

6. Syntaxgesteuerte Übersetzung

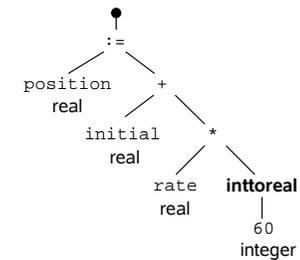
Inhalte der Vorlesung

1. Einführung
2. Lexikalische Analyse
3. Der Textstrom-Editor sed
4. Der Scanner-Generator lex
5. Syntaxanalyse und der Parser-Generator yacc
- 6. Syntaxgesteuerte Übersetzung
7. Übersetzungssteuerung mit make

6. Syntaxgesteuerte Übersetzung

- 6.1 Syntaxgesteuerte Definitionen
- 6.2 Konstruktion expliziter Syntaxbäume
- 6.3 Aufsteigende Auswertung

Attributierter Syntaxbaum



- Knoten mit Werten
 - Wert: durch Semantikregel des Knotens
- Semantikregel mit Seitenwirkungen möglich
 - Ausgabe
 - globale Variable verändern

Abhängigkeiten von Attributen

- synthetisiertes Attribut
 - Wert hängt nur von Nachfolgeknoten ab
- ererbtes Attribut
 - Wert hängt von Vorgängern und Geschwistern ab

- Auswertungsreihenfolge
 - muß Abhängigkeitsgraph beachten
 - Baum: nicht unbedingt explizit aufgebaut

Syntaxgesteuerte Definition

- ist kontextfreie Grammatik
plus:
 - Attributmenge für jedes Grammatiksymbol
 - Menge von Semantikregeln für jede Grammatikregel

- Attributgrammatik:
 - ist syntaxgesteuerte Definition
plus
 - Semantikregeln ohne Seitenwirkungen

Syntaxgesteuerte Definition eines einfachen Taschenrechners

Grammatikregel	Semantikregel
$l \rightarrow e \text{ NL}$	<code>print(e.val)</code>
$e \rightarrow e_1 + t$	<code>e.val := e₁.val + t.val</code>
$e \rightarrow t$	<code>e.val := t.val</code>
$t \rightarrow t_1 * f$	<code>t.val := t₁.val · f.val</code>
$t \rightarrow f$	<code>t.val := f.val</code>
$f \rightarrow (e)$	<code>f.val := e.val</code>
$f \rightarrow \text{NUMBER}$	<code>f.val := NUMBER.val</code>

S-attributierte Definition

- syntaxgesteuerte Definition
- nur synthetisierte Attribute
 - Wert hängt nur von Nachfolgeknoten ab

- Beispiel:
 - Taschenrechner eben
- kann von Blättern zur Wurzel ausgewertet werden
 - leicht

Erebt Attribute

- Wert hängt von Vorgängern und Geschwistern ab
- für Abhängigkeit von Kontext
 - Beispiel: Typ eines Bezeichners

Erebt Attribute: Beispiel

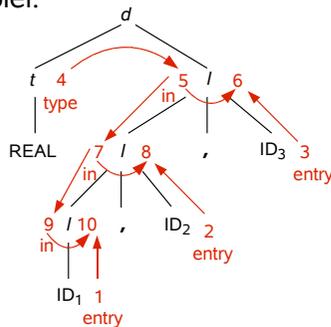
Grammatikregel	Semantikregel
$d \rightarrow t \mid$	$l.in := t.type$
$t \rightarrow INT$	$t.type := integer$
$t \rightarrow REAL$	$t.type := real$
$l \rightarrow l_1 , ID$	$l_1.in := l.in$ $addtype(ID.entry, l.in)$
$l \rightarrow ID$	$addtype(ID.entry, l.in)$

Ableitungsbaum mit Attribut „in“ an l -Knoten



Abhängigkeitsgraph

- Abhängigkeiten zwischen Attributen
- wird über Ableitungsbaum konstruiert
- Beispiel:



Abhängigkeitsgraph (2)

- wenn möglich: Abhängigkeitsgraphen schrittweise an Tafel unter Beamer-Bild der Vor-Folie malen
- Der Abh.-Graph hat für jedes Attribut einen Knoten
- „1“, „...“: Sind die Knoten des Abh.-Graphen
- „1“, „2“, „3“: Wir fangen hier mit den Attributen an, die von nichts abhängen
 - ID_3 hat Attribut „entry“
- „6“, „7“, „8“: Knoten für Scheinattribute
 - je eines für jeden Prozeduraufruf
- Siehe auch Drachenbuch S. 346f.
- „l“ = steht für „list (of IDs)“

Auswertungsreihenfolge

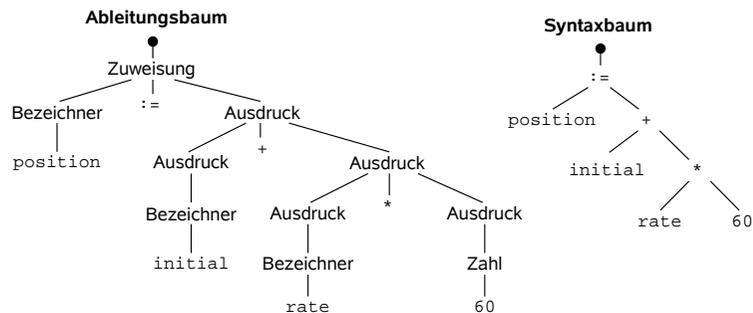
- nötig: topologische Sortierung
 - Reihenfolge für Auswertung, die Abhängigkeiten einhält
- verschiedene Methoden:
 - Syntaxbaummethoden
 - explizite topologische Sortierung
 - für jeden Syntaxbaum wieder neu
 - regelbasierte Methoden
 - feste, aber explizite Reihenfolge für jede Regel
 - von Hand oder automatisch
 - unbewußte Methoden
 - beliebiges festes Schema
 - z.B. so, wie Parser gerade die Grammatikregeln erkennt

6. Syntaxgesteuerte Übersetzung

- 6.1 Syntaxgesteuerte Definitionen
- 6.2 Konstruktion expliziter Syntaxbäume
- 6.3 Aufsteigende Auswertung

Syntaxbaum

- verdichteter Ableitungsbaum
 - Operatoren und Schlüsselworte in Knoten verschoben



Konstruktion expliziter Syntaxbäume

- Vorteil:
 - entkoppelt Syntaxanalyse und Übersetzung
 - Reihenfolge der Analyse beschränkt nicht Generierung
- Nachteil:
 - Platzbedarf ggf. hoch

Konstruktion expliziter Syntaxbäume für Ausdrücke

- ein Knoten pro Operator und pro Operand
 - Knoten z.B. implementiert als Record
 - OO-Sprachen: Knoten als Objekt
 - Felder:
 - Werte
 - Zeiger auf andere Records

Rekursive Auswerter für Attribute

- für explizite Syntaxbäume
 - Auswertung der Attribute erst *nach* Fertigstellung des Syntaxbaums
- Realisierung: gegenseitig rekursive Funktionen
 - eine Funktion pro Nichtterminal bzw. Symbol
 - Aufruf der Funktionen für Unterknoten in geeigneter Reihenfolge:
 - L-attributierte Definition: links nach rechts
 - sonst: geeignete andere Reihenfolge
- evtl. mehrere Durchläufe zur vollständigen Auswertung nötig

6. Syntaxgesteuerte Übersetzung

- 6.1 Syntaxgesteuerte Definitionen
- 6.2 Konstruktion expliziter Syntaxbäume
- 6.3 Aufsteigende Auswertung

S-attributierte Definition

- nur synthetisierte Attribute
 - Wert hängt nur von Nachfolgeknoten ab
- kann von Parser mit aufsteigender Analyse „nebenher“ ausgewertet werden
 - *ohne* expliziten Syntaxbaum
- Idee der Auswertung:
 - synthetisierte Attribute mit auf Parser-Stack
 - bei jeder Reduktion:
 - Werte mit von Stack nehmen
 - neuen Wert berechnen
 - neuen Wert mit neuem Nichtterminal auf Stack legen

L-attributierte Definition

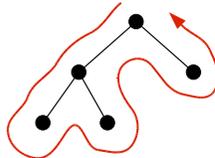
- syntaxgesteuerte Definition
- Bedingung für ererbtes Attribut von b_j :
 - sei:
Regel $a \rightarrow b_1 b_2 \dots b_{j-1} b_j \dots b_n$
 - dann:
Attribut von b_j hängt nur ab von
 1. Attributen von weiter links stehenden b_1, b_2, \dots, b_{j-1} und
 2. ererbten Attributen von a
- keine Einschränkungen für synthetisierte Attribute

Eigenschaften L-attributierter Definitionen

- erlaubt auch (bestimmte) ererbte Attribute
- kann immer durch Depth-First-Traversierung (s.u.) des Syntaxbaums ausgewertet werden
- alle syntaxgesteuerten Definitionen über LL(1)-Grammatiken sind L-attributiert
 - jede LL(1)-Grammatik ist auch eine LR(1)-Grammatik
 - daher immer durch yacc auswertbar
- viele syntaxgesteuerten Definitionen über LR(1)-Grammatiken sind L-attributiert
- Name „L“:
Attributinformation fließt von links nach rechts

Depth-First-Traversierung des Syntaxbaums

- **procedure** dfvisit(k : knoten);
begin
 for n := alle Nachfolger von k , links \rightarrow rechts **do**
 begin
 berechne ererbte Attribute von n ;
 dfvisit(n);
 end;
 berechne synthetisierte Attribute von k ;
end



Übersetzungsschema

- Notation zur Spezifikation der Übersetzung während der Syntaxanalyse
 - kein expliziter Syntaxbaum notwendig
- kontextfreie Grammatik
 - Attribute an Grammatiksymbolen
 - synthetisiert und ererbt
 - semantische Aktionen in rechte Seiten der Regeln eingefügt, eingeschlossen in Klammern { }
 - Zeitpunkt der Auswertung dadurch explizit
- yacc verwendet Übersetzungsschemata

Übersetzungsschema: Beispiel

- Taschenrechner für ganze Zahlen mit yacc
 - hier nur synthetisierte Attribute
 - daher alle semantischen Aktionen immer am Ende

Demo



Übersetzungsschema: Beispiel (2)

- calc-int.l

```
%{
#include <math.h>
#include "calc-int.tab.h"
}%
%option noyywrap
DIGIT [0-9]
%%
{DIGIT}+      {
                yy1val = atoi(yytext);
                return NUMBER;
              }

"="           {
                }
"("           {
                }
")"           {
                }
"+"           {
                }
"_"           {
                }
"*"           {
                }
"/"           { return *yytext; }
[[:space:]]   { /* ignoriere Whitespace */ }
.             { return ILLEGAL_CHAR; }
```



Übersetzungsschema: Beispiel (3)

- calc-int.y

```
%{
#include <stdio.h>
#define YERROR_VERBOSE
void yyerror(char *);
}%
%token NUMBER ILLEGAL_CHAR
%left '-' '+'
%left '*' '/'
%%
eingabe: /* empty */
        | eingabe berechnung
        ;
berechnung:
        term '=' { printf("Ergebnis: %d\n", $1); }
        ;
term:
        NUMBER
        | term '+' term { $$ = $1 + $3; }
        | term '-' term { $$ = $1 - $3; }
        | term '*' term { $$ = $1 * $3; }
        | term '/' term { $$ = $1 / $3; }
        | '(' term ')' { $$ = $2; }
        ;
%%
void yyerror(char *msg) {
    printf("\nEingabefehler: %s\n", msg);
}
int main() {
    return yyparse();
}
```



Übersetzungsschema: Beispiel (4)

- Variable `yy1val`: Attributwert für Token
 - von lex gesetzt, von yacc gelesen
- semantische Aktionen in Programmiersprache C
- „\$\$“: Wert des Attributs des Nichtterminals auf der linken Seite
- „\$1“: Wert des Attributs des Grammatiksymbols auf der rechten Seite ganz links
 - liegt auf Parser-Stack
- „\$2: ... 2. Symbol von links ...

Verschiedene Typen für die Attributwerte

- Default: Attribute sind int
- aber Normalfall: Typen der Attribute verschieden
- Beispiel:
Taschenrechner für Fließkommazahlen und Strings

Demo



Verschiedene Typen für die Attributwerte (2)

- calc-types.l

```
%{
#include <stdlib.h>
#include <math.h>
#include <string.h>
#include "calc-types.tab.h"
char *dequote(const char *);
%}
%option noyywrap
DIGIT [0-9]
%%
{DIGIT}+{".{DIGIT}"}? {
    yyval.zahl = atof(yytext);
    return NUMBER;

}

\**{[^\\n]+}\\** {
    yyval.string = dequote(yytext);
    return STRING;

}

"+="
"+("
"+){"
"++"
"+-="
"+){"
"+){"
/* ignore white space */
[:space:] {
    return ILLEGAL_CHAR;
}
.
%%
char *retVal(const char *s) {
    char *retVal;
    retVal = (char *) malloc(strlen(s)+2);
    strncpy(retVal, s, strlen(s)-2);
    return retVal;
}
```



Verschiedene Typen für die Attributwerte (3)

- calc-types.y

```
%{
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#define YERROR_VERBOSE
void yyerror(char *);
char *addString(const char *, const char *);
%}
union {
    double zahl;
    char * string;
}
token ILLEGAL_CHAR
token <zahl> NUMBER
token <string> STRING
%token "+"
%left "*"
%type <zahl> zahlTerm
%type <string> stringTerm
%%
eingabe: /* empty */
    | eingabe zahlBerechnung
    | eingabe stringBerechnung
    ;
zahlBerechnung: zahlTerm "=" { printf("Ergebnis: %g\n", $1); }
stringBerechnung: stringTerm "=" { printf("Ergebnis: \"%s\"\n", $1); }
    ;
zahlTerm: NUMBER
    | zahlTerm "+" zahlTerm { $$ = $1 + $3; }
    | zahlTerm "-" zahlTerm { $$ = $1 - $3; }
    | zahlTerm "*" zahlTerm { $$ = $1 * $3; }
    | zahlTerm "/" zahlTerm { $$ = $1 / $3; }
    | "(" zahlTerm ")" { $$ = $2; }
    ;
stringTerm: STRING
    | stringTerm "+" stringTerm { $$ = addString($1, $3); }
    ;
%}
char *addString(const char *s1, const char *s2) {
    char *retVal;
    retVal = (char *) malloc(strlen(s1)+strlen(s2)+1);
    strcpy(retVal, s1);
    strcat(retVal, s2);
    return retVal;
}
void yyerror(char *msg) {
    printf("Syntaxfehler: %s\n", msg);
}
int main() {
    return yyparse();
}
```



Verschiedene Typen für die Attributwerte (4)

- %union
 - definiert alle mögliche Typen von Attributen
 - wie C-union
- Zugriff auf `yylval`
 - jetzt durch `yylval.xy`
 - auch in lex-Datei
- Zuordnung eines Typs zu Nichtterminal
 - %type <xy> meinNichtTerminal
- Zuordnung eines Typs zu Symbol
 - %token <xy> meinToken
- Alternative Typzuordnung: `(xy)`, `$(xy)>1`, ...

Die Default-Aktion von yacc

- falls keine Aktion für eine Regel
- Default: „ $$$ = 1 ;“
 - nur, falls Typen zusammenpassen
 - nur, falls nicht leere Regel (kein „ $$1$ “)
- Implementation:
 - $$1$ und $$$$ haben ohnehin gleichen Platz auf Stack

Implizites Attribut für Position in der Eingabe

- Position von Symbolen/Nichtterminalen wichtig für Fehlermeldungen
- Beispiel:
Taschenrechner für ganze Zahlen, erweitert
 - (nur sinnvoll, wenn er aus Datei liest)

Demo



Implizites Attribut für Position in der Eingabe (2)

- calc-int-loc.l

```
{
#include <math.h>
#include "calc-int-loc.tab.h"
extern YYSTYPE yylloc;
#define YY_USER_INIT { \
    yylloc.first_line = 1; yylloc.first_column = 0; \
    yylloc.last_line = 1; yylloc.last_column = 0; \
}
#define YY_USER_ACTION { \
    yylloc.first_column = yylloc.last_column+1; \
    yylloc.last_column += yyleng; \
}
}
%option noyywrap
DIGIT [0-9]
%%
[DIGIT]+ {
    yyval = atoi(yytext);
    return NUMBER;
}
==
+|-
+|-
+|-
+|-
+|-
/*
[ \t] { /* ignoriere Whitespace */
}
\n { /* ignoriere Whitespace, setze Position in nächste Zeile */
    yylloc.first_line++;
    yylloc.first_column = 0;
    yylloc.last_line++;
    yylloc.last_column = 0;
}
. { return ILLEGAL_CHAR; }
```



Implizites Attribut für Position in der Eingabe (3)

- `./calc-int-loc < calc-int-loc.txt`
- lex muß yacc Position des Symbols mitteilen:
Variable `yylloc`
 - von yacc definiert, falls „ $@$$ “, ... (s.u.) verwendet
 - Default: 4 Felder, wie angegeben
- Makro `YY_USER_INIT` bei Initialisierung von Lexer ausgeführt
- Makro `YY_USER_ACTION` vor jeder Lexer-Aktion ausgeführt
- Newline muß Position auf nächste Zeile setzen



Implizites Attribut für Position in der Eingabe (4)

- calc-int-loc.y

```

11
12 #include <stdio.h>
13 #define YYERROR_VERBOSE
14 void yyerror(char *);
15
16 #token NUMBER ILLEGAL_CHAR
17 %left '-' '+'
18 %left '*' '/'
19 %
20
21 eingabe:
22     /* empty */
23     | eingabe berechnung
24
25 berechnung:
26     term '-' { printf("Ergebnis: %d\n", $1); }
27     | error '-'
28
29 term:
30     NUMBER { $0 = $1; }
31     | term '*' term { $0 = $1 * $3; }
32     | term '^' term { $0 = $1 ^ $3; }
33     | term '**' term { $0 = $1 * $3; }
34     | term '/' term
35     {
36         if ($3) {
37             $0 = $1 / $3;
38         }
39         else {
40             $0 = 0;
41             printf("Warn Bereich %d(%d)-%d(%d): ",
42                   $1.first_line, $1.first_column,
43                   $3.last_line, $3.last_column);
44             printf("Division durch Null\n");
45             YYERROR;
46         }
47     }
48     | (" term ") { $0 = $2; }
49
50
51 void yyerror(char *msg) {
52     printf("Warn Position %d(%d): ",
53           yyloc.first_line, yyloc.first_column);
54     printf("%s\n", msg);
55 }
56
57 int main() {
58     yysetdebug(YYDEBUG);
59 }

```



Implizites Attribut für Position in der Eingabe (5)

- „@\$“, ... muß mind. einmal verwendet sein, damit yylloc existiert
 - notfalls irgendwo redundante Aktion mit Zuweisung „@\$ = @1“
 - wie hier hinter „NUMBER“
 - „@1“: Position des ersten Grammatiksymbols
 - „@\$“ Position des Nichtterminals für ganze Regel
- Zugriff:
 - in Aktion mit „@1“, ...
 - in yyerror() nur auf Eingabeposition in yylloc



Implizites Attribut für Position in der Eingabe (6)

- Default-Aktion für Eingabeposition:
 - „@\$“: erstes Zeichen von „@1“ bis letztes Zeichen des letzten Symbols
 - Beispiel: Division durch Null, Klammerausdruck im Nenner
- Makro YYERROR: löst Syntaxfehler aus
 - abgefangen durch *error*-Regel für *berechnung*

Übersetzungsschema mit semantischen Aktionen in der Mitte

- bekanntes Beispiel: Variablendeklaration in C
 - „int i, j, k“ und „real i, j, k“
 - hier nur zwei Typen

Demo



Übersetzungsschema mit semantischen Aktionen in der Mitte (2)

- typedecl-glob.l

```
%{
#include <string.h>
#include "typedecl-glob.tab.h"
%}
%option noyywrap
IDENT  [[[:alpha:]]_]+
%%
"int"      { return INT; }
"real"     { return REAL; }
{IDENT}    {
            yylval.name = strdup(yytext);
            return ID;
          }
",,"       { return *yytext; }
[[[:space:]] { /* ignoriere Whitespace */ }
.          { return ILLEGAL_CHAR; }
```



Übersetzungsschema mit semantischen Aktionen in der Mitte (3)

- typedecl-glob.y

```

%{
#include <stdio.h>
#define YYERROR_VERBOSE
void yyerror(char *);
void addtype(char *, int);
#define T_INT 0
#define T_REAL 1
int curr_type;
%}
%union {
int idtype;
char * name;
}
%token INT REAL ID ILLEGAL_CHAR
%type <name> ID
%type <idType> type
%verbose
%%
def:      type { curr_type = $1; } list
;
type:     INT { $$ = T_INT; }
| REAL { $$ = T_REAL; }
list:     list ',' ID { addtype($3, curr_type); }
| ID { addtype($1, curr_type); }
;
%%
void addtype(char *id, int type) {
printf("Adding type %s for id '%s'\n", (type ? "REAL" : "INT"), id);
}
void yyerror(char *msg) {
printf("\n%s\n", msg);
}
int main() {
return yyparse();
}

```



Übersetzungsschema mit semantischen Aktionen in der Mitte (4)

- *keine* ererbten Attribute verwendet
 - stattdessen Kommunikation über globale Variable
 - Problem: geht nicht rekursiv
 - *hier* aber OK

Aufsteigende Auswertung ererbter Attribute

- Problem: Erben abwärts, Auswertung aufwärts
- (meist) Lösung: L-attributierte Definitionen
- hinreichende Regeln für semantische Aktionen:
 1. ererbtes Attribut für b_j bereits links davon berechnet
 2. kein Zugriff auf synthetisierte Attribute weiter rechts
 - läßt yacc sowieso nicht zu
 3. synthetisiertes Attribut für a erst ganz rechts berechnet

Problem mit eingebetteten Aktionen in Übersetzungsschemata

- Parser reduziert, wenn ganze Regel gelesen
- aber: eingebettete Aktion kommt vorher

- Lösung: Transformation der Grammatik:
Entfernen eingebetter Aktionen
 - yacc macht es intern automatisch

Entfernen eingebetteter Aktionen aus Übersetzungsschemata

- Transformation der Grammatik
- führe für jede eingebettete Aktion i ein neues Markierungsnichtterminal m_i ein und eine Regel $m_i \rightarrow \epsilon$
- füge m_i an die Stelle der Aktion ein und verschiebe die Aktion ans Ende der neuen Regel

Demo



Entfernen eingebetteter Aktionen aus Übersetzungsschemata (2)

- siehe typedecl-glob.output
- Problem: neue Regel kann evtl. Konflikt erzeugen, weil sie früher reduziert

Erben von Attributen auf dem Parser-Stack

- eine Implementierung des Erbens von Attributen
 - erfüllt automatisch die Bedingungen für L-attributierte Definitionen
 - yacc macht es so
- Idee: die bisher berechneten Attribute für die aktuelle Regel liegen alle im Parser-Stack
 - Zugriff durch geeignete Indizierung
 - Attribut des Grammatiksymbols direkt links von der Aktion liegt ganz oben

Demo



Erben von Attributen auf dem Parser-Stack (2)

- typedecl.y

```
#include <stdio.h>
#define YYERROR_VERBOSE
void yyerror(char *);
void addtype(char *, int);
#define T_INT 0
#define T_REAL 1
}
%union {
    int idtype;
    char * name;
}
%token INT REAL ID ILLEGAL_CHAR
%type <name> ID
%type <idType> type
%verbose
%%
def:      type { $<idType>$ = $1; } list
;
type:    INT { $$ = T_INT; }
        | REAL { $$ = T_REAL; }
;
list:    '.' ID { addtype($3, $<idType>0); }
        | ID { addtype($1, $<idType>0); }
;

%%
void addtype(char *id, int type) {
    printf("Adding type %s for id '%s'\n", (type ? "REAL" : "INT"), id);
}
void yyerror(char *msg) {
    printf("\n%s\n", msg);
}
int main() {
    return yyparse();
}
```



Erben von Attributen auf dem Parser-Stack (3)

- yacc:

- „\$0“: oberster Wert auf Parser-Stack
 - „\$-1“: zweitoberster Wert, ...
- „\$\$“ in eingebetteter Regel:
 - andere Bedeutung:
 - Wert für diese Aktion, nicht für die ganze Regel
 - \$<idType>\$: explizite Typangabe nötig, da unbenannt

- **Vorsicht:**

Bei jedem Reduzieren von *list* muß das Richtige auf Stack sein!

- Aktion „\$\$ = \$1“ hier redundant, aber ggf. sicherer
- alle Verwendungen von *list* prüfen!

Inhalte der Vorlesung

1. Einführung
2. Lexikalische Analyse
3. Der Textstrom-Editor sed
4. Der Scanner-Generator lex
5. Syntaxanalyse und der Parser-Generator yacc
- 6. Syntaxgesteuerte Übersetzung
7. Übersetzungssteuerung mit make