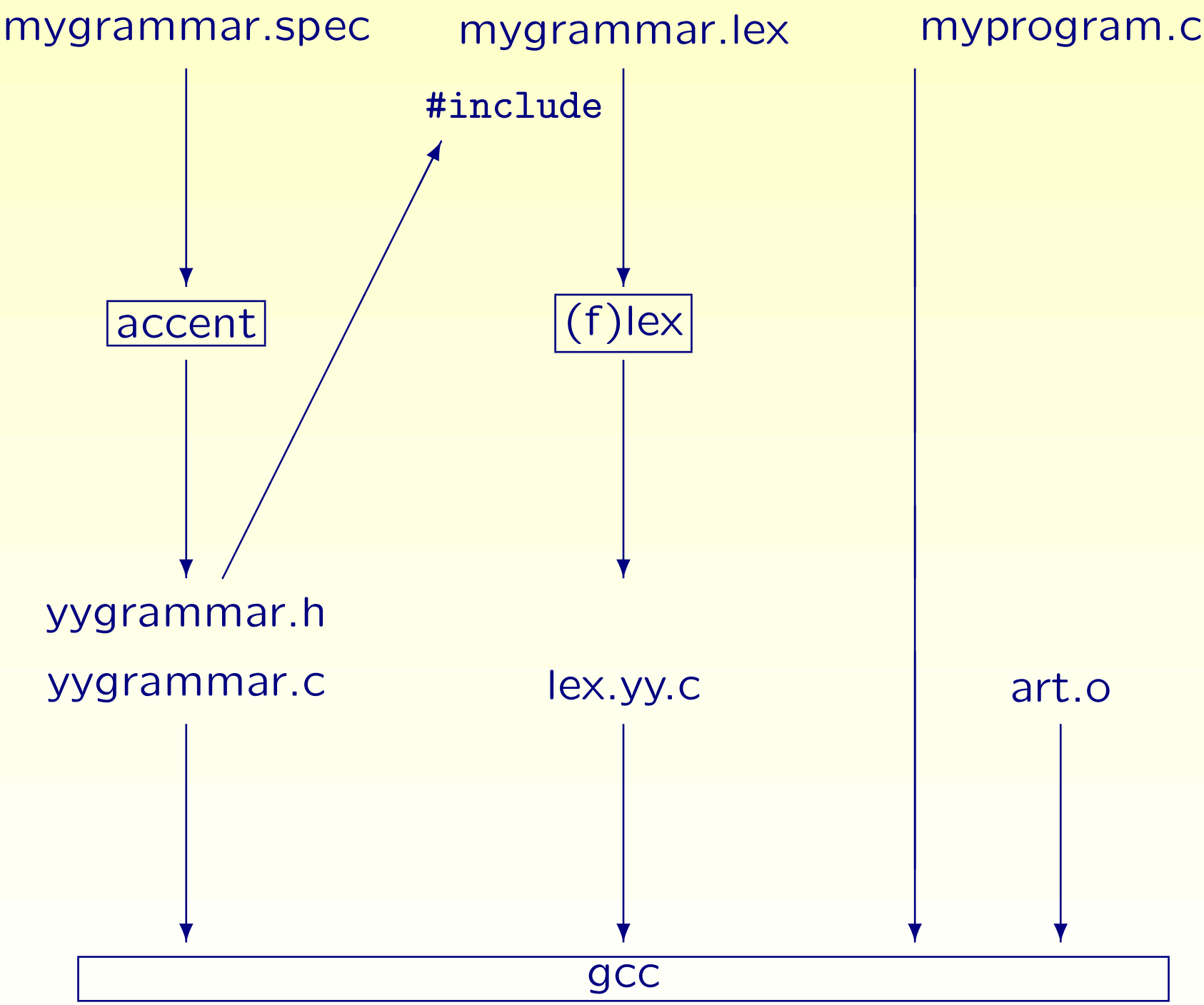
- Einleitung/Überblick
- Integration und benötigte Lexerfunktionen
- Accent Parsergenerator
- Parsertechnik und generierter Code
- Fazit

Accent: Ein kurzer Überblick

- erzeugt C-Code aus Spezifikationsdatei
  In dieser definiert:
  - Regeln der Grammatik (in EBNF Variante)
  - "Semantik" zu den Regeln (C-Code)
- jede EBNF Grammatik ist erlaubt
- auf Zusammenarbeit mit (f)lex ausgelegt


mygrammar.spec
accent
yygrammar.h
yygrammar.c


mygrammar.spec
mygrammar.lex
#include
accent
(f)lex
yygrammar.h
yygrammar.c
lex.yy.c


mygrammar.spec
mygrammar.lex
myprogram.c
#include
accent
(f)lex
yygrammar.h
yygrammar.c
lex.yy.c
art.o
gcc

Benötigte Lexerfunktionen

muss bereitgestellt werden:

- Funktion `int yylex ();`
- Variable `YYSTYPE yylvalue;`

Rückgabewert `yylex` : numerische ID des aktuellen Tokens gemäß erzeugter Datei `yygrammar.h`

Macro `YYSTYPE` standardmäßig `long`, kann vom Nutzer umdefiniert werden

Einbau in Hauptprogramm

Integration des generierten Parsers in ein Programm:

- `yyerror (char* msg)` muss implementiert werden wird vom Parser im Fehlerfall aufgerufen
- bei Benutzung von (f)lex ist ggf. `yyin` zu setzen und `yywrap` zu implementieren

start des Parsers durch Aufruf von `yyparse ();`

Aufbau Accent-Spezifikationsdatei ("spec-file")

Drei Teile:

- "global prelude" (optional)
- Token-Deklarationen (optional)
- Regeln und Semantik

global prelude

```
%prelude {
  /* beliebiger C-Code */
}
```

- Inhalt des Blockes wird an den Anfang der generierten Datei übernommen
- kann weggelassen werden, muss bei Verwendung aber am Anfang des Spec-files stehen

Token Deklarationen

- wird mit `%token` eingeleitet
- enthält durch Kommata separierte Liste der verwendeten Terminal-Bezeichner
- abgeschlossen durch Semikolon
  Beispiel:

  ```
  %token KW_IF, KW_ELSE, ... ;
  ```

- Einzeichen-Token müssen nicht deklariert werden

Regeln

eine Regel (ohne Semantik) hat die Form

```
nichtterminalID :
    RegelKoerper
;
```

wobei ein elementarer Regelkörper aus Listen von Nichtterminalen und Tokens besteht

erste Regel im spec-file stellt Startsymbol

Operatoren auf Regelkörpern

Sind φ, ψ Regelkörper, so sind auch

$\varphi \mid \psi$	(Alternative)
$\varphi\,?$	(Option)
$\varphi\,+$	(Wiederholung)
$(\ \varphi\ )$	(Gruppierung)

Regelkörper.

Interpretation der Operatoren gemäß EBNF Semantik

Semantik zuweisen: “semantic blocks”

semantik blocks...

- sind C-Statements in geschweiften Klammern
- dürfen an beliebiger Stelle im Regelkörper stehen:

```
fiend:
  'c' { prf("Got Cop"); } 0 'p'
| { prf("Going"); } 'c' 0 {prf(" to get Cow")} 'w'
;
```

- werden entsprechend der Reihenfolge, in der sie im Ableitungsbaum auftreten, aufgerufen

Semantik zuweisen: “semantic blocks”

semantik blocks...

- sind C-Statements in geschweiften Klammern
- dürfen an beliebiger Stelle im Regelkörper stehen:

```
fiend:
  'c' { prf("Got Cop"); } 0 'p'
| { prf("Going"); } 'c' 0 {prf(" to get Cow")} 'w'
;
```

- werden entsprechend der Reihenfolge, in der sie im Ableitungsbaum auftreten, aufgerufen
  Beispiel macht deutlich: Gesammter Ableitungsbaum muss bekannt sein, bevor Semantik-Blöcke ausgeführt werden können !

Semantik zuweisen: Parameter

- Nichtterminalen können optionale Parameter zugeordnet werden
  - unterteilt in Eingabe- und Ausgabeparameter
  - können in Semantik-Blöcken beschrieben und gelesen werden
- Liste der Parameter wird in Regelkopf deklariert:

  ```
  nonterminal <%in paramlist  %out paramlist >
  ```

  wobei `paramlist` eine durch Kommata getrennte Liste von typisierten Bezeichnern ist

Beispiel:

```
blowup_term<%in float p, float q  %out float n> :
  ...
```

Semantik zuweisen: Parameter

- bei Benutzung eines Nichtterminals im Regelkörper müssen dessen Parameter mit Bezeichnern belegt werden:

```
blowup_term<%in float p, float q  %out float n> :
  blowup_term<q,q,x> '+' factor<p,q,y> { *n = (x+y)*p; }
```

innerhalb eines Semantik-Blockes verfügbar:

- alle im Regelkopf deklarierten Eingabe-Parameter
- alle im Regelkopf deklarierten Ausgabe-Parameter
- alle Ausgabe-Parameter von im Regelkörper vorhergehenden Einträgen

Semantik zuweisen: Parameter

Kurzschreibweisen:

- Typen können optional ausgelassen werden, Parameter erhalten automatisch den Typ `YYSTYPE`
- werden `%in`, `%out` ausgelassen, so werden alle Parameter zu Ausgabeparametern

Beispiel:

```
A<%out YYSTYPE a, YYSTYPE b>
```

ist äquivalent zu

```
A<a,b>
```

Semantik zuweisen: Parameter

jedem Token ist generisch ein Ausgabeparameter vom Typ YYSTYPE zugewiesen:

```
%token NUMBER;


sum<n>:
  NUMBER<x> '+' NUMBER<y> {*n = x + y;}
```

Mehrdeutige Grammatiken

es gibt Grammatiken, in welchen ein Wort verschiedene Ableitungen besitzt

- Accent merkt dies nicht während der Parsergeneration
- zur Laufzeit wird festgestellt, falls Eingabe mehrere Ableitungspfade besitzt

Mehrdeutige Grammatiken

es gibt Grammatiken, in welchen ein Wort verschiedene Ableitungen besitzt

- Accent merkt dies nicht während der Parsergeneration
- zur Laufzeit wird festgestellt, falls Eingabe mehrere Ableitungspfade besitzt

Accent Parser bricht beim Auftreten einer Mehrdeutigkeit mit Fehlerausgabe ab !

Auflösung von Mehrdeutigkeiten

Prioritäten für Alternativen vergeben

- anfügen von `%prio <const int>` hinter jeder Alternative
- bei Konflikten zwischen mehreren möglichen Alternativen wird Pfad über Alternative mit der höchsten Priorität gewählt

Auflösung von Mehrdeutigkeiten

Auswahl kürzester/längster Ableitungen

- Einfügen des Schlüsselwortes `%short` vor einem Nichtterminal bewirkt, das die kürzest mögliche Ableitung ausgewählt wird
- analog bewirkt `%long` Auswahl der längsten möglichen Ableitung

Technik hinter Accent

Jede EBNF Grammatik ist auflösbar nach BNF.

Wir betrachten im folgenden nur noch Grammatiken, deren Regelkörper lediglich Alternativen beinhalten !

Erzeugter Code: Realisierung der Semantik

Ableitungsbaum zum Zeitpunkt der Ausführung bereits bekannt

$\Rightarrow$ zu jeder Regel kann eine einfache Funktion generiert werden

$A ::= B|C$ wird zu

```
void A() {
  switch (yyselect()) {
    case B:
      B (); break;
    case C:
      C (); break;
  }
}
```

Erzeugte Funktion für Regel

- Parameter der Regel werden zu Aufrufparametern der Funktion
- ein `switch`-statement für den gesammten Regelkörper:
  - `case`s stehen für einzelne Alternativen
  - rekursiver Aufruf der Funktionen von in Alternative enthalten Nichtterminalen
- Semantik-Blöcke werden 1:1 an die entsprechenden Stellen der Funktion kopiert

Parsertechnik: Exhaustive Parsing

Exhaustive Parsing ist eine effiziente Realisierung einer Breitensuche über den Syntaxbaum zum finden einer Ableitung des Eingabewortes.

- klappt mit allen kontextfreien Sprachen
- kubische Laufzeit zum finden einer Ableitung

Parsertechnik: Exhaustive Parsing

jeder Regel $R ::= A_1 A_2 \ldots | B_1 B_2 \ldots | C_1 \ldots$ wird pro Alternative eine (benannte) Liste zugeordnet:

$R : A_1 A_2 \ldots$

$R : B_1 B_2 \ldots$

$R : \ldots$

$\ldots$

eine Instanz ist eine dieser Listen zusammen mit einem Positionsindex und einem Zeiger auf eine andere Instanz

Parsertechnik: Exhaustive Parsing

jeder Regel $R ::= A_1 A_2 \ldots | B_1 B_2 \ldots | C_1 \ldots$ wird pro Alternative eine (benannte) Liste zugeordnet:

$R : A_1 A_2 \ldots$

$R : B_1 B_2 \ldots$

$R : \ldots$

$\ldots$

eine Instanz ist eine dieser Listen zusammen mit einem Positionsindex und einem Zeiger auf eine andere Instanz

Notation (ohne Zeiger): $R : A_1 * A_2 A_3$, wobei $*$ den Positionsindex kennzeichnet

Parsertechnik: Exhaustive Parsing

Sei S Startsymbol der Grammatik

Parser erzeugt schrittweise Mengen $I_0, I_1, \ldots$ von Instanzen:

- gestartet wird mit der Menge $\{\emptyset : *S\langle eof\rangle\}$
- im Schritt i wird die Menge I_{i-1} zusammen mit gelesenen Token zur Menge I_i transformiert.
- Akzeptanzbedingung: $\{\emptyset : S\langle eof\rangle *\} \in I_i$
- Verwerfen bei $S_i = \emptyset$

Parsertechnik: Exhaustive Parsing

ein Schritt beinhaltet drei Aktionen:

- Vorhersage
- Vervollständigung
- Scanner-Schritt

Exhaustive Parsing: Vorhersage

Wir befinden uns in Schritt i.

falls Instanz k in I_i der Form

$k = R : \cdots * N \ldots$ N ist nicht-Terminal

existiert, ersetzen wir diese durch neue Instanzen

$N : *N_1 \ldots$ $N : N_1 \ldots$ ist gültige Liste

und setzen den Zeiger dieser Instanzen auf k

Exhaustive Parsing: Vervollständigung

falls Instanz k in I_i existiert, deren Positionsindex das Ende der Liste erreicht hat, wird diese durch Kopie der Zeiger-Instanz ersetzt

Der Positionsindex jener Kopie wird erhöht.

Exhaustive Parsing: Scanner-Schritt

Sei t das in Schritt i gelesene Token.

für alle Instanzen in I_i der Form:

$R : \cdots * t \ldots$

wird der Positionsindex erhöht

alle anderen Instanzen werden aus I_i entfernt

Exhaustive Parsing: Schritt

Ablauf Schritt i :

- starte mit $I_i := I_{i-1}$
- führe wiederholt Vorhersage und Vervollständigung aus, bis sich nichts mehr ändert
- mache Scanner-Schritt

Korrektheit des Verfahrens ist leicht einzusehen

Laufzeitanalyse kompliziert, Ergebnis: $O(n^3)$

(n ist Anzahl der Token)

Fazit

- Grammatiken ohne grossen Aufwand implementierbar
- Möglichkeiten, Semantik auf einfache Art einzubauen

Jedoch:

- muss man sich weiterhin mit lex "ärgern"
- Fehlerbehandlung nicht sinnvoll möglich
- Speicherbedarf und Abarbeitungszeit deutlich höher als bei prädiktiven Parsern

somit nur bedingt einsetzbar im Compilerbau

schnelle Entwicklung diverser Tools oder Prototypen möglich